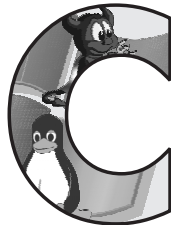


# **PROGRAMANDO EM**



**PARA LINUX, UNIX E WINDOWS**

Escrever um livro não é uma tarefa fácil, nunca foi. Toma tempo, exige pesquisa e dedicação. Como as editoras não desejam mais publicar este título, nada mais natural que tentar comercializa-lo na forma de arquivo (semelhante a um ebook). Mas, esta atividade também toma tempo e exige dedicação.

Pensando assim, resolvi liberar este livro para consulta pública, se você acha que este livro te ajudou e quiser colaborar comigo, passe numa lotérica, e deposite o valor que achar que deve.

Terminou de pagar a conta de luz, telefone ou água e sobrou troco, você pode depositar na minha conta. Eu acredito que nós dois podemos sair ganhando, você porque teve acesso a um bom material que te ajudou e eu como incentivo a continuar escrevendo livros. Caso de certo, talvez os próximos livros nem sejam publicados por uma editora e estejam liberados diretamente para sua consulta.

Qualquer valor depositado será direcionado para a conta poupança do meu filho, para quando ele estiver na maioridade ter recursos para começar um negócio próprio, financiar seus estudos, etc.

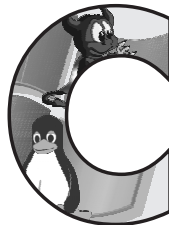
Dados para depósito:

Marcos Aurelio Pchek Laureano.  
Banco: 104 - Caixa Econômica Federal  
Agência: 1628  
Operação: 001  
Conta: 6012-2

Curitiba, 13 de março de 2012.



# **PROGRAMANDO EM**



**PARA LINUX, UNIX E WINDOWS**

**Marcos Laureano**

Copyright© 2005 por Brasport Livros e Multimídia Ltda.

Todos os direitos reservados. Nenhuma parte deste livro poderá ser reproduzida, sob qualquer meio, especialmente em fotocópia (xerox), sem a permissão, por escrito, da Editora.

Editor: Sergio Martins de Oliveira  
Diretora Editorial: Rosa Maria Oliveira de Queiroz  
Assistente de Produção: Marina dos Anjos Martins de Oliveira  
Revisão: Maria Helena dos Anjos Martins de Oliveira  
Editoração Eletrônica: Abreu's System Ltda.  
Capa: UseDesign

Dados Internacionais de Catalogação na Publicação (CIP)  
(Câmara Brasileira do Livro, SP, Brasil)

Laureano, Marcos

Programando em C para Linux, Unix e Windows / Marcos Laureano. – Rio de Janeiro: Brasport, 2005.

Bibliografia

ISBN 85-7452-233-3

1. C (Linguagem de programação para computadores) 2. LINUX (Sistema operacional de computador) 3. UNIX (Sistema operacional de computador) 4. WINDOWS (Sistema operacional de computador) I. Título

05-6860

CDD-005.133

Índices para catálogo sistemático:

1. C : Linguagem de programação : Computadores :  
Processamento de dados 005.133

**BRASPORT Livros e Multimídia Ltda.**

Rua Pardal Mallet, 23 – Tijuca

20270-280 Rio de Janeiro-RJ

Tels. Fax: (21) 2568.1415/2568.1507/2569.0212/2565.8257

e-mails: [brasport@brasport.com.br](mailto:brasport@brasport.com.br)

[vendas@brasport.com.br](mailto:vendas@brasport.com.br)

[editorial@brasport.com.br](mailto:editorial@brasport.com.br)

site: [www.brasport.com.br](http://www.brasport.com.br)



## Agradecimentos

Este trabalho não teria saído se não fosse pelo apoio da minha esposa Margarete e do meu querido filho Luiz Otavio. Foram eles que agüentaram o meu mau humor após longas noites de trabalho.

Agradeço à Brasport pela oportunidade de publicar meu livro sobre um tema onde vários autores já trabalharam (é claro que este livro tem um diferencial em relação aos demais!).

Aos meus colegas professores e alunos que ajudaram a melhorar este material nos últimos anos.

*Copiar de um autor é plágio;  
copiar de muitos autores é pesquisa.*

Wilson Mizner, escritor americano.





## Sobre o Autor

**Marcos Laureano** é tecnólogo em Processamento de Dados pela ESSEI, Pós-graduado em Administração pela FAE Business School e Mestre em Informática Aplicada pela Pontifícia Universidade Católica do Paraná. Doutorando na Universidade de Lisboa, onde irá desenvolver trabalhos na área de segurança em máquinas virtuais e sistemas embarcados. Trabalha com programação em C no ambiente Unix (AIX/HP-UX) desde 1997 e Linux desde 2000, sendo especialista em segurança de sistemas operacionais. É professor de graduação e pós-graduação, tendo lecionado em várias instituições nos últimos anos. É autor de vários guias de utilização/configuração de vários aplicativos para os ambientes Unix e Linux. Possui vários artigos publicados sobre programação e segurança de sistemas.

Atualmente leciona disciplinas relacionadas com segurança, programação e sistemas operacionais nos cursos de graduação e pós-graduação na FAE Centro Universitário, e atua como consultor na área de projetos de desenvolvimento e segurança de sistemas.

O autor pode ser contactado pelo e-mail [marcos@laureano.eti.br](mailto:marcos@laureano.eti.br) ou através de sua página [www.laureano.eti.br](http://www.laureano.eti.br), onde está disponível vasto material sobre programação, segurança de sistemas e sistemas operacionais. O autor mantém um fórum em seu site para discussão sobre segurança de sistemas, sistemas operacionais e programação.







# Introdução

*Ser professor é... ter breves momentos de satisfação  
num dia-a-dia feito de desgostos !  
(Anônimo)*

**E**ste material não pretende ser o guia definitivo sobre programação em C (nem é a sua pretensão), mas sim introduzir as informações necessárias para começar a programar nesta poderosa linguagem e, principalmente, mostrar que a programação em C para o ambiente Linux e Unix é mais simples do que parece.

É recomendável que o leitor tenha conhecimentos prévios sobre o sistema Linux/Unix, principalmente com relação aos comandos básicos de utilização. Diferente do que muitos pensam, para você desenvolver programas para o ambiente Linux/Unix é necessário conhecer apenas alguns comandos do sistema operacional (`grep`, `find`, `ps`, `ls`, `mv`, `cp`, `rm`, `rmdir`, `mkdir`, `vi`, `more` e é claro o `gcc/cc`).

Para a plataforma Linux é utilizado o compilador GNU C Compiler ou simplesmente `gcc`. É possível obter mais detalhes sobre o `gcc` em <http://www.gnu.org> ou <http://gcc.gnu.org>.

Para a plataforma Windows é utilizado o Borland C++ 5.5 ou simplesmente `bcc32`. É possível obter mais detalhes sobre o `bcc32` em [www.borland.com](http://www.borland.com) ou [http://www.borland.com/products/downloads/download\\_cbuilder.html](http://www.borland.com/products/downloads/download_cbuilder.html).

## 2 ♦ Programando em C para Linux, Unix e Windows

Ainda para a plataforma Windows é utilizado o LCC-Win32, que é um ambiente integrado e simples para escrever programas em C. O LCC-Win32 pode ser encontrado em <http://www.cs.virginia.edu/~lcc-win32/>.

Os três compiladores podem ser utilizados livremente em seu computador. Os programas exemplos foram testados nos três compiladores e funcionam de forma idêntica (exceto na declaração da função `main` e capítulos específicos).

Embora a linguagem C seja portátil, algumas funções são específicas para cada sistema operacional. Do capítulo 1 até o 17, as informações vistas são válidas para qualquer sistema operacional. A partir do capítulo 18 até o 22 são vistas informações a respeito de particularidades do C para o sistema Unix e Linux. O capítulo 23 trata de programação para rede. Embora as opções de programação para rede sejam vastas, neste capítulo é vista apenas uma pequena parte (introdutória). No apêndice A são vistas questões sobre recursividade, pesquisas e ordenação. Os apêndices B e C mostram como obter ajuda no sistema Linux e Unix e como compilar programas em C nestes ambientes. O apêndice D mostra como instalar e compilar um programa utilizando o LCC-Win32. O apêndice E contém uma relação das funções mais utilizadas no ambiente Unix e Linux.

Caso o leitor ache alguma inconsistência (pode ocorrer, apesar de todas as precauções tomadas) peço que me contate através do e-mail [marcos@laureano.eti.br](mailto:marcos@laureano.eti.br).



*A programação hoje é uma corrida entre os engenheiros de software que lutam para construir programas maiores e mais à prova de idiotas enquanto o universo tenta produzir idiotas maiores e melhores. Até agora, o universo está vencendo.*

Rick Cook

## 1.1 História

A linguagem de programação C é uma linguagem estruturada e padronizada criada na década de 1970 por Ken Thompson e Dennis Ritchie para ser usada no sistema operacional Unix. Desde então espalhou-se por muitos outros sistemas operacionais e tornou-se uma das linguagens de programação mais usadas. A linguagem C tem como ponto forte a sua eficiência e é a linguagem de programação de preferência para o desenvolvimento de aplicações para sistemas operacionais, apesar de também ser usada para desenvolver aplicações mais complexas.

O desenvolvimento inicial da linguagem C ocorreu nos laboratórios *Bell* da AT&T entre 1969 e 1973. Deu-se o nome "C" à linguagem porque muitas das suas características derivaram de uma linguagem de programação anterior chamada "B". Há vários relatos que se referem à origem do nome "B": Ken Thompson dá crédito à linguagem de programação BCPL, mas ele também criou uma outra linguagem de programação chamada Bon, em honra da sua mulher Bonnie. Por volta de 1973, a linguagem C tinha-se tornado suficientemente poderosa para que grande parte do núcleo de Unix, originalmente escrito na linguagem de programação PDP-11/20 assembly, fosse reescrito em C. Este foi um dos primeiros núcleos de sistema operacional que foi implementado

## 4 ♦ Programando em C para Linux, Unix e Windows

numa linguagem sem ser o `assembly`, sendo exemplos anteriores o sistema Multics (escrito em `PL/I`) e TRIPOS (escrito em `BCPL`).

### 1.2 C de K&R

Em 1978, Ritchie e Brian Kernighan publicaram a primeira edição do livro *The C Programming Language*. Esse livro, conhecido pelos programadores de C como "K&R", serviu durante muitos anos como uma especificação informal da linguagem. A versão da linguagem C que ele descreve é usualmente referida como "C de K&R". K&R introduziram as seguintes características na linguagem:

- o Tipos de dados `struct`;
- o Tipos de dados `long int`;
- o Tipos de dados `unsigned int`;
- o O operador `==` foi alterado para `==`, e assim sucessivamente (o analisador léxico do compilador confundia o operador `=`. Por exemplo, `i == 10` e `i = +10`).

C de K&R é frequentemente considerada a parte mais básica da linguagem que é necessário que um compilador C suporte. Nos anos que se seguiram à publicação do C de K&R, algumas características "não-oficiais" foram adicionadas à linguagem, suportadas por compiladores da AT&T e de outros fornecedores. Estas incluíam:

- o Funções `void` e tipos de dados `void *`;
- o Funções que retornam tipos `struct` ou `union`;
- o Campos de nome `struct` num espaço de nome separado para cada tipo `struct`;
- o Atribuição a tipos de dados `struct`;
- o Qualificadores `const` para criar um objecto só de leitura;
- o Uma biblioteca-padrão que incorpora grande parte da funcionalidade implementada por vários fornecedores;
- o Enumerações;
- o O tipo de ponto flutuante de precisão simples.

### 1.3 C ANSI e C ISO

Durante os finais da década de 1970, a linguagem C começou a substituir a linguagem `BASIC` como a linguagem de programação de microcomputadores mais usada. Durante a década de 1980, foi adotada para uso no `PC IBM`, e a sua popularidade começou a aumentar significativamente. Ao mesmo tempo, Bjarne Stroustrup, juntamente com outros nos laboratórios Bell, começou a

trabalhar num projeto onde se adicionava programação orientada a objetos à linguagem C. A linguagem que eles produziram, chamada C++, é nos dias de hoje a linguagem de programação de aplicações mais comum no sistema operacional Windows da Microsoft, enquanto o C permanece mais popular no mundo Unix. Em 1983, o instituto norte-americano de padrões (ANSI) formou um comitê, X3j11, para estabelecer uma especificação do padrão da linguagem C. Após um processo longo e árduo, o padrão foi completo em 1989 e ratificado como ANSI X3.159-1989 "Programming Language C". Esta versão da linguagem é freqüentemente referida como C ANSI. Em 1990, o padrão C ANSI, após sofrer umas modificações menores, foi adotado pela Organização Internacional de Padrões (ISO) como ISO/IEC 9899:1990. Um dos objetivos do processo de padronização C ANSI foi o de produzir um sobreconjunto do C K&R, incorporando muitas das características não-oficiais subseqüentemente introduzidas. Entretanto, muitos programas já tinham sido escritos e não compilavam em certas plataformas, ou com um certo compilador, devido ao uso de bibliotecas não-padrão (por exemplo, interfaces gráficas) alguns compiladores não aderirem ao padrão C ANSI.

## 1.4 C99

Após o processo ANSI de padronização, as especificações da linguagem C permaneceram relativamente estáticas por algum tempo, enquanto que a linguagem C++ continuou a evoluir. Contudo, o padrão foi submetido a uma revisão nos finais da década de 1990, levando à publicação da norma ISO 9899:1999 em 1999. Este padrão é geralmente referido como "C99". O padrão foi adotado como um padrão ANSI em março de 2000. As novas características do C99 incluem:

- o Funções em linha ;
- o Levantamento de restrições sobre a localização da declaração de variáveis (como em C++) ;
- o Adição de vários tipos de dados novos, incluindo o `long long int` (para minimizar a dor da transição de 32-bits para 64-bits), um tipo de dados `boolean` explícito e um tipo `complex` que representa números complexos;
- o Disposições de dados de comprimento variável;
- o Suporte oficial para comentários de uma linha iniciados por `//`, emprestados da linguagem C++;
- o Várias funções de biblioteca novas, tais como `snprintf`;
- o Vários arquivos-cabeçalho novos, tais como `stdint.h`.

O interesse em suportar as características novas de C99 parece depender muito das entidades. Apesar do gcc e vários outros compiladores suportarem grande

## 6 ♦ Programando em C para Linux, Unix e Windows

parte das novas características do C99, os compiladores mantidos pela Microsoft e pela Borland não, e estas duas companhias não parecem estar muito interessadas adicionar tais funcionalidades, ignorando por completo as normas internacionais.

### 1.5 Comentários

Os comentários de um programa devem ser colocados entre `'/*'` e `'*/'`. O compilador ANSI C aceita os comentários entre `/*` e `*/`. Quaisquer textos colocados entre estes dois símbolos serão ignorados pelo compilador.

Um outro ponto importante é que se deve colocar um comentário no início de cada função do programa explicando a função, seu funcionamento, seus parâmetros de entrada e quais são os possíveis retornos que esta função pode devolver.

Alguns compiladores mais novos (e seguindo a padronização da linguagem C99) aceitam como comentários o `"/"`. A diferença básica é que com o `//` é possível fazer comentários apenas em uma linha.

Exemplos:

```
/* Isto é um comentário */
int i;           // índice do vetor de saída
float soma;      /* Soma dos valores pagos */
char letra;      /* este é um comentário de
                  02 linhas */
```

### 1.6 Constantes Numéricas

Sempre é preciso colocar constantes em um programa. A linguagem C permite a colocação de constantes, exigindo, porém, uma sintaxe diferenciada para que o compilador identifique o tipo da constante e realize o processamento adequado dela.

Na linguagem C tem-se os seguintes formatos para as constantes numéricas:

- o Números inteiros – De uma maneira geral basta colocar o número no programa para que o compilador entenda o formato e trabalhe de maneira adequada.
- o Números reais – A exceção vale quando se quer colocar uma constante `float`. Neste caso, é preciso indicar o formato através da letra `F` no final ou utilizar o formato de ponto flutuante (exemplo: 1.0) para que o compilador entenda. Pode-se também utilizar o formato científico para isto (exemplo: 0.1E+1).

- o Números octais – Se um número iniciar por zero, o compilador irá considerar este número como OCTAL, mudando o valor final. Exemplo: Se no programa for colocado 010, o compilador entenderá que foi colocado o valor 8 no programa, pois 010 é a representação octal do número 8. Outro exemplo: Se colocado 0019 será gerado um erro de compilação, pois o compilador não aceita os dígitos 8 e 9 em um número octal.
- o Números hexadecimais – Uma outra maneira de indicar um número para o compilador é o formato em hexadecimal. Este formato é muito útil quando se trabalha com *bits* e operações para ligar ou desligar determinados *bits* de uma variável. Uma constante será considerada em hexadecimal se a mesma começar por "0x". Neste caso são aceitos os dígitos 0 a 9 e as letras 'a' a 'f', maiúsculas ou minúsculas.

Exemplos:

Constante	Tipo
0xEF	Constante Hexadecimal (8 bits)
0x12A4	Constante Hexadecimal (16 bits)
03212	Constante Octal (12 bits)
034215432	Constante Octal (24 bits)
10 ou 34	Constante inteira
78U	Constante inteira sem sinal
20	Constante long
10F ou 1,76E+2	Constante de ponto flutuante

## 1.7 Outras Constantes

Além das constantes numéricas pode-se colocar constantes do tipo caractere. O compilador identifica estas constantes através dos apóstrofes. O compilador permite que se coloque um único caractere entre apóstrofo.

Existe uma maneira alternativa de se indicar o caractere quando o mesmo é um caractere de controle. Para isto basta colocar entre apóstrofo a barra invertida e o código ASCII do caractere desejado. Por exemplo, para se colocar um <Ctrl>A em uma variável, pode-se colocar '\1' e para se colocar um <Carriage Return> ou <ENTER> utiliza-se a constante '\13'.

Como alguns caracteres de controle são muito usados, existe uma maneira especial de se indicar estes caracteres. A seguir estão alguns formatos interpretados pelo compilador:

## 8 ♦ Programando em C para Linux, Unix e Windows

Código	Significado
<code>\b</code>	Retrocesso ( <i>back</i> )
<code>\f</code>	Alimentação de formulário ( <i>form feed</i> )
<code>\n</code>	Nova linha ( <i>new line</i> )
<code>\t</code>	Tabulação horizontal ( <i>tab</i> )
<code>\"</code>	Aspas
<code>\'</code>	Apóstrofo
<code>\0</code>	Nulo (0 em decimal)
<code>\\</code>	Barra invertida
<code>\v</code>	Tabulação vertical
<code>\a</code>	Sinal sonoro ( <i>beep</i> )
<code>\N</code>	Constante octal (N é o valor da constante)
<code>\xN</code>	Constante hexadecimal (N é o valor da constante)

De uma maneira geral, toda vez que o compilador encontrar a barra invertida ele não processará o próximo caractere, a não ser que seja um dos indicados antes.

O compilador também permite a criação de *strings* de caracteres. Para se colocar em constantes deste tipo, deve-se colocar a *string* entre aspas. Podem-se colocar caracteres especiais utilizando o formato visto antes dentro da *string*, que o compilador irá gerar o código adequado.

### Exemplos:

Caracteres

Caractere - `'a'`, `'F'`, `'('`, `'0'`

Código - `'\10'`, `'\0'`, `'\9'`

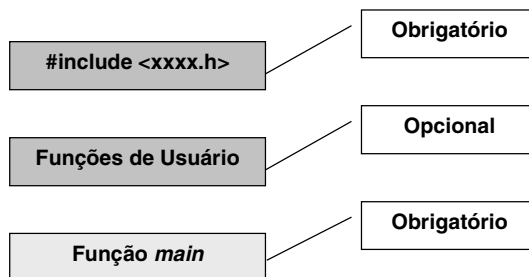
Especiais - `'\n'`, `'\t'`

Strings

`"Sistema de Controle\tRelatorio\n"`

## 1.8 Estrutura de um Programa

Um programa básico em C possui os seguintes blocos:





Um programa C deve possuir uma certa estrutura para ser válido. Basicamente têm-se três blocos distintos nos programas. Inicialmente deve-se ter uma seção onde serão feitos os *includes* necessários para o programa (será visto com mais detalhes). Por enquanto deve-se colocar a seguinte linha em todos os programas:

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

O segundo bloco é o bloco das funções definidas pelo usuário. Este bloco não é obrigatório e só existirá se o usuário definir uma função.

O último bloco, chamado de bloco principal, é obrigatório em qualquer programa C. Nele está definida a função `main`, que será a função por onde o programa começará a ser executado.

## 1.9 Função main

Todo programa em C deve ter uma função chamada `main`. É por esta função que será iniciada a execução do programa. Deve-se especificar o tipo da saída da função, que pode ser `int` ou `void`.

Caso seja colocado `int`, o valor retornado pela função `main` estará disponível para teste no sistema operacional.

Caso o retorno da função seja declarado como `void`, nada será retornado ao sistema operacional. Alguns compiladores podem exigir que o retorno da função `main` seja declarado como `int`.

Veja o exemplo:

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

```
void main ()
{
    printf ("Hello World\n");
}
```

ou

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

```
int main ()
{
    printf ("Hello World\n");
}
```

## 10 ♦ Programando em C para Linux, Unix e Windows

ou

```
#include <stdio.h>
int main(void)
{
    printf ("Hello World\n");
}
```

### 1.10 O que main devolve

De acordo com o padrão ANSI, a função `main` devolve um inteiro para o processo chamador (geralmente o sistema operacional). Devolver um valor em `main` é equivalente a chamar a função `exit` (capítulo 6) com o mesmo valor. Se `main` não devolve explicitamente um valor, o valor passado para o processo chamador é tecnicamente indefinido. Na prática, a maioria dos compiladores C devolvem 0 (zero).

Também é possível declarar `main` como `void` se ela não devolve um valor. Alguns compiladores geram uma mensagem de advertência (*warning*) se a função não é declarada como `void` e também não devolve um valor.

### 1.11 O C é "Case Sensitive"

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

Veja o Exemplo:

```
#include <stdio.h>

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

Programa 1.1

Resultado do Programa 1.1

Ola! Eu estou vivo!

A linha `#include <stdio.h>` diz ao compilador que ele deve incluir o arquivo-cabeçalho `stdio.h`. Neste arquivo existem declarações de funções úteis para entrada e saída de dados (*std* = *standard*, padrão em inglês; *io* = *Input/Output*, entrada e saída → *stdio* = Entrada e saída padronizadas). Sempre que for utilizada uma destas funções deve-se incluir este comando. O C possui diversos arquivos-cabeçalho.

A linha `int main()` indica que está sendo definida uma função de nome `main`. Todos os programas em C têm que ter uma função `main`, pois é esta função que será chamada quando o programa for executado. O conteúdo da função é delimitado por chaves `{ }`. O código que estiver dentro das chaves será executado seqüencialmente quando a função for chamada. A palavra `int` indica que esta função retorna um inteiro. Este retorno será visto posteriormente, quando estudarmos um pouco mais detalhadamente as funções do C. A última linha do programa, `return(0);`, indica o número inteiro que está sendo retornado pela função, no caso o número 0.

A única coisa que o programa realmente faz é chamar a função `printf()`, passando a *string* (uma *string* é uma seqüência de caracteres, que será visto posteriormente) `"Ola! Eu estou vivo!\n"` como argumento. É por causa do uso da função `printf()` pelo programa que deve-se incluir o arquivo-cabeçalho `stdio.h`. A função `printf()` neste caso irá apenas colocar a *string* na tela do computador. O `\n` é uma constante chamada de constante barra invertida. No caso, o `\n` é a constante barra invertida de *new line* e ele é interpretado como um comando de mudança de linha, isto é, após imprimir `Ola! Eu estou vivo!` o cursor passará para a próxima linha. É importante observar também que os comandos do C terminam com `;` (ponto-e-vírgula).

## 1.12 Palavras Reservadas do C

Todas as linguagens de programação têm palavras reservadas. As palavras reservadas não podem ser usadas a não ser nos seus propósitos originais, isto é, não é possível declarar funções ou variáveis com os mesmos nomes. Como o C é *case sensitive* pode-se declarar uma variável `For`, apesar de haver uma palavra reservada `for`, mas isto não é uma coisa recomendável de se fazer, pois pode gerar confusão.

A seguir são apresentadas as palavras reservadas do ANSI C:

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



*Aprecie as pequenas coisas, pois um dia você pode olhar para trás e perceber que elas eram as grandes coisas.*  
Robert Brault, jornalista.

## 2.1 Tipos Básicos

Para criar variáveis em um programa C deve-se indicar para o compilador qual o tipo desta variável. Uma variável pode ter um tipo básico, intrínseco à linguagem C ou tipo estruturado, montado pelo programador. Nesta seção serão vistos os tipos básicos já existentes na linguagem e como usá-los.

A linguagem C define os seguintes tipos básicos de variáveis:

- o `int` – Variável tipo inteira. Deve ser utilizada para se armazenar valor inteiro, com ou sem sinal.
- o `char` – Variável do tipo caracteres. Servirá para se armazenar um único caractere.
- o `float` – Para valores com casas decimais (reais) deve-se utilizar este tipo. Ele pode armazenar números reais com até 6 dígitos significativos.
- o `double` – É o mesmo que o anterior, só que pode armazenar mais dígitos, dando uma precisão maior nos cálculos com casas decimais.

O tipo `void` deve ser utilizado não para variáveis, mas sim para indicar que uma função não tem nenhum valor retornado ou não possui nenhum parâmetro de entrada.

A padronização ANSI C 99 especifica ainda mais 2 tipos de variáveis:

- o `_Bool` – Variável tipo booleana (verdadeiro ou falso). Ressalta-se que na linguagem C, em comparações lógicas, 0 (zero) é considerado falso e diferente de 0 (zero) é considerado verdadeiro.
- o `complex` – Variável para trabalhar com valores complexos (raízes imaginárias, por exemplo).

## 2.2 Abrangência e Modificadores de Tipo

A linguagem ANSI C determina para cada tipo intrínseco um certo tamanho em *bytes*. Este tamanho irá determinar a escala de valores que pode ser colocada dentro de um determinado tipo.

A seguir estão os limites de valores aceitos por cada tipo e o seu tamanho ocupado na memória. Também nesta tabela está especificado o formato que deve ser utilizado para ler/imprimir os tipos de dados com a função `scanf` e `printf` (vistos com mais detalhes no capítulo 3):

Tipo	Número de Bits	Formato para leitura e impressão	Início	Fim
<code>_Bool</code>	8	Não tem (pode-se utilizar <code>%d</code> )	0	1
<code>char</code>	8	<code>%c</code>	-128	127
<code>unsigned char</code>	8	<code>%c</code>	0	255
<code>signed char</code>	8	<code>%c</code>	-128	127
<code>int</code>	32	<code>%i</code>	-2.147.483.648	2.147.483.647
<code>unsigned int</code>	32	<code>%u</code>	0	4.294.967.295
<code>signed int</code>	32	<code>%i</code>	-2.147.483.648	2.147.483.647
<code>short int</code>	16	<code>%hi</code>	-32.768	32.767
<code>unsigned short int</code>	16	<code>%hu</code>	0	65.535
<code>signed short int</code>	16	<code>%hi</code>	-32.768	32.767
<code>long int</code>	32	<code>%li</code>	-2.147.483.648	2.147.483.647
<code>signed long int</code>	32	<code>%li</code>	-2.147.483.648	2.147.483.647
<code>unsigned long int</code>	32	<code>%lu</code>	0	4.294.967.295
<code>float</code>	32	<code>%f</code>	3,4E-38	3,4E+38
<code>double</code>	64	<code>%lf</code>	1,7E-308	1,7E+308
<code>long double</code>	80	<code>%Lf</code>	3,4E-4932	3,4E+4932

## 14 ♦ Programando em C para Linux, Unix e Windows

Estes limites podem ser verificados no arquivo `limits.h` do pacote C e são válidos para plataformas de 32 *bits*. Em plataformas de 16 *bits*, o `int` era definido com 16 *bits* (o equivalente a `short int` na plataforma de 32 *bits*). Em plataformas de 64 *bits*:

Tipo	Número de Bits	Formato para leitura e impressão	Início	Fim
<code>long</code>	64	<code>%li</code>	-9223372036854775806	9223372036854775807
<code>unsigned long</code>	64	<code>%lu</code>	0	18446744073709551615

Pode-se modificar o comportamento de um tipo básico, tanto no tamanho (espaço em memória) como no seu sinal (positivo ou negativo). Os modificadores de sinais indicam para o compilador considerar ou não valores negativos para o tipo inteiro. Apesar de existir a palavra `signed`, ela é padrão, não precisando ser colocada, mas pode sofrer influência do sistema operacional. A palavra `unsigned` indica para o compilador não considerar o *bit* de sinal, estendendo assim o limite da variável inteira.

Pode-se modificar o tamanho das variáveis do tipo `int` para que ele ocupe somente dois *bytes* na memória, reduzindo assim o limite de abrangência para -32768 a +32767. Para isto, coloca-se o modificador `short` na definição da variável, indicando que serão utilizados somente dois *bytes* de tamanho. Devem-se tomar alguns cuidados com a portabilidade, pois num sistema a variável pode ser considerada `signed` e em outros `unsigned`; em alguns sistemas operacionais (variações de Unix, OS/2 da IBM etc.) as variáveis são definidas por padrão com `unsigned` (sem sinal), em outros como `signed` (com sinal).

## 2.3 Nomenclatura de Variáveis

Toda variável de um programa deve ter um nome único dentro do contexto de existência dela. Para se formar o nome de uma variável é necessário seguir algumas regras impostas pelo compilador. Estas regras são:

1. O nome de uma variável deve começar por uma letra ou por um caractere `"_"` ("underline").
2. Os demais caracteres de uma variável podem ser letras, dígitos e `"_"`.
3. O compilador reconhece os primeiros 31 caracteres para diferenciar uma variável de outra.
4. Um ponto importante a ser ressaltado é que para o compilador C as letras maiúsculas são diferentes das letras minúsculas.

O processo de escolha de nome de uma variável é importante para a legibilidade de um programa em manutenções posteriores. Há algumas regras básicas que, se seguidas, irão melhorar muito a futura manutenção do programa.

1. Não utilize nomes que não tenham significados com o uso da variável. Por exemplo: uma variável "cont" utilizada para se guardar a soma de um procedimento. Melhor seria utilizar uma variável com o nome de "soma".
2. Se uma variável for utilizada para guardar a soma de um valor, por exemplo, total de descontos, além da função coloque também o conteúdo da mesma, chamando a variável de SomaDesconto.
3. Se desejar, coloque uma letra minúscula no início indicando o tipo da variável. Isto facilita muito o entendimento na fase de manutenção. Esta técnica é chamada de *Nomenclatura Húngara*. Procure utilizar esta técnica em todo o programa e mantenha uma única maneira de se indicar o tipo, pois pior que não ter uma indicação de tipos de variáveis no seu nome é ter duas maneiras diferentes de indicar isto. Pode-se juntar mais de uma letra, caso o tipo de uma variável seja composta.

Tipo	Prefixos	Exemplo
char	(ch)	chOpt
short	(sh)	shTipo
int	(i)	iNum
long	(l)	lValor
float	(f)	fImposto
double	(db)	dbGraus
string	(s)	Stela
string c/ "\0"	(sz)	szNome
structs (definição)	(ST_)	ST_Monit
structs	(st)	stFile
union (definição)	(U_)	U_Registro
union	(un)	unBuff
ponteiros	(p)(tipo)	pchOpt
Variáveis Globais	(G_)(tipo)	G_lValor

4. Procure usar somente abreviaturas conhecidas, como por exemplo: Vlr, Cont, Tot, Deb, Cred etc. Quando o significado não puder ser abreviado, utilize a forma integral. Exemplos: Balanceamento, GiroSemanal etc.
5. Se a variável possui mais de uma palavra em seu nome, procure colocar sempre a primeira letra maiúscula e as demais minúsculas em cada palavra. Exemplos: GiroSemanal, ContContasNegativas, SomaValorSaldo, TotDebitos.

## 2.4 Definição de Variáveis

Para se usar uma variável em C, ela deve ser definida indicando o seu tipo e o seu nome. Para se fazer isto, deve-se usar a seguinte sintaxe:

*tipo nome1 [, nome2]... ;*

Pode-se definir em uma mesma linha mais de uma variável, bastando para isto colocar os nomes das variáveis separados por vírgulas. Isto deve ser usado somente quando as variáveis são simples e não se precisa explicar o uso das mesmas. Como sugestão deve-se colocar sempre uma única variável por linha e, após a definição da mesma, colocar um comentário com a descrição mais completa.

Exemplos:

```
float fValorSalário;  
char cSexo;  
int i,k,j;
```

## 2.5 Atribuição de Valores

Às vezes, ao se definir uma variável, é desejável que a mesma já tenha um valor predefinido. A linguagem C permite que quando se defina uma variável se indique também o valor inicial da mesma. Deve-se colocar após a definição da variável o caractere "=" seguido de um valor compatível com o tipo da variável.

Exemplos:

```
float fValorSalário = 15000; /* Sonhar não paga imposto */  
char cSexo = 'M';  
int i,k,j;
```

## 2.6 Definição de Constantes

Às vezes também é desejável que, além de uma variável possuir um valor predefinido, este valor não seja modificado por nenhuma função de um programa.

Para que isto aconteça, deve-se colocar a palavra `const` antes da definição da variável, indicando ao compilador que, quando detectar uma mudança de valor da variável, seja emitida uma mensagem de erro.



Este instrumento é muito utilizado para documentar a passagem de parâmetros de uma função, indicando que um determinado parâmetro não será alterado pela função.

Exemplos:

```
const float fValorSalário = 5000; /* Se for constante nunca
receberei aumento??*/
const char cSexo = 'M'; /* Com certeza !!! */
```

## 2.7 Conversão de Tipos

De uma forma geral, pode-se realizar a conversão de um tipo para outro da linguagem C, utilizando o que se chama de *typecast*. Esta técnica é muito utilizada para se melhorar o entendimento de alguns trechos de programas. Outras vezes utiliza-se o *typecast* para compatibilizar um determinado tipo de um parâmetro na chamada de uma função para o tipo do parâmetro esperado por aquela função.

A sintaxe do “*typecasting*” é a seguinte:

(tipo) valor\_constante

(tipo) variável

No primeiro formato, o compilador irá realizar a transformação da constante indicada para o tipo indicado entre parênteses durante o processo de compilação. No segundo formato, o compilador irá gerar o código adequado para que a conversão ocorra em tempo de execução.

Exemplos:

```
int iASCII = (int) 'E'; /* Código ASCII do 'E' */
/* converter o resultado de uma divisão para inteiro */
short int si;
float f;
int i;
i = (int) (f/si);
```

## 2.8 Declaração typedef

Na linguagem C pode-se dar um outro nome a um tipo determinado. Isto é feito através da declaração *typedef*. Isto é muito usado para se manter a compatibilidade entre os sistemas operacionais e também para encurtar algumas definições longas, simplificando o programa.

Também o *typedef* é muito usado para criar tipos na própria língua do programador, criando-se tipos equivalentes.

## 18 ♦ Programando em C para Linux, Unix e Windows

Exemplos:

```
typedef unsigned char uchar;
typedef unsigned float ufloat;
typedef signed int sint;
/* Declarando as variáveis */
uchar cSexo;
ufloat fSalário;
sint i;
```

## 2.9 Operador sizeof

Existe um operador em C que indica o tamanho em *bytes* que uma determinada variável está utilizando na memória. Pode-se também colocar um determinado tipo como parâmetro que o resultado será o mesmo.

Este operador é muito utilizado para fazer alocações dinâmicas de memória ou movimentações diretas na memória.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int          a;
    short int    b;
    long int     c;
    unsigned int d;
    unsigned short int e;
    unsigned long int f;
    float        g;
    long float   h;
    double       i;
    long double  j;
    char         k;

    printf ("Tamanho do a : %d\n", sizeof(a));
    printf ("Tamanho do b : %d\n", sizeof(b));
    printf ("Tamanho do c : %d\n", sizeof(c));
    printf ("Tamanho do d : %d\n", sizeof(d));
    printf ("Tamanho do e : %d\n", sizeof(e));
    printf ("Tamanho do f : %d\n", sizeof(f));
    printf ("Tamanho do g : %d\n", sizeof(g));
    printf ("Tamanho do h : %d\n", sizeof(h));
    printf ("Tamanho do i : %d\n", sizeof(i));
    printf ("Tamanho do j : %d\n", sizeof(j));
    printf ("Tamanho do k : %d\n", sizeof(k));
}
```

Programa 2.1

**Resultado do programa 2.1:**

```
Tamanho do a : 4
Tamanho do b : 2
Tamanho do c : 4
Tamanho do d : 4
Tamanho do e : 2
Tamanho do f : 4
Tamanho do g : 4
Tamanho do h : 4
Tamanho do i : 8
Tamanho do j : 8
Tamanho do k : 1
```

**Observações:**

- 3 Nas especificações atuais da linguagem C, o `long` e o `int` têm o mesmo tamanho e o modificador `long` não altera os tamanhos de `float` e `double`.
- 3 Uma variável não precisa ter o seu conteúdo especificado para se obter o seu tamanho.
- 3 O operador `sizeof` pode ser utilizado direto com o tipo da variável. Por exemplo, uma linha de código com `printf("Tamanho de int = ", sizeof(int));` resultaria em `Tamanho de int = 4`.



*Cada saída é a entrada para algum outro lugar.*  
Tom Stoppard, autor de teatro tcheco

## 3.1 Função printf

Sintaxe:

```
printf("formato", argumentos);
```

Para realizar a impressão de textos no terminal, deve-se utilizar a função `printf`. Ela possui um número variado de parâmetros, tantos quantos forem necessários.

O primeiro parâmetro da função `printf` deve ser uma *string* indicando o texto a ser mostrado. Nesta *string* devem ser colocados formatadores de tipo para cada variável que será impressa. No texto também podem ser colocados alguns caracteres especiais, indicados através da barra invertida, a serem impressos na saída.

Se a função `printf` não possuir nenhum parâmetro, não será necessário colocar os formatadores de tipo em seu parâmetro de texto. Pode-se também colocar no texto caracteres indicados através da barra invertida.

Exemplos:

- o Para se imprimir um texto somente:

```
printf ("Sistema de Controle de Estoque");
```

- o Para se imprimir um valor de uma variável `b` do tipo inteiro:  

```
printf ("%d", b);
```
- o Misturando texto e valor de variáveis:  

```
printf ("Acumulado:%d - Contas %d", iTotAcum, iTotConta);
```
- o Com caracteres indicados através da barra invertida:  

```
printf ("%d \t-\t %d\n", b, c);
```

## 3.2 Formatadores de Tipos

Para cada variável colocada no comando `printf` deve-se indicar qual o formato desejado de saída. Isto é feito colocando-se o caractere `'%'` seguido de uma letra dentro do texto informado como primeiro parâmetro da função `printf`.

A função `printf` não faz nenhuma verificação entre o tipo real da variável e o caractere formatador indicado. Também não é feita a verificação do número correto de formatadores, um para cada variável. Quando isto acontecer, só será percebido quando da execução do programa, gerando resultados imprevisíveis.

Os formatadores que podem ser utilizados devem ser os seguintes:

Formato	Tipo da variável	conversão realizada
<code>%c</code>	Caracteres	<code>char</code> , <code>short int</code> , <code>int</code> , <code>long int</code>
<code>%d</code>	Inteiros	<code>int</code> , <code>short int</code> , <code>long int</code>
<code>%e</code>	Ponto flutuante, notação científica	<code>float</code> , <code>double</code>
<code>%f</code>	Ponto flutuante, notação decimal	<code>float</code> , <code>double</code>
<code>%lf</code>	Ponto flutuante, notação decimal	<code>double</code>
<code>%g</code>	O mais curto de <code>%e</code> ou <code>%f</code>	<code>float</code> , <code>double</code>
<code>%o</code>	Saída em octal	<code>int</code> , <code>short int</code> , <code>long int</code> , <code>char</code>
<code>%s</code>	String	<code>char *</code> , <code>char []</code>
<code>%u</code>	Inteiro sem sinal	<code>unsigned int</code> , <code>unsigned short int</code> , <code>unsigned long int</code>
<code>%x</code>	Saída em hexadecimal (0 a f)	<code>int</code> , <code>short int</code> , <code>long int</code> , <code>char</code>
<code>%X</code>	Saída em hexadecimal (0 a F)	<code>int</code> , <code>short int</code> , <code>long int</code> , <code>char</code>
<code>%ld</code>	Saída em decimal longo	Usado quando <code>long int</code> e <code>int</code> possuem tamanhos diferentes

## 3.3 Indicando o Tamanho

Quando é feita a saída do valor de uma variável, além de se especificar o tipo (formato) que deve ser mostrado, pode-se indicar o tamanho da saída. A indi-

## 22 ♦ Programando em C para Linux, Unix e Windows

cação do tamanho depende do tipo da variável. Para os números inteiros (`int`, `short int`, `long int`, `unsigned int`, `unsigned short int` e `unsigned long int`) a especificação do tamanho tem a seguinte sintaxe:

```
% [tam].[qtd_dig]d
```

Onde:

- o `tam` – Indica o tamanho mínimo que deve ser colocado na saída caso o número possua quantidade menor de dígitos. Se o número possuir quantidade de dígitos maior que o valor, o número não será truncado.
- o `qtd_dig` – Quantidade de dígitos que deve ser mostrada. Caso o número possua quantidade menor que o indicado, serão colocados zeros à esquerda até se completar o tamanho indicado.

Para os números reais (`float` e `double`), tem-se o seguinte formato:

```
% [tam].[casa_dec]f
```

Onde:

- o `tam` – É o mesmo que o descrito antes para os números inteiros. Vale completar que, neste tamanho, estão consideradas as casas decimais inclusive.
- o `casa_dec` – Número de casas decimais que devem ser mostradas. Caso o número possua número menor de decimais, o número será completado com zeros até o tamanho indicado. Se o número possuir um número de casas decimais maior que o indicado, a saída será truncada para o tamanho indicado.

Para as variáveis do tipo *string* pode-se indicar o tamanho mínimo e máximo a ser mostrado através da seguinte sintaxe:

```
%[tam].[tam_max]s
```

Neste caso, se a string possuir tamanho menor que o indicado a saída será completada com brancos à esquerda.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    printf ("%8.6d|\n", 820);
    printf ("%10d|\n", 820);
}
```

Espaço reservado de 8 caracteres (mínimo),  
preenchendo com zeros à esquerda até o  
máximo de 6 caracteres.

Espaço reservado de 10 caracteres.

```

printf ("%08d\n", 820);
printf ("%08d\n", 820);
printf ("%2.2f\n", 1223.4432);
printf ("%10.2f\n", 1223.4432);
printf ("%20f\n", 1223.4432);
printf ("%2f\n", 1223.4432);
printf ("%10s\n", "abcdefghijklmnopqrstuvwxyz");
printf ("%10.10s\n", "abcdefghijklmnopqrstuvwxyz");
printf ("%10s\n", "abcde");
}

```

Preenche com zeros à esquerda até o máximo de 8 caracteres.  
 Espaço reservado de 2 caracteres (mínimo), com 2 casas decimais (o número é arredondado).  
 Espaço reservado de 10 caracteres (mínimo), com 2 casas decimais (o número é arredondado).  
 Espaço reservado de 20 caracteres (mínimo), a quantidade de casas decimais é especificada pelo número a ser impresso.  
 A quantidade de caracteres utilizados é especificada pelo número a ser impresso, com 2 casas decimais (o número é arredondado).  
 Mínimo de 10 caracteres.  
 Mínimo e máximo de 10 caracteres. Ocorre um truncamento do campo a ser impresso se este for maior que 10 caracteres.  
 Mínimo de 10 caracteres. Preenche com espaços em branco até o limite de 10 caracteres.

### Programa 3.1

#### Resultado do programa 3.1:

```

| 000820|
|      820|
|00000820|
|00000820|
|1223.44|
|   1223.44|
|      1223.443200|
|1223.44|
|abcdefghijklmnopqrstuvwxyz|
|abcdefghij|
|      abcde|

```

## 3.4 Função putchar

Sintaxe:

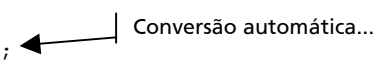
```
putchar(argumento);
```

Esta função é uma maneira simplificada de mostrar um único caractere na tela. O argumento passado será convertido para caractere e mostrado na tela.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    char      cLetra_a      = 'a';
    short int iCod_ASCII    = 65; /* Código ASCII do 'A' */

    putchar (cLetra_a);
    putchar (iCod_ASCII);
}
```



Programa 3.2

Resultado do programa 3.2:

aA

**Observação:** Na linguagem C, a conversão ocorre diretamente, ou seja, o valor 65 foi convertido no momento da impressão para o caractere ASCII correspondente ao valor 65. O inverso também pode ocorrer; se um valor 'A' foi atribuído a uma variável inteira, a conversão será feita automaticamente para 65.

## 3.5 Função scanf

Sintaxe:

```
scanf("formato", endereços_argumentos);
```

Para realizar a entrada de valores para as variáveis deve ser utilizada a função `scanf`. A sintaxe desta função é muito parecida com o `printf`. Primeiramente, são informados quais os formatos que serão fornecidos no terminal, depois os endereços das variáveis que irão receber estes valores.

O formato segue a mesma sintaxe do comando `printf`, sendo obrigatório colocar um formato, especificado através do caractere '%', e a letra indicando o formato.

Para especificar o endereço de uma variável, necessário para que a função `scanf` localize a variável na memória, deve-se colocar o caractere '&' antes do

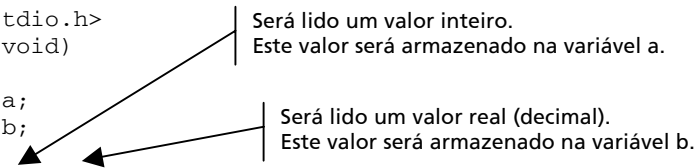


nome da variável, indicando assim que se está passando o endereço da variável e não o seu valor. Veja o exemplo:

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

    scanf ("%d %f", &a, &b);

    printf ("Inteiro %d\n", a);
    printf ("Real    %f\n", b);
}
```

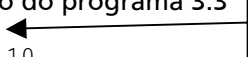


Será lido um valor inteiro.  
Este valor será armazenado na variável a.

Será lido um valor real (decimal).  
Este valor será armazenado na variável b.

**Programa 3.3**

**Resultado do programa 3.3**



Valores digitados separados por espaço.

```
10 20.0
Inteiro 10
Real    20.000000
```

## 3.6 Função getchar

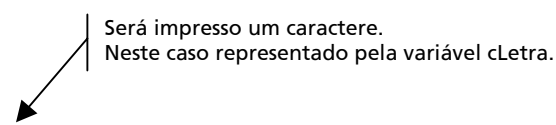
Sintaxe:

```
var = getchar();
```

Quando for necessário realizar a entrada de um único caractere, pode ser utilizada esta função. Ela lê um caractere do terminal e devolve o código ASCII do mesmo. Sendo assim, é possível assinalar o valor da função para uma variável do tipo caractere (`char`). Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    char  cLetra;

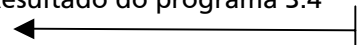
    cLetra = getchar();
    printf ("Letra digitada %c\n", cLetra);
}
```



Será impresso um caractere.  
Neste caso representado pela variável cLetra.

**Programa 3.4**

**Resultado do programa 3.4**



Valor digitado.

```
f
Letra digitada f
```



*Se você pensar sobre isso tempo suficiente,  
perceberá que isso é óbvio.*  
Saul Gorn, professor americano

## 4.1 Operadores Aritméticos

Operador	Operação
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)

Todas estas operações exigem dois operandos (números).

## 4.2 Operadores Unários

Operador	Operação
++	Incremento
--	Decremento

A posição relativa destes operadores em relação à variável influencia o seu funcionamento. Se os operadores forem colocados antes da variável em uma expressão, inicialmente será efetuado o incremento e depois será utilizado este novo valor na expressão.

Se os operadores forem colocados após a variável, primeiro será utilizado o valor atual da variável na expressão, e após, será realizado o incremento ou decremento. Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    vlr1 = 10, vlr2 = 5, vlr3 = 8;
    int    resultado;

    resultado = vlr1++ + 9;
    printf ("Resultado 1 %d\n", resultado);

    resultado = --vlr2 + 10;
    printf ("Resultado 2 %d\n", resultado);

    resultado = ++vlr3 * ++vlr3;
    printf ("Resultado 3 %d\n", resultado);

    resultado = vlr1++ * vlr1++;
    printf ("Resultado 4 %d\n", resultado);
}
```

Retorna 10 para vlr1  
resultado = 10 + 9  
vlr1 = vlr1 + 1

vlr2 = vlr2 - 1  
Retorna 4 para vlr2  
resultado = 4 + 10

vlr3 = vlr3 + 1  
Retorna 9 para vlr3  
vlr3 = vlr3 + 1  
Retorna 10 para vlr3  
resultado = 9 \* 10

Retorna 11 para vlr1  
vlr1 = vlr1 + 1  
Retorna 12 para vlr1  
vlr1 = vlr1 + 1  
resultado = 11 \* 12

**Programa 4.1**

Resultado do programa 4.1:

```
Resultado 1 19
Resultado 2 14
Resultado 3 90
Resultado 4 132
```

## 4.3 Operadores de Atribuição

Operador	Operação
+=	Adição
-=	Subtração
*=	Multiplicação
/=	Divisão
%=	Módulo

Como é comum a atribuição onde uma variável é somada ou diminuída de um valor e o resultado é atribuído à mesma variável, a linguagem C disponibiliza uma maneira curta de realizar este tipo de operação. Veja o exemplo:

## 28 ♦ Programando em C para Linux, Unix e Windows

<code>d += 5;</code>	equivale a <code>d = d + 5;</code>
<code>b -= (c*8);</code>	equivale a <code>b = b - (c*8);</code>
<code>x *= 2;</code>	equivale a <code>x = x * 2;</code>

### 4.4 Operadores Relacionais

Operador	Operação
<	Menor que
>	Maior que
<=	Menor ou igual a
>=	Maior ou igual a
==	Igual
!=	Diferente

Os operadores relacionais servem para realizar a comparação de dois valores distintos. A implementação da linguagem C retorna como resultado destas operações os seguintes valores:

- o 0 – Operação obteve o resultado falso;
- o 1 – Operação obteve o resultado verdadeiro.

Exemplos:

<code>4 &lt; 5</code>	resulta 1
<code>50 = 43</code>	resulta 0
<code>4 &gt;= 0</code>	resulta 1
<code>4 != 9</code>	resulta 1

### 4.5 Prioridade de Avaliação

Operadores	Prioridade
<code>++ --</code>	Prioridade mais alta
<code>* / %</code>	Prioridade média
<code>+ -</code>	Prioridade baixa

Quando é utilizada uma série de operadores em uma expressão, deve-se sempre ter em mente a prioridade em que o compilador irá realizar essas operações.

Os operadores unários têm a maior prioridade e serão executados por primeiro. Depois serão executadas todas as operações multiplicativas (\*, / e %). Por último serão executadas as operações aditivas (+ e -). Veja o exemplo:

```

#include <stdio.h>
void main (void)
{
    int    vlr1    = 10;
    int    resultado;

    resultado = 10 - 5 * 8;
    printf ("Resultado 1 %d\n", resultado);

    resultado = 7 * --vlr1 - 4;
    printf ("Resultado 2 %d\n", resultado);
}

```

Ocorre a multiplicação ( $5 \cdot 8 = 40$ )  
Subtrai 10 ( $10 - 40 = -30$ )

Retornar 9 para vlr1.  
Realiza a multiplicação ( $7 \cdot 9 = 63$ ).  
Subtrai 4 ( $63 - 4 = 59$ ).

Programa 4.2

Resultado do Programa 4.2:

Resultado 1 -30  
Resultado 2 59

## 4.6 Operadores Lógicos

Operadores	Prioridade
&&	Operação lógica E
	Operação lógica OU
!	Operação lógica negação

Os operadores lógicos consideram dois operandos como valores lógicos (verdadeiro e falso) e realizam a operação binária correspondente. A linguagem C considera como valor verdadeiro qualquer valor diferente de zero. O valor falso será indicado pelo valor zero.

Estas operações, quando aplicadas, irão retornar 1 ou zero, conforme o resultado seja verdadeiro ou falso. Veja o exemplo:

```

#include <stdio.h>
void main (void)
{
    int    vlr1    = 10;
    int    resultado;

    resultado = (vlr1 > 8) && (vlr1 != 6);
}

```

10 > 8? A expressão é verdadeira  
então é retornado 1.

10 é diferente de 6? A expressão é verdadeira  
então é retornado 1.

A expressão lógica 1 e 1 retorna 1.  
Lembrando que 0 representa o valor lógico falso e diferente de zero (1,2,3 etc.)  
representa o valor lógico verdadeiro.

## 30 ♦ Programando em C para Linux, Unix e Windows

```
printf ("Resultado: %d\n", resultado);  
}
```

### Programa 4.3

Resultado do Programa 4.3:

Resultado: 1

## 4.7 Assinalamento de Variáveis

Para a linguagem C o assinalamento é considerado um operador. Isto significa que podem ser colocados assinalamentos dentro de qualquer comando que exija um valor qualquer.

Sempre que um assinalamento é efetuado, a linguagem C retorna como resultado do assinalamento o valor que foi colocado em variáveis. Por isso pode-se utilizar este valor retornado pelo operador "=" e colocar como se fosse um novo assinalamento, concatenando os mesmos. Veja o exemplo:

```
#include <stdio.h>  
void main (void)  
{  
    int    i,j,k,w;  
  
    i = j = k = 20;  
  
    printf ("Variavel i: %d\n", i );  
    printf ("Variavel j: %d\n", j );  
    printf ("Variavel k: %d\n", k );  
  
    printf ("Resultado da expressao: %d\n", w = 90);  
    printf ("Variavel w: %d\n", w );  
  
}
```

Atribuição múltipla de 20 para as variáveis i, j e k.  
Equivalente a:  
i = 20;  
j = 20;  
k = 20;

Atribui 90 à variável w e depois o conteúdo da variável é impresso.

### Programa 4.4

Resultado do Programa 4.4:

Variavel i: 20  
Variavel j: 20  
Variavel k: 20  
Resultado da expressao: 90  
Variavel w: 90

## 4.8 Parênteses e Colchetes como Operadores

Em C, parênteses são operadores que aumentam a precedência das operações dentro deles. Colchetes realizam a indexação de matrizes (será visto mais tarde). Dada uma matriz, a expressão dentro de colchetes provê um índice dentro dessa matriz.

A precedência dos operadores em C.

Maior	( ) [ ] ->
	! ~ ++ -- -(tipo) * & sizeof
	* / %
	+ -
	<< >>
	<=> >=>
	== !=
	&
	^
	&&
	?
	= += -= *= /=
Menor	,

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int w,k;
    int resultado;

    printf ("Resultado da expressao 1: %d\n", 9*(w = 90));
    printf ("Variavel w: %d\n", w );
    printf ("Resultado da expressao 2: %d\n", resultado = (20*(k=50)));

    printf ("Variavel k: %d\n", k );
    printf ("Variavel resultado: %d\n", resultado );
}
```

Atribui 90 à variável w.  
Realiza a multiplicação (9\*90=810)  
Imprime o resultado (810)

Atribui 50 à variável k.  
Realiza a multiplicação (20\*50=1000).  
Atribui 1000 à variável resultado.  
Imprime o conteúdo da variável resultado (1000).

Programa 4.5

## 32 ♦ Programando em C para Linux, Unix e Windows

### Resultado do Programa 4.5:

```
Resultado da expressao 1: 810
Variavel w: 90
Resultado da expressao 2: 1000
Variavel k: 50
Variavel resultado: 1000
```

**Observação:** Embora a linguagem C permita o assinalamento de variáveis no meio de expressões, seu uso não é recomendado, pois dificulta o entendimento do programa.

## 4.9 Operador & e \* para Ponteiros

O primeiro operador de ponteiro é `&`. Ele é um operador unário que devolve o endereço na memória de seu operando. Por exemplo:

```
m = &count;
```

põe o endereço na memória da variável `count` em `m`. Esse endereço é a posição interna da variável no computador e não tem nenhuma relação com o valor de `count`. Você pode imaginar `&` como significando o “endereço de”.

O segundo operador é `*`, que é o complemento de `&`. O `*` é um operador unário que devolve o valor da variável localizada no endereço que o segue. Por exemplo, se `m` contém o endereço da variável `count`:

```
q = *m;
```

coloca o valor de `count` em `q`. Pense no `*` como significando “no endereço de”.

### Observações:

- 3 Cuidado para não confundir `*` como multiplicação na utilização de ponteiros e vice-versa;
- 3 Cuidado para não confundir `&` como o operador relacional `&&` e vice-versa. Será vista com detalhes a utilização de ponteiros no capítulo 11.





# Comandos de Seleção

*Se não fosse para cometer erros, eu não tomaria decisões.*  
Robert Wood Johnson, empresário americano

## 5.1 Comando if

Sintaxe:

```
if (condição)
{
    bloco de comandos
}
```

O comando `if` funciona da seguinte maneira: primeiramente, a expressão da condição é avaliada. Caso o resultado seja verdadeiro (diferente de zero, o que significa verdadeiro em C), o bloco de comandos entre `{ }` é executado. Caso a expressão resulte em falso (igual a zero), o bloco de comandos não será executado.

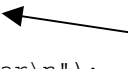
Se o bloco de comandos na realidade representar um único comando, não será necessário colocar `{ }`, bastando o comando após o `if`.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    vlr1;
    int    resultado;
```

## 34 ♦ Programando em C para Linux, Unix e Windows

```
printf ("Entre com um numero :");
scanf ("%d", &vlr1);
if ((vlr1 % 2) == 0)
{
    printf ("Numero Par\n");
}
```

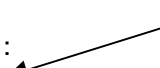


Se o resto da divisão for zero, o número é par.  
Exemplo: 4 dividido por 2 = 2 com resto 0.

### Programa 5.1

#### Resultado do Programa 5.1:

Entre com um numero :20  
Numero Par



Valor digitado.

## 5.2 Comando if...else...

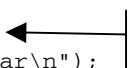
### Sintaxe:

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

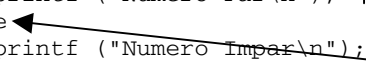
Podem-se também selecionar dois trechos de um programa baseados em uma condição. Para isto utiliza-se a construção `if...else...`. Este comando inicialmente testa a condição. Caso seja verdadeiro, o bloco de comando será executado. Caso a condição resulte em valor falso, será executado o bloco de comandos 2.

### Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    vlr1;
    int    resultado;
    printf ("Entre com um numero : ");
    scanf ("%d", &vlr1);
    if ((vlr1 % 2) == 0)
        printf ("Numero Par\n");
    else
        printf ("Numero Impar\n");
}
```



Se o resto da divisão for zero, o número é par.  
Exemplo: 4 dividido por 2 = 2 com resto 0.



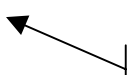
Se o resto da divisão for 1, o número é ímpar.  
Exemplo: 7 dividido por 2 = 3 com resto 1.

### Programa 5.2

**Resultado do Programa 5.2:****1ª Execução**

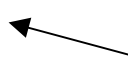
Entre com um numero : 21  
Numero Impar

Valor digitado.


**2ª Execução**

Entre com um numero : 24  
Numero Par

Valor digitado.



## 5.3 Operador ? :

**Sintaxe:**

(cond? bloco\_verd : bloco\_falso)

O operador ? : é uma maneira simplificada de escrever um `if...else`. Apesar de possuir a mesma funcionalidade, não se deve usar este operador quando os comandos envolvidos são complexos.

Primeiramente a condição é avaliada. Dependendo do resultado o bloco respectivo será executado.


**Veja o exemplo:**

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

```
void main (void)
```


```
{
    int    vlr1;
    printf ("Entre com um numero : ");
    scanf ("%d", &vlr1);
    printf ((vlr1%2 == 0? "Numero Par\n" : "Numero Impar\n"));
}
```

Esta linha é idêntica ao programa 5.2


**Programa 5.3****Resultado do Programa 5.3****1ª Execução**


Entre com um numero : 37  
Numero Impar

Valor digitado.


**2ª Execução**

Entre com um numero : 10  
Numero Par

Valor digitado.



## 5.4 Comando switch...case

Sintaxe:

```
switch (expressão) {
    case const1:{ bloco_1...;break;}
    case const2:{ bloco_2...;break;}
    .....
    default : { bloco_n... }
}
```

Caso seja necessário realizar operações baseadas em um valor de uma expressão ou variável, em vez de se construir para isto uma cadeia de `if...else...if...else..if...else`, pode-se utilizar o comando de seleção múltipla `switch...case`.

Inicialmente o valor da expressão é avaliado. Depois é feita uma comparação com cada valor colocado na seção `case`. Caso o valor seja coincidente, o bloco ligado ao `case` será executado. Convém ressaltar que a execução continuará na ordem em que os comandos aparecem, indiferentemente se eles fazem parte de outro `case`. Para interromper a execução deve-se utilizar a cláusula `break`, indicando que deve ser interrompida a execução e passar a executar os comandos após o `switch`.

Existe a possibilidade de colocar uma condição para que, se nenhum `case` foi selecionado, um bloco seja executado. A palavra `default` indicará este bloco padrão a ser executado.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    vlr1;
    printf ("Entre com um numero : ");
    scanf ("%d", &vlr1);
    switch (vlr1) ← Aqui vai a variável a ser avaliada.
    {
        case 1 : {
            printf ("Um\n");
            break;}
        case 2 : {
            printf ("Dois\n");
            break;}
        case 3 : {
            printf ("Tres\n");
            break;}
    }
```

← Aqui vai o valor (constante) que será utilizado na comparação.

```

    case 4 : {
        printf ("Quatro\n");
        break;}
    case 5 : {
        printf ("Cinco\n");
        break;}
    case 6 : {
        printf ("Seis\n");
        break;}
    case 7 : {
        printf ("Sete\n");
        break;}
    case 8 : {
        printf ("Oito\n");
        break;}
    case 9 : {
        printf ("Nove\n");
        break;}
    default :
        printf ("Valor nao associado\n");
        break;
}

```

Se nenhuma opção anterior corresponder à variável informada.

Não precisa deste comando aqui, afinal não existem mais condições para serem avaliadas dentro da estrutura switch.

Programa 5.4

#### Resultado do Programa 5.4

Entre com um numero : 8  
Oito

Valor digitado.

Caso não seja utilizado o comando `break`, todos os demais comandos/instruções serão executados, até encontrar o próximo comando `break`, após a primeira condição verdadeira.

Veja o exemplo:

```

#include <stdio.h>
void main (void)
{
    int    vlr1;
    printf ("Entre com um numero : ");
    scanf ("%d", &vlr1);
    switch (vlr1)
    {
        case 1 : {
            printf ("Um\n");
            break;}
        case 2 : {
            printf ("Dois\n");
            break;}
    }
}

```

## 38 ♦ Programando em C para Linux, Unix e Windows

```
case 3 : {  
    printf ("Tres\n");  
    break;}  
case 4 : {  
    printf ("Quatro\n");  
    break;}  
case 5 : {  
    printf ("Cinco\n");  
    break;}  
case 6 : {  
    printf ("Seis\n");  
    break;}  
case 7 : {  
    printf ("Sete\n");  
    }  
case 8 : {  
    printf ("Oito\n");  
    }  
case 9 : {  
    printf ("Nove\n");  
    break;}  
default :  
    printf ("Valor nao associado\n");  
    break;  
}  
}
```

← Não colocado o comando break.  
Todos os comandos serão avaliados até o próximo comando break.

← Não colocado o comando break.  
Todos os comandos serão avaliados até o próximo comando break.

### Programa 5.5

#### Resultado do Programa 5.5:

##### 1ª Execução:

Entre com um numero : 6  
Seis

Valor digitado.

##### 2ª Execução:

Entre com um numero : 7  
Sete  
Oito  
Nove

Valor digitado.

##### 3ª Execução:

Entre com um numero : 8  
Oito  
Nove

Valor digitado.



# Comandos de Repetição

*Como eu disse antes, eu nunca me repito.*  
(Anônimo)

## 6.1 Comando for

Sintaxe:

```
for(inicialização;  
    condição de fim;  
    incremento)  
{  
    bloco de comandos  
}
```

Quando se quer executar um bloco de comando um número determinado de vezes, deve-se utilizar o comando `for`. Na sua declaração, o comando `for` determina três áreas distintas. A primeira área são os comandos que serão executados inicialmente. Deve-se colocar nesta área comandos de inicialização de variáveis.

A segunda área é a de teste. A cada interação, as condições colocadas aí são testadas e, caso sejam verdadeiras, segue-se com a execução do bloco de comandos.

A última área possui comandos que serão executados ao final da interação. Geralmente são colocados nesta área os comandos de incrementos de variáveis.

## 40 ♦ Programando em C para Linux, Unix e Windows

Pode-se omitir os comandos da área de inicialização e de incremento, bastando-se colocar o ponto e vírgula. Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
```

```
    int    vlr1;
    int    i;
```

```
    printf ("Contar ate : ");
    scanf ("%d", &vlr1);
```

```
    for (i=1; i <= vlr1; i++)
        printf ("%d\n", i);
}
```

Programa 6.1

Atribuição inicial. Executado somente 1 vez, sempre no início.

A condição sempre será avaliada antes da execução das instruções agrupadas embaixo do comando `for`.

O incremento (ou decremento) sempre ocorrerá após a execução das instruções agrupadas embaixo do comando `for`.

### Resultado do Programa 6.1

#### 1ª Execução

Contar ate : 0

Valor digitado. Repare que não é contado nada, pois a condição analisa a variável `i` (assinalada com o valor 1) perguntando se `i` é menor ou igual à variável `vlr1` (neste caso informado 0). Ou seja: `1 <= 0` que resulta no valor lógico falso.

#### 2ª Execução

Contar ate : 5

Valor digitado.

1  
2  
3  
4  
5

## 6.2 Comando while

Sintaxe:

```
while (condição)
{
    bloco de comandos
}
```

O comando `while` deve ser usado quando não se pode determinar com certeza quantas vezes um bloco de comandos será executado. Inicialmente a condição é testada. Caso seja falso, o programa não executará o bloco de comando indicado e continuará no comando após o comando `while`.

Caso a condição seja verdadeira, o bloco de comando é executado. Ao final da execução do bloco, volta-se a testar a condição. O bloco de comandos, portan-



to, será executado até que se alcance uma condição falsa. De uma outra maneira, o bloco de comando será executado enquanto a condição for verdadeira. Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    vlr1;
    int    i;

    printf ("Contar ate : ");
    scanf ("%d", &vlr1);
    i =1;
    while (i <= vlr1)
    {
        printf ("%d\n", i);
        i++;
    }
}
```

A condição é avaliada antes da execução das operações agrupadas embaixo do comando while.

Programa 6.2

#### Resultado do Programa 6.2

##### 1ª Execução

Contar ate : 0

##### 2ª Execução

Contar ate : 3

1  
2  
3

Valor digitado. Repare que não é contado nada, pois a condição analisa a variável *i* (assinalada com o valor 1) perguntando se *i* é menor ou igual à variável *vlr1* (neste caso informado 0). Ou seja:  $1 \leq 0$  que resulta no valor lógico falso.

Valor digitado.

## 6.3 Comando do...while

Sintaxe:

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

O comando `do...while` diferencia-se do comando `while` em somente um detalhe. O bloco de comando indicado é sempre executado pelo menos uma vez. Após a execução do bloco, a condição é testada. Caso seja verdadeira, o bloco continua a ser executado.

A execução passará para o próximo comando somente quando a condição re-tornar falso. Veja o exemplo:

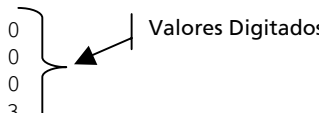
## 42 ♦ Programando em C para Linux, Unix e Windows

```
#include <stdio.h>
void main (void)
{
    int    vlr1; | Executa o conjunto de instruções...
    do
    {
        printf ("Entre com um numero diferente de zeros: ");
        scanf ("%d", &vlr1);
    }
    while (vlr1 == 0); | ...enquanto a condição for verdadeira.
    printf ("Valor digitado: %d\n", vlr1);
}
```

### Programa 6.3

#### Resultado do Programa 6.3

```
Entre com um numero diferente de zeros: 0
Entre com um numero diferente de zeros: 0
Entre com um numero diferente de zeros: 0
Entre com um numero diferente de zeros: 3
Valor digitado: 3
```



## 6.4 Comando break

### Sintaxe:

```
while (condição)
{
    bloco de comandos;
    break;
}
```

Às vezes é necessário quebrar a execução de um comando repetitivo devido a uma condição determinada. Pode-se programar esta condição no próprio local da condição dos comandos repetitivos ou colocar um teste dentro do bloco de comandos.

Caso a condição seja alcançada, pode-se sair do comando repetitivo de uma maneira não usual, terminando a execução deste comando. Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    vlr1;
    int    i;
    char   resp;

    printf ("Contar ate : ");
    scanf ("%d", &vlr1);
```

```

i =1;
while (i <= vlr1)
{
    printf ("\n%d", i++);
    printf ("\nTermina (S/N)?");
    scanf ("%c", &resp);
    if (resp == 's' || resp == 'S')
        break;
}
printf ("\nContagem Encerrada");
}

```

**Programa 6.4**

Se for respondido Sim...  
...interrompe a execução...  
...desviando o programa para a próxima instrução depois do } (fecha chaves) do while.

**Resultado do Programa 6.4**

```

Contar ate : 5
1
Termina (S/N)?n
2
Termina (S/N)?n
3
Termina (S/N)?n
4
Termina (S/N)?s
Contagem Encerrada

```

Valor digitado.

**6.5 Comando continue****Sintaxe:**

```

while (condição)
{
    bloco de comandos;
    continue;
}

```

Às vezes é necessário que se volte para testar a condição indicada quando ocorre uma situação. Neste caso será utilizado o comando `continue`. Toda vez que este comando for executado, será feito um desvio de execução para a condição do comando repetitivo. Veja o exemplo:

```

#include <stdio.h>
void main (void)
{
    int i;

    for (i=1; i < 30; i++)
    {
        if (i > 10 && i < 20)

```

De 1 até 29.  
Se a variável i estiver entre 11 e 19...

## 44 ♦ Programando em C para Linux, Unix e Windows

```
        continue; ← ...o comando continue desvia o controle  
        printf ("%d\n", i); do programa para o comando for.  
    }  
}
```

### Programa 6.5

#### Resultado do Programa 6.5

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29
```

## 6.6 Comando goto

### Sintaxe:

```
...  
goto saida;  
...  
saida: comandos  
...
```

O comando `goto` realiza o desvio da execução para o comando que possuir o *label* indicado. Apesar de existir este comando, todas as boas técnicas de programação dizem que seu uso deve ser evitado. Deve ser usado somente em processamento de exceção, desviando para uma área específica caso ocorra algum erro grave na execução de algum comando. Veja o exemplo:

```
#include <stdio.h>  
  
void main (void)  
{  
    int    vlr_a;  
    int    vlr_b;
```

```

while (1)
{
    printf ("Valores:");
    scanf ("%d %d", &vlr_a, &vlr_b);
    if (vlr_a == 0)
        goto fim;
    if (vlr_b == 0)
        goto erro;
    printf ("Divisao : %d\n", vlr_a / vlr_b);
}
erro:
    printf ("Divisao por zero\n");
fim:
    printf ("Fim da execucao do programa\n");
}

```

Caso seja informado 0 para `vlr_b`, o programa é desviado para o *label* erro através do comando `goto..`

Executado somente se for informado o valor 0 para a variável `vlr_b`.

Esta linha é executada sempre, pois todos os comandos após um *label* `goto` serão interpretados. Mesmo que faça parte de outro *label* `goto`.

### Programa 6.6

#### Resultado do Programa 6.6

##### 1ª Execução

```

Valores:23 234
Divisao : 0
Valores:24 2
Divisao : 12
Valores:0 23
Fim da execucao do programa

```

Valores Digitados.

Valores Digitados.

Valores Digitados.

##### 2ª Execução

```

Valores:24 0
Divisao por zero
Fim da execucao do programa

```

Valores Digitados.

## 6.7 Comando exit

### Sintaxe:

```
exit (valor_de_retorno);
```

A função `exit` deve ser usada quando se quer terminar a execução do programa, retornando para o sistema operacional um indicativo. Tanto em Unix/Linux como em Windows/DOS existem maneiras de se obter o número retornado.

O retorno 0 (zero) indica para o sistema operacional que o programa terminou corretamente, um retorno diferente de 0 (zero) indica um erro. Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
void main (void)

```

## 46 ♦ Programando em C para Linux, Unix e Windows

```
{
    int    vlr_a;
    int    vlr_b;

    while (1)
    {
        printf ("Valores:");
        scanf ("%d %d", &vlr_a, &vlr_b);
        if (vlr_a == 0)
            exit (0);
        if (vlr_b == 0)
            exit (11);
        printf ("Divisao : %d\n", vlr_a / vlr_b);
    }
}
```

Programa 6.7

### Resultado do Programa 6.7

#### No Unix/Linux:

##### 1ª Execução

Valores:24 23

Divisao : 12

Valores:0 12

Comando no Unix/Linux que mostra o valor retornado pelo programa.

\$> echo \$?

0

Valor retornado.

##### 2ª Execução

Valores:24 0

Comando no Unix/Linux que mostra o valor retornado pelo programa.

\$> echo \$?

11

Valor retornado.

#### No Windows:

##### 1ª Execução

C:\> echo %errorlevel%

0

Valor retornado.

Comando no Windows/DOS que mostra o valor retornado pelo programa.

##### 2ª Execução

C:\> echo %errorlevel%

11

Valor retornado.

Comando no Windows/DOS que mostra o valor retornado pelo programa.



# Definições de Funções

*Prefiro ser esta metamorfose ambulante do que ter  
aquela velha opinião formada sobre tudo.*

Raul Seixas, cantor e compositor de rock brasileiro

## 7.1 Criação de Funções

A boa técnica de programação diz para, sempre que possível, evitar códigos extensos, separando o mesmo em funções, visando um fácil entendimento e uma manutenção facilitada. De acordo com a técnica, devem-se agrupar códigos correlatos em uma função.

Uma outra utilização de função é quando um trecho de código será utilizado muitas vezes no programa. Deve-se colocar este trecho em uma função e, sempre que for preciso, chamar a função.

A Linguagem C possibilita criar funções, sendo possível passar parâmetros para elas e retornar valores, tanto no nome da função, como em algum parâmetro passado.

## 7.2 Função e Protótipo (assinatura da função)

O uso de funções na linguagem C exige certas regras. Primeiramente, a função deve estar definida, isto é, deve-se indicar para o compilador qual o nome da função e quais são os parâmetros esperados.

## 48 ♦ Programando em C para Linux, Unix e Windows

Uma maneira simples de resolver isso é a colocação da função antes de seu uso, ou seja, coloca-se a função dentro do programa fonte antes das posições onde ela é chamada.

Quando se têm sistemas grandes, não é recomendável ter um único arquivo fonte, pois a manutenção seria impraticável. Neste caso é possível ter uma função definida em um programa fonte e seu uso em outro programa fonte. Para resolver este problema, a linguagem C criou uma definição chamada de *protótipo*.

No protótipo de uma função é definido somente o necessário para o compilador não acusar erros. A definição do protótipo geralmente é colocada dentro de arquivos *header* e incluída dentro dos programas fontes.

No protótipo somente são informados o nome da função, o seu tipo de retorno e o tipo de cada parâmetro esperado.

## 7.3 Definindo Funções

Sintaxe:

```
tp_ret nome (tipo_par1 nome_par1, tipo_par2 nome_par2)
{
}
```

Para se definir uma função deve-se indicar o tipo do retorno da função, seu nome e os parâmetros da mesma.

Uma função pode ou não retornar um valor. Se uma função não retorna nenhum valor, seu retorno deve ser definido como *void*.

Os parâmetros devem ser definidos, um por um, indicando o seu tipo e nome separado por vírgula.

Veja o exemplo:

```
#include <stdio.h>
int soma (int, int);
```

Declaração da função (protótipo da função).  
Esta declaração indica que a função soma irá receber 2 valores inteiros e vai retornar 1 valor inteiro.

```
void main (void)
{
    int    vlr_a;
    int    vlr_b;
    int    resultado;
```



```

printf ("Entre com os valores:");
scanf ("%d %d", &vlr_a, &vlr_b);
resultado = soma (vlr_a, vlr_b);
printf ("Soma : %d\n", resultado);
}
int soma (int a, int b)
{
    return a + b;
}

```

Chamada da função.

Função soma. Recebe 2 inteiros.

Retorna 1 valor inteiro.

#### Programa 7.1

#### Resultado do Programa 7.1

Entre com os valores:10 20  
Soma : 30

Valores digitados.

## 7.4 Comando return

#### Sintaxe:

```
return expressão;
```

Quando uma função deve retornar valores, utiliza-se o comando `return`. Quando este comando é executado, o valor indicado é retornado para a função e a mesma encerra a sua execução, independentemente do local onde o `return` se encontra.

#### Veja o exemplo:

```

#include <stdio.h>
int le_numero (void);
int soma (int, int);

void main (void)
{
    int    vlr_a;
    int    vlr_b;

    vlr_a = le_numero();
    vlr_b = le_numero();
    printf ("Soma : %d\n", soma (vlr_a, vlr_b));
}

int le_numero (void)
{
    char    ch;
    int     valor;
}

```

Declaração das funções (protótipos).

O resultado de uma função pode ser utilizado diretamente em outra função.

## 50 ♦ Programando em C para Linux, Unix e Windows

```
printf ("Entre com um numero :");
ch = getchar();
while (ch < '0' || ch > '9')
    ch = getchar ();
valor = 0;
while (ch >= '0' && ch <= '9')
{
    valor *= 10;
    valor += (int) ch - (int) '0';
    ch = getchar ();
}
while (ch != '\n')
    ch = getchar ();
return valor;
}

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

Uma forma complicadíssima de se ler valores numéricos.

- 1º É verificado se o caractere digitado encontra-se entre 0 e 9.
- 2º Multiplica-se por 10, para deslocar uma casa para a direita. Por exemplo, se o valor atual for 2, vira 20.
- 3º Convertem-se os valores digitados, que estão em caracteres, para numéricos e subtrai-se o valor numérico representando o 0 na tabela ASCII. Por exemplo: o valor 9 na tabela ASCII é representado pelo número 57 e o 0 por 48, então  $57 - 48 = 9$ .
- 4º Soma-se o valor encontrado. Por exemplo, se o primeiro caractere for 2, ele é multiplicado por 10 e torna-se 20, se o segundo caractere for 9, é somado à variável e obtém-se o valor 29.

Recebe 2 valores inteiros...

... soma e retorna o valor somado.

**Programa 7.2**

### Resultado do Programa 7.2

```
Entre com um numero :10
Entre com um numero :45
Soma : 55
```

Valor digitado.

Valor digitado.

## 7.5 Escopo de Variáveis

Entende-se como escopo de variáveis a área onde o valor e o nome dela tem significado. Pode-se ter dois tipos de variáveis na linguagem C. O primeiro tipo é a variável global. Uma variável é global quando a mesma é definida fora de qualquer função. Esta variável pode ser usada em qualquer função e o significado dela abrange todo o programa fonte.

As variáveis locais são definidas dentro de funções e o seu significado é somente válido dentro da função. Assim têm-se duas variáveis com o mesmo nome em funções diferentes.

## 7.6 Variáveis Globais

As variáveis globais são definidas fora de qualquer função e o seu nome é válido para todo o programa. Qualquer função que alterar o seu conteúdo estará alterando para todo o programa, pois estas variáveis ficam em uma área de dados disponível para todo o programa.

## 7.7 Variáveis Locais

Quando uma variável é definida dentro de uma função, está sendo definida uma variável local à função. Esta variável utiliza a pilha interna da função como memória; portanto, ao final da função, este espaço de memória é liberado e a variável não existe mais. A definição da variável só é válida dentro da função.

Os parâmetros de uma função também são considerados variáveis locais e também utilizam a pilha interna para a sua alocação.

Veja o exemplo:

```
#include <stdio.h>
int soma (int, int);
int diferenca (int, int);
void le_valores(void);

int    vlr_a;
int    vlr_b;

void main (void)
{
    int    resultado;

    le_valores();
    printf ("Soma : %d\n", soma (vlr_a, vlr_b));
    printf ("Diferenca : %d\n", diferenca (vlr_a, vlr_b));
}

void le_valores(void)
{
    printf ("Entre com os valores:");
    scanf ("%d %d", &vlr_a, &vlr_b);
}

int soma (int a, int b)
{
    int    resultado;

    resultado = a + b;
    return resultado;
}

int diferenca (int a, int b)
{
    int    resultado;
```

Declarando as variáveis como públicas, ou seja, elas estarão disponíveis para uso em todo o programa.

Variável local e portanto somente disponível para a função main.

Variáveis públicas declaradas anteriormente.

Variáveis públicas declaradas anteriormente.

Variáveis declaradas como parâmetros de função sempre são locais, portanto as variáveis a e b estão disponíveis para uso somente na função soma.

Variável local e portanto somente disponível para a função soma. Importante: não é a mesma variável definida anteriormente.

Variáveis declaradas como parâmetros de função sempre são locais, portanto as variáveis a e b estão disponíveis para uso somente na função diferenca, embora na função soma também tenha sido utilizado o mesmo nome para as variáveis.

## 52 ♦ Programando em C para Linux, Unix e Windows

```
    resultado = a - b;  
    return resultado;  
}
```

### Programa 7.3

#### Resultado do Programa 7.3

```
Entre com os valores:10 20  
Soma : 30  
Diferenca : -10
```

Valores digitados.

## 7.8 Definição extern

Quando o sistema é separado em vários programas, pode-se ter o problema de acesso a certas variáveis globais, pois a definição da mesma pode estar em um programa fonte e é necessário acessar estas variáveis em outro programa fonte.

Como na linguagem C deve-se sempre definir uma variável antes de usá-la, quando ocorrer a situação descrita, deve ser indicado no programa que irá usar a variável que a mesma está definida em outro programa.

Para fazer isto basta colocar a palavra `extern` na frente da definição da variável juntamente com a definição do seu tipo e nome. Feito isto, está sendo indicado para o compilador o necessário para que não sejam gerados erros, e indicado que a variável já foi definida em outro arquivo fonte.

Veja o exemplo do programa principal e do programa auxiliar:

#### Programa principal:

```
#include <stdio.h>  
void imprime_soma (void);
```

Declaração do protótipo da função. Mesmo que a função não esteja no mesmo código fonte, é importante "informar" ao compilador que esta função existe, senão ocorrerá erro na compilação.

```
int    vlr_a;  
int    vlr_b;
```

Declarando as variáveis como públicas, ou seja, elas estarão disponíveis para uso em todo o programa.

```
void main (void)  
{  
    int    resultado;  
  
    printf ("Entre com os valores:");  
    scanf ("%d %d", &vlr_a, &vlr_b);  
    imprime_soma();  
}
```

#### Programa 7.4.1

Programa auxiliar:

```
#include <stdio.h>
extern int    vlr_a;
extern int    vlr_b;
```

Declarando que **existem**, em outro programa, as variáveis públicas, ou seja, elas estarão disponíveis para uso em todo o programa.

```
void imprime_soma (void)
{
    printf ("Soma %d\n", vlr_a + vlr_b);
}
```

Uso das variáveis públicas.

Programa 7.4.2

Resultado da execução do programa 7.4.1 e 7.4.2

```
Entre com os valores:20 70
Soma 90
```

Valores digitados.

## 7.9 Definição static

Como padrão, toda variável definida dentro de uma função é alocada na pilha interna de execução da função. Ao final da função, a pilha é liberada, liberando assim a memória alocada pela variável. Na próxima chamada à função é feita uma nova alocação na pilha e assim por diante.

Quando for necessário que uma variável local de uma função permaneça com o seu valor mantido, mesmo depois que a função termine, permitindo assim na próxima chamada utilizar o valor anterior, deve-se indicar através da palavra `static` na definição da mesma.

Veja o exemplo:

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

```
int somatorio (int, int);
```

```
void main (void)
{
```

```
    int    vlr_a;
    int    i;
```

```
    i = 1;
```

```
    while (1)
```

```
    {
```

```
        printf ("Entre com um valor:");
```

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

```
        if (vlr_a == 0)
```

```
            break;
```

Quando o valor for zero, o programa é encerrado.

## 54 ♦ Programando em C para Linux, Unix e Windows

```
        printf ("Somatorio %d\n", somatorio (i++, vlr_a));
    }
}

int somatorio (int cont, int vlr)
{
    static int    soma;
    if (cont == 1)
        soma = vlr;
    else
        soma += vlr;
    return soma;
}
```

Declara a variável como `static`, ou seja, diz ao compilador que o conteúdo da variável `soma` deve ser guardado na memória até o final da execução do programa.

Na primeira execução, a variável `soma` somente recebe o valor digitado...

... nas demais execuções, a variável `soma` recebe o seu próprio valor somado do conteúdo da variável `vlr`.

**Programa 7.5**

### Resultado do Programa 7.5

```
Entre com um valor:1
Somatorio 1
Entre com um valor:5
Somatorio 6
Entre com um valor:10
Somatorio 16
Entre com um valor:0
```

Valor digitado.

Valor digitado.

Valor digitado.

## 7.10 Função `atexit`

Sintaxe:

```
int atexit(void (*func) (void))
```

Conforme definido no ANSI C, pode-se registrar até 32 funções que serão automaticamente executadas quando um processo termina. Estas funções são chamadas de *exit handlers* e são registradas através da chamada à função `atexit()`.

As funções serão chamadas na ordem inversa ao seu registro. Pode-se registrar a mesma função mais de uma vez que a mesma será chamada tantas vezes quantas registradas.

A função a ser registrada não deve esperar nenhum parâmetro, bem como não é esperado que a função retorne nenhum valor, devendo ser definida como:

```
void funcao(void);
```

Veja o exemplo:

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

void saindo1(void)
{
    printf("\nFinalizando 1.");
}

void saindo2(void)
{
    printf("\nFinalizando 2.");
}

int main(void)
{
    printf("\nRegistrando funções..");
    atexit(saindo1);
    atexit(saindo2);
    printf("\nNo meio do programa...");
    exit(0);
}
```

A função não recebe e nem devolve nenhum valor.

Registro das funções que serão executadas antes do término do programa.

### Programa 7.6

#### Resultado do Programa 7.6

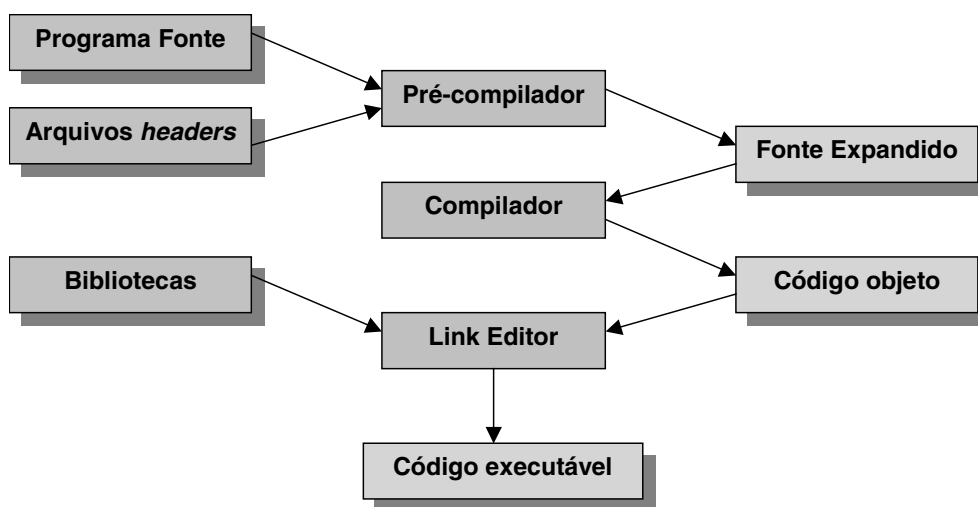
```
Registrando funções..
No meio do programa...
Finalizando 2.
Finalizando 1.
```

Execução na ordem inversa.

# **Pré-Compilação**

*Apenas porque tudo é diferente não significa que algo mudou.*  
Irene Peter, escritora americana

## 8.1 Fases de uma compilação



O processo de compilação de um programa é constituído de três fases distintas: pré-compilação, compilação e link-edição. Na fase de pré-compilação, o programa fonte é lido e, caso encontre comandos do pré-compilador, eles se-



rão processados. O pré-compilador gera então um código intermediário que será lido pelo compilador. O compilador interpreta a linguagem deste fonte intermediário e gera o código objeto, que é um código em *assembler*, pronto para ser utilizado.

A última fase do processo de compilação é a link-edição. Nesta fase, o link-editor lê o código objeto gerado e identifica nele quais são as funções do sistema que foram utilizadas e busca o código das mesmas nas bibliotecas de sistema. Por fim, o link-editor agrupa todos os códigos objetos e gera o programa executável final.

## 8.2 Diretiva `#include`

Sintaxe:

```
#include <arquivo.h>
#include "arquivo.h"
```

Todos os comandos para o pré-compilador começam com o caractere '#'. O comando `#include` indica para o pré-compilador ler o arquivo indicado e colocar o mesmo no programa fonte intermediário.

O arquivo incluído possui o nome de arquivo *header* e geralmente possui protótipo de funções a serem utilizadas por todos os programas de um sistema. Possui também as declarações de tipos existentes (`typedef`) no sistema.

Quando um arquivo *header* pertence ao sistema (ou seja, ao compilador) deve-se colocar o nome dele entre '<' e '>'. Se o arquivo *header* for local (criado pelo programador) deve-se colocar o nome dele entre aspas. Na prática, esta regra não precisa ser seguida, pois a utilização de " " indica ao compilador primeiro procurar o arquivo *header* no diretório local e depois nos diretórios do sistema, e o < > indica para o compilador primeiro procurar o arquivo *header* nos diretórios do sistema e depois no diretório local.

Veja os programas principal, auxiliar e o arquivo *header*:

Programa principal:

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

```
int    vlr_a;
int    vlr_b;
void main (void)
{
```

Repare na forma como os arquivos headers foram inseridos. O arquivo *header* padrão do sistema tem o seu nome preenchido entre < > e o arquivo *header* local tem o seu nome preenchido entre " ".

## 58 ♦ Programando em C para Linux, Unix e Windows

```
int      resultado;

printf ("Entre com os valores:");
scanf ("%d %d", &vlr_a, &vlr_b);
imprime_soma();
}
```

← Esta função de usuário está declarada no arquivo *header soma.h*.

### Programa 8.1.1

Programa auxiliar:

```
#include <stdio.h>
extern int      vlr_a;
extern int      vlr_b;

void imprime_soma (void)
{
    printf ("Soma %d\n", vlr_a + vlr_b);
}
```

### Programa 8.1.2

Arquivo *header (soma.h)*:  
void imprime\_soma (void);

← Declaração da função *imprime\_soma*.

Resultado dos Programas 8.1.1 e 8.1.2  
Entre com os valores:10 50  
Soma 60

← Valores digitados.

## 8.3 Diretiva #define

Sintaxe:

```
#define nome_constante valor_constante
```

É possível definir constantes para o pré-compilador, fazendo com que ele atribua um valor a uma variável. Um detalhe importante a ressaltar é que esta variável é uma variável do pré-compilador e não do programa.

Cada vez que o pré-compilador encontrar esta variável, ele substituirá pelo conteúdo definido anteriormente, não levando em consideração o contexto de compilação.

Vale ressaltar que a definição de uma variável de pré-compilação é uma pura substituição de caracteres.

Veja o exemplo:

```
#include <stdio.h>
#define VALOR_MAGICO 34

void main (void)
{
    int    vlr_a;

    while (1)
    {
        printf ("Entre com o valor:");
        scanf ("%d", &vlr_a);
        if (vlr_a == VALOR_MAGICO)
            break;
    }
}
```

Como são definições de pré-processamento....

A definição VALOR\_MAGICO...

Programa 8.2 (antes do pré-processamento)

```
void main (void)
{
    int    vlr_a;

    while (1)
    {
        printf ("Entre com o valor:");
        scanf ("%d", &vlr_a);
        if (vlr_a == 34)
            break;
    }
}
```

...simplesmente não estão mais disponíveis para o compilador.

...é substituído pelo valor 34, definido no início do programa.

Programa 8.2 (após o pré-processamento)

Resultado do programa 8.2

```
Entre com o valor:20
Entre com o valor:23
Entre com o valor:34
```

## 8.4 Diretivas #if, #else e #endif

Sintaxe:

```
#if condição
    bloco de condição verdadeiro
#else
    bloco condição falso
#endif
```

## 60 ♦ Programando em C para Linux, Unix e Windows

Às vezes é preciso selecionar um trecho de um código de acordo com uma condição preestabelecida, de forma que se compile ou não um trecho do código.

Esta técnica, chamada compilação condicional, é muito usada quando se tem um programa que será usado em diversas plataformas (Linux, Windows etc.) e somente um pequeno trecho de programa difere de um sistema para outro. Como é extremamente desejável que se tenha um único código, simplificando a manutenção e evitando riscos de alterar em um sistema e esquecer de alterar em outro, utiliza-se compilação condicional nos trechos diferentes.

Pode-se selecionar somente um trecho com o `#if` ou selecionar entre dois trechos com o `#if...#else`. O final do trecho, em qualquer um dos casos, é delimitado pela diretiva `#endif`.

Por último, deve-se ressaltar que o `#if` só será executado se na fase de pré-compilação for possível resolver a expressão condicional colocada. Portanto, não é possível fazer compilação condicional baseada em valores de variáveis da linguagem C, pois o valor da variável só estará disponível quando o programa for executado e não durante a compilação.

A variável do teste pode ser definida internamente ou ser diretamente definida quando se chama o comando de compilação, tornando assim bem dinâmico o processo. Para se definir um valor para uma variável ao nível de comando de compilação deve-se usar a opção a seguir:

Plataforma Linux:

```
gcc progxx.c -Dvar=valor -o progxx
```

Plataforma Windows:

LCC-Win32

Colocar no campo `#defines` (opção *compiler*) do projeto da aplicação.

Veja o exemplo:

```
#include <stdio.h>
#define PULA 1
void main (void)
{
    int i;

    for (i=1; i < 30; i++)
    {
```

← Qualquer valor pode ser atribuído ao define.

```

#if PULA == 1
    if (i > 10 && i < 20)
        continue;
#endif
    printf ("%d\n", i);
}

```

Todo este trecho estará disponível para o compilador.

### Programa 8.3 (antes do pré-processamento)

```

void main (void)
{
    int i;

    for (i=1; i < 30; i++)
    {
        if (i > 10 && i < 20)
            continue;
        printf ("%d\n", i);
    }
}

```

Repare que somente as diretivas de pré-processamento foram suprimidas

### Programa 8.3 (após o pré-processamento)

#### Resultado do Programa 8.3

```

1
2
3
4
5
6
7
8
9
10
20
21
22
23
24
25
26
27
28
29

```

## 8.5 Diretivas #ifdef e #ifndef

Sintaxe:

```
#ifdef variável_pré_definida
    bloco de condição verdadeiro
#else
    bloco de condição falso
#endif
```

ou

```
#ifndef variável_pré_definida
    bloco de condição verdadeiro
#else
    bloco de condição falso
#endif
```

Pode-se implementar também a compilação condicional baseada na existência de uma variável e não em seu conteúdo. Para isto é utilizada a diretiva `#ifdef`. Quando a variável especificada estiver definida, o trecho entre o `#ifdef` e o `#endif` será compilado, caso contrário, não.

Pode-se definir a variável também ao nível de comando de compilação evitando assim a alteração de código quando da geração de versões diferentes. Usar a seguinte sintaxe para se fazer isto:

Plataforma Linux:

```
gcc progxx.c -Dvariavel -o progxx
```

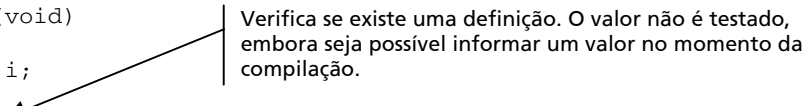
Plataforma Windows:

Colocar no campo `#defines` (opção *compiler*) do projeto da aplicação.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    i;

#ifdef DOS
    clrscr(); /* Para DOS, chamar funcao de limpar tela*/
#else
#ifdef UNIX
                /* <Esc> [ 2 J <Esc> [ H          */
    printf ("^[[2J^[[H"); /*LINUX sequencia de caracteres */
#endif
#endif
```



Verifica se existe uma definição. O valor não é testado, embora seja possível informar um valor no momento da compilação.

```
#endif
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

#### Programa 8.4

```
void main (void)
{
    int    i;
    clrscr();
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

#### Programa 8.4 (após pré-processamento compilado no ambiente Windows)

```
void main (void)
{
    int    i;
    printf ("^[2J^[[H");
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

#### Programa 8.4 (após pré-processamento compilado no ambiente Unix/Linux)

#### Resultado do Programa 8.4

```
1
2
3
4
5
6
7
8
9
```

## 8.6 Diretiva #undef

#### Sintaxe:

```
#undef variável_pré_definida
```

Na construção de dependências pode-se ter uma situação em que seja necessário desabilitar alguma variável de pré-compilação, mesmo que ela seja definida ao nível de comando de compilação.

Para isto é utilizada a diretiva `#undef`, que irá retirar a definição da variável especificada.

## 64 ♦ Programando em C para Linux, Unix e Windows

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    i;

    #ifdef DOS
    #undef UNIX
    clrscr();
    #endif
    #ifdef UNIX
    printf ("^[2J^[[H");
    #endif

    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

Mesmo que, por engano, sejam especificadas as duas opções de compilação. O compilador irá "destruir" a segunda definição...

... e portanto não estará disponível aqui.

### Programa 8.5

```
void main (void)
{
    int    i;
    printf ("^[2J^[[H");
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

### Programa 8.5 (após pré-processamento compilado no ambiente Unix/Linux)

```
void main (void)
{
    int    i;
    clrscr();
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

### Programa 8.5 (após pré-processamento compilado no ambiente Windows)

```
#include <stdio.h>
void main (void)
{
    int    i;

    #ifdef DOS
    clrscr();
    #endif
    #ifdef UNIX
    printf ("^[2J^[[H");
    #endif
}
```

Sem a diretiva #undef...

..e se esta definição existir...

...este comando também pode ser considerado.



```
#endif

    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

**Programa 8.6** (igual ao programa 8.5, sem a diretiva `#undef`)

```
void main (void)
{
    int      i;

    clrscr();
    printf ("^[2J^[H"); } Os 2 comandos estão disponíveis.

    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

**Programa 8.6** (após pré-processamento compilado para os ambientes Unix/Linux e Windows por “engano”)

Resultado do Programa 8.5 e 8.6.

```
1
2
3
4
5
6
7
8
9
```

## 8.7 Diretiva `#error`

Sintaxe:

```
#error mensagem
```

Esta diretiva deve ser usada quando se quer exigir a definição de uma ou outra variável ao nível de compilação ou internamente no programa.

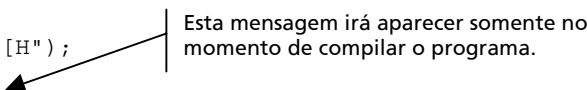
Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int      i;
```

## 66 ♦ Programando em C para Linux, Unix e Windows

```
#ifndef DOS
    clrscr(); /* Para sistemas DOS, chamar funcao de limpar tela*/
#else
    #ifndef UNIX
        printf ("^[[2J^[[H");
    #else
        #error Especificar -DUNIX ou -DDOS na compilacao
    #endif
#endif

    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```



Esta mensagem irá aparecer somente no momento de compilar o programa.

### Programa 8.7

```
void main (void)
{
    int    i;
    #error Especificar -DUNIX ou -DDOS na compilacao
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
}
```

### Programa 8.7 (após pré-processamento compilado)

- o Compilando no Unix/Linux sem colocar a definição  
"p8\_7.c", line 12.8: 1540-086: (S) Especificar -DUNIX ou -DDOS na compilacao
- o Compilando no Windows sem colocar a definição (LCC-Win32)  
cpp: c:\marcos\c\p8\_7.c:12 #error directive: Especificar -DUNIX ou -DDOS na compilacao

## 8.8 Variáveis predefinidas

O pré-compilador disponibiliza uma série de variáveis de pré-compilação para serem utilizadas no programa. Essas variáveis geralmente são utilizadas para código de "debug" ou "log" a ser gerado por programas. São elas:

__LINE__	Número da linha no arquivo fonte.
__FILE__	Nome do arquivo fonte
__DATE__	Data da compilação
__TIME__	Hora da compilação

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    i;

#ifdef DEBUG
    printf ("Inicio do programa %s\n", __FILE__);
    printf ("Versao de %s-%s\n", __DATE__, __TIME__);
#endif
    for (i=1; i < 10; i++)
        printf ("%d\n", i);
#ifdef DEBUG
    printf("A contagem parou! Estamos na linha %d\n", __LINE__);
#endif
    printf("Fim da execução\n");
#ifdef DEBUG
    printf("A última linha do programa é: %d\n", __LINE__);
#endif
}
```

A diretiva `__FILE__` sempre vai ter o nome do programa fonte, mesmo que o executável tenha outro nome.

Estas diretivas sempre terão a data e a hora da compilação do programa.

A diretiva `__LINE__` vai ser sempre substituída pelo compilador por um número que representa a linha dentro do arquivo fonte.

### Programa 8.8

```
void main (void)
{
    int    i;

    for (i=1; i < 10; i++)
        printf ("%d\n", i);

    printf("Fim da execução\n");
}
```

**Programa 8.8** (após pré-processamento compilado sem a opção definindo a diretiva `DEBUG`)

```
void main (void)
{
    int    i;

    printf ("Inicio do programa %s\n", "p8_8.c");
    printf ("Versao de %s-%s\n", "Mar 28 2005", "09:57:20");

    for (i=1; i < 10; i++)
        printf ("%d\n", i);

    printf("A contagem parou! Estamos na linha %d\n", 13);
}
```

## 68 ♦ Programando em C para Linux, Unix e Windows

```
printf("Fim da execução\n");  
  
printf("A última linha do programa é: %d\n", 17);  
  
}
```

**Programa 8.8** (após pré-processamento compilado com a opção definindo a diretiva DEBUG)

**Resultado do Programa** (compilado com a opção definindo a diretiva DEBUG)

```
Inicio do programa p8_8.c  
Versao de Mar 28 2005-09:56:31  
1  
2  
3  
4  
5  
6  
7  
8  
9  
A contagem parou! Estamos na linha 13  
Fim da execução  
A última linha do programa é: 17
```



*Estabeleça suas limitações e depois certifique-se de que elas são suas.*

Richard Bach, escritor americano

## 9.1 Definindo Vetores

Sintaxe:

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

Define-se como vetor uma variável que possui várias ocorrências de um mesmo tipo. Cada ocorrência é acessada através de um índice. Os vetores também são chamados de matrizes unidimensionais por possuírem somente um índice.

Para definir um vetor em C, deve-se indicar a quantidade de ocorrência que o mesmo terá, colocando na sua definição o valor entre '[' e ']'.

Os índices de um vetor em C irão sempre começar de zero, fato que deve ser lembrado, pois geralmente este detalhe é um grande causador de problemas. Portanto, para acessar a primeira ocorrência de um vetor, deve-se indicar o índice zero. Veja o exemplo:

```
#include <stdio.h>
#define TAMANHO    5

void main (void)
{
    int    i;
```

## 70 ♦ Programando em C para Linux, Unix e Windows

```
int    vlr_a;
int    soma;
int    vetor [TAMANHO];

for (i=0; i < TAMANHO; i++)
{
    printf ("Entre com o valor %d:", i + 1);
    scanf ("%d", &vlr_a);
    vetor [i] = vlr_a;
}
soma = 0;
for (i=0; i < TAMANHO; i++)
    soma += vetor [i];
printf ("Media : %f\n", soma / (TAMANHO * 1.0));
}
```

Na linguagem C, um vetor sempre começa na posição 0.

Esta multiplicação serve para converter o valor 5 (diretiva TAMANHO) que é inteira em real.

### Programa 9.1

#### Resultado do Programa 9.1

```
Entre com o valor 1:10
Entre com o valor 2:20
Entre com o valor 3:30
Entre com o valor 4:40
Entre com o valor 5:50
Media : 30.000000
```

Valores 10, 20, 30, 40 e 50 digitados.

## 9.2 Definindo Matrizes

### Sintaxe:

```
tipo nome[quantidade_linhas][quantidade_colunas];
```

Para definir matrizes basta adicionar mais uma dimensão na definição da variável. Por compatibilidade com a matemática, a primeira dimensão é chamada de linha e a segunda de colunas.

Para se acessar um item de uma matriz, deve-se indicar os dois índices.

### Veja o exemplo:

```
#include <stdio.h>
#define TAM    2

void main (void)
{
    int    i,j;
    int    determ;
    int    vlr_a;
    int    matriz [TAM] [TAM];
```

```

for (i=0; i < TAM; i++)
    for (j=0; j < TAM; j++)
    {
        printf ("Entre item %d %d:", i + 1, j + 1);
        scanf ("%d", &vlr_a);
        matriz [i][j] = vlr_a;
    }
determ = matriz[0][0] * matriz [1][1] -
        matriz[0][1] * matriz [1][0];
printf ("Determinante : %d\n", determ);
}

```

Uma regra que pode-se sempre levar em consideração: para cada dimensão de uma matriz, sempre haverá um laço (normalmente um `for`). Se houver 2 dimensões, então haverá 2 laços.

### Programa 9.2

#### Resultado do Programa 9.2

```

Entre item 1 1:10
Entre item 1 2:20
Entre item 2 1:30
Entre item 2 2:40
Determinante : -200

```

Valores 10, 20, 30 e 40 digitados.

## 9.3 Matrizes n-Dimensionais

### Sintaxe:

```
Tipo nome[dimensão_1][dimensão_2][dimensão_3][dimensão_4];
```

O conceito de dimensão pode ser estendido para mais de duas dimensões, criando-se matrizes n-dimensionais. Apesar de terem pouco uso prático, deve-se lembrar que sempre cada dimensão definida terá o índice começando de zero e terminando em uma unidade antes do tamanho especificado para aquela dimensão. Veja o exemplo:

```

#include <stdio.h>
#define DIM_1    2
#define DIM_2    5
#define DIM_3    3
#define DIM_4    4

void main (void)
{
    int    i,j,k,l;
    int    matriz [DIM_1][DIM_2][DIM_3][DIM_4];

    /*Codigo para zerar uma matriz de 4 dimensoes */
    for (i=0; i < DIM_1; i++)
        for (j=0; j < DIM_2; j++)
            for (k=0; k < DIM_3; k++)
                for (l=0; l < DIM_4; l++)

```

Uma regra que pode-se sempre levar em consideração: para cada dimensão de uma matriz, sempre haverá um laço (normalmente um `for`). Se houver 4 dimensões, então haverá 4 laços.

## 72 ♦ Programando em C para Linux, Unix e Windows

```
        matriz [i][j][k][l] = i+j+k+l;

for (i=0; i < DIM_1; i++)
    for (j=0; j < DIM_2; j++)
        for (k=0; k < DIM_3; k++)
            for (l=0; l < DIM_4; l++)
                printf("\nValor para matriz em [%d] [%d] [%d] [%d] =
%d", i,j,k,l, matriz[i][j][k][l]);
}
```

### Programa 9.3

## 9.4 Inicializando Matrizes

Sintaxe:

```
tipo vetor[5]={vlr_1, vlr_2, vlr_3, vlr_4, vlr_5 };
tipo matriz[2][2] = { {vlr_11,vlr_12}, {vlr_21,vlr_22} };
```

Pode-se, ao mesmo tempo em que se define a matriz, inicializá-la com valores, utilizando a seguinte sintaxe.

- o Os valores devem ser colocados de acordo com as dimensões.
- o Cada dimensão deve ser colocada dentro de '{' e '}'.
- o Não se pode pular valores; todos os valores devem ser colocados.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    i,j, k;
    int    matriz1 [5] = {1, 2, 3, 4, 5};
    int    matriz2 [3][3] = {{11, 12, 13},
                             {21, 22, 23},
                             {31, 32, 33}};
    int    matriz3 [3][2][2] = {{{111, 112},
                                 {121, 122}},
                                 {{211, 212},
                                 {221, 222}},
                                 {{311, 312},
                                 {321, 322}}};

    printf ("Primeira Matriz\n");
    for (i=0; i < 5; i++)
        printf ("%d ", matriz1 [i]);
    printf ("\n\n");

    printf ("Segunda Matriz\n");
    for (i=0; i < 3; i++)
```



```

{
    for (j=0; j < 3; j++)
        printf ("%d ", matriz2 [i][j]);
    printf ("\n");
}
printf ("\n");

printf ("Terceira Matriz\n");
for (i=0; i < 3; i++){
    for (j=0; j < 2; j++){
        for (k=0; k < 2; k++)
            printf ("%d ", matriz3 [i][j][k]);
        printf ("\n");
    }
    printf ("\n");
}
}

```

#### Programa 9.4

#### Resultado do Programa 9.4

Primeira Matriz  
1 2 3 4 5

Segunda Matriz  
11 12 13  
21 22 23  
31 32 33

Terceira Matriz  
111 112  
121 122

211 212  
221 222

311 312  
321 322

## 9.5 Matrizes como Parâmetros

Quando se coloca um vetor como parâmetro, a linguagem C passa somente o seu endereço, não fazendo uma cópia na pilha. Portanto, pode-se definir o parâmetro sem a quantidade de elementos, pois, como só será recebido o endereço, pode-se acessar toda a matriz através deste endereço.

A mesma regra se aplica a matrizes para o caso da primeira dimensão. Pode-se não informar a quantidade de elementos da primeira dimensão. Devido à

## 74 ♦ Programando em C para Linux, Unix e Windows

construção sintática da linguagem, deve-se porém informar as demais dimensões para que o compilador gere código corretamente. Veja o exemplo:

```
#include <stdio.h>

void imprime_1(int vet[])
{
    int i;
    printf ("Primeira Matriz\n");
    for (i=0; i < 5; i++)
        printf ("%d ", vet [i]);
    printf ("\n\n");
}

void imprime_2(int mat[][3])
{
    int i,j;
    printf ("Segunda Matriz\n");
    for (i=0; i < 3; i++)
    {
        for (j=0; j < 3; j++)
            printf ("%d ", mat [i][j]);
        printf ("\n");
    }
}

void main (void)
{
    int matriz1 [5] = {1, 2, 3, 4, 5};
    int matriz2 [3][3] = {{11, 12, 13},
                          {21, 22, 23},
                          {31, 32, 33}};

    imprime_1(matriz1);
    imprime_2(matriz2);
}
```

← Não é preciso informar o tamanho do índice...

← ...mas deve-se tomar cuidado na hora de manipular o vetor, pois caso o programa "tente" acessar um índice que não existe, o resultado será indesejado.

← Para a primeira dimensão, não é necessário informar a quantidade de índices. Mas para as demais é necessário.

### Programa 9.5

#### Resultado do programa 9.5

Primeira Matriz  
1 2 3 4 5

Segunda Matriz  
11 12 13  
21 22 23  
31 32 33



*O difícil é aprender a ler. O resto está escrito.*  
(Anônimo)

## 10.1 Implementação de Strings

A linguagem C implementa o conceito de cadeia de caracteres, ou *strings*, utilizando um vetor de caracteres. Ao definir uma *string*, portanto, deve-se definir um vetor de caracteres com determinado tamanho.

A marcação do fim da *string* será indicada através da colocação de um caractere zerado, chamado tecnicamente de caractere `NULL`. Este caractere pode ser indicado, testado, usado através do literal definido `NULL` quando se incluem arquivos headers `stdio.h` ou `strings.h` ou através da forma de constante de caractere `'\0'`.

Portanto, como a *string* deve ser terminada por um caractere `NULL`, sempre deve ser considerada uma posição a mais no vetor para estes caracteres. Por exemplo, se for definido um vetor de nomes com até 30 caracteres deve-se definir um vetor com 31 posições.

Na definição de uma *string* pode-se também já assinalar um valor, bastando para isto colocar após o `'='` a constante de inicialização entre aspas.

Para se imprimir uma *string*, deve ser utilizado o formato `'%s'`, conforme já visto.

## 76 ♦ Programando em C para Linux, Unix e Windows

Veja o exemplo:

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

```
void main (void)
{
```

```
    char    nome [40] = "Pacífico Pacato Cordeiro Manso";
```

```
    printf ("%s\n", nome);
```

```
    printf ("%d\n", sizeof(nome));
```

```
    nome [0] = NULL;
```

```
    printf ("%s\n", nome);
```

```
    printf ("%d\n", sizeof(nome));
```

```
}
```

Programa 10.1

A definição é idêntica a um vetor, neste caso têm-se 39 posições para caracteres. Lembrar que a última posição deve ser reservada para o caractere que indica o final da *string* (`\0`).

Este comando simplesmente “limpa” o conteúdo da variável.

O operador `sizeof` irá mostrar o tamanho reservado para a *string* e não o seu tamanho atual.

Resultado do Programa 10.1

```
[Pacífico Pacato Cordeiro Manso]
[40]
[]
[40]
```

## 10.2 Entrada/Saída de Strings

Sintaxe:

```
scanf( "%s", endereçoString );
gets(endereçoString);
puts(endereçoString);
```

Para realizar a entrada de *strings*, pode-se utilizar a função `scanf`, usando como formato o `'%s'`. Um detalhe importante sobre esta forma de entrada de *string* é o fato de somente ser lida a sequência de caracteres até ser encontrado um branco, não sendo possível ler caracteres brancos para uma variável usando o `scanf`.

Uma maneira mais útil de se fazer a leitura de *strings* do terminal é utilizar a função `gets`. Esta função lê toda a linha e coloca o conteúdo na variável indicada, permitindo assim a entrada de caracteres em branco para dentro da cadeia.

É de inteira responsabilidade do programador reservar espaço suficiente na variável para ser realizada a entrada de dados. Caso entrem mais caracteres que o reservado, ocorrerá invasão de memória, cancelando o programa.

Para realizar a saída de uma *string*, pode-se utilizar a função `printf` com o formato `'%s'` ou a função específica `puts`.

Veja o exemplo:

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

void main (void)
{
    char    nome [30];
    char    frase [100];

    printf ("Entre com uma frase : \n");

    gets (frase); } Leitura e impressão de uma string. O término da leitura
    puts (frase); } ocorrerá quando for pressionado <ENTER>.

    printf ("Entre com o seu nome : ");
    scanf ("%s\n", &nome); ← Leitura de uma string. O término da leitura
                                ocorrerá após ser pressionado <ENTER>.

    printf ("Sr(a). %s seja bem vindo ao curso\n\n", nome);
}
```

### Programa 10.2

#### Resultado do Programa 10.2

```
Entre com uma frase :
Bem vindo a aula de strings.
Bem vindo a aula de strings.
Entre com o seu nome : Marcos Aurelio Pchek Laureano
Sr(a). Marcos seja bem vindo ao curso
```

Valor digitado. →

Valor digitado. →

← A função `scanf` lê os dados somente até encontrar o 1º caractere branco.

## 10.3 String como vetor

Devido ao fato de uma *string* ser implementada como um vetor de caracteres, nada impede que se faça o acesso ao vetor utilizando índices, acessando assim cada caractere da *string* de maneira direta.

Veja o exemplo:

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

## 78 ♦ Programando em C para Linux, Unix e Windows

```
void main (void)
{
    char    nome [40];
    int     i;

    printf ("Entre com o seu nome : \n");
    gets (nome);

    printf ("Nome digitado\n");
    for (i=0; i < strlen(nome); i++)
        putchar (nome [i]);
    putchar ('\n');
}
```

Como a definição de uma *string* é idêntica à definição de um vetor...

...o tratamento de uma *string* pode ser feito igual ao tratamento de um vetor qualquer.

### Programa 10.3

#### Resultado do Programa 10.3

```
Entre com o seu nome :
Marcos Aurelio
Nome digitado
Marcos Aurelio
```

String digitada.

## 10.4 Função strlen

#### Sintaxe:

```
int strlen(endereçoString);
```

Para obter o tamanho de uma *string* utiliza-se a função `strlen`. Esta função irá retornar a quantidade de caracteres existentes em uma *string*, não considerando o caractere `NULL` na contagem dos caracteres.

#### Veja o exemplo:

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

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

    printf ("Entre com uma frase : \n");
    gets (frase);

    tamanho = strlen (frase);
}
```

A função `strlen` informa a quantidade de caracteres utilizados, ou seja, são levados em consideração todos os caracteres até ser encontrado o `\0`.

```

printf ("A frase possui %d caracteres\n", tamanho);
printf("A variável tem tamanho %d\n",sizeof(frase));
}

```

#### Programa 10.4

#### Resultado do Programa 10.4

Entre com uma frase :

Este eh o teste da funcao strlen ← *String digitada.*

A frase possui 32 caracteres

A variável tem tamanho 100

← Só para lembrar que o tamanho reservado é diferente do tamanho efetivamente usado.

## 10.5 Função strcat

Sintaxe:

```
strcat( endereçoString1, endereçoString2 );
```

Pode-se fazer a concatenação de dois *strings*, colocando um ao final do outro. A função para fazer isto é `strcat`. Esta função irá concatenar a segunda *string* ao final da primeira *string*.

O primeiro parâmetro da função, portanto, deve ser uma variável e possuir o espaço suficiente para o resultado. A função não irá testar se existe espaço fazendo a movimentação de caracteres do segundo parâmetro para o final do primeiro. O segundo parâmetro pode ser uma variável ou uma constante delimitada por aspas.

Veja o exemplo:

```

#include <stdio.h>
#include <strings.h>

```

```
void main (void)
```

```

{
    char    mensagem [100] = "Sr(a). ";
    char    nome [40];

```

```

    printf ("Entre com o seu nome : \n");
    gets (nome);

```

```

    strcat (mensagem, nome);
    strcat (mensagem, ". Bem vindo ao curso");
    puts (mensagem);
}

```

← A concatenação ocorre logo após o último caractere da primeira *string*. Seria o equivalente em algoritmo a `var_string = var_string + nova_string`, embora na linguagem C não se possa trabalhar com *strings* desta forma.

#### Programa 10.5

## 80 ♦ Programando em C para Linux, Unix e Windows

### Resultado do Programa 10.5

```
Entre com o seu nome :      String digitada.
Marcos Laureano ←
Sr(a). Marcos Laureano. Bem vindo ao curso
```

## 10.6 Função strcpy

Sintaxe:

```
strcpy( endereçoString1, endereçoString2 );
```

Quando quiser copiar o conteúdo de uma *string* para outro, deve-se utilizar a função `strcpy`. O conteúdo da segunda variável ou constante informada será copiado para a área indicada no primeiro parâmetro.

Como sempre, é função do programador garantir espaço suficiente para que a cópia seja realizada.

Veja o exemplo:

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

```
void main (void)
{
    char    backup [40];
    char    nome [40];

    printf ("Entre com o seu nome : \n");
    gets (nome);
```

```
    strcpy (backup, nome);
    puts (backup);
}
```

← Seria o equivalente em algoritmo a `var_string = nova_string`, embora na linguagem C não se possa trabalhar com *strings* desta forma.

### Programa 10.6

### Resultado do Programa 10.6

```
Entre com o seu nome :
Pacifico Pacato Cordeiro Manso ← String digitada.
Pacifico Pacato Cordeiro Manso
```

## 10.7 Função strcmp

Sintaxe:

```
int strcmp(endereçoString1, endereçoString2 );
```

Para comparar o conteúdo de duas *strings* deve-se usar a função `strcmp`. Esta função irá fazer a comparação, caractere a caractere, dos dois parâmetros in-



formados. Como não é alterado o conteúdo de nenhum parâmetro, pode ser informado um valor constante em qualquer um deles, apesar de fazer mais sentido usar a constante como segundo parâmetro.

Como resultado da comparação serão obtidos os seguintes valores: -1 indicando que o parâmetro 1 é menor que o parâmetro 2; 0 indicando que os parâmetros são iguais e 1 caso o primeiro seja maior que o segundo parâmetro.

Veja o exemplo:

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

void main (void)
{
    char    nome [80];
    int     tamanho;

    while (1)
    {
        printf ("Entre com nomes (fim p/ terminar): \n");
        gets (nome);

        if (strcmp (nome, "fim") == 0)
            break;
        tamanho = strlen (nome);
        printf ("Nome com %d caracteres\n", tamanho);
    }
}
```

← Importante lembrar que a comparação é feita até encontrar o caractere \0.

### Programa 10.7

#### Resultado do Programa 10.7

```
Entre com nomes (fim p/ terminar):
Castro Alves ← String digitada.
Nome com 12 caracteres
Entre com nomes (fim p/ terminar):
Machado de Assis ← String digitada.
Nome com 16 caracteres
Entre com nomes (fim p/ terminar):
Julio Verne ← String digitada.
Nome com 11 caracteres
Entre com nomes (fim p/ terminar):
Conan Doyle ← String digitada.
Nome com 11 caracteres
Entre com nomes (fim p/ terminar):
fim ← String digitada.
```

## 10.8 Função sprintf

Sintaxe:

```
sprintf( endereçoString, "formato", variável1, variável2, ...);
```

A função `sprintf` tem a mesma funcionalidade da função `printf`, exceto que a saída resultante, após a execução dos formatos, será colocada na variável indicada como primeiro parâmetro.

Veja o exemplo:

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

void main (void)
{
    char    nome [30];
    char    mensagem [100];

    printf ("Entre com o seu nome : ");
    gets (nome);

    sprintf (mensagem, "Sr. %s seja bem vindo ao curso\n\n", nome);
    puts (mensagem);
}
```

A vantagem da função `sprintf` é poder formatar qualquer dado dentro de uma *string*.

Programa 10.8

Resultado do Programa 10.8

```
Entre com o seu nome : Marcos Laureano
Sr. Marcos Laureano seja bem vindo ao curso
```

String digitada.

## 10.9 Função sscanf

Sintaxe:

```
sscanf(string, "formato", endereços_argumentos);
```

A função `sscanf` é idêntica à função `scanf`, mas os dados são lidos da *string*. O valor devolvido é igual ao número de variáveis, às quais foram realmente atribuídos valores. Esse número não inclui variáveis que foram saltadas devido ao uso do especificador de formato `*`. Um valor zero significa que nenhum campo foi atribuído; EOF indica que ocorreu um erro antes da primeira atribuição.

Veja o exemplo:

```
#include <stdio.h>
void main(void)
{
    char str[80];
    int i;
    sscanf("Alo 1 2 3 4 5", "%s%d", str, &i);
    printf("\n%s %d\n", str, i);
}
```

Vai ler uma *string* (formato %s) e depois um valor inteiro (formato %d). As variáveis devem ser informadas na mesma sequência.

Programa 10.9

Resultado do Programa 10.9

Alo 1

## 10.10 Função strncat

Sintaxe:

```
strncat( endereçoString1, endereçoString2, quantidade );
```

A função `strncat` tem o mesmo comportamento da função `strcat`, exceto por concatenar não mais que `quantidade` caracteres da string apontada por `endereçoString2` à string apontada por `endereçoString1`.

Lembrando que não ocorre nenhuma verificação de limite, é responsabilidade do programador assegurar que `endereçoString1` seja suficientemente grande para armazenar seu conteúdo original como também o de `endereçoString2`.

Veja o exemplo:

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

void main(void)
{
    char s1[80], s2[80];
    unsigned int tam;
    printf("\nEntre com uma frase:");
    gets(s1);
    printf("\nEntre com outra frase:");
    gets(s2);

    tam = 79 - strlen(s2);

    strncat(s2, s1, tam);
```

Cálculo simples para garantir que a *string* não tenha o seu tamanho ultrapassado.

A única diferente em relação à função `strcat`, é o último parâmetro, que informa a quantidade de caracteres que devem ser concatenadas.

## 84 ♦ Programando em C para Linux, Unix e Windows

```
printf("\n%s",s2);  
}
```

### Programa 10.10

#### Resultado do Programa 10.10

Entre com uma frase:Então cuidado !

String digitada.

Entre com outra frase:Deve-se respeitar o limite (tamanho) da variável utilizada.

String digitada.

Deve-se respeitar o limite (tamanho) da variável utilizada. Então cuidado !

## 10.11 Função strncpy

Sintaxe:

```
strncpy( endereçoString1, endereçoString2,quantidade );
```

A função `strncpy` tem o mesmo comportamento da função `strcpy`, exceto por copiar até quantidade caracteres da string apontada por `endereçoString2` na string apontada por `endereçoString1`.

Veja o exemplo:

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

```
void main(void)
```

```
{  
    char str1[]="123456789\0";  
    char str2[4];
```

Copia os 3 primeiros caracteres da string str1.

```
    strncpy(str2, str1, 3);
```

```
    printf("\nstr1 = [%s]", str1);  
    printf("\nstr2 = [%s]", str2);
```

É necessário sempre acrescentar o `\0` ao final da string, pois a função `strncpy` copia até o número de caracteres indicados e nenhum dos caracteres pode ser o `\0` ocasionando em resultados "não controlados".

```
    str2[3]='\0';  
    printf("\nstr2 = [%s]", str2);
```

```
}
```

### Programa 10.11

#### Resultado do Programa 10.11

```
str1 = [123456789]  
str2 = [123iP-%iP-%iP-%iP-%i]  
str2 = [123]
```

Resultados não controlados....

## 10.12 Função strncmp

Sintaxe:

```
int strncmp(endereçoString1, endereçoString2, quantidade );
```

Esta função irá fazer a comparação, caractere a caractere, dos dois parâmetros informados, como a função `strcmp`, exceto por comparar até `quantidade` caracteres.

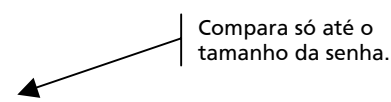
Veja o exemplo:

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

void main(void)
{
    char senha[]="xP1247";
    char s1[80];
    int tam;
    printf("\nEntre com a senha para ver a mensagem:");
    gets(s1);

    tam = strlen(senha);

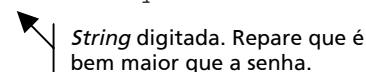
    if( strncmp( s1, senha, tam ) == 0 )
        printf("\nAcertou a senha..");
    else
        printf("\nTente novamente..");
}
```



### Programa 10.12

#### Resultado do Programa 10.12

Entre com a senha para ver a mensagem:xP1247 acho que eh  
Acertou a senha..



String digitada. Repare que é bem maior que a senha.



*Chegar no meio do caminho é não chegar a lugar nenhum.*

Tom Peters, consultor de negócios americano

## 11.1 Conceito Básico

Existe na linguagem C o conceito de ponteiro. Deve-se entender que o ponteiro é um tipo de dado como `int`, `char` ou `float`. A diferença do ponteiro em relação aos outros tipos de dados é que uma variável que seja ponteiro irá guardar um endereço de memória. Através desse endereço pode-se acessar a informação, dizendo que a variável ponteiro aponta para uma posição de memória. O maior problema em relação ao ponteiro é entender quando se está trabalhando com o seu valor, ou seja, o endereço, e quando se está trabalhando com a informação apontada por ele.

Por ser um endereço, deve-se especificar que tipo de variável será encontrado na posição apontada pelo ponteiro. Assim é informado que foi criado um ponteiro para um inteiro, um ponteiro para uma estrutura ou um ponteiro para um arquivo. Quando isto é definido, quer dizer que no endereço indicado pelo ponteiro será encontrado um valor inteiro e o compilador deve gerar código para tratar este endereço como tal.

## 11.2 Definição de Ponteiros

Sintaxe:

```
tipo * variável;
```

Para definir uma variável do tipo ponteiro, deve-se colocar um asterisco ('\*') na frente de uma definição normal daquele tipo. O asterisco deve ser colocado entre o tipo e o nome da variável. Pode-se ter um ponteiro para qualquer tipo de variável possível em C, como inteiro, ponto flutuante, estruturas, arquivos etc.

Convém lembrar que a definição de um ponteiro é somente a definição de um espaço de memória que conterá outro endereço. Portanto, ao definir um ponteiro, somente é alocado o espaço do endereço e não do valor.

Para utilizar um ponteiro, é preciso sempre inicializar o mesmo, ou seja, colocar um endereço válido para depois realizar o acesso. Veja o exemplo:

```
void main (void)
{
    int *a; /*ponteiro para inteiro */
    char *b; /*ponteiro para um caractere */
    float *e; /*ponteiro para um ponto flutuante */
}
```

## 11.3 Uso de Ponteiros

Um ponteiro pode ser utilizado de duas maneiras distintas. Uma maneira é trabalhar com o endereço armazenado no ponteiro e outro modo é trabalhar com a área de memória apontada pelo ponteiro. É importantíssimo diferenciar estes dois modos para não causar problemas.

Quando se quiser trabalhar com o endereço armazenado no ponteiro, utiliza-se o seu nome sem o asterisco na frente. Sendo assim, qualquer operação realizada será feita no endereço do ponteiro.

Normalmente, trabalha-se com área de memória indicada pelo ponteiro, alterando ou lendo o valor desta área. Para tal é preciso colocar um asterisco antes do nome do ponteiro (\*nome), desta forma o compilador entenderá que deve ser acessada a memória e não o endereço do ponteiro. Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    *a;
    int    variavel = 10;
    a = &variavel;
    printf ("Endereco %d\n", a);
    printf ("Valor    %d\n", *a);
}
```

Ponteiro para um inteiro.

O ponteiro a recebe o endereço de memória da variável variavel.

```

*a = 15;
printf ("Valor alterado %d\n", variavel);
printf ("Endereco %d\n", a);

```

Indica que endereço de memória, representado pelo ponteiro, irá receber o valor 15.

Programa 11.1

**Resultado do Programa 11.1**

```

Endereco 804399236
Valor 10
Valor alterado 15
Endereco 804399236

```

...embora o valor contido seja diferente.

O endereço de memória é o mesmo...

## 11.4 Parâmetros de Saída

Para fazer a saída de valores via parâmetros de função, deve-se definir este parâmetro como sendo do tipo ponteiro. Na chamada da função deve-se colocar o endereço de uma variável local ou global neste parâmetro.

Para retornar um valor, deve-se utilizar o asterisco antes do nome do parâmetro, indicando assim que está sendo mudado o valor naquele endereço passado como parâmetro. Veja o exemplo:

```

#include <stdio.h>
void soma (int, int, int *);
void main (void)
{
    int    vlr_a;
    int    vlr_b;
    int    resultado;

    printf ("Entre com os valores:");
    scanf ("%d %d", &vlr_a, &vlr_b);
    soma (vlr_a, vlr_b, &resultado);
    printf ("Soma : %d\n", resultado);
}

```

Indica que o último parâmetro é um ponteiro para inteiro.

Como está sendo passado o endereço de memória da variável (ponteiro), qualquer alteração estará sendo realizada na memória.

```

void soma (int a, int b, int *valor)
{
    *valor = a + b;
}

```

Alterando diretamente na memória.

Programa 11.2

**Resultado do Programa 11.2**

```

Entre com os valores:15 20
Soma : 35

```

Valores digitados.



## 11.5 Operações com Ponteiros

É possível realizar as operações de soma e subtração do valor do ponteiro, ou seja, do endereço armazenado na variável.

Um detalhe a ser observado é que esta soma estará condicionada ao tamanho do tipo que o ponteiro aponta. Explicando melhor, suponha que exista um ponteiro para um inteiro, que ocupa 4 *bytes* na memória. Ao se somar uma unidade neste ponteiro (+ 1) o compilador interpretará que se deseja somar um valor que permita acessar o próximo inteiro e irá gerar código para somar 4 unidades no endereço do ponteiro. Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
    int    *pt_int;
    int     ivalor;
    char   *pt_char;
    char    cvalor;

    pt_int = &ivalor;
    pt_char = &cvalor;

    printf ("Endereco de pt_int = %d\n", pt_int);
    printf ("Endereco de pt_char = %d\n", pt_char);

    pt_int++;
    pt_char++;

    printf ("\nEndereco de pt_int = %d\n", pt_int);
    printf ("Endereco de pt_char = %d\n", pt_char);
}
```

← Adicionando "uma unidade" aos ponteiros.

Programa 11.3

### Resultado do programa 11.3

Endereco de pt\_int = 804399220  
Endereco de pt\_char = 804399228

Endereco de pt\_int = 804399224  
Endereco de pt\_char = 804399229

Ponteiro para inteiro, adicionando uma "unidade", tem-se o acréscimo de 4 bytes

Ponteiro para caractere, adicionando uma "unidade", tem-se o acréscimo de 1 byte.

## 11.6 Ponteiros e Matrizes

Quando é passado um vetor ou matriz como parâmetro, a linguagem C coloca o endereço na pilha. Pode-se então definir o tipo do parâmetro como um ponteiro e acessar a matriz dentro da função como se fosse um ponteiro.

## 90 ♦ Programando em C para Linux, Unix e Windows

Veja o exemplo:

```
#include <stdio.h>
void imprime_1(int *vet)
{
    int i;
    for (i=0; i < 5; i++){
        printf("\nConteudo na matriz na posicao %d=%d", i, *vet);
        printf("\nEndereço de memória = %d", vet );
        vet++;
    }
    printf ("\n");
}

void main (void)
{
    int matriz1 [5] = {1, 2, 3, 4, 5};
    printf("\nTamanho da matriz = [%d]", sizeof(matriz1));
    imprime_1(matriz1);
}
```

Seria o equivalente:  
imprima vet[i];  
i++

Pegando o tamanho da variável na memória e não o tamanho da matriz.

### Programa 11.4

#### Resultado do Programa 11.4

Tamanho da matriz = [20]

Conteudo na matriz na posicao 0=1

Endereço de memória = 804399216

Conteudo na matriz na posicao 1=2

Endereço de memória = 804399220

Conteudo na matriz na posicao 2=3

Endereço de memória = 804399224

Conteudo na matriz na posicao 3=4

Endereço de memória = 804399228

Conteudo na matriz na posicao 4=5

Endereço de memória = 804399232

Embora o vetor tenha apenas 5 posições, é um vetor de inteiros. Como um inteiro ocupa 4 bytes...

Lembrando que o acréscimo de uma "unidade" causa o salto de 4 bytes (inteiro) na memória.

## 11.7 Ponteiros e Strings

Acessar um vetor como ponteiro e vice-versa é muito comum quando são utilizadas *strings*. Quando é definido um vetor de caracteres, pode-se acessar o mesmo através de um ponteiro para caractere. Este ponteiro estará sempre apontando para um único caractere da *string* e através de operações sobre o ponteiro (incremento ou decremento) pode-se caminhar no vetor.

Veja o exemplo:

```
#include <stdio.h>
void main (void)
{
```

```

char    nome [30] = "Marcos Aurelio";
char    *pt;
pt = (char *) &nome;

printf("\nTamanho da string = %d", sizeof(nome));
printf("\nEndereço da 1a. posicao = %d\n", pt );
while (*pt != NULL)
{
    putchar (*pt);
    pt++;
}
putchar ('\n');
printf("\nEndereço da última posicao = %d", pt );
}

```

Aponta para o primeiro caractere da *string* ou vetor.

Pegando o tamanho da variável na memória e não o tamanho da matriz.

Seria o equivalente: imprima vet[i]; i++

#### Programa 11.5

#### Resultado do Programa 11.5

```

Tamanho da string = 30
Endereço da 1a. posicao = 804399184
Marcos Aurelio
Endereço da última posicao = 804399198

```

Endereço inicial da *string* ou vetor...

... e 14 posições depois o endereço atual.

## 11.8 Argumentos de Entrada

De dentro de um programa C pode-se acessar a linha de comando que ativou o programa, permitindo assim passar valores para o programa na sua chamada.

Os valores passados para o programa são chamados de argumentos e pode-se acessá-los colocando-se dois parâmetros na definição da função `main`. O primeiro parâmetro deve ser do tipo inteiro e conterá a quantidade de argumentos passados na linha de comando. É importante observar que o nome do programa é um argumento e portanto será contado como tal. Posto isto, vale dizer que sempre este parâmetro irá considerar o nome do programa como argumento.

O outro parâmetro que deve ser colocado é um vetor de ponteiros. Cada ocorrência deste vetor será um ponteiro para uma *string* contendo o argumento passado para o programa.

Veja o exemplo:

```

#include <stdio.h>
void main (int  argc, char *argv[])
{

```

Contém o número de argumentos passados. Será sempre pelo menos 1, pois o nome do programa é sempre passado como 1º argumento.

Conterá os argumentos passados. Os argumentos são separados por um espaço em branco ao serem passados na linha de comando.

## 92 ♦ Programando em C para Linux, Unix e Windows

```
int i;

printf ("Argumentos digitados\n");
for (i=0; i < argc; i++)
    printf ("Argumento %d - %s\n", i + 1, argv[i]);
}
```

### Programa 11.6

#### Resultado do Programa 11.6

##### 1ª Execução

#> p11\_6 ← Linha de comando.

Argumentos digitados

Argumento 1 - p11\_6 ← Só o nome do programa, não foram passados outros argumentos.

##### 2ª Execução

#> p11\_6 1 2 3 4 5 ← Linha de comando.

Argumentos digitados

Argumento 1 - p11\_6

Argumento 2 - 1

Argumento 3 - 2

Argumento 4 - 3

Argumento 5 - 4

Argumento 6 - 5

Os espaços em branco na linha comando são considerados os delimitadores entre os argumentos.

##### 3ª Execução

#> p11\_6 "1 2 3 4 5" ← Linha de comando.

Argumentos digitados

Argumento 1 - p11\_6

Argumento 2 - 1 2 3 4 5 ← Como foi passado entre " " (aspas), o programa recebeu como um único argumento.

## 11.9 Função strstr

Sintaxe:

```
char *strstr( endereçoStr1, endereçoStr2);
```

A função `strstr` devolve um ponteiro para a primeira ocorrência da *string* apontada por `endereçoStr2` na *string* apontada por `endereçoStr1`. Ela devolve um ponteiro nulo se não for encontrada nenhuma coincidência.

Veja o exemplo:

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

```
void main(void)
{
```

```
char *p;
char frase[] = "isto e um teste";
char *pt_char;
```

```
pt_char = frase;
printf("\nEndereço Inicial = %d", pt_char );
```

```
p = strstr(frase, "to");
```

```
pt_char=p;
```

```
printf("\nEndereço inicial para a pesquisa = %d\n", pt_char );
```

### Programa 11.7

#### Resultado do Programa 11.7

```
Endereço Inicial = 804399220
```

```
Endereço inicial para a pesquisa = 804399222
```

```
to e um testeb
```

Parâmetro de pesquisa. A função irá retornar o endereço correspondente à localização deste parâmetro.

Posição inicial na memória da frase original.

Resultado da pesquisa.

Posição inicial na memória do resultado da pesquisa. Neste exemplo, 2 "unidades" (o char ocupa 1 byte) depois do endereço inicial da frase original.

## 11.10 Função strtok

### Sintaxe

```
char * strtok(endereçoStr1, endereçoStr2);
```

A função `strtok` devolve um ponteiro para a próxima palavra na *string* apontada por `endereçoStr1`. Os caracteres que formam a *string* apontada por `endereçoStr2` são os delimitadores que terminam a palavra. Um ponteiro nulo é devolvido quando não há mais palavras na string.

Na primeira chamada à função `strtok`, o `endereçoStr1` é realmente utilizado na chamada. Nas chamadas seguintes deve-se usar um ponteiro nulo como primeiro argumento.

Pode-se utilizar um conjunto diferente de delimitadores para cada chamada à `strtok`.

Veja o exemplo.

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

## 94 ♦ Programando em C para Linux, Unix e Windows

```
void main(void)
{
    char *p;
    char frase[]="Mario Quintana, o maior poeta gaúcho";

    printf("\nFrase = %s", frase);
    p = strtok(frase, " ");
    printf("\nP = %s", p);
    printf("\nFrase = %s", frase);
    do
    {
        p = strtok('\0', ", ");
        if(p)
            printf("\nP = %s", p);
    } while(p);
}
```

Primeira pesquisa por espaço em branco.

Nas próximas chamadas, deve-se passar um ponteiro "nulo". Isto "indica" para a função que a pesquisa deve continuar no ponteiro anterior.

Demais pesquisas por , (vírgula) ou espaço em branco.

Um valor nulo (NULL) é considerado sempre falso em comparações booleanas (verdadeiro ou falso).

### Programa 11.8

#### Resultado do Programa 11.8

Frase = Mario Quintana, o maior poeta gaúcho

P = Mario

Frase = Mario

P = Quintana

P = o

P = maior

P = poeta

P = gaúcho

A variável original é "modificada"... portando cuidado.



# Manipulação de Arquivos (padrão ANSI)

*Lógica é um método sistemático de chegar à conclusão errada com confiança.*  
Arthur Bloch, escritor americano

## 12.1 Conceitos Importantes

Na linguagem C pode-se trabalhar com arquivos de duas maneiras distintas. Uma maneira é utilizar as funções disponibilizadas pelo sistema operacional para fazer a entrada e a saída de dados. As funções disponibilizadas pelo sistema operacional são mais básicas.

Outra forma, mais adequada, é realizar a entrada de dados utilizando as funções disponibilizadas pela biblioteca de funções do próprio C. Estas funções possuem um nível mais elaborado, facilitando a entrada e a saída de dados. São estas funções que serão utilizadas aqui.

Para se fazer uso destas funções deve ser incluído o arquivo *header* contendo a descrição dos protótipos, algumas constantes utilizadas etc.

Quando um arquivo é aberto através da função `fopen` será retornado um ponteiro para uma estrutura de controle do arquivo. Esta estrutura está definida no arquivo *header* e possui um `typedef` chamado `FILE`. Para se realizar qualquer operação sobre o arquivo deve-se informar este ponteiro como parâmetro.

Na linguagem C, um arquivo pode ser qualquer coisa, desde um arquivo em disco até um terminal ou uma impressora. Basta associar uma `stream` (ponteiro

para arquivos) com um arquivo específico realizando uma operação de abertura. Uma vez o arquivo aberto, informações podem ser trocadas entre ele e o seu programa.

Nem todos os arquivos apresentam os mesmos recursos. Por exemplo, um arquivo em disco pode suportar acesso aleatório, enquanto um teclado não pode. Isso revela um ponto importante sobre o sistema de E/S da linguagem C: todas as *streams* são iguais, mas não todos os arquivos.

Se o arquivo pode suportar acesso aleatório (algumas vezes referido como solicitação de posição), abrir esse arquivo também inicializa o indicador de posição no arquivo para o começo do arquivo. Quando cada caractere é lido ou escrito no arquivo, o indicador de posição é incrementado, garantindo progressão através do arquivo.

Um arquivo é desassociado de uma *stream* específica através de uma operação de fechamento. Se um arquivo aberto para saída for fechado, o conteúdo, se houver algum, de sua *stream* associada é escrito no dispositivo externo. Esse processo é geralmente referido como descarga (*flushing*) da *stream* e garante que nenhuma informação seja acidentalmente deixada no *buffer* de disco. Todos os arquivos são fechados automaticamente quando o programa termina, normalmente com `main` retornando ao sistema operacional ou uma chamada à função `exit`. Os arquivos não são fechados quando um programa quebra (*crash*).

## 12.2 Ponteiro para Arquivos

Como visto, para cada arquivo que se quer acessar, deve-se definir uma variável do tipo ponteiro com o tipo predefinido `FILE`.

Internamente este ponteiro irá apontar para uma estrutura de controle, onde serão armazenadas todas as informações para se acessar o arquivo, a posição em que se está trabalhando no arquivo etc.

Veja o exemplo:

```
FILE *arquivo_in;  
FILE *arquivo_out;
```

## 12.3 Função `fopen`

Sintaxe:

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

Para fazer qualquer operação de entrada ou saída de dados de um arquivo deve-se abrir o mesmo.



A função `fopen` irá abrir o arquivo com o nome fornecido no primeiro parâmetro. Neste nome pode-se indicar somente o nome, ou também indicar o caminho completo do arquivo com diretório e subdiretórios.

O segundo parâmetro do `fopen` indicará qual o tipo de acesso que será permitido fazer no arquivo. Este parâmetro é uma *string* informando a modalidade de acesso a ser realizada. Pode ser definido nesta string o seguinte:

Parâmetro	Significado
<code>r</code>	Abre o arquivo para leitura somente.
<code>w</code>	Abre o arquivo para gravação, criando o arquivo, caso não exista, ou limpando o conteúdo dele, caso ele já exista.
<code>a</code>	Abre o arquivo para gravação, mantendo o conteúdo do arquivo. O sistema se posiciona no final do arquivo.
<code>r+</code>	Abre o arquivo para atualização. Pode-se ler ou gravar no arquivo.
<code>w+</code>	Abre o arquivo para atualização, permitindo leitura e gravação. Caso o arquivo não exista será criado, caso exista será truncado.
<code>a+</code>	Abre o arquivo para atualização, permitindo leitura e gravação. Se o arquivo existir, será posicionado no final do mesmo, mantendo o conteúdo anterior.

Pode-se colocar um caractere `'b'` ao final da *string* do tipo, informando que será realizada a leitura binária do arquivo. Caso não seja colocado este `'b'`, o arquivo será considerado como um arquivo texto. Um arquivo texto é terminado quando se encontra um `<Ctrl> Z` no mesmo.

O resultado da função `fopen` será o ponteiro para o arquivo. Este valor deve ser colocado na variável definida para o arquivo. Se ocorrer erro na abertura do arquivo, a função retornará um valor nulo, que pode ser testado contra a constante `NULL`.

Veja o exemplo:

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

```
void main (void)
{
```

```
    FILE      *arquivo;
```

```
    printf("\nAbrindo o arquivo pessoa.dat");
```

```
    arquivo = fopen ("pessoa.dat", "r");
```

```
    if (arquivo == NULL)
```

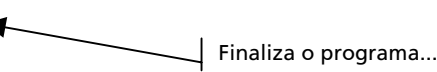
```
    {
```

```
        printf ("\nErro na abertura do arquivo");
```

Tenta abrir o arquivo para leitura no diretório corrente.

## 98 ♦ Programando em C para Linux, Unix e Windows

```
        exit (0);  
    }  
    else  
    {  
        printf("\nArquivo aberto para operações.. ");  
    }  
    /* Operacoes sobre o arquivo */  
}
```



Programa 12.1

### Resultado do Programa 12.1

#### 1ª Execução (arquivo não existe)

Abrindo o arquivo pessoa.dat  
Erro na abertura do arquivo

#### 2ª Execução (arquivo existe)

Abrindo o arquivo pessoa.dat  
Arquivo aberto para operações..

## 12.4 Função fclose

Sintaxe:

```
int fclose(FILE *arquivo)
```

Quando são feitas gravações em um arquivo, o sistema operacional, visando otimizar o tempo, não grava efetivamente os dados no disco, mantendo *buffers* na memória.

Para efetivar todas as alterações realizadas no arquivo, deve-se fechar o mesmo, indicando para o sistema operacional que realize todas as gravações pendentes.

A boa técnica de programação também indica que qualquer arquivo aberto, mesmo sendo para leitura, deve ser fechado ao final do processamento.

Veja o exemplo:

```
#include <stdio.h>  
#include <stdlib.h>  
  
void main (void)  
{  
    FILE      *arquivo;  
  
    printf("\nAbrindo o arquivo pessoa.dat");
```

```

arquivo = fopen ("pessoa.dat", "r");
if (arquivo == NULL)
{
    printf ("\nErro na abertura do arquivo");
    exit (0);
}
else
{
    printf("\nArquivo aberto para operações.. ");
}
/* Operacoes sobre o arquivo */
printf("\nFechando o arquivo... ");

fclose(arquivo);

```

### Programa 12.2

#### Resultado do Programa 12.2

```

Abrindo o arquivo pessoa.dat
Arquivo aberto para operações..
Fechando o arquivo...

```

## 12.5 Função fread

#### Sintaxe:

```
int fread( void *memoria, int tamanho, int quantidade, FILE *arquivo);
```

A função `fread` realiza a leitura de dados do arquivo e transfere os mesmos para o endereço de memória fornecido no parâmetro. A quantidade de *bytes* que serão lidos estará baseada no tamanho fornecido multiplicado pela quantidade indicada. A função irá retornar a quantidade de itens lidos efetivamente.

#### Veja o exemplo:

```

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

void main (void)
{
    FILE      *arquivo;
    int       vetor [100];
    int       qtd;
    int       i;

    arquivo = fopen ("pessoa.dat", "r");
    if (arquivo == NULL)

```

## 100 ♦ Programando em C para Linux, Unix e Windows

```
{
    printf ("Erro na abertura do arquivo\n");
    exit (0);
}

qtd = fread (vetor, sizeof (int), 100, arquivo);
printf ("Foram lidos %d itens\n", qtd);
for (i=0; i<qtd; i++)
    printf ("%d\n", vetor [i]);

fclose (arquivo);
}
```

Lendo valores inteiros. Cada inteiro tem 4 bytes, ou seja, esta chamada à função `fread` irá ler até 400 bytes.

O retorno contém a quantidade de itens e não bytes.

### Programa 12.3

#### Resultado do Programa 12.3

```
Foram lidos 3 itens
27
12
14
```

## 12.6 Função `fwrite`

### Sintaxe:

```
int fwrite( void *memoria, int tamanho, int quantidade, FILE
            *arquivo);
```

Para gravar informações no arquivo, deve-se utilizar a função `fwrite`. Deve ser informado para esta função o ponteiro do arquivo aberto para gravação, o endereço de memória de onde serão buscados os dados para gravação, a quantidade de itens a serem gravados e o tamanho de cada item.

Como resultado da função será retornada a quantidade de itens efetivamente gravados no arquivo.

### Veja o exemplo:

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

void main (void)
{
    FILE      *arquivo;
    int       vetor [100];
    int       qtd;
    int       i;
```

```

arquivo = fopen ("pessoa.dat", "w");
if (arquivo == NULL)
{
    printf ("Erro na abertura do arquivo\n");
    exit (0);
}

i = 0;
while (1)
{
    printf ("Entre com um valor :");
    scanf ("%d", &vetor [i]);
    if (vetor [i] == 0)
        break;
    i++;
}
qtd = fwrite (vetor, sizeof (int), i, arquivo);
printf ("Foram gravados %d itens\n", i);

fclose (arquivo);
}

```

Gravando os valores lidos no vetor.  
A quantidade de *bytes* efetivamente gravados pode ser obtido através de `sizeof(int) * i`.

#### Programa 12.4

#### Resultado do Programa 12.4

```

Entre com um valor :27
Entre com um valor :12
Entre com um valor :14
Entre com um valor :0
Foram gravados 3 itens

```

Valores 27, 12, 14 e 0 digitados.

## 12.7 Função fgets

#### Sintaxe:

```
char *fgets( char *string, int tam_max, FILE *arquivo);
```

A função `fgets` possui a mesma funcionalidade da função `gets`, exceto que a leitura é realizada do arquivo indicado e não do terminal.

A função irá retornar um valor nulo quando não for possível ler do arquivo.

#### Veja o exemplo:

```

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

void main (void)
{

```

## 102 ♦ Programando em C para Linux, Unix e Windows

```
FILE      *arquivo;
char       vetor [100][100];
char       *nome;
int        i, j;

arquivo = fopen ("texto.dat", "r");
if (arquivo == NULL)
{
    printf ("Erro na abertura do arquivo\n");
    exit (0);
}

i = 0;
while (1)
{
    nome = fgets (vetor [i], 100, arquivo);
    if (nome == NULL)
        break;
    i++;
}
printf ("Linhas lidas\n");
for (j=0; j < i; j++)
    printf ("Linha %2d - %s",j+1, vetor[j]);

fclose (arquivo);
}
```

← Vetor de *strings*.

← Cada linha do arquivo não poderá ter mais que 100 caracteres.

### Programa 12.5

#### Resultado do Programa 12.5

```
Linhas lidas
Linha  1 - INICIO DA MENSAGEM
Linha  2 - Este arquivo
Linha  3 - irá conter 5 linhas
Linha  4 - para demonstrar a utilidade da funcao fgets.
Linha  5 - FIM DA MENSAGEM
```

Arquivo texto.dat (pode ser criado com o bloco de notas do Windows ou vi no Linux).

```
INICIO DA MENSAGEM
Este arquivo
irá conter 5 linhas
para demonstrar a utilidade da funcao fgets.
FIM DA MENSAGEM
```

## 12.8 Função fseek

Sintaxe:

```
int fseek( FILE *arquivo, long int valor, int posição);
```

O sistema operacional, quando abre um arquivo para leitura e gravação, controla internamente a posição em que está no arquivo. Se for realizada uma leitura, será a partir deste ponto que os dados serão lidos.

No caso de gravação, serão gravados os dados a partir desta posição. Se o ponteiro do arquivo estiver posicionado no meio do arquivo, os dados gravados anteriormente serão perdidos (sobrescritos).

Pode-se controlar manualmente a posição do arquivo através da função `fseek`. Como parâmetros da função, além do ponteiro para o arquivo, deve-se informar um valor de deslocamento em *bytes* e a partir de qual posição deve ocorrer este deslocamento.

Pode-se indicar a posição relativa ao início do arquivo, utilizando a constante predefinida `SEEK_SET`. Pode-se indicar o deslocamento em relação ao final do arquivo utilizando-se a constante `SEEK_END`.

Finalmente, pode-se também realizar o deslocamento em relação à posição atual do arquivo, bastando para isto informar `SEEK_CUR` como parâmetro de posição.

Veja o exemplo:

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


```
void main (void)
```

```
{
    FILE      *arquivo;
    int        pos, valor;
```

```
    arquivo = fopen ("pessoa.dat", "r");
    if (arquivo == NULL)
    {
        printf ("Erro na abertura do arquivo\n");
        exit (0);
    }
```

```
    printf ("Qual posicao deseja ler? ");
    scanf ("%d", &pos);
```

Este arquivo pode ser gerado com o programa 12.4



## 104 ♦ Programando em C para Linux, Unix e Windows

```
pos--;
pos *= sizeof (int);
if (fseek (arquivo, pos, SEEK_SET) == -1)
    printf ("Erro no arquivo, verifique\n");
else
{
    fread (&valor, sizeof (int), 1, arquivo);
    printf ("Valor lido %d\n", valor);
}
fclose (arquivo);
}
```

Diminui uma posição para parar exatamente no início da posição desejada.

A quantidade em bytes é obtida por `sizeof(int) * pos` ou seja `4 * pos`.

Deslocando *n bytes* a partir do início do arquivo.

### Programa 12.6

#### Resultado do Programa 12.6

```
Qual posicao deseja ler? 2
Valor lido 12
```

Valor digitado.

Utilizado o arquivo criado anteriormente pelo programa 12.4

## 12.9 Função feof

Sintaxe:

```
int feof(FILE *arquivo);
```

A função `feof` indica quando o final do arquivo (*end of file*) foi atingido. Passa-se o ponteiro para o arquivo e será recebido como retorno o seguinte:

Zero	Não está posicionada no final do arquivo
Diferente de zero	Está no final do arquivo.

Veja o exemplo:

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

void main (void)
{
    FILE      *arquivo;
    int       pos, valor;

    arquivo = fopen ("pessoa.dat", "r");
    if (arquivo == NULL)
    {
```



```

    printf ("Erro na abertura do arquivo\n");
    exit (0);
}

printf ("Qual posicao deseja ler? ");
scanf ("%d", &pos);
pos--;
pos *= sizeof (int);
if (fseek (arquivo, pos, SEEK_SET) == -1)
    printf ("Erro no arquivo, verifique\n");
else
{
    fread (&valor, sizeof (int), 1, arquivo);
    if (feof (arquivo))
        printf ("Nao existe este registro\n");
    else
        printf ("Valor lido %d\n", valor);
}
fclose (arquivo);
}

```

Se a quantidade de bytes deslocando for maior que o tamanho do arquivo, a função `fseek` vai posicionar no último *byte* do arquivo.....

...a leitura não irá retornar nada...

...e o final do arquivo será atingido.

### Programa 12.7

#### Resultado do Programa 12.7

##### 1ª Execução

Qual posicao deseja ler? 3

Valor lido 14

Valor digitado.

Utilizado o arquivo criado anteriormente pelo programa 12.4

##### 2ª Execução

Qual posicao deseja ler? 99

Nao existe este registro

Valor digitado.

## 12.10 Função fprintf

#### Sintaxe:

```
int fprintf( FILE *arquivo, const char *formato, [argumentos,] ...);
```

Toda a funcionalidade do comando `printf` pode ser direcionada para um arquivo através desta função. A função `fprintf` possui a sintaxe idêntica ao `printf`, aplicando-se os mesmos formatos. O único parâmetro adicional é o ponteiro para o arquivo.

Veja o exemplo:

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

## 106 ♦ Programando em C para Linux, Unix e Windows

```
void main (void)
{
    FILE    *arq_log;
    int      i;

    arq_log = fopen ("log.dat", "a");

    if (arq_log == NULL)
    {
        printf ("Erro na abertura do arquivo\n");
        exit (0);
    }

    fprintf (arq_log, "Inicio do programa %s\n", __FILE__);
    fprintf (arq_log, "Versao de %s-%s\n", __DATE__, __TIME__);

    for (i=1; i < 10; i++)
        printf ("%d\n", i);

    fclose (arq_log);
}
```

Arquivo aberto para *append*.

A única diferença da função `printf` é o ponteiro para o arquivo.

### Programa 12.8

#### Resultado do Programa 12.8

```
1
2
3
4
5
6
7
8
9
```

Conteúdo do arquivo log.dat  
Inicio do programa p12\_8.c  
Versao de Mar 30 2005-16:10:35

## 12.11 Função `fscanf`

### Sintaxe:

```
int fscanf( FILE *arquivo, const char *formato, [endereços,] ...);
```

Da mesma maneira, pode-se realizar a leitura utilizando a mesma funcionalidade encontrada na função `scanf`, só que realizando a leitura de um arquivo previamente aberto para leitura.

Veja o exemplo:

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

void main (void)
{
    FILE    *arquivo;
    int      vlr1;
    int      soma;
    int      qtd;

    arquivo = fopen ("numeros.dat", "r");
    if (arquivo == NULL)
    {
        printf ("Erro na abertura do arquivo\n");
        exit (0);
    }

    soma = 0;
    qtd = 0;
    fscanf (arquivo, "%d", &vlr1);
    while (! feof (arquivo))
    {
        printf("%d,",vlr1);
        soma += vlr1;
        qtd++;
        fscanf (arquivo, "%d", &vlr1);
    }
    if (qtd != 0)
    {
        printf ("\nQuantidade de numeros lidos %d\n", qtd);
        printf ("Media dos numeros lidos %f\n",
                soma /(qtd * 1.0));
    }
    else
        printf ("Nao foi lido nenhum numero do arquivo\n");
}
```

Arquivo gerado como programa 12.9.2 listado a seguir...

A única diferença da função scanf é o ponteiro para arquivo.

Enquanto não for o final do arquivo...

### Programa 12.9.1

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

void main(void)
{
    FILE * fp;
    int i;
    int k;
```

## 108 ♦ Programando em C para Linux, Unix e Windows

```
int termo;

if( (fp = fopen("numeros.dat", "w"))==NULL)
{
    printf("\nErro ao abrir arquivo numeros.dat");
    exit(0);
}
i = 0;
k = 1;
termo = 0;
while( termo < 10 )
{
    fprintf( fp, "%d\n", i);
    fprintf( fp, "%d\n", k);
    i = k+i;
    k = k+i;
    termo+=2;
}
fclose(fp);
}
```

Posso abrir um arquivo, assinalar uma variável e já comparar o resultado para verificar se a função obteve sucesso.

Uma das muitas formas de se gerar a sequência de Fibonacci.

### Programa 12.9.2

Resultado do Programa 12.9.1

0,1,1,2,3,5,8,13,21,34, ← Sequência de Fibonacci...

Quantidade de numeros lidos 10

Media dos numeros lidos 8.800000

## 12.12 Função fflush

Sintaxe:

```
int fflush(FILE *stream);
```

Para otimizar as operações de saída de dados, o sistema operacional deixa os dados a serem gravados em um *buffer* na memória. Estes dados serão gravados efetivamente no arquivo quando o *buffer* estiver cheio ou o arquivo é fechado.

Às vezes é necessário que, logo após uma gravação, a mesma seja realmente efetivada em disco para que não haja perda. Para isto deve-se chamar a função `fflush` e indicar o ponteiro para o arquivo.

Veja o exemplo:

```
fwrite( buf, sizeof(data_type), 1, fp)
fflush(fp);
```

Garante que o *buffer* será descarregado do sistema operacional.

## 12.13 Função ftell

Sintaxe:

```
long ftell(FILE *stream);
```

A função `ftell` devolve o valor atual do indicador de posição de arquivo para o ponteiro do arquivo especificado. Para arquivos abertos em modo binário, o valor é o número de bytes cujo indicador está a partir do início do arquivo. Para arquivos abertos em modo texto, o valor de retorno pode não ser significativo, exceto como um argumento de `fseek`, devido às possíveis traduções de caracteres. Por exemplo, retornos de carro/alimentações de linha podem ser substituídos por novas linhas, o que altera o tamanho aparente do arquivo.

Veja o exemplo:

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

void main(void)
{
    FILE * fp;
    long i;
    if( (fp = fopen("texto.dat", "r")) == NULL )
    {
        printf("\nErro ao abrir o arquivo pessoa.dat");
        exit(0);
    }
    if((i=ftell(fp))==-1L)
    {
        printf("\nErro no arquivo");
    }
    printf("\nPosição atual do arquivo %d", i);

    if( fseek(fp, 0L, SEEK_END) != 0 )
    {
        printf("\nErro no arquivo");
    }
    if((i=ftell(fp))==-1L)
    {
        printf("\nErro no arquivo");
    }

    printf("\nO arquivo tem %d bytes\n", i);

    fclose(fp);
}
```

Posição inicial do arquivo. Como o retorno da função é um long, a comparação deve ser feita com um valor do tipo long (-1L).

Posicionando no final do arquivo...

...e pegando o tamanho atual do arquivo.

Programa 12.10

## 110 ♦ Programando em C para Linux, Unix e Windows

### Resultado do programa 12.10

Posição atual do arquivo 0  
O arquivo tem 113 bytes

Arquivo texto.dat (pode ser criado com o bloco de notas do Windows ou vi no Linux).

```
INICIO DA MENSAGEM
Este arquivo
irá conter 5 linhas
para demonstrar a utilidade da funcao fgets.
FIM DA MENSAGEM
```

## 12.14 Função `ferror` e `clearerr`

Sintaxe:

```
int ferror(FILE *stream);
void clearerr (FILE *stream);
```

A função `ferror` verifica a ocorrência de erros no arquivo especificado pelo ponteiro. Um valor de retorno zero indica que nenhum erro ocorreu, enquanto um valor diferente de zero significa um erro.

A função `clearerr` desliga tanto o indicativo de fim de arquivo como o indicativo de erro sobre o arquivo informado.

O indicador de erro associado ao arquivo permanece ativado até que o arquivo seja fechado ou `clearerr` seja chamado.

Veja o exemplo:

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

```
void main(void)
{
    FILE * fp;
    long i;
    if( (fp = fopen("pessoa.dat","r")) == NULL )
    {
        printf("\nErro ao abrir o arquivo pessoa.dat");
        exit(0);
    }

    fseek(fp,0,SEEK_END);
    if( ferror(fp) )
```

Este teste pode ser realizado sempre que houver alguma operação em um arquivo.

```

        printf("\nErro ao posicionar no arquivo");
        fclose(fp);
        exit(0);
    }

    if((i=ftell(fp))==-1L)
    {
        printf("\nErro no arquivo");
    }

    printf("\nO arquivo tem %d bytes", i);

    fclose(fp);
}

```

#### Programa 12.11

##### Resultado do programa 12.11

O arquivo tem 12 bytes

## 12.15 Streams Padrão

Sempre que um programa em C começa a execução, três *streams* são abertas automaticamente. Elas são a entrada padrão (*stdin* – *standart input*), a saída padrão (*stdout* – *standart output*) e a saída de erro padrão (*stderr* – *standart error*). Normalmente, essas *streams* referem-se ao console, mas podem ser redirecionadas pelo sistema operacional para algum outro dispositivo em ambientes que suportam redirecionamento de E/S. E/S redirecionadas são suportadas pelos Unix, Linux, DOS e Windows, por exemplo.

Como as *streams* padrão são ponteiros de arquivos, elas podem ser utilizadas pelo sistema para executar operações de E/S no console. Em geral, *stdin* é utilizada para ler do console e *stdout* e *stderr*, para escrever no console. *stdin*, *stdout* e *stderr* podem ser utilizadas como ponteiros de arquivo em qualquer função que usa uma variável do tipo `FILE*`. Por exemplo, você pode utilizar `fprintf` para escrever uma string no console usando uma chamada como esta:

```
fprintf( stdout, "Ola mundo!");
```

Tenha em mente que *stdin*, *stdout* e *stderr* não são variáveis no sentido normal e não podem receber nenhum valor usando `fopen`. Além disso, da mesma maneira que são criados automaticamente no início do seu programa, os ponteiros são fechados automaticamente no final.

## 112 ♦ Programando em C para Linux, Unix e Windows

Para redirecionar a entrada padrão e saída padrão:

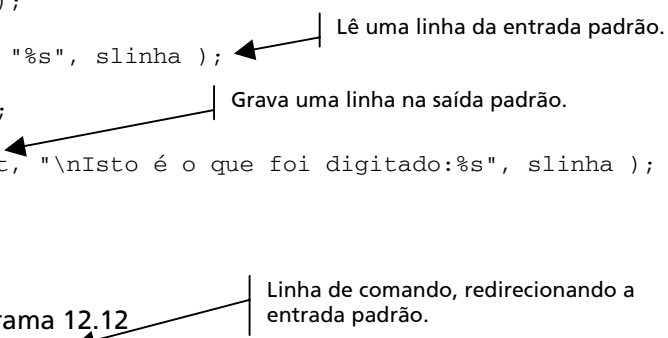
- o Entrada: nomeprograma < entrada.txt
- o Saída: nomeprograma > saída.txt
- o Entrada e saída: nomeprograma < entrada.txt > saída.txt

Veja o exemplo:

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

int main(void)
{
    char slinha[20];
    fprintf(stdout, "Este programa irá ler da entrada padrão e gravar
na saída padrão");

    fscanf(stdin, "%s", slinha );
    fflush(stdin);
    fprintf(stdout, "\nIsto é o que foi digitado:%s", slinha );
}
```



### Programa 12.12

#### Resultado do Programa 12.12

```
$> p12_12 < teste.txt
Este programa irá ler da entrada padrão e gravar na saída padrão
Isto é o que foi digitado:Teste
```

#### Conteúdo do arquivo teste.txt

```
Teste de mensagem
Teste de mensagem
Teste de mensagem
```

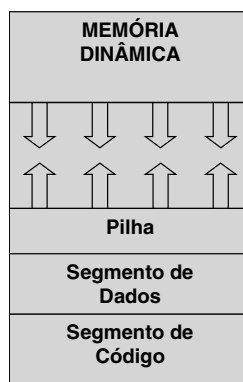


  
**13**

# Alocação de Memória

*640 K é mais do que suficiente para qualquer um.*  
Bill Gates, em 1981

## 13.1 Configuração da Memória



Quando um programa está em execução, ele ocupa um determinado espaço de memória. Tecnicamente, o programa fica dividido na memória em pedaços chamados segmentos.

Cada programa possui o segmento de código, segmento de dados, a pilha para controle das chamadas de funções e uma área chamada memória dinâmica. Conforme diagrama anterior, tanto a pilha como a memória dinâmica possu-

em tamanho variável e as duas crescem em sentido contrário, permitindo assim um melhor aproveitamento da memória.

Na memória dinâmica pode-se alocar espaço para utilização do programa. A vantagem deste tipo de abordagem é que só será utilizada a memória que realmente é necessária, podendo liberá-la após o uso.

## 13.2 Função malloc

Sintaxe:

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

É a função `malloc` que realiza a alocação de memória. Deve-se informar para a função a quantidade de *bytes* para alocação. A função irá retornar, se existir memória suficiente, um endereço que deve ser colocado em uma variável do tipo ponteiro.

Como a função retorna um ponteiro para o tipo `void`, deve-se utilizar o *type-cast*, transformando este endereço para o tipo de ponteiro desejado. Veja o exemplo:

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

```
void main (void)
{
```

```
    int    *valores, *aux;
    int     qtd,i;
    int     vlr;
```

```
    printf( "\nEntre com a quantidade de números:");
    scanf("%d", &qtd);
```

```
    if( qtd == 0)
        exit(0);
```

← Aloca memória necessária para os dados que serão digitados.

```
    valores = (int *) malloc (sizeof (int) * qtd);
```

```
    aux = valores;
```

← Guarda o 1º endereço (referente ao início) da posição da memória alocada.

```
    for (i=1; i <= qtd; i++)
    {
```

```
        printf("Entre com número %d ->", i);
```

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

```
        *aux = vlr;
```

← "Pula" para a próxima posição da memória.

```
        aux++;
```

```
    }
```

```

aux = valores;
for (i=1; i <= qtd; i++)
{
    printf ("%d\n", *aux);
    aux++;
}

free (valores);

```

Posiciona no início da memória alocada...

...e imprime o conteúdo da memória atual.

"Pula" para a próxima posição da memória.

Libera a memória utilizada.

### Programa 13.1

#### Resultado do Programa 13.1

```

Entre com a quantidade de números:5
Entre com número 1 ->3
Entre com número 2 ->6
Entre com número 3 ->9
Entre com número 4 ->12
Entre com número 5 ->15
3
6
9
12
15

```

Valores digitados.

## 13.3 Função free

Sintaxe:

```
void free (void *ponteiro);
```

Quando não se deseja mais uma área alocada, deve-se liberá-la através da função `free`. Deve ser passado para a função o endereço que se deseja liberar, que foi devolvido quando a alocação da memória ocorreu. Veja o código exemplo (igual ao código anterior):

```

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

void main (void)
{
    int    *valores, *aux;
    int    qtd,i;
    int    vlr;
    printf( "\nEntre com a quantidade de números:");
    scanf("%d", &qtd);

```

## 116 ♦ Programando em C para Linux, Unix e Windows

```
if( qtd == 0)
    exit(0);

valores = (int *) malloc (sizeof (int) * qtd);

aux = valores;
for (i=1; i <= qtd; i++){
    printf("Entre com número %d ->", i);
    scanf("%d", &vlr);
    *aux = vlr;
    aux++;
}

aux = valores;
for (i=1; i <= qtd; i++){
    printf ("%d\n", *aux);
    aux++;
}

free (valores);
```

Aloca memória necessária para os dados que serão digitados.

Guarda o 1º endereço (referente ao início) da posição da memória alocada.

"Pula" para a próxima posição da memória.

Posiciona no início da memória alocada...

...e imprime o conteúdo da memória atual.

"Pula" para a próxima posição da memória.

Libera a memória utilizada.

### Programa 13.2

#### Resultado do Programa 13.2

```
Entre com a quantidade de números:5
Entre com número 1 ->3
Entre com número 2 ->6
Entre com número 3 ->9
Entre com número 4 ->12
Entre com número 5 ->15
3
6
9
12
15
```

Valores digitados.

## 13.4 Função calloc

### Sintaxe:

```
void *calloc(int qtd,int tam);
```

Em vez de se alocar uma quantidade de *bytes* através da função `malloc`, pode-se usar a função `calloc` e especificar a quantidade de bloco de um determinado tamanho. Funcionalmente, a alocação irá ocorrer de maneira idêntica.

A única diferença entre o `malloc` e o `calloc` é que a última função, além de alocar o espaço, também inicializa o mesmo com zeros. Veja o exemplo:

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

```

#include <stdlib.h>

void main (void)
{
    int    *valores, *aux;
    int     qtd,i;
    int     vlr;

    printf( "\nEntre com a quantidade de números:");
    scanf("%d", &qtd);

    if( qtd == 0)
        exit(0);

    valores = (int *) calloc (qtd, sizeof (int));

    aux = valores;
    for (i=1; i <= qtd; i++)
    {
        printf("Entre com número %d ->", i);
        scanf("%d", &vlr);
        *aux = vlr;
        aux++;

        aux = valores;
        for (i=1; i <= qtd; i++)
        {
            printf ("%d\n", *aux);
            aux++;

            free (valores);
        }
    }
}

```

Aloca memória necessária para os dados que serão digitados. A multiplicação realizada na chamada malloc é feita internamente pela função calloc.

Guarda o 1º endereço (referente ao início) da posição da memória alocada.

"Pula" para a próxima posição da memória.

Posiciona no início da memória alocada...

...e imprime o conteúdo da memória atual.

"Pula" para a próxima posição da memória.

Libera a memória utilizada.

### Programa 13.3

#### Resultado do Programa 13.3

```

Entre com a quantidade de números:3
Entre com número 1 ->2
Entre com número 2 ->4
Entre com número 3 ->6
2
4
6

```

## 13.5 Função realloc

Sintaxe:

```
void *realloc(void *pt_alocado,int novo_tam);
```

## 118 ♦ Programando em C para Linux, Unix e Windows

Às vezes é necessário expandir uma área alocada. Para isto deve-se usar a função `realloc`. Deve-se passar para ela o ponteiro retornado pelo `malloc` e a indicação do novo tamanho. A realocação de memória pode resultar na troca de blocos na memória. Veja o exemplo:

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

int main(void)
{
    char *p;
    p = (char *)malloc(23);
    if(!p)
    {
        printf("\nErro de alocação de memória - abortando.");
        exit(1);
    }
    strcpy(p, "isso são 22 caracteres");
    printf("\n%s", p);
    p = (char *)realloc(p, 24);
    if(!p)
    {
        printf("\nErro de alocação de memória - abortando.");
        exit(1);
    }
    strcat(p, ".");
    printf("\n%s", p);
    free(p);
}
```

Alocando espaço para 23 bytes...

...usando 23 bytes (22 da frase mais o caractere \0)...

...realocando espaço para 24 bytes...

...utilizando o último byte alocado.

### Programa 13.4

Resultado do Programa 13.4

```
isso são 22 caracteres
isso são 22 caracteres.
```

Repare no " "(ponto) final.

## 13.6 Função memset

Sintaxe:

```
memset( variável, byte, quantidade )
```

A função `memset` copia o `byte` nos primeiros `quantidade` caracteres da matriz apontada por `variável`. A matriz será modificada na memória e terá o novo conteúdo.

O uso mais comum de `memset` é na inicialização de uma região de memória com algum valor conhecido.

Veja o exemplo:

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

int main(void)
{
    char p[20];
    int i;
    printf("\n[");
    for( i=0; i<20;i++)
    {
        printf("%c", p[i]);
    }
    printf("]\n[");

    strcpy( p, "teste");
    for( i=0; i<20;i++)
    {
        printf("%c", p[i]);
    }
    printf("]\n[");

    memset( p, 0, sizeof(p));
    for( i=0; i<20;i++)
    {
        printf("%c", p[i]);
    }
    printf("]\n[");

    strcpy( p, "teste");
    for( i=0; i<20;i++)
    {
        printf("%c", p[i]);
    }
    printf("]\n");
}
```

Toda vez que uma variável é declarada...

...ela já tem um "conteúdo". Normalmente, algum "lixo" que ficou na memória...

...e mesmo que venha a ser assinalado algum valor para a variável...

...o espaço de memória não utilizado continua com o "conteúdo"...

...então a melhor opção é preencher todo o espaço de memória com algum valor conhecido, normalmente 0...

...assim o "conteúdo" indesejado é descartado...

...evitando possíveis problemas na utilização da variável em qualquer ponto do programa.

Programa 13.5

Resultado do Programa 13.5

```
[ZZ· ZZ· ZZ· ZZ· ZZ· ]
[teste · ZZ· ZZ· ZZ· ]
[
[teste
]
```

Conteúdo indesejável.

Variável ainda com conteúdo indesejável.

Memória limpa e com um valor "controlado".

A variável com o conteúdo "correto".



*Não há nada novo sob o sol,  
mas há muitas coisas velhas que não conhecemos.*  
Ambrose Bierce, jornalista americano

## 14.1 Definição de Estruturas

Sintaxe:

```
struct sTnomeStruct
{
    tipo var1;
    tipo var2;
    ...
} [variavel];
```

É muito comum implementar uma entidade dentro de um sistema e para isto definir uma série de variáveis que represente a entidade. A linguagem C permite que se faça um agrupamento destas variáveis, criando o que é chamado de estrutura.

A vantagem de ter uma estrutura é que ela passa a ser um tipo definido, podendo definir de maneira simplificada uma ou mais variáveis. Cada variável desta estrutura é chamada de campo da estrutura.

Com a estrutura definida pode-se fazer atribuição de variáveis do mesmo tipo de maneira simplificada.



Veja o exemplo:

```
struct Funcionario
{
    char nome [40];
    char departamento[10];
    int dataNasc;
    float salario;
};
```

## 14.2 Utilização de Estruturas

Para fazer o acesso de um único campo, deve-se utilizar o nome da estrutura seguida de um ponto e do nome do campo desejado da estrutura. A partir daí são aplicadas as regras de uma variável discreta em C.

Veja o exemplo:

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

void main (void)
{
    char        linha [80];
    struct st_aluno
    {
        char nome [80];
        char turno;
        int media;
    } aluno;

    printf ("Entre com o nome..... : ");
    gets (linha);
    strcpy (aluno.nome, linha);

    printf ("Entre com o turno (M/T/N) : ");
    gets (linha);
    aluno.turno = linha [0];

    printf ("Entre com a media (0-100) : ");
    gets (linha);
    aluno.media = atoi (linha);
}
```

Acessando uma posição da estrutura. Lembrando que nome\_estrutura.nome\_campo.

Uma forma de se converter *string* para numérico.

Programa 14.1

## 122 ♦ Programando em C para Linux, Unix e Windows

### Resultado do Programa 14.1

```
Entre com o nome..... : Marcos Laureano
Entre com o turno (M/T/N) : M
Entre com a media (0-100) : 97
```

} Dados digitados.

## 14.3 Definindo mais Estruturas

Sintaxe:

```
struct NomeStruc Nova_var;
```

Ao se definir uma estrutura, também se está definindo um tipo a mais na linguagem. Portanto, é possível definir mais variáveis do mesmo tipo, bastando para isto colocar a palavra reservada `struct` seguida no nome da estrutura definida e o nome da nova variável.

Veja o exemplo:

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

void main (void)
{
    char    linha [80];
    struct st_aluno ← Criando uma estrutura chamada st_aluno...
    {
        char nome [80];
        char turno;
        int media;
    } aluno;
    struct st_aluno backup; ← ...e criando outra variável a partir da estrutura.

    printf ("Entre com o nome..... : ");
    gets (linha);
    strcpy (aluno.nome, linha);

    printf ("Entre com o turno (M/T/N) : ");
    gets (linha);
    aluno.turno = linha [0];

    printf ("Entre com a media (0-100) : ");
    gets (linha);
    aluno.media = atoi (linha);
    backup = aluno; ← Uma forma de copiar uma variável para outra.
    printf ("Nome : %s\n", backup.nome);
    printf ("Turno : %c\n", backup.turno);
}
```

```
printf ("Media : %d\n", backup.media);
}
```

### Programa 14.2

#### Resultado do Programa 14.2

```
Entre com o nome..... : Pacifico Pacato Cordeiro Manso
Entre com o turno (M/T/N) : N
Entre com a media (0-100) : 75
Nome : Pacifico Pacato Cordeiro Manso
Turno : N
Media : 75
```

} Dados digitados.

## 14.4 Estruturas e o typedef

Pode-se também utilizar o `typedef` com uma estrutura, gerando assim um sinônimo para a estrutura. Quando é usado o `typedef` na definição de uma estrutura, não é preciso mais utilizar a palavra `struct` para definir mais variáveis do mesmo tipo.

Veja o exemplo:

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

void main (void)
{
    char    linha [80];
    typedef struct st_aluno
    {
        char nome [80];
        char turno;
        int media;
    } ALUNO;

    ALUNO aluno, backup;

    FILE * fp;

    if( (fp = fopen("notas.dat", "w"))==NULL)
    {
        printf("\nErro ao abrir arquivo.");
        exit(0);
    }

    printf("\nEntre com os dados ou FIM para terminar");

    while(1)
    {
```

Criando a estrutura st\_aluno...

...criando um "apelido" ou "sinônimo" chamado ALUNO para a estrutura st\_aluno...

...e criando outras estruturas a partir do typedef definido.

## 124 ♦ Programando em C para Linux, Unix e Windows

```
printf ("\nEntre com o nome..... : ");
gets (linha);
strcpy (aluno.nome, linha);

if( strcmp(aluno.nome,"FIM") == 0)
    break;

printf ("Entre com o turno (M/T/N) : ");
gets (linha);
aluno.turno = linha [0];

printf ("Entre com a media (0-100) : ");
gets (linha);
aluno.media = atoi (linha);

backup = aluno;
printf ("Nome : %s\n", backup.nome);
printf ("Turno : %c\n", backup.turno);
printf ("Media : %d\n", backup.media);

fwrite(&backup, sizeof(ALUNO), 1,fp);
}
fclose(fp);
}
```

Gravando toda a estrutura.

### Programa 14.3

#### Resultado do Programa 14.3

```
Entre com os dados ou FIM para terminar
Entre com o nome..... : Marcos Laureano
Entre com o turno (M/T/N) : M
Entre com a media (0-100) : 98
Nome : Marcos Laureano
Turno : M
Media : 98

Entre com o nome..... : Jose Silva
Entre com o turno (M/T/N) : T
Entre com a media (0-100) : 76
Nome : Jose Silva
Turno : T
Media : 76

Entre com o nome..... : FIM
```

Dados digitados.

Dados digitados.

## 14.5 Estruturas Aninhadas

Pode-se aninhar uma definição de estrutura dentro de outra estrutura. O acesso a um item aninhado é o mesmo visto para a estrutura.

Veja o exemplo:

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

typedef struct st_func
{
    char nome [40];
    struct data;
    {
        unsigned short int dia;
        unsigned short int mês;
        unsigned short int ano;
    } data_admiss;
    float salário;
} FUNCIONARIO;

void main(void)
{
    FUNCIONARIO f;
    char linha[80];

    printf("\nEntre com o nome do funcionário:");
    gets(f.nome);

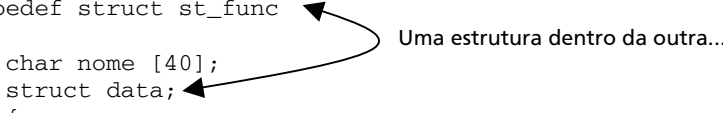
    printf("\nEntre com dia de admissao do funcionário:");
    gets(linha);
    f.data_admiss.dia = atoi(linha);

    printf("\nEntre com mes de admissao do funcionário:");
    gets(linha);
    f.data_admiss.mes = atoi(linha);

    printf("\nEntre com ano de admissao do funcionário:");
    gets(linha);
    f.data_admiss.ano = atoi(linha);

    printf("\nEntre com o salário:");
    scanf("%f", &f.salario );

    printf("\nNome = %s foi contratado em %02d/%02d/%02d com o salá-
rio de %.2f.",f.nome,f.data_admiss.dia, f.data_admiss.mes,
        f.data_admiss.ano,
        f.salario );
}
```



Uma estrutura dentro da outra...

**Programa 14.4**

**Resultado do Programa 14.4**

```

Entre com o nome do funcionário:Marcos Laureano
Entre com dia de admissao do funcionário:1
Entre com mes de admissao do funcionário:2
Entre com ano de admissao do funcionário:1999
Entre com o salário:2100
Nome = Marcos Laureano foi contratado em 01/02/1999 com o salário de
2100.00.

```

**14.6 Estruturas e Matrizes**

Vale lembrar que a estrutura é um tipo da linguagem C, portanto, pode-se construir um vetor de estruturas. Para se fazer acesso a um campo da estrutura deve-se indicar o índice do vetor seguido do ponto e o nome do campo da estrutura.

Veja o exemplo:

```

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

typedef struct st_aluno
{
    char nome [80];
    char turno;
    int media;
} ALUNO;

void main (void)
{
    FILE      *arquivo;
    ALUNO      aluno[50];
    int        i;
    arquivo = fopen ("notas.dat", "r");
    if (arquivo == NULL)
    {
        printf ("Erro na abertura do arquivo\n");
        exit (0);
    }

    i = 0;
    while (1)
    {
        if (fread (&aluno[i], sizeof (ALUNO), 1, arquivo) != 1)
            break;
        printf ("Nome   : %s\n", aluno[i].nome);
        printf ("Turno  : %c\n", aluno[i].turno);
        printf ("Media  : %d\n", aluno[i].media);
    }
}

```

Definindo um vetor de estruturas.

Arquivo criado com o programa 14.3.

Lendo, posição a posição, toda a estrutura (gravada anteriormente com o mesmo formato).

Acesso ao um vetor de estruturas.

```

        i++;
    }
    fclose (arquivo);
}

```

#### Programa 14.5

#### Resultado do Programa 14.5

```

Nome   : Marcos Laureano
Turno  : M
Media  : 98
Nome   : Jose Silva
Turno  : T
Media  : 76

```

## 14.7 Estruturas e Ponteiros

Pode-se também definir um ponteiro para uma estrutura. Para alocar a memória deve ser utilizada a função `malloc` e fornecido o tamanho da estrutura obtido através da função `sizeof`.

Veja o exemplo:

```

struct teste{
    int var_a;
    int var_b;
} *pVariavel;
pVariavel = (struct teste *) malloc (sizeof(struct teste));

```

## 14.8 Pointer Member

Quando é definido um ponteiro para uma estrutura, deve-se acessar os campos desta estrutura utilizando o “->”, que é o chamado *pointer member*, no lugar do ponto.

Veja o exemplo:

```

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

typedef struct st_aluno
{
    char nome [80];
    char turno;
    int media;
} ALUNO;

```

## 128 ♦ Programando em C para Linux, Unix e Windows

```
void main (void)
{
    FILE      *arquivo;
    ALUNO      *aluno;

    arquivo = fopen ("notas.dat", "r");
    if (arquivo == NULL)
    {
        printf ("Erro na abertura do arquivo\n");
        exit (0);
    }

    aluno = (ALUNO *) malloc (sizeof (ALUNO));

    while (1)
    {
        if (fread (aluno, sizeof (ALUNO), 1, arquivo) != 1)
            break;
        printf ("Nome   : %s\n", aluno->nome);
        printf ("Turno  : %c\n", aluno->turno);
        printf ("Media  : %d\n", aluno->media);
    }
    fclose (arquivo);
}
```

Definição do ponteiro.

Arquivo criado com o programa 14.3.

Alocação da memória para a estrutura.

Pelo fato de aluno ser um ponteiro, não é necessário utilizar o & para passar o endereço da memória para a função fread.

Acesso dos campos da estrutura via *pointer member*.

### Programa 14.6

#### Resultado do Programa 14.6

```
Nome   : Marcos Laureano
Turno   : M
Media   : 98
Nome   : Jose Silva
Turno   : T
Media   : 76
```





*Antes de os relógios existirem, todos tinham tempo.*

*Hoje, todos têm relógios.*

Eno Theodoro Wanke, poeta brasileiro

## 15.1 Função time

Sintaxe:

```
time_t time(time_t *variavel)
```

A função `time` retorna a quantidade de segundos decorridos no sistema desde 1 de janeiro de 1970 a 00:00:00 hora.

Apesar de, internamente, a quantidade de segundos ser representada em um `long`, deve-se sempre utilizar o tipo `time_t` visando dar portabilidade ao programa. O tipo `time_t` está definido no arquivo `time.h` que deve ser incluído no programa para o uso desta função.

A função pode retornar o valor no endereço colocado como parâmetro ou retornar como o resultado de sua execução. Caso se escolha a segunda opção pode-se passar um ponteiro nulo (`NULL`) para a mesma.

Veja o exemplo:

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

## 130 ♦ Programando em C para Linux, Unix e Windows

```
#ifndef WINDOWS
#include <unistd.h>
#endif

int main (void)
{
    time_t hora;

    hora = time(NULL);

    printf ("Numero de segundos antes : %d\n", hora);
    printf ("Dormindo 5 segundos\n");

#ifdef WINDOWS
    sleep(5*1000);
#else
    sleep(5);
#endif

    time(&hora);

    printf ("Numero de segundos depois : %d\n", hora);
}
```

Necessário no Unix/Linux para a função `sleep`.

Para armazenar a quantidade de segundos.

Pegando o número de segundos a partir do retorno da função.

No windows, a função `sleep` dorme em milissegundos

No Linux/Unix, a função `sleep` dorme em segundos

Pegando o número de segundos como parâmetro (ponteiro) para a função.

### Programa 15.1

#### Resultado do Programa 15.1

```
Numero de segundos antes : 1112194207
Dormindo 5 segundos
Numero de segundos depois : 1112194212
```

Quantidade de segundos desde 1 de janeiro de 1970 a 00:00:00 horas...

...e 5 segundos depois...

## 15.2 Trabalhando com datas

Pode-se dizer que existem três formatos de representar a data que podem ser usados como origem ou destino das funções de conversão de datas. São eles:

- o tipo `time_t` – Formato interno, representado por um `long`. Este formato é o resultado da chamada da função `time`.
- o *string* formatada – Pode-se obter a data em uma *string* formatada de acordo com uma máscara definida pelo usuário. Colocando-se a máscara em um arquivo, pode-se realizar a conversão ao contrário, sendo a *string* utilizada como entrada para a conversão.
- o `struct tm` – Também chamada de data expandida. É uma estrutura definida no `time.h` onde cada campo da data/hora está representado separadamente por um campo discreto. Os campos são os seguintes:

```
struct tm
{
    int tm_sec; /* seconds after the minute - [0,61]*/
```

```

int tm_min; /* minutes after the hour - [0,59] */
int tm_hour; /* hours - [0,23] */
int tm_mday; /* day of month - [1,31] */
int tm_mon; /* month of year - [0,11] */
int tm_year; /* years since 1900 */
int tm_wday; /* days since Sunday - [0,6] */
int tm_yday; /* days since January 1 - [0,365] */
int tm_isdst; /* daylight savings time flag */
};

```

## 15.3 Funções asctime e ctime

Sintaxe:

```

char * asctime( variável data da struct tm)
char * ctime(variável data da struct tm)

```

As funções `asctime` e `ctime` fazem a conversão da data no formato expandido.

Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main (void)
{
    time_t hora_sist;
    struct tm *hora_exp;

    hora_sist = time (NULL);
    hora_exp = localtime (&hora_sist);

    printf ("Usando ctime () : %s", ctime(&hora_sist));

    printf ("Usando asctime() : %s", asctime(hora_exp));
}

```

Para armazenar a quantidade de segundos

Para armazenar o formato expandido da data.

Obtendo a data e hora atual. A função `localtime` será detalhada a seguir.

Obtendo a data e hora por extenso a partir da quantidade de segundos.

Obtendo a data e hora por extenso a partir da estrutura `tm`.

Programa 15.2

Resultado do Programa 15.2

```

Usando ctime () : Wed Mar 30 16:20:06 2005
Usando asctime() : Wed Mar 30 16:20:06 2005

```

## 15.4 Funções `gmtime` e `localtime`

Sintaxe:

```
gmtime( variável do tipo time_t )
localtime( variável do tipo time_t )
```

As funções `gmtime` e `localtime` transformam a data do formato interno em segundos para o formato expandido na estrutura `tm`. A diferença entre elas está no tratamento dado ao *timezone*. A função `localtime` utiliza o valor da variável de ambiente `TZ` para a conversão do formato. A função `gmtime` converte a data UTC.

Um detalhe importante que se deve observar sobre o campo `tm_mon` da estrutura `tm` é seu intervalo de 0 a 11, ou seja, o mês está subtraído de um.

Para configurar *timezone* no Unix/Linux:

Ver documentação, pois a configuração é diferente em cada ambiente. Normalmente `export TZ="Brazil/East"` (para Linux e se você estiver no horário de Brasília).

Para configurar *timezone* no Windows:

```
set TZ=GMT-3
```

Veja o exemplo:

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

int main (int argc, char *argv[])
{
    time_t    hora_sist; /* Quantidade de segundos */
    struct tm *hora_exp; /* Data no formato expandido */

    hora_sist = time(NULL); ← Tenho que obter a hora em segundos para
                             depois passar nas funções time e gmtime.

    hora_exp = gmtime (&hora_sist); ← Obtendo a data expandida sem
                                     considerar o timezone.

    printf("Dados expandidos sem considerar timezone\n");
    printf("Dia      :tm_mday :%d\n", hora_exp->tm_mday);
    printf("Mes      :tm_mon  :%d\n", hora_exp->tm_mon);
    printf("Ano      :tm_year :%d\n", hora_exp->tm_year);
    printf("Hora     :tm_hour :%d\n", hora_exp->tm_hour);
    printf("Minuto   :tm_min  :%d\n", hora_exp->tm_min);
    printf("Segundo  :tm_sec  :%d\n", hora_exp->tm_sec);
    printf("Dia Semana :tm_wday :%d\n", hora_exp->tm_wday);
```

```

printf("Dt Juliana :tm_yday :%d\n", hora_exp->tm_yday);
printf("Hor. verao:tm_isdst:%d\n", hora_exp->tm_isdst);
printf("Expandida :%s", asctime(hora_exp));

hora_exp = localtime (&hora_sist);

printf("\nDados expandidos considerando timezone\n");
printf("Dia :tm_mday :%d\n", hora_exp->tm_mday);
printf("Mes :tm_mon :%d\n", hora_exp->tm_mon);
printf("Ano :tm_year :%d\n", hora_exp->tm_year);
printf("Hora :tm_hour :%d\n", hora_exp->tm_hour);
printf("Minuto :tm_min :%d\n", hora_exp->tm_min);
printf("Segundo :tm_sec :%d\n", hora_exp->tm_sec);
printf("Dia Semana :tm_wday :%d\n", hora_exp->tm_wday);
printf("Dt Juliana :tm_yday :%d\n", hora_exp->tm_yday);
printf("Hor. verao:tm_isdst:%d\n", hora_exp->tm_isdst);
printf("Expandida :%s", asctime(hora_exp));
}

```

Obtendo a data expandida considerando o *timezone*.

### Programa 15.3

#### Resultado do Programa 15.3

Dados expandidos sem considerar timezone

```

Dia :tm_mday :30
Mes :tm_mon :2
Ano :tm_year :105
Hora :tm_hour :19
Minuto :tm_min :27
Segundo :tm_sec :13
Dia Semana :tm_wday :3
Dt Juliana :tm_yday :88
Hor. verao:tm_isdst:0
Expandida :Wed Mar 30 19:27:13 2005

```

Dados expandidos considerando timezone

```

Dia :tm_mday :30
Mes :tm_mon :2
Ano :tm_year :105
Hora :tm_hour :16
Minuto :tm_min :27
Segundo :tm_sec :13
Dia Semana :tm_wday :3
Dt Juliana :tm_yday :88
Hor. verao:tm_isdst:0
Expandida :Wed Mar 30 16:27:13 2005

```

Na Inglaterra...

Diferença de fusos horários...

...e aqui.

## 15.5 Função mktime

Sintaxe:

```
mktime( struct tm *hora)
```

A função `mktime` realiza a conversão de uma data no formato expandido (estrutura `tm`) para um formato em segundos. Com o `mktime` é possível realizar operações com datas, para tal, basta no campo da estrutura `tm` somar ou diminuir os valores desejados e passar a estrutura alterada para a função `mktime`, que retorna o novo valor em segundos.

Veja o exemplo:

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

int main (int argc, char *argv[])
{
    time_t hora_sist;
    struct tm *hora_exp;
    time_t qtd_dias;

    hora_sist = time (NULL);
    printf ("Data atual      : %s\n", ctime (&hora_sist));

    hora_exp = localtime(&hora_sist);

    printf("Quantidade de dias no futuro/passado:");
    scanf ("%d", &qtd_dias);

    hora_exp->tm_mday += qtd_dias;

    hora_sist = mktime (hora_exp);

    printf ("\nData futura: %s\n", ctime (&hora_sist));
}
```

Quantidade de dias a ser acrescida ou subtraída da data atual...

... e cálculo da nova data.

### Programa 15.4

#### Resultado do Programa 15.4

##### 1ª Execução

```
Data atual      : Mon Apr  4 16:32:08 2005
Quantidade de dias no futuro/passado:5
Data futura: Sat Apr  9 16:32:08 2005
```

Valor digitado..

... data 5 dias depois.

**2ª Execução**

```
Data atual      : Mon Apr  4 16:33:56 2005
Quantidade de dias no futuro/passado:-30
Data futura: Sat Mar  5 16:33:56 2005
```

Valor digitado...  
... e data 30 dias antes.

## 15.6 Função `strftime`

**Sintaxe:**

```
strftime( string, tamanho, formato, data a ser convertida)
```

A função `strftime` converte uma data no formato expandido para um formato *string* que será colocado no parâmetro informado na função. Deve-se também informar o tamanho máximo da *string* para que a função não invada a memória caso o formato seja maior que o tamanho do parâmetro informado.

Deve-se informar em uma *string* o formato da data a ser gerada, usando-se a notação do formato parecida com a função `printf`, onde os campos são representados por seqüências “%<char>”. Qualquer outro caractere constante da *string* de formato será transcrito como informado para a *string* de saída. Também é aceita a notação especial de barra invertida (“\n” para *Line Feed*, por exemplo) no campo formato.

A função retorna o número de caracteres, excluindo o terminador “\0”, movimentados para a *string* de saída. A função irá retornar zero caso o formato exceda os tamanhos informados, ficando a *string* de saída com um conteúdo imprevisível.

As seqüências de diretivas aceitas pela função e definidas como ANSI C são as seguintes:

Formato	Descrição
%a	Nome do dia da semana abreviado
%A	Nome completo do dia da semana
%b	Nome do mês abreviado
%B	Nome completo do mês
%c	Data no formato do comando <code>date()</code>
%C	O número do século como um número decimal [00-99]
%d	Dia do mês com duas posições [01,31]
%D	Equivalente ao formato “%m/%d/%y”
%e	Dia do mês com duas posições, sendo que dias com um dígito são precedidos por um branco [1,31]
%h	Equivalente a %b
%H	Hora com dois dígitos [00,23]
%I	Hora no formato de 0 a 12 horas com dois dígitos [01,12]

Formato	Descrição
%j	Dia do ano. Chamado de data juliana [001,366]
%m	Número do mês com dois dígitos [01,12]
%M	Minutos com dois dígitos [00,59]
%n	Caractere Line Feed
%p	Indicativo de antes ou pós meio-dia [AM/PM]
%r	Notação POSIX. Equivalente à “%l:%M:%S %p”
%R	Hora no formato de 0 a 24 horas com minutos. Equivalente a “%H:%M”
%S	Número de segundos com dois dígitos [00,61]
%t	Caractere de tabulação
%T	A hora no formato hora/minuto/segundo. Equivalente a “%H:%M:%S”
%u	O dia da semana no formato numérico. 1 para Segunda-feira, 2 para Terça-feira, etc. [1,7]
%U	Número da semana no ano, considerando o domingo como primeiro dia da semana [00,53]. Todos os dias precedentes ao primeiro domingo do ano serão considerados como sendo a semana 0.
%V	Número da semana no ano, considerando a Segunda-feira como primeiro dia da semana [00,53]. Se a semana contendo o dia 1 de Janeiro tem quatro ou mais dias no novo ano, então ela é considerada semana 1, caso contrário será considerada semana 53 do ano anterior.
%w	O dia da semana no formato numérico. 0 para Domingo, 1 para Segunda-feira etc. [1,6]
%W	Número da semana no ano, considerando a Segunda-feira como primeiro dia da semana [00,53]. Todos os dias precedendo a primeira Segunda-feira do ano serão considerados como semana 0.
%x	Data no formato configurado por LANG
%X	Hora no formato configurado por LANG
%y	Número do ano sem o século, ou seja, com dois dígitos somente [00,99]
%Y	Número do ano com quatro dígitos
%Z	Nome do timezone configurado
%%	Caractere “%”

Veja o exemplo:

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

int main (int argc, char *argv[])
{
    time_t hora_sist;
    struct tm *hora_exp;
    char linha[80];

    hora_sist = time (NULL);
    hora_exp = localtime (&hora_sist);
```



← Tamanho máximo a ser utilizado para a mensagem.

```

    strftime(linha, 80, "Estamos no dia %j do ano.\n", hora_exp);
    printf (linha);

```

Formato de data desejado. A chamada da função strftime é parecido com a função printf.

```

    strftime(linha, 80, "A hora atual e %H:%M:%S\n", hora_exp);
    printf (linha);

    strftime(linha, 80, "O dia da semana e %A\n", hora_exp);
    printf (linha);

    strftime(linha, 80, "Estamos no mes de %B\n", hora_exp);
    printf (linha);
}

```

### Programa 15.5

#### Resultado do Programa 15.5

```

Estamos no dia 094 do ano.
A hora atual e 16:41:07
O dia da semana e Monday
Estamos no mes de April

```



# Tratamento de Erros

*O maior erro que você pode cometer na vida é ficar  
o tempo todo com o medo de cometer algum.*  
Elbert Hubbard, escritor americano

## 16.1 Variável `errno`

A maioria das funções devolve somente uma indicação de que houve erro em sua execução, seja através de um valor negativo, seja através de um ponteiro nulo.

O erro ocorrido na função é armazenado na variável `errno`, definida internamente no sistema e disponibilizada no programa através da colocação do arquivo `errno.h` na compilação do programa. Dentro deste arquivo *header* também são definidas as constantes mnemônicas dos possíveis erros que podem acontecer nas funções.

Em caso de se necessitar testar um erro específico, sugere-se sempre usar este mnemônico em lugar do número, pois assim o sistema ficará mais portátil e imune às mudanças futuras de versões de Sistema Operacional.

Um fator importante de ser citado é que as funções não zeram o valor da variável `errno` caso não ocorra erro nas funções. Isto obriga ao programador usar a variável `errno` somente depois de ter verificado se a função realmente retornou erro.

Veja o exemplo:

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

```
int main (void)
{
```

```
    FILE * fp;
```

```
    printf ("\nAbrindo um arquivo que nao existe\n");
    fp = fopen("arquivo_nao_existe", "r");
```

```
    if (fp==NULL)
    {
```

```
        printf ("Codigo de Erro      : %d\n", errno);
```

```
    }
```

```
    printf ("\nAbrindo um arquivo que existe\n");
    fp = fopen("pessoa.dat", "r");
```

```
    printf ("Codigo de Erro: %d\n", errno);
```

```
}
```

Programa 16.1

Utilizando a variável `errno` para mostrar o código do erro.

ATENÇÃO: para utilizar a variável `errno` é necessário que tenha acontecido algum erro, pois a variável continua com o valor do último erro ocorrido.

### Resultado do Programa 16.1

```
Abrindo um arquivo que nao existe
Codigo de Erro : 2
```

```
Abrindo um arquivo que existe
Codigo de Erro : 2
```

Como não ocorreu um erro ao abrir o arquivo, a variável `errno` continua com o valor anterior.

## 16.2 Função `strerror`

Sintaxe:

```
char * strerror(int error);
```

Caso queira mostrar a mensagem correspondente ao erro ocorrido, ou queira gerar um erro dentro do programa que utilize a mesma mensagem padrão do sistema operacional, deve-se usar a função `strerror`.

Esta função mapeia o número do erro passado como parâmetro e retorna um ponteiro para a mensagem de erro correspondente. A mensagem de erro retornada não possui uma quebra de linha ao seu final.

Veja o exemplo:

## 140 ♦ Programando em C para Linux, Unix e Windows

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

int main (void)
{
    FILE * fp;

    printf ("\nAbrindo um arquivo que nao existe\n");
    fp = fopen("arquivo_nao_existe", "r");

    if (fp==NULL)
    {
        printf ("Codigo de Erro: %d\n", errno);
        printf ("Erro: %s\n", strerror (errno));
    }

    printf ("\nAbrindo um arquivo que existe\n");
    fp = fopen("pessoa.dat", "r");

    printf ("Codigo de Erro: %d\n", errno);
    printf ("Erro: %s\n", strerror (errno));
}
```

Retorno da mensagem de erro a partir do código do erro contido na variável `errno`.

ATENÇÃO: para utilizar a variável `errno` é necessário que tenha acontecido algum erro, pois a variável continua com o valor do último erro ocorrido.

### Programa 16.2

#### Resultado do Programa 16.2

```
Abrindo um arquivo que nao existe
Codigo de Erro: 2
Erro: No such file or directory
```

```
Abrindo um arquivo que existe
Codigo de Erro: 2
Erro: No such file or directory
```

Como não ocorreu um erro ao abrir o arquivo, a variável `errno` continua com o valor anterior.

## 16.3 Função perror

Sintaxe:

```
perror( mensagem );
```

Como a maioria dos erros ocorridos deve ser mostrada de maneira idêntica na saída de erro padrão, e baseado principalmente no valor da variável `errno`, pode-se usar a função `perror` que realiza todas estas tarefas automaticamente.

A mensagem é mostrada na saída de *standart error*. Inicialmente será mostrada a *string* passada como parâmetro, seguida de dois pontos e um caractere em branco. A mensagem correspondente será mostrada de acordo com o valor da variável `errno`. Por último, será feita uma quebra de linha.

Veja o exemplo:

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

int main (int argc, char *argv[])
{
    FILE * fp;

    printf ("\nAbrindo um arquivo que nao existe\n");
    fp = fopen("arquivo_nao_existe", "r");

    if (fp==NULL)
        perror(argv[0]);
}
```

### Programa 16.3

#### Resultado do Programa 16.3

Abrindo um arquivo que nao existe  
p16\_3: No such file or directory

Nome do programa. Contido em argv[0].

Mensagem de erro (igual ao retorno da função strerror).



# 17

## Definições Avançadas

*Calma. São apenas zeros e uns.*  
(Anônimo)

### 17.1 Definição de Uniões

Sintaxe:

```
union nomeUniao
{
    tipo var1;
    tipo var2;
    ...
} [variavel];
```

O conceito de união significa uma série de variáveis com o mesmo endereço de memória, ou seja, cada elemento de uma união compartilha a sua memória com os demais.

Como todos os campos da união começam no mesmo endereço, o tamanho da variável será determinado pelo tamanho do maior campo.

### 17.2 Utilização de Uniões

Para se usar uma união deve-se seguir a mesma sintaxe e regras das estruturas. A definição de uma união também pode ser usada em outra variável pois na sua definição está sendo criado um tipo.

Para acessar um determinado campo de uma união é usada a mesma sintaxe da estrutura, colocando o nome da variável seguida de um ponto e o nome do campo que se quer acessar.

Veja o exemplo:

```
#include <stdio.h>
typedef union teste
{
    unsigned short int    valor;
    char                 caractere[2];
} TESTE;

void main (void)
{
    TESTE    uniao;

    printf ("Tamanho da uniao  : %d\n", sizeof (TESTE));

    uniao.caractere [0] = '\1'; ← 00000001
    uniao.caractere [1] = '\0'; ← 00000000

    printf ("Valor  : %d\n", uniao.valor);
}
```

Programa 17.1

#### Resultado do Programa 17.1

```
Tamanho da uniao  : 2
Valor  : 256 ← 0000000100000000
```

## 17.3 Lista Enumerada

Sintaxe:

```
enum nome {
    const1, const2, const3, ...
} [Variavel];
```

Às vezes, para se simplificar a leitura de um programa, pode-se utilizar no lugar de constantes, variáveis predefinidas no pré-compilador. Uma maneira de se fazer isto utilizando o compilador é definindo-se uma lista enumerada.

A lista enumerada nada mais é que uma lista em que o compilador, para cada constante colocada, irá atribuir um valor interno.

Toda vez que o compilador encontrar esta constante ele a substituirá pelo valor interno.

Veja o exemplo:

```
#include <stdio.h>

typedef enum cores {azul, vermelho, amarelo, verde} CORES;

void main (void)
{
    CORES carro;
    carro = amarelo;
    printf ("A cor do carro eh %d\n", carro);
}
```

Será associada uma sequência numérica começando em 0.

Como amarelo é o 3º termo da sequência numérica, a variável `carro` irá receber o número 2.

Programa 17.2

Resultado do Programa 17.2

A cor do carro eh 2

Representa o amarelo.

## 17.4 Estruturas de Bits

Pode-se definir uma estrutura e indicar para cada campo da mesma a quantidade de *bits* que o campo deve usar. Isto é muito usado quando se têm armazenadas situações binárias e é preciso otimizar o uso de memória.

Veja o exemplo:

```
#include <stdio.h>

typedef struct byte
{
    int meio_byte_high : 4;
    int meio_byte_low : 4;
} BYTE;

typedef union letra
{
    char caractere;
    BYTE byte;
} LETRA;

void main (void)
{
    LETRA valor;
```

São utilizados 4 bits para cada campo da estrutura.



```

    valor.byte.meio_byte_high = 0x03;
    valor.byte.meio_byte_low = 0x01;
    printf ("Caractere '%c'\n", valor.caractere);
}

```

← 0011 em binário.  
← 0001 em binário.

### Programa 17.3

Resultado do Programa 17.3  
Caractere '1' ← 00110001 em binário ou 49 em decimal.

## 17.5 Operadores Bit a Bit

Além de definir campos com um número determinado de *bits* é possível realizar operações com os *bits* de variáveis. São possíveis as seguintes operações binárias:

Operador	Operação
&	E
	OU
^	OU Exclusivo
~	Negação

Convém lembrar que estas operações não usam o valor, não consideram o tipo e nem o sinal, somente trabalhando com os *bits* dos operandos.

Veja o exemplo:

```

#include <stdio.h>

void main (void)
{
    unsigned char vlr1;
    unsigned char vlr2;

    vlr1 = 0x13;
    vlr2 = 0x4f;

    printf ("Valor 1 '%X'\n", vlr1);
    printf ("Valor 2 '%X'\n", vlr2);
    printf ("AND logico '%2.2X'\n", vlr1 & vlr2);
    printf ("OR logico '%2.2X'\n", vlr1 | vlr2);
    printf ("XOR logico '%2.2X'\n", vlr1 ^ vlr2);
    printf ("NOT logico '%2.2X'\n", ~vlr1);
}

```

← 00010011  
← 01001111

### Programa 17.4

**Resultado do Programa 17.4**

```

Valor 1 '13'
Valor 2 '4F'
AND logico '03'
OR logico '5F'
XOR logico '5C'
NOT logico 'FFFFFFEC'

```

**17.6 Deslocamento de Bits**

Também é possível realizar o deslocamento de *bits* de uma variável tanto para a direita como para a esquerda.

Operador	Operação
<<	Deslocamento à esquerda
>>	Deslocamento à direita

Nestas operações deve ser informada a quantidade de posições que o valor do primeiro operador será deslocado.

Veja o exemplo:

```

#include <stdio.h>

void main (void)
{
    unsigned long int vlr1;
    int i;

    vlr1 = 4000;

    printf ("Valor '%d'\n", vlr1);
    printf ("Deslocando para a direita\n");
    for (i=1; i <= 4; i++)
        printf ("Deslocando %d posicoes - Resultado %d\n",
            i, vlr1 >> i);
    printf ("Deslocando para a esquerda\n");
    for (i=1; i <= 4; i++)
        printf("Deslocando %d posicoes - Resultado %d\n",i,
            vlr1 << i);
}

```

**Programa 17.5****Resultado do Programa 17.5**

```

Valor '4000'
Deslocando para a direita
Deslocando 1 posicoes - Resultado 2000

```

```
Deslocando 2 posicoes - Resultado 1000
Deslocando 3 posicoes - Resultado 500
Deslocando 4 posicoes - Resultado 250
Deslocando para a esquerda
Deslocando 1 posicoes - Resultado 8000
Deslocando 2 posicoes - Resultado 16000
Deslocando 3 posicoes - Resultado 32000
Deslocando 4 posicoes - Resultado 64000
```

## 17.7 Deslocamento de Bits Circular

A linguagem C não implementa o deslocamento de *bits* de forma circular, ou seja, os *bits* deslocados são perdidos. Mas implementar o deslocamento circular em C é possível utilizando a combinação de alguns operadores. O deslocamento circular é muito utilizado em algoritmos de criptografia.

Veja o exemplo de deslocamento circular de variáveis do tipo `char`.

```
#include <stdio.h>

#define shift_esquerda(c,bits) (((c) << (bits)) | ((c) >> (8-(bits))))
#define shift_direita(c,bits) (((c) >> (bits)) | ((c) << (8-(bits))))

void main(void)
{
    char i = 255;
    char r;
    printf("\nOriginal = %d", i );
    r = shift_esquerda(i,4);
    printf("\n%d", r);
    r = shift_direita(r,3);
    printf("\n%d", r);

    i = 129;
    printf("\nOriginal = %d", i );
    r = shift_esquerda(i,4);
    printf("\n%d", r);
    r = shift_direita(r,4);
    printf("\n%d", r);
}
```

### Programa 17.6

#### Resultado do Programa 17.6

```
Original = 255
255
255
Original = 129
24
129
```



# Manipulação de Arquivos (padrão Linux e Unix)

*Uma coisa é sempre totalmente diferente da outra,  
a não ser quando as duas de assemelham.*  
Jô Soares, humorista brasileiro

## 18.1 O Sistema de Arquivo Tipo Unix

Como a linguagem C foi originalmente desenvolvida sobre o sistema operacional Unix, ela inclui um segundo sistema de E/S com arquivos em disco que reflete basicamente as rotinas de arquivo em disco de baixo nível do Unix. O sistema de arquivo tipo Unix usa funções que são separadas das funções do sistema de arquivo com `buffer`.

Estas funções são parte integrante do sistema operacional e são conhecidas como sendo funções de nível 2 devido a serem definidas na sessão 2 do manual do Unix ou como sistema de arquivos sem `buffer`. As funções de E/S são listadas a seguir:

Nome	Função
<code>read()</code>	Lê um buffer de dados
<code>write()</code>	Escreve um buffer de dados
<code>open()</code>	Abre um arquivo em disco
<code>close()</code>	Fecha um arquivo em disco
<code>lseek()</code>	Move ao byte especificado em um arquivo
<code>unlink()</code>	Remove um arquivo do diretório
<code>remove()</code>	Remove um arquivo do diretório
<code>rename()</code>	Renomeia um arquivo no diretório.

Existe um conjunto de operações de E/S de mais alto nível, chamadas funções de nível 3, definidas como parte integrante da biblioteca padrão do ANSI C (vistas anteriormente no capítulo 12). Estas funções de alto nível fazem uso das funções de nível 2 também.

Com as funções de nível 2 é possível se implementar qualquer tipo de acesso ou método de acesso a arquivos.

Ao contrário do sistema de E/S de alto nível (nível 3), o sistema de baixo nível (nível 2) não utiliza ponteiros de arquivo do tipo `FILE`, mas descritores de arquivo do tipo `int`. Na função de abertura de arquivo, o sistema operacional devolve para o programa o descritor de arquivo que ele atribuiu ao arquivo. Todas as outras funções devem receber este descritor para identificar sobre qual arquivo estamos querendo realizar a operação.

## 18.2 Descritores Pré-alocados

Desde os primeiros sistemas Unix, por convenção, o interpretador de comandos (*shell*) sempre que cria um processo, abre três arquivos e passa os descritores dos mesmos para o processo. Estes descritores são utilizados por todas as funções de nível 2 e 3 do sistema.

O descritor de número 0 (zero) representa a `Entrada Padrão` (nas funções de nível 3, `stdin`). Todas as funções de entrada de dados que não especificam um descritor de arquivos irão ler os seus dados deste arquivo.

O descritor número 1 representa a `Saída Padrão` (nas funções de nível 3, `stdout`). Todas as funções que fazem a saída de dados e não especificam um descritor irão utilizar este arquivo como saída de dados.

Por último, o descritor 2 é reconhecido pelo sistema como sendo a `Saída de Erro Padrão` do sistema (nas funções de nível 3, `stderr`). Todas as funções que emitem mensagens de erro irão utilizar este descritor com saída das informações.

## 18.3 Função open

Sintaxe:

```
int open(const char *path,
         int oflag,
         /*[mode_t mode]*/);
```

Para abrir um arquivo deve-se chamar a função `open`. A função recebe o nome do arquivo como parâmetro juntamente com o *flag* indicando o modo de abertura e *flags* informando opções adicionais de abertura e/ou tratamento de arquivo. Pode-se colocar todo o caminho do arquivo juntamente com o seu nome.

A função irá retornar um número inteiro positivo em caso de sucesso no processo de abertura. Este número é o descritor do arquivo e deve ser armazenado e passado para as demais funções a serem realizadas no arquivo.

O sistema Unix/Linux garante que o número retornado é o menor número de descritor ainda não utilizado pelo sistema. Isto pode ser utilizado dentro de um programa de maneira a abrir um outro arquivo para a Entrada Padrão, Saída Padrão ou Erro padrão. Por exemplo, ao se fechar a Entrada Padrão, descritor 1, disponibilizando-se o número 1 como sendo o menor descritor. Ao abrir um novo arquivo, o número será usado para este arquivo. Como as funções do sistema sempre gravam as suas informações na Saída Padrão, estarão gravando no arquivo e não mais na tela.

A função no processo de abertura irá verificar se o arquivo existe, se o usuário tem permissão sobre o diretório onde o arquivo se encontra e também se tem permissão para realizar a operação indicada no parâmetro (leitura/gravação).

Caso ocorra algum erro na abertura, a função irá retornar o valor `-1` como resultado. Na variável `errno` estará indicado o erro ocorrido na abertura.

O segundo parâmetro da função `open` irá indicar qual é o modo de abertura sendo realizado no arquivo. Deve-se obrigatoriamente indicar um e somente um dos seguintes *flags*:

`O_RDONLY` – O arquivo está sendo aberto para leitura. Caso não se coloque nenhum flag adicional e caso o arquivo não exista, a função de abertura retornará um erro.

`O_WRONLY` – O arquivo está sendo aberto para gravação. Caso o arquivo não exista e não se coloque nenhum flag adicional, a função retornará um erro.

`O_RDWR` – O arquivo estará sendo aberto para leitura e gravação. Também neste caso, se o arquivo não existir e nenhum flag adicional foi colocado, a função retorna erro.

Podem-se colocar alguns *flags* adicionais com um dos *flags* antes definidos. Estes *flags* irão determinar as ações a serem realizadas no processo de abertura do arquivo e devem ser colocadas com `OR` binário no segundo parâmetro.

Os flags possíveis são:

- o `O_APPEND` – Cada gravação a ser realizada no arquivo será feita no final do mesmo.
- o `O_CREAT` – Caso o arquivo não exista, ele será criado no diretório. Esta opção exige que se coloque no terceiro parâmetro o modo de proteção a ser usado na criação do arquivo.
- o `O_EXCL` – Este flag deve ser usado em conjunto com o flag `O_CREAT`. Ele indica para a função `open` retornar um erro caso o arquivo já exista.
- o `O_TRUNC` – Se o arquivo existir no diretório, o conteúdo do mesmo será eliminado, deixando o arquivo com o tamanho de zero *bytes*.
- o `O_SYNC` – Indica para o Unix que cada gravação espere que a gravação física dos dados seja efetuada. Isto garante que as informações gravadas pelo processo sejam efetivamente gravadas em disco, evitando perda de informações caso o processo seja cancelado. Em contrapartida, o desempenho do processo será seriamente prejudicado.

O terceiro parâmetro deve ser fornecido caso na abertura se tenha especificado o *flag* `O_CREAT`. Este terceiro parâmetro indica a máscara de proteção a ser usada na criação do arquivo. Pode-se especificar mais de um *flag* neste campo, bastando fazer o binário entre os seguintes valores:

```
S_IRUSR – Leitura para usuário
S_IWUSR – Gravação para usuário
S_IXUSR – Execução para usuário

S_IRGRP – Leitura para grupo
S_IWGRP – Gravação para grupo
S_IXGRP – Execução para grupo

S_IROTH – Leitura para outros
S_IWOTH – Gravação para outros
S_IXOTH – Execução para outros
```

Pode-se também indicar a proteção a ser usada no arquivo, usando-se o modo numérico octal de permissão aceito pelo sistema Unix (veja no *help* do sistema o comando `chmod`). As permissões de um arquivo sofrem a influência da configuração atual do sistema. Veja no *help* do sistema o funcionamento do comando `umask`.

Veja o exemplo:

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

## 152 ♦ Programando em C para Linux, Unix e Windows

```
#include <errno.h>
#include <fcntl.h>
#include <string.h>

void main (int argc, char *argv[])
{
    int fd; ← | Descritor do arquivo a ser aberto

    printf("Tentando abrir o arquivo 'file1' para leitura\n");

    fd = open ("file1", O_RDONLY); ← | Abrindo um arquivo em modo leitura.
                                   | Este arquivo não existe no diretório
                                   | local, ocasionando erro na abertura.
    if (fd < 0)
        fprintf (stderr, "Erro : %s\n", strerror(errno));
    else
        printf ("Arquivo aberto\n");

    close (fd); ← | Fechando o arquivo....

    printf("Tentando abrir o arquivo 'file2' para gravação\n");

    fd = open("file2", O_WRONLY); ← | Abrindo um arquivo em modo gravação.
                                   | Este arquivo não existe no diretório local,
                                   | ocasionando erro na abertura.
    if (fd < 0)
        fprintf(stderr, "Erro : %s\n", strerror(errno));
    else
        printf("Arquivo aberto\n");

    close(fd);

    printf("Tentando abrir o arquivo 'file3' para leitura\n");


    fd = open("file3", O_RDONLY | O_CREAT, 0744); ← | Abrindo um arquivo para leitura que não existe. Como foi
                                                    | utilizado o flag O_CREAT, se o arquivo não existir, ele será
                                                    | criado com a permissão 0744 (leitura, gravação e execução
                                                    | para usuário e leitura para grupo e outros).

    if (fd < 0)
        fprintf(stderr, "Erro : %s\n", strerror(errno));
    else
        printf("Arquivo aberto\n");

    close (fd);

    printf("Tentando abrir o arquivo 'file4' para gravacao\n");
```





Abrindo um arquivo para gravação. Este arquivo não existe no diretório local, ocasionando erro na abertura. Se o arquivo existisse, como foi especificado no *flag* `O_TRUNC`, o arquivo seria truncado (zerado).

```

fd = open("file4", O_WRONLY | O_TRUNC);

if (fd < 0)
    fprintf (stderr, "Erro : %s\n", strerror(errno));
else
    printf ("Arquivo aberto\n");

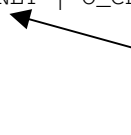
close (fd);

printf("Tentando abrir o arquivo 'file5' para gravacao\n");

fd = open ("file5", O_WRONLY | O_CREAT | O_EXCL, 0755);

if (fd < 0)
    fprintf (stderr, "Erro : %s\n", strerror(errno));
else
    printf ("Arquivo aberto\n");

close (fd);
}
    
```



Abrindo um arquivo para gravação com os *flags* `O_CREAT` e `O_EXCL`, ou seja, o arquivo somente será criado se já não existir no sistema de arquivos.

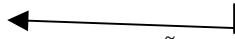
### Programa 18.1

#### Resultado do Programa 18.1

##### 1ª Execução

```

Tentando abrir o arquivo 'file1' para leitura
Erro : No such file or directory
Tentando abrir o arquivo 'file2' para gravação
Erro : No such file or directory
Tentando abrir o arquivo 'file3' para leitura
Arquivo aberto
Tentando abrir o arquivo 'file4' para gravacao
Erro : No such file or directory
Tentando abrir o arquivo 'file5' para gravacao
Arquivo aberto
    
```



Arquivo não existe...

##### 2ª Execução

```

Tentando abrir o arquivo 'file1' para leitura
Erro : No such file or directory
Tentando abrir o arquivo 'file2' para gravação
Erro : No such file or directory
    
```

## 154 ♦ Programando em C para Linux, Unix e Windows

```
Tentando abrir o arquivo 'file3' para leitura
Arquivo aberto
Tentando abrir o arquivo 'file4' para gravacao
Erro : No such file or directory
Tentando abrir o arquivo 'file5' para gravacao
Erro : File exists
```

O arquivo já existia...

### Configuração do ambiente

```
$> umask
0027
```

Verificando a máscara padrão para criação de arquivos no sistema. O resultado 0027 indica permissão total para usuário (leitura, gravação e execução), leitura e execução para grupo e nenhuma permissão para outros.

### Arquivos criados pelo programa (obtidos através do comando `ls -l`):

```
-rwxr----- 1 laureano prof 0 May 13 09:18 file3
-rwxr-x--- 1 laureano prof 0 May 13 09:18 file5
```

Arquivos criados pelo programa. Repare que nos arquivos as permissões ficarão diferentes do que foi especificado pelo programa. As permissões especificadas no programa somente serão utilizadas se a configuração do sistema (`umask`) permitir o uso das permissões.

## 18.4 Função `creat`

Sintaxe:

```
int creat(const char *path, mode_t mode);
```

Quando se quer abrir um arquivo e criar o mesmo caso não exista, ou truncar o mesmo caso já exista, pode-se usar a função `creat`.

O arquivo será aberto somente para gravação pois esta função é equivalente à chamada da função `open` com os parâmetros seguintes.

```
open (path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

A função `creat` exige que se coloque o parâmetro de permissão conforme já definido na função `open`.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>
```

```

void main (int argc, char *argv[])
{
    int      fd;

    if( argc < 2 )
    {
        fprintf(stderr, "Obrigatório informar o nome do arquivo\n");
        exit(1);
    }

    printf("Criando o arquivo %s com a funcao 'creat()'\n", argv[1]);

    fd = creat (argv[1], 0755); ← O código da permissão (0775) é
                                obrigatório informar na função create.

    if (fd < 0)
    {
        fprintf (stderr, "Erro : %s\n", strerror(errno));
        exit(errno); ← Informando o código do erro para o
                      sistema operacional.
    }
    printf ("Arquivo criado\n");

    close (fd);

    exit (0);
}

```

## Programa 18.2

### Resultado do Programa 18.2

**1ª Execução** ← Linha de comando.  
 \$> p18\_2  
 Obrigatório informar o nome do arquivo

**2ª Execução**  
 \$> p18\_2 file6  
 Criando o arquivo file6 com a funcao 'creat()'  
 Arquivo criado

Arquivos criados pelo programa (obtidos através do comando `ls -l`):  
 -rwxr-x--- 1 laureano prof 0 May 13 10:01 file6

## 18.5 Função close

Sintaxe:

```
int close (int fildes);
```

Quando um arquivo é fechado, todas as operações de saída pendentes em memória são gravadas no disco e as estruturas de controle interno são liberadas pelo sistema operacional.

Quando um processo termina normalmente com a chamada à função `exit()` ou executa-se o último comando da rotina `main` do programa, todos os arquivos abertos pelo mesmo são fechados automaticamente pelo sistema operacional.

A utilização da função `close` pode ser observada nos exemplos anteriores (programas 18.1 e 18.2)

## 18.6 Função read

Sintaxe:

```
ssize_t read (int fildes, void *buf, size_t nbyte);
```

A função `read` realiza a leitura de dados do arquivo para a memória. Deve-se informar de qual descritor devem ser lidos os *bytes*. O descritor deve estar aberto com opção `O_RDONLY` ou `O_RDWR`. Os dados serão lidos a partir da posição corrente.

Deve-se informar o endereço onde a informação lida será armazenada. Também se informa para a função a quantidade de *bytes* que devem ser lidos do arquivo. É responsabilidade do programador reservar o espaço necessário para que a função não invada memória.

A função irá retornar a quantidade de *bytes* realmente lidos do arquivo. Caso ocorra algum erro na leitura do arquivo, a função irá retornar `-1` indicando erro. A descrição do erro estará disponível na variável `errno`.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>
#include <fcntl.h>
```

```
void main(int argc, char *argv[])
{
    int      fd;

    char      sNome [_POSIX_PATH_MAX];
    ssize_t   iQtdeLida;
    char      aBuffer[100];
```

Esta constante indica o tamanho (de caracteres incluindo diretórios e subdiretórios) que um arquivo pode possuir. Está definido no arquivo `limits.h`.

Reservando espaço para a leitura.

```
printf ("Entre com o nome do arquivo : ");
```

```

gets (sNome);
fd = open (sNome, O_RDONLY);
if (fd < 0)
{
    perror (argv[0]);
    exit (errno);
}
printf ("Arquivo '%s' aberto\n", sNome);

printf("\nTentando ler 100 bytes do arquivo indicado\n");

iQtdeLida = read (fd, &aBuffer, 100);

if (iQtdeLida < 0)
{
    perror (argv[0]);
    exit (errno);
}

printf("\nForam lidos %d bytes do arquivo '%s'\n", iQtdeLida,
sNome);
close (fd);
exit (0);
}

```

Deve-se passar o endereço de memória da variável para onde os dados serão lidos.

### Programa 18.3

#### Resultado do Programa 18.3

##### 1ª Execução

Entre com o nome do arquivo : file6  
 Arquivo 'file6' aberto  
 Tentando ler 100 bytes do arquivo indicado  
 Foram lidos 0 bytes do arquivo 'file6'

##### 2ª Execução

Entre com o nome do arquivo : file7  
 Arquivo 'file7' aberto  
 Tentando ler 100 bytes do arquivo indicado  
 Foram lidos 100 bytes do arquivo 'file7'

##### 3ª Execução

Entre com o nome do arquivo : file8  
 Arquivo 'file8' aberto  
 Tentando ler 100 bytes do arquivo indicado  
 Foram lidos 88 bytes do arquivo 'file8'

A função read não retorna um erro se o arquivo tiver menos bytes que o indicado para leitura. Neste caso, a função read retorna o que foi possível ser lido.

#### Arquivos utilizado pelo programa (obtidos através do comando `ls -l`):

```

-rwxr-x---  1 laureano  prof          0 May 13 10:15 file6
-rw-r-----  1 laureano  prof       110 May 13 10:15 file7
-rw-r-----  1 laureano  prof        88 May 13 10:15 file8

```

## 18.7 Função write

Sintaxe:

```
ssize_t write (int fildes, const void *buf, size_t nbyte);
```

A função `write` grava no arquivo indicado pelo descritor as informações obtidas do endereço fornecido. Os dados serão gravados a partir da posição atual do arquivo. Caso a opção `O_APPEND` tenha sido especificada na abertura, a posição atual do arquivo será antes atualizada com o valor do tamanho do arquivo. Após a gravação, a posição atual do arquivo será somada da quantidade de *bytes* gravados no arquivo.


Deve-se informar a quantidade de *bytes* a ser gravada no arquivo. Caso ocorra algum erro, a função irá retornar `-1`, e a descrição do erro estará disponível na variável `errno`. Caso não ocorra erro a função irá retornar a quantidade de *bytes* gravados no arquivo, que deve ser igual à quantidade informada como parâmetro.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <string.h>

void main (int argc, char *argv[])
{
    int      fd;
    ssize_t  iQtdeWrite;
    char     aBuffer[100];
    if( argc < 2 )
    {
        fprintf(stderr, "Obrigatório informar os nomes dos arquivos\n");
        exit(1);
    }

    fd = open (argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0755);
    if (fd < 0)
    {
        perror (argv[0]);
        exit (errno);
    }
    printf ("Arquivo '%s' aberto\n", argv[1]);
    strcpy (aBuffer, "Esta linha foi gravada pelo programa.");
```



Abrindo o arquivo indicado, truncando o mesmo, caso já exista.

```

iQtdeWrite = write (fd, &aBuffer, strlen (aBuffer));
if (iQtdeWrite < strlen (aBuffer))
{
    perror (argv[0]);
    exit (errno);
}
printf("\nForam gravado %d bytes no arquivo '%s'\n", iQtdeWrite,
argv[1]);

close(fd);
exit(0);
}

```

Gravando um texto no arquivo. Deve-se sempre passar o endereço de memória da variável cujo conteúdo se deseja para gravar.

Sempre deve ser verificado se todos os dados foram gravados. Basta comparar a quantidade de *bytes* gravados com o tamanho da variável que se deseja gravar.

Programa 18.4

## Resultado do Programa 18.4

```

>$ p18_4 file10
Arquivo 'file10' aberto
Foram gravado 37 bytes no arquivo 'file10'

```

Linha de comando.

## 18.8 Função lseek

Sintaxe:

```
off_t lseek(int fildes, off_t offset, int whence);
```

Como visto anteriormente, tanto a leitura como a gravação de informações no arquivo são realizadas a partir da posição atual do arquivo.

Com a função `lseek` pode-se posicionar em um determinado ponto do arquivo antes da leitura ou gravação de dados. O primeiro *byte* do arquivo é a posição 0 (zero).

Após a execução da função, ela retorna a posição atual do arquivo. Caso ocorra algum erro no posicionamento, a função irá retornar `-1`.

O posicionamento via função `lseek` é realizado através da informação de um deslocamento positivo ou negativo a partir de posições definidas no terceiro parâmetro. É possível colocar as seguintes posições:

`SEEK_SET` – O deslocamento informado será baseado no início do arquivo. O deslocamento deve ser zero ou um valor positivo pois não tem sentido se posicionar antes do início do arquivo.

## 160 ♦ Programando em C para Linux, Unix e Windows

**SEEK\_CUR** – O deslocamento será efetuado baseado na posição atual do arquivo. Para se posicionar antes da posição atual, passa-se um valor negativo no deslocamento. Para se avançar no arquivo, coloca-se um valor positivo no deslocamento.

**SEEK\_END** – O deslocamento será efetuado a partir do final do arquivo. Nesta situação pode-se colocar um deslocamento negativo ou positivo. Caso se indique um deslocamento positivo e for feita uma gravação no arquivo, este estará sendo estendido.

Veja o exemplo:

```
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>

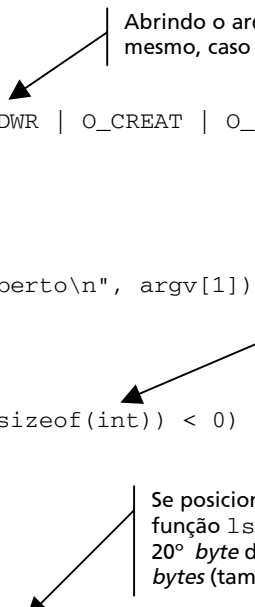
void main (int argc, char *argv[])
{
    int    fd;
    int    i;

    if( argc < 2 )
    {
        fprintf(stderr, "Obrigatório informar o nome do arquivo\n");
        exit(1);
    }

    fd = open (argv[1], O_RDWR | O_CREAT | O_TRUNC, 0755);
    if (fd < 0)
    {
        perror (argv[0]);
        exit (errno);
    }
    printf ("Arquivo '%s' aberto\n", argv[1]);

    for (i= 1; i<= 10; i++)
    {
        if (write (fd, &i, sizeof(int)) < 0)
        {
            perror (argv[0]);
            exit (errno);
        }
    }

    if (lseek (fd, 5 * sizeof(int), SEEK_SET) < 0)
```



Abrindo o arquivo, truncando o mesmo, caso já exista.

Gravando números no arquivo.

Se posicionando no sexto registro com a função lseek. O sexto registro começa na 20ª byte do arquivo, ou seja, 5 registros \* 4 bytes (tamanho do int).



```

{
    perror (argv[0]);
    exit (errno);
}
i = 127;
if (write (fd, &i, sizeof(int)) < 0)
{
    perror (argv[0]);
    exit (errno);
}

if (lseek (fd, 0, SEEK_SET) < 0)
{
    perror (argv[0]);
    exit (errno);
}

while(read (fd, &i, sizeof(int)) > 0)
    printf ("Valor lido |%d|\n", i);

close (fd);
exit (0);
}

```

E gravando o número 127 nesta posição.

Posicionando-se no início do arquivo...

...e lendo o arquivo até o final.

### Programa 18.5

#### Resultado do Programa 18.5

```

$> p18_5 file10
Arquivo 'file10' aberto
Valor lido |1|
Valor lido |2|
Valor lido |3|
Valor lido |4|
Valor lido |5|
Valor lido |127|
Valor lido |7|
Valor lido |8|
Valor lido |9|
Valor lido |10|

```

Linha de comando.

## 18.9 Função remove

Sintaxe:

```
int remove(const char *path);
```

A função `remove` apaga o arquivo especificado pela variável `path`. Ela devolve 0 (zero) se a operação foi um sucesso e um valor diferente de zero se ocorreu um erro.

Veja o exemplo:

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

void main(int argc, char *argv[])
{
    if( argc < 2 )
    {
        fprintf(stderr, "Obrigatório informar o nome do arquivo\n");
        exit(1);
    }

    printf("\nExcluindo o arquivo %s", argv[1]);
    if(remove(argv[1])==-1)
    {
        printf("\nErro na exclusão do arquivo.");
        exit(errno);
    }
    printf("\nArquivo excluído!");
    exit(0);
}
```

### Programa 18.6

#### Resultado do Programa 18.6

```
$> p18_6 file10
Excluindo o arquivo file10
Arquivo excluído!
```

← Linha de comando.

## 18.10 Função unlink

Sintaxe:

```
int unlink(const char *path);
```

A remoção de um arquivo de um sistema de arquivo é feita através da função `unlink`. Este nome é devido à função simplesmente receber um nome de arquivo como parâmetro e decrementar o número de links existente no *inode* do mesmo (consulte o *help* do sistema e veja o comando `ln`). Caso o número de links atinja o valor zero, então os dados do arquivo serão liberados para o sistema como áreas livres para uso.

Isto permite que se faça a remoção de arquivos ainda abertos, sem que haja a perda de dados enquanto o arquivo estiver aberto. Esta técnica permite a criação de arquivos temporários através da abertura dos mesmos. Logo em segui-

da remove-se o arquivo com a função `unlink` que irá decrementar o atributo correspondente.

Os dados criados pelo processo continuam a valer até que o processo feche o arquivo. Nesta ocasião seriam alterados os atributos do *inode* com as informações pertinentes, mas como o número de links está zerado, os dados são eliminados do sistema. Com isto estamos prevendo que, caso o processo cancele por algum motivo, o arquivo temporário criado pelo mesmo será removido automaticamente pelo sistema.

Veja o exemplo:

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <string.h>

void main (int argc, char *argv[])
{
    int iFd;
    char *szArquivo;
    char szBuffer[25];

    szArquivo = tempnam ("/tmp", "temp");

    printf ("Abrindo o arquivo %s'\n", szArquivo);
    iFd = open (szArquivo, O_RDWR | O_CREAT | O_TRUNC, 0777);
    if (iFd < 0)
    {
        perror (argv[0]);
        exit (1);
    }

    printf ("Removendo o arquivo ABERTO\n");
    if (unlink (szArquivo) < 0)
    {
        perror (argv[0]);
        exit (1);
    }

    printf ("Gravando informacoes no arquivo\n");
    strcpy (szBuffer, "123456789-123456789-");
    if (write (iFd, szBuffer, 20) < 20)
    {
        perror (argv[0]);
        exit (1);
    }
}
```

Função que retorna um nome aleatório baseado nos parâmetros informados. O nome retornado é único no sistema.

Criando um arquivo temporário.

Removendo o arquivo que está aberto!

Gravando dados no arquivo que foi excluído.

## 164 ♦ Programando em C para Linux, Unix e Windows

```
printf ("Posicionando no inicio do arquivo e lendo as informacoes\n");
strcpy (szBuffer, "+++++++");
lseek (iFd, 0, SEEK_SET);
if (read (iFd, szBuffer, 20) < 20)
{
    perror (argv[0]);
    exit (1);
}

printf ("Informacoes gravadas e lidas : %s\n", szBuffer);
close (iFd);
exit (0);
}
```

Programa 18.7

### Resultado do Programa 18.7

```
Abrindo o arquivo /tmp/tempJxCWia'
Removendo o arquivo ABERTO
Gravando informacoes no arquivo
Posicionando no inicio do arquivo e lendo as informacoes
Informacoes gravadas e lidas : |123456789-123456789-|
```

## 18.11 Função rename

Sintaxe:

```
int rename (const char *source, const char *target);
```

A função `rename` faz com que o arquivo indicado no primeiro parâmetro tenha o seu nome trocado pelo nome informado no segundo parâmetro.

Caso os nomes se referenciem ao mesmo diretório, está sendo realizada uma troca de nomes de arquivos. Caso o diretório origem seja diferente do diretório destino, está sendo realizada uma movimentação de um arquivo de um diretório para o outro, com o mesmo ou outro nome. Caso o arquivo indicado no segundo parâmetro já exista no sistema de arquivo, ele é removido antes da troca de nomes. Consulte o *help* do sistema para ver o comando `mv` do Unix/Linux.

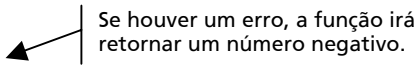
Os arquivos de origem e destino devem ser do mesmo tipo e devem residir no mesmo sistema de arquivos. Caso o arquivo de origem seja um *link* simbólico, o sistema irá trocar o nome do *link* simbólico e não do arquivo apontado pelo mesmo.

Veja o exemplo:

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

void main (int argc, char *argv[])
{
    if( argc < 3 )
    {
        fprintf(stderr, "Obrigatório informar os nomes dos arquivos\n");
        exit(1);
    }

    if (rename (argv[1], argv[2]) < 0)
    {
        perror (argv[0]);
        exit (9);
    }
    exit(0);
}
```



Se houver um erro, a função irá retornar um número negativo.

**Programa 18.8**



## Buscando Algumas Informações no Linux e Unix

*Se essa é a idade da informação, por que ninguém sabe nada?*

Robert Mankoff, cartunista americano

### 19.1 Funções `getpid` e `getppid`

Sintaxe:

```
pid_t getpid(void);  
pid_t getppid(void);
```

Quando um programa está rodando, ele é chamado de processo e ocupa um espaço dentro do Gerenciador de Processos do Linux/Unix. Para identificar este processo, o sistema operacional atribui a ele um número inteiro positivo, chamado de `Process Identification`, `Process ID` ou `PID`.

O sistema operacional garante que enquanto o processo estiver ativo na máquina ele será o único com este número.

Juntamente com o `PID`, o processo possui o `PPID`, ou `Parent Process ID`, que é o `PID` do processo pai que ativou o processo. Na maioria dos casos, quando um programa é executado, o `PPID` dele será o `PID` do processo *shell* que está sendo executado.

Este vínculo de um processo com seu pai é devido às características de dependência existentes entre os processos pai e filho. Quando um processo pai termina por qualquer razão, todos os processos filhos do mesmo são também eliminados do sistema.

A função `getpid` retorna o `Process ID` do processo corrente. Não é necessário informar nenhum parâmetro para a função e ela retorna uma variável do tipo `pid_t` (`long int`). A função `getppid` retorna o `PID` do processo pai do processo corrente. As funções não retornam erro, pois é garantido que um processo sempre tenha um `PID` e um `PPID`.

Veja o exemplo:

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

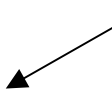
void main (int argc, char *argv[])
{
    pid_t pid_processo;
    pid_t pid_pai;

    pid_processo = getpid();
    pid_pai = getppid();

    printf ("Valores retornados\n");
    printf ("Process ID      : |%d|\n", pid_processo);
    printf ("Process ID pai   : |%d|\n", pid_pai);

    printf ("\nDigite 'echo $$' para obter o PID do processo shell");
    printf (" que chamou este programa\n");

    exit (0);
}
```



Obtendo os valores de identificação deste processo.

### Programa 19.1

#### Resultado do Programa 19.1

```
$> ./p19_1
Valores retornados
Process ID      : |19629|
Process ID pai   : |19579|
```

Digite 'echo \$\$' para obter o `PID` do processo `shell` que chamou este programa

```
$> echo $$
19579
```

## 19.2 Funções `getuid` e `geteuid`

Sintaxe:

```
uid_t getuid(void);  
uid_t geteuid(void);
```

Além das informações do `PID` e `PPID`, o processo possui mais informações associadas. O processo possui o `User Identification (UID)` e o `Group Identification (GID)` reais que indicam para o sistema quem o usuário realmente é. Estes dois identificadores são obtidos quando o processo *shell* da sessão é iniciada e os valores são retirados do arquivo `/etc/passwd`. Normalmente estes valores não mudam durante a sessão, a não ser que o usuário possua permissão de superusuário.

Estes valores serão usados pelo sistema operacional para a verificação das permissões quando o processo faz algum acesso a um recurso do sistema (arquivos, diretórios, dispositivos, memória compartilhada, filas, named pipes).

O segundo conjunto de identificadores é chamado de identificadores efetivos. Eles quase sempre são iguais aos identificadores reais, a menos que o processo tenha sido gerado a partir de um programa que tenha os seus bits de `set-user-ID` e/ou `set-group-ID` ligados (consulte o comando `chmod` no manual do sistema).

São os identificadores efetivos que serão usados para a verificação de permissão quando o processo fizer algum acesso aos recursos do sistema.

Para se obter o `UID` real de um processo é utilizada a função `getuid`. Não é necessário se passar nenhum parâmetro e ela retornará um valor do tipo `uid_t`. Este `UID` é sempre o `UID` de quem executou o programa que virou processo.

A função `geteuid` retorna o `UID` efetivo do processo. Na maioria das vezes o `UID` efetivo é igual ao `UID` real. Será diferente caso o arquivo com o programa que deu início ao processo possua o `set-user-ID` ligado com o comando `chmod u+s`.

Como todo processo sempre possui os identificadores do usuário, as funções citadas não retornam erro.



Veja o exemplo:

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

void main (void)
{
    uid_t uid_real;
    uid_t uid_efet;

    uid_real = getuid();
    uid_efet = geteuid();

    printf("Valores retornados\n");
    printf("User Id real      : |%d|\n", uid_real);
    printf("User Id efetivo   : |%d|\n", uid_efet);

    printf("\nDigite 'id' para obter o UID do processo shell");
    printf(" que chamou este programa\n");

    exit(0);
}
```

Declaração das variáveis. Repare no tipo da variável (uid\_t).

Pegando as informações...

## Programa 19.2

### Resultado do Programa 19.2

```
Valores retornados
User Id real      : |10004|
User Id efetivo   : |10004|
```

Digite 'id' para obter o UID do processo shell que chamou este programa

```
$> id
uid=10004(laureano) gid=502(laureano) grupos=502(laureano)
```

Verificação com o comando id.

## 19.3 Função uname

Sintaxe:

```
int uname(struct utsname *name);
```

A função `uname` retorna no parâmetro uma estrutura que conterá informações sobre o sistema operacional. A estrutura devolvida possui as seguintes informações:

```
#define UTSLEN  9
#define SNLEN   15
```

## 170 ♦ Programando em C para Linux, Unix e Windows

```
struct utsname
{
    char sysname[UTSLEN];
    char nodename[UTSLEN];
    char release[UTSLEN];
    char version[UTSLEN];
    char machine[UTSLEN];
    char idnumber[SNLEN];
};
```

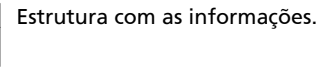
Caso ocorra algum erro na função, será retornado `-1` como resultado. O erro ocorrido será colocado na variável `errno`. A função retorna valor zero caso a função não apresente problemas.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/utsname.h>

void main(void)
{
    struct utsname  info;
    if (uname(&info) < 0)
    {
        perror("");
        exit(errno);
    }
    printf("Nome do Sistema   : |%s|\n", info.sysname);
    printf("Nome do Node      : |%s|\n", info.nodename);
    printf("Release do S.O.       : |%s|\n", info.release);
    printf("Versao do S.O.         : |%s|\n", info.version);
    printf("Tipo de Hardware      : |%s|\n", info.machine);

    exit (0);
}
```



Estrutura com as informações.

### Programa 19.3

#### Resultado do Programa 19.3

```
Nome do Sistema   : |Linux|
Nome do Node      : |hercules.ppgia.pucpr.br|
Release do S.O.   : |2.4.30|
Versao do S.O.    : |#1 SMP Qua Mai 4 10:37:07 BRT 2005|
Tipo de Hardware  : |i686|
```

## 19.4 Função access

Sintaxe:

```
int access(char *path, int amode);
```

A função `access` é uma maneira de se testar a permissão para um determinado arquivo. No primeiro parâmetro é passado o nome do arquivo e no segundo parâmetro deve-se informar através de constantes definidas no arquivo `unistd.h` qual é a permissão que se quer testar. Pode-se agrupar mais de uma permissão através de "OU" binário para testar mais de uma permissão. A função irá retornar zero caso a permissão indicada esteja assinalada no arquivo.

A função `access` irá considerar o `UID` e `GID` real do processo no teste das permissões e não os valores efetivos do processo. As constantes permitidas na função são:

`R_OK` – Permissão de leitura.  
`W_OK` – Permissão de gravação.  
`X_OK` – Permissão de execução.  
`F_OK` – Teste para arquivo comum.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
void main (int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "Uso : %s <arquivo>\n", argv[0]);
        exit(1);
    }
    printf("Permissoes para o arquivo %s\n", argv[1]);

    if(access(argv[1], R_OK) == 0)
        printf("\tLeitura\n");

    if(access(argv[1], W_OK) == 0)
        printf("\tGravacao\n");

    if(access(argv[1], X_OK) == 0)
        printf("\tExecucao\n");

    exit(0);
}
```

Obtendo as permissões do arquivo indicado no argumento.

Programa 19.4

## 172 ♦ Programando em C para Linux, Unix e Windows

### Resultado do Programa 19.4

- o Verificando as permissões:

```
$> ls -l p19_4.c p19_4
```

```
-rwx----- 1 laureano laureano 14272 Mai 22 11:10 p19_4  
-rw----- 1 laureano laureano 897 Mai 22 11:06 p19_4.c
```

#### 1ª Execução

```
$> p19_4 p19_4
```

```
Permissoes para o arquivo p19_4
```

```
Leitura  
Gravacao  
Execucao
```

Verificando as permissões do arquivo executável (próprio arquivo)...

#### 2ª Execução

```
$> p19_4 p19_4.c
```

```
Permissoes para o arquivo p19_4.c
```

```
Leitura  
Gravacao
```

Verificando as permissões do arquivo fonte...

## 19.5 Função stat

Sintaxe:

```
int stat(const char *ph, struct stat *bf);
```

Os arquivos gravados no sistema de arquivos possuem uma série de informações pertinentes a eles, chamadas de atributos. São exemplos de atributos: dono, permissão, tamanho, inode etc.

A função `stat` é utilizada para obter os atributos de um arquivo armazenado no sistema de arquivo, fornecendo para a função o nome do arquivo, bem como o seu caminho no sistema de arquivos. A função retorna os atributos do sistema de arquivos e coloca em uma estrutura do tipo `stat`.

As informações retornadas na estrutura são as seguintes:

```
struct stat {  
    dev_t      st_dev;      /* device */  
    ino_t      st_ino;      /* inode */  
    mode_t     st_mode;     /* protection */  
    nlink_t    st_nlink;    /* number of hard links */  
    uid_t      st_uid;      /* user ID of owner */  
    gid_t      st_gid;      /* group ID of owner */  
    dev_t      st_rdev;     /* device type (if inode device) */  
    off_t      st_size;     /* total size, in bytes */  
    blksize_t  st_blksize;  /* blocksize for filesystem I/O */  
};
```

```

blkcnt_t    st_blocks;    /* number of blocks allocated */
time_t      st_atime;     /* time of last access */
time_t      st_mtime;     /* time of last modification */
time_t      st_ctime;     /* time of last status change */
};

```

O campo `st_mode` possui bits indicando três informações: tipo do arquivo, atributos adicionais e permissões de acesso. O tipo do arquivo está representado de maneira binária no campo `st_mode` da estrutura `stat` retornada pela função `stat`.

Para obter um determinado bit deste campo deve-se fazer-se um “E” binário com uma máscara que indicará quais bits está se querendo obter. O sistema possui macros definidas no arquivo `sys/stat.h` para testar diretamente estes bits. As macros definidas irão retornar verdadeiro caso o campo `st_mode` passado como parâmetro estiver indicando os seguintes tipos de arquivos:

Macro	Função
<code>S_ISREG()</code>	Arquivo comum
<code>S_ISDIR()</code>	Diretório
<code>S_ISCHR()</code>	Arquivo de dispositivo orientado a caractere
<code>S_ISBLK()</code>	Arquivo de dispositivo orientado a bloco
<code>S_ISFIFO()</code>	O arquivo é um pipe ou um arquivo FIFO
<code>S_ISLNK()</code>	O arquivo é um link simbólico
<code>S_ISSOCK()</code>	O arquivo é um arquivo do tipo socket

O campo `st_mode` possui também a permissão de acesso ao arquivo codificado em bits dentro deste campo. Para testar uma determinada permissão, deve-se fazer um “E” binário com constantes indicando os bits desejados.

Para facilitar o processo, estão definidas no arquivo `sys/stat.h` as constantes a seguir, indicando cada uma um bit de permissão. Estas constantes são usadas fazendo o “E” binário delas com o campo `st_mode`. Quando se desejar obter mais de um bit de permissão, deve-se juntar as constantes anteriores com um “OU” binário antes de se fazer o “E” binário com o campo.

As constantes definidas representam os seguintes bits de permissão:

- o Permissão para o Usuário
  - `S_IRUSR` – Leitura.
  - `S_IWUSR` – Gravação.
  - `S_IXUSR` – Execução.
  - `S_RWXU` – `S_IRUSR` | `S_IWUSR` | `S_IXUSR`

- o Permissão para o Grupo
  - `S_IRGRP` – Permissão de leitura.
  - `S_IWGRP` – Permissão de gravação.
  - `S_IXGRP` – Permissão de execução.
  - `S_RWXG` – `S_IRGRP` | `S_IWGRP` | `S_IXGRP`
- o Permissão para os Outros
  - `S_IROTH` – Permissão de leitura.
  - `S_IWOTH` – Permissão de gravação.
  - `S_IXOTH` – Permissão de execução.
  - `S_RWXO` – `S_IROTH` | `S_IWOTH` | `S_IXOTH`

Além destes bits de permissão, existem os bits chamados de `set-user-ID` e `set-group-ID` usados para programas. Quando um programa é executado, o sistema operacional atribui ao processo o `UID` e `GID` de quem está chamando o programa. Caso um dos bits esteja ligado, o sistema irá atribuir ao novo processo o `UID` ou `GID` do arquivo e não do usuário que está chamando o programa. Para testar estes bits do campo `st_mode` são usadas as seguintes constantes:

`S_ISUID` – Set-user-ID bit ligado.  
`S_ISGID` – Set-group-ID bit ligado.

Veja o exemplo:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

void main (int argc, char *argv[])
{
    struct stat info;

    if(argc != 2)
    {
        fprintf(stderr, "Uso : %s <arquivo>\n", argv[0]);
        exit(1);
    }

    if(stat(argv[1], &info) < 0)
    {
        printf("Erro ao obter informacoes de %s\n", argv[1] );
        exit(1);
    }
}
```

← Obtendo os atributos do arquivo.

```

                                Mostrando os atributos padrões.
printf("st_ino           : %d\n", info.st_ino);
printf("st_mode (oct)   : %o\n", info.st_mode);
printf("st_nlink        : %d\n", info.st_nlink);
printf("st_uid          : %d\n", info.st_uid);
printf("st_gid          : %d\n", info.st_gid);
printf("st_size         : %d\n", info.st_size);
printf("st_atime        : %s",   ctime (&info.st_atime));
printf("st_mtime        : %s",   ctime (&info.st_mtime));
printf("st_ctime        : %s",   ctime (&info.st_ctime));
printf("st_blksize(Kb) : %d\n", info.st_blksize / 1024);

                                Mostrando o tipo do arquivo usando as macros.
if(S_ISREG(info.st_mode))
    printf("%s --> Arquivo comum\n", argv[1]);

if(S_ISDIR(info.st_mode))
    printf("%s --> Diretorio\n", argv[1]);

if(S_ISCHR(info.st_mode))
    printf ("%s --> Dispositivo orientado a caracter\n", argv[1]);

if(S_ISBLK(info.st_mode))
    printf ("%s --> Dispositivo orientado a bloco\n", argv[1]);

if(S_ISFIFO(info.st_mode))
    printf ("%s --> Arquivo pipe ou FIFO\n", argv[1]);

if(S_ISLNK(info.st_mode))
    printf ("%s --> Link simbolico\n", argv[1]);

printf("Permissao do arquivo %s : ", argv[1]);
printf(info.st_mode & S_IRUSR? "r" : "-");
printf(info.st_mode & S_IWUSR? "w" : "-");
printf(info.st_mode & S_IXUSR? "x" : "-");
printf(" ");
printf(info.st_mode & S_IRGRP? "r" : "-");
printf(info.st_mode & S_IWGRP? "w" : "-");
printf(info.st_mode & S_IXGRP? "x" : "-");
printf(" ");
printf(info.st_mode & S_IROTH? "r" : "-");
printf(info.st_mode & S_IWOTH? "w" : "-");
printf(info.st_mode & S_IXOTH? "x" : "-");
printf("\n");

exit(0);
}

```

Verificando as permissões do arquivo.

Programa 19.5

## Resultado do Programa 19.5

- o Verificando as informações (`ls -li /usr/bin/passwd`)

```
1391286 -r-s--x--x 1 root root 19336 Set 7 2004 /usr/bin/passwd
```

```
$> p19_5 /usr/bin/passwd
st_ino      : 1391286
st_mode (oct) : 104511
st_nlink    : 1
st_uid      : 0
st_gid      : 0
st_size     : 19336
st_atime    : Wed May 18 21:11:38 2005
st_mtime    : Tue Sep 7 05:11:03 2004
st_ctime    : Tue May 10 04:18:14 2005
st_blksize(Kb): 4
/usr/bin/passwd --> Arquivo comum
Permissao do arquivo /usr/bin/passwd : r-x --x --x
```

## 19.6 Função umask

Sintaxe:

```
mode_t umask(mode_t cmask);
```

Toda vez que o sistema cria um arquivo ou diretório, ele utiliza uma máscara padrão de permissão. Esta máscara sempre está assinalada para um valor e nela deve-se colocar quais são as permissões que devem ser desligadas na criação de arquivos e diretórios. Veja o comando `umask` do sistema operacional.

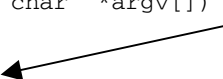
Com a função `umask` a máscara é alterada somente para o processo em questão. A máscara informada na função `umask` tem prioridade sobre a máscara informada na criação de arquivos via função `open` ou `creat`.

Pode-se passar como parâmetro o formato octal de permissão aceito pelo comando `chmod` (desde que precedido por zero para indicar o formato octal) ou então utilizar as mesmas constantes definidas no `sys/stat.h` para obter a permissão do campo `st_mode` da estrutura `stat`.

Veja o exemplo:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

void main (int argc, char *argv[])
{
    mode_t mask_old;
    | Máscara anterior do sistema.
```





```

mask_old = umask(S_IWGRP | S_IRWXO);
printf("Mascara anterior do sistema : '%03o'\n", mask_old);

printf("\nCriando o arquivo 'arq19_6.1'\n");
creat("arq19_6.1", 0777);

mask_old = umask(0);
printf("Mascara anterior do sistema : '%03o'\n", mask_old);

printf("\nCriando o arquivo 'arq19_6.2'\n");
creat("arq19_6.2", 0777);

exit(0);
}

```

Assinalando a máscara padrão para desligar os *bits* do grupo e usuário.

Assinalando a máscara padrão para ligar todas as permissões de um arquivo.

### Programa 19.6

#### Resultado do Programa 19.6

```

Mascara anterior do sistema : '022'
Criando o arquivo 'arq19_6.1'
Mascara anterior do sistema : '027'
Criando o arquivo 'arq19_6.2'

```

- o Verificando as permissões dos arquivos criados (`ls -l`)
 

```

$> ls -l
-rwxr-x--- 1 laureano laureano 0 Mai 22 19:00 arq19_6.1
-rwxrwxrwx 1 laureano laureano 0 Mai 22 19:00 arq19_6.2

```
- o Verificando as permissões de criação do arquivo do sistema (`umask`)
 

```

$> umask
0022

```



*Se você não pode descrever o que está fazendo como um processo,  
você não sabe o que está fazendo.*

W. Edwards Deming, consultor em qualidade americano

## 20.1 Criação de processos no sistema operacional

Com exceção de um processo do sistema operacional (`init`), que é criado durante o processo de inicialização (`boot`) da máquina, todos os demais processos são criados através do uso da função interna do *kernel* chamada `fork`.

A função `fork` duplica o processo atual dentro do sistema operacional. O processo que inicialmente chamou a função `fork` é chamado de *processo pai*. O novo processo criado pela função `fork` é chamado de *processo filho*. Todas as áreas do processo são duplicadas dentro do sistema operacional (código, dados, pilha, memória dinâmica). Para maiores informações sobre funcionamento de processos, recomenda-se a leitura de um livro específico sobre o assunto.

Portanto, a função `fork` é chamada uma única vez e retorna duas vezes, uma no processo pai e outra no processo filho. Como exemplo, o processo filho herda informações do processo pai:

- o Usuários (`user id`) Real, efetivo.
- o Grupos (`group id`) Real, efetivo.
- o Variáveis de ambiente.

- o Descritores de arquivos.
- o Prioridade.
- o Todos os segmentos de memória compartilhada assinalados.
- o Diretório corrente de trabalho.
- o Diretório Raiz.
- o Máscara de criação de arquivos.

O processo filho possui as seguintes informações diferentes do processo pai:

- o PID único dentro do sistema.
- o Um PPID diferente. (O PPID do processo filho é o PID do processo pai que inicialmente ativou a função `fork`).
- o O conjunto de sinais pendentes para o processo é inicializado como estando vazio.
- o *Locks* de processo, código e dados não são herdados pelo processo filho.
- o Os valores da contabilização do processo obtida pela função `time` são inicializados com zero.
- o Todos os sinais de tempo são desligados.

## 20.2 Função `fork`

Sintaxe:


```
pid_t fork(void);
```


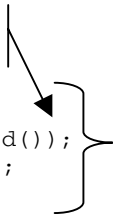
Para criar um novo processo basta chamar a função `fork`. É criado um novo processo, chamado de processo filho, que é uma cópia quase fiel do processo pai. Quando do uso da função `fork`, em ambos os processos será executada a linha seguinte à chamada da função `fork`. Para identificar de dentro de um processo qual é o código do pai e qual é o código do filho, pois geralmente possuem lógicas distintas, deve-se testar o retorno da função `fork`.


Caso a função `fork` retorne 0 (zero), o processo filho está sendo executado. Caso a função retorne um valor diferente de 0 (zero), mas positivo, o processo pai está sendo executado. O valor retornado representa o `PID` do processo filho criado. A função retorna -1 em caso de erro, provavelmente devido a ter atingido o limite máximo de processos por usuário configurado no sistema. Veja o exemplo:

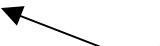
```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
```

## 180 ♦ Programando em C para Linux, Unix e Windows

```
void main(int argc, char *argv[])
{
    int iPid;  Variável que irá conter o Process ID retornado pela função fork.

    printf ("\nDuplicando o processo\n");
    iPid = fork ();
    if (iPid < 0)  Duplicando o processo e verificando
    {                                     algum possível erro.
        perror(argv[0]);
        exit(errno);
    }
     Este código será executado apenas no pai, embora o if
    também esteja disponível para o processo filho.

    if(iPid != 0){
        printf("\nCódigo executado no processo pai\n");
        printf("\nPAI -Processo pai.  PID:|%d|\n", getpid());
        printf("\nPAI -Processo filho.PID:|%d|\n", iPid);
    }
    else
    {
        printf("\nCódigo executado pelo filho");
    }
     Este código será executado apenas no filho, embora o
    if também esteja disponível para o processo pai.


    if(iPid == 0) {
        printf("\nCódigo executado no processo filho\n");
        printf("\nFILHO-Processo pai.  PID:|%d|\n",getppid());
        printf("\nFILHO-Processo filho.PID:|%d|\n",getpid());
    }
    else
    {
        printf("\nCódigo executado pelo pai");
    }
    printf("\nEsta mensagem será impressa 2 vezes");
    exit(0);
}
 Este código está disponível para o pai e para o
filho.
```

### Programa 20.1

#### Resultado do Programa 20.1

Duplicando o processo

```
Código executado pelo filho
Codigo executado no processo filho
FILHO-Processo pai.  PID:|17216|
FILHO-Processo filho. PID:|17217|
Esta mensagem será impressa 2 vezes
Codigo executado no processo pai
PAI-Processo pai.  PID:|17216|
PAI-Processo filho. PID:|17217|
Código executado pelo pai
Esta mensagem será impressa 2 vezes
```

 As mensagens enviadas por vários processos simultâneos, quase sempre não são sincronizadas para impressão na tela.

## 20.3 Função `wait`

Sintaxe:

```
pid_t wait (int *stat_loc);
```

O processo pai pode esperar o término de um processo filho através da chamada da função `wait`. A função `wait` irá devolver o *status* de retorno de qualquer processo filho que termine. O processo que chamar a função irá apresentar um dos seguintes comportamentos:

- o Bloquear a sua execução até o término de algum processo filho.
- o Retornar imediatamente com o status de término de um processo filho caso o filho já tenha terminado (zumbi) e esteja esperando o processo pai receber o seu *status*.
- o Retornar imediatamente com um erro caso não se tenha nenhum processo filho rodando.

A função irá retornar, além do *status* de término no parâmetro, o `PID` do processo filho que terminou. Isto é útil caso vários processos filhos tenham sido ativados e se quer ter controle sobre o término de cada um deles.

O *status* retornado pela função, apesar de ser um número inteiro, possuirá informações codificadas indicando se o processo terminou normalmente ou cancelou o código de retorno do processo. O arquivo `sys/wait.h` contém um conjunto de macros para testar o motivo do término do processo filho, bem como para obter o código de retorno caso o processo tenha sido terminado normalmente, ou o número do sinal caso o processo tenha sido cancelado.

As macros para testar o motivo do término são as seguintes:

- o `WIFEXITED(*stat_loc)` – Retorna verdadeiro caso o processo filho tenha terminado normalmente através da chamada das funções `exit` ou `_exit`.
- o `WIFSIGNALED(*stat_loc)` – Retorna verdadeiro caso o processo filho tenha terminado através do recebimento de um sinal.
- o `WIFSTOPPED(*stat_loc)` – Retorna verdadeiro caso o processo filho tenha recebido um sinal de *stop*.

Para obter o *status* de término são usadas as seguintes macros:

- o `WEXITSTATUS(*stat_loc)` – Caso o processo tenha terminado normalmente, esta macro retorna o código de retorno do processo filho.
- o `WTERMSIG(*stat_loc)` – Caso o processo tenha sido cancelado por um sinal, esta macro retorna o número do sinal que cancelou o processo filho.

## 182 ♦ Programando em C para Linux, Unix e Windows

- o `WCOREDUMP(*stat_loc)` – Caso o processo tenha sido cancelado por um sinal, esta macro irá retornar verdadeiro caso o processo filho em seu cancelamento tenha gerado um core dump.
- o `WSTOPSIG(*stat_loc)` – Caso o processo filho tenha sido parado por um sinal, esta macro retorna o número do sinal que parou o processo filho.

Veja o exemplo:

```
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>

void main (int argc, char *argv[])
{
    int iPid;
    int iStatus; ← Status de retorno do processo filho.
    int iVlr;

    printf ("\nCriando um processo filho\n");
    iPid = fork ();
    if (iPid < 0)
    {
        perror(argv[0]);
        exit(errno);
    }

    if (iPid != 0)
    {
        wait (&iStatus);
        if (WIFEXITED(iStatus))
            printf ("\nTermino normal : |%d|\n", WEXITSTATUS(iStatus));
        else if (WIFSIGNALED(iStatus))
            printf ("\nCancelado por sinal : |%d|\n", WTERMSIG(iStatus));
    }
    if(iPid == 0)
    {
        printf("\nProcesso filho-PID-%d\n", getpid());
        while(1) /* Loop infinito */
        {
            printf ("Digite um numero (0 p/ terminar) : ");
            scanf ("%d", &iVlr);
            if(iVlr == 0)
                exit(1);
        }
    }
    exit(0);
}
```

Programa 20.2

## Resultado do Programa 20.2

- o **1ª Execução**  

```
$> p20_2
Criando um processo filho
Executando o processo filho - PID - 17521
Digite um numero (0 p/ terminar) : 2
Digite um numero (0 p/ terminar) : 0
Termino normal : |1|
```
- o **2ª Execução (executado o comando `kill -9 17523`)**  

```
$> p20_2
Criando um processo filho
Executando o processo filho - PID - 17523
Digite um numero (0 p/ terminar) :
Cancelado por sinal : |9|
```

## 20.4 Função `waitpid`

Sintaxe:

```
pid_t waitpid (pid_t pid, int *stat_loc, int options);
```

A função `wait` apresenta duas características que nem sempre são desejáveis. A primeira é que ela trava o processo que a chamou até a ocorrência do término de um processo filho e a outra é que não podemos monitorar um único processo filho, pois a mesma retorna o *status* de qualquer processo filho que tenha terminado.

Com a função `waitpid` tem-se um controle melhor sobre os processos filhos que estão sendo executados, pois pode-se monitorar um único processo filho caso se deseje, sem que ocorra o travamento do processo pai. O primeiro parâmetro da função irá determinar sobre qual ou quais processos filhos estamos interessados. O parâmetro pode ser um dos seguintes valores:

- o `-1` – Com este valor, a função `waitpid` irá processar qualquer processo filho que esteja sendo executado. Caso o terceiro parâmetro seja `0` (zero), a função tem o mesmo comportamento da função `wait`.
- o `>0` – Neste caso o parâmetro está indicando o `PID` do processo filho que será aguardado pela função.
- o `0` – Com este valor, a função irá aguardar qualquer filho que faça parte do mesmo processo `group ID` do processo pai.
- o `< -1` – Indica para a função aguardar todos os processos filhos cujo `process group ID` está indicado pelo valor absoluto do parâmetro.

O terceiro parâmetro da função `waitpid` indica a ação a ser executada pela função. Ele deve ser um dos valores a seguir definidos ou uma conjunção binária (OU binário) destes valores.

## 184 ♦ Programando em C para Linux, Unix e Windows

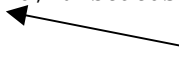
- o WNOHANG – A função `waitpid` não irá esperar o término do processo filho, retornando o valor zero para o processo que a ativou.
- o WUNTRACED – Caso se passe este *flag* para a função `waitpid`, ela retornará a informação que o processo filho está parado (*stopped*) devido ao recebimento de algum sinal.

Sobre o status de um processo no sistema operacional, pode-se utilizar o comando `ps` do Linux/Unix (veja o seu funcionamento no manual *on-line* do sistema). Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>

void main(int argc, char *argv[])
{
    int iPid;
    int iStatus;

    printf("\nCriando um processo filho\n");
    iPid = fork();
    if(iPid < 0)
    {
        perror(argv[0]);
        exit(errno);
    }

    if (iPid != 0)
    {
        while(1)
        {
            printf("\nVerificando o processo filho\n");
            sleep(2);
            if(waitpid(iPid, &iStatus, WNOHANG) != 0)
            {
                 Verificando se o processo filho terminou...
                if(WIFEXITED(iStatus))
                    printf("\nTermino normal:|%d|\n", WEXITSTATUS(iStatus));
                else if (WIFSIGNALED(iStatus))
                    printf("\nCancelado por sinal:|%d|\n", WTERMSIG(iStatus));
                break;
            }
        }
    }
    if (iPid == 0)
    {
        int iVlr;
    }
}
```



```

printf("\nExecutando o processo filho\n");
while(1)    /* Loop infinito */
{
    printf("Digite um numero (0 p/ terminar) : ");
    scanf ("%d", &iVlr);
    if(iVlr == 0)
        exit(27);
}
exit (0);
}

```

← Valor que será retornado para o pai.

### Programa 20.3

#### Resultado do Programa 20.3

```

Criando um processo filho
Executando o processo filho
Digite um numero (0 p/ terminar) :
Verificando o processo filho
Verificando o processo filho
0
Verificando o processo filho
Termino normal:|27|

```

## 20.5 Função `execl`

### Sintaxe:

```
int execl(const char *path, const char *arg0, ..., (char *) 0);
```

É muito comum que ocorra a duplicação de um processo para posterior substituição do processo filho por um outro programa. Existem seis funções que realizam a carga de um programa executável no processo, cada uma com um tipo de parâmetro. Estas funções são conhecidas como sendo da família `exec`.

Com a chamada de alguma função da família `exec`, o código atual do processo será substituído pelo novo código carregado do disco. Portanto, as funções da família `exec` não possuem valor de retorno, pois o código é substituído pelo novo programa.

Caso a função retorne para o processo, estará se configurando algum erro na carga do novo programa, seja por problemas de permissão de execução, seja por não se ter localizado o programa fonte indicado no parâmetro das funções da família `exec`. A função `execl`, onde o `l` significa "lista de argumentos", possui por característica receber os argumentos a serem passados para o novo programa através de ponteiros para uma *string* em C com o valor do argumento.

O primeiro parâmetro indica o programa a ser executado. O fim dos argumentos a serem passados para o novo processo deve ser indicado através da colocação de

um parâmetro representando um ponteiro nulo ( `(char *)0` ). Caso este ponteiro nulo não seja colocado, os resultados serão imprevisíveis. Veja o exemplo:

```
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>

void main (int argc, char *argv[])
{
    int iPid;
    int iStatus;

    printf("\nDuplicando o processo\n");
    iPid = fork ();
    if(iPid < 0)
    {
        perror(argv[0]);
        exit(errno);
    }
    if(iPid != 0)
    {
        wait (&iStatus);
        printf("\nStatus de termino :|%d|\n",
               WEXITSTATUS(iStatus));
    }

    if(iPid == 0)
    {
        printf("\nExecutando o comando 'ls /home'\n");
        execl("/bin/ls", "ls", "/home", (char *) 0);
        perror (argv[0]);
        exit(errno);
    }
    exit(0);
}
```

Execução do comando ls.

Estas instruções serão executadas no filho somente em caso de erro.

#### Programa 20.4

##### Resultado do Programa 20.4

```
Duplicando o processo
Executando o comando 'ls /home'
aluteste      espec  grad      mest  prof      seq
aquota.user  ftp    lost+found  old   projeto
Status de termino : |0|
```

##### o Verificando o diretório /home

```
$> ls /home
aluteste      espec  grad      mest  prof      seq
aquota.user  ftp    lost+found  old   projeto
```

## 20.6 Função system

Sintaxe:

```
int system (const char *command);
```

Um procedimento comum a ser realizado é a necessidade de executar um comando do sistema operacional qualquer dentro do programa, esperar o seu término e após seguir com o processamento. Isto pode ser feito através da execução da função `fork`, da execução da função `waitpid` no processo pai para esperar o término do processo filho e da execução de alguma função da família `exec` ativando o interpretador ou o comando no processo filho criado.

A função `system` faz esta tarefa automaticamente. A função irá retornar `-1` caso não seja possível efetuar o `fork` ou o `waitpid` devido a algum problema no sistema e irá retornar `127` caso o `exec` não tenha encontrado o comando especificado ou o programa indicado esteja com problemas de permissão.

Veja o exemplo:

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

void main (int argc, char *argv[])
{
    int iStatus;

    printf("\nRelação de arquivos do diretório /home\n");
    iStatus = system ("ls /home");
    if(iStatus < 0)
    {
        perror(argv[0]);
        exit(errno);
    }
    printf("\ncomando terminou com status |%d|\n", iStatus);

    exit(0);
}
```

Execução do comando ls.

### Programa 20.5

#### Resultado do Programa 20.5

```
Relação de arquivos do diretório /home
aluteste      espec  grad      mest  prof      seq
aquota.user  ftp    lost+found old    projeto
```

```
comando terminou com status |0|
```



21

## Tratamento de Sinais em Linux e Unix

*A tecnologia moderna é capaz de realizar a produção sem emprego.*

*O diabo é que a economia moderna não consegue inventar o consumo sem salário.*

Betinho, sociólogo brasileiro

### 21.1 Conceito e tratamento de Sinais

O sistema operacional Linux e Unix permite que se enviem sinais a processos. Estes sinais são tratados pelo processo como sendo interrupções de software e possuem sempre um número associado ao mesmo. Como este número é diferente entre as diversas versões de Unix e Linux, os nomes de sinais e seu tratamento foram padronizados. Estes nomes estão definidos no arquivo `sys/signal.h` e sempre começam com as letras `SIG` seguidas de um mnemônico relativo ao sinal. Por exemplo, o sinal `SIGKILL` significa o sinal de cancelamento do processo.

Os sinais enviados a um processo ocorrem de maneira assíncrona, ou seja, não existe certeza da ocorrência e da hora de recebimento de um sinal, podendo ocorrer em qualquer ponto do programa. Este fato exige um certo cuidado no desenho da aplicação.

Os sinais podem ser gerados para um processo de diversas fontes. São elas:

- o Terminal – Ao se pressionarem certas teclas do terminal, o `device handler` do mesmo envia um determinado sinal ao processo atual que está lendo do terminal. Na grande maioria dos sistemas estas teclas são configuradas pelo comando `stty`.

- o Problemas de hardware – Alguns problemas acontecidos no hardware são interceptados pelo *kernel*, que envia o sinal correspondente para o processo que causou o erro. São exemplos de erro desta categoria a divisão por zero, endereço de memória inválido, instrução ilegal etc.
- o Função `kill` – Usando a função de nível 2 do *kernel*, pode-se enviar um sinal de um processo para um processo ou para um grupo de processos. Tecnicamente, pode-se enviar qualquer sinal válido de um processo para outro.
- o Comando `kill` – Com este comando pode-se também enviar da linha de comando qualquer sinal para um processo ou para um grupo de processos. Veja o manual *on-line* do sistema sobre o funcionamento do mesmo e a relação de sinais disponíveis.
- o Problemas de software – Existem certas condições que ocorrem em algum módulo do sistema operacional que caracterizam problemas de software. Quando ocorrem o *kernel* gera o sinal respectivo para o processo causador dos problemas. São exemplos de problemas de software: tentativa de gravar em um `pipe` que não está aberto pelo processo de leitura, problemas de invasão de área em dados lidos de redes etc.

Quando o kernel envia um sinal a um processo, o processo pode indicar que se execute uma das ações seguintes:

- o Ignorar o sinal – Neste caso, o processo indica que o sinal não será tratado, devendo ser descartado pelo *kernel*. O processo continua o seu processamento. Existem sinais que não podem ser ignorados pelo processo.
- o Processar o sinal – Nesta situação, o processo instala uma função que será responsável pelo tratamento do sinal. Esta função é chamada de `signal handler`. É possível colocar funções específicas para cada sinal ou se implementar uma única função que fará o tratamento de todos os sinais recebidos. Existem sinais que não podem ser tratados pelo processo.
- o Aplicar a ação *default* – Neste caso, o processo não instala nenhum `signal handler` nem indica para o *kernel* ignorar o sinal. O próprio *kernel* irá determinar a ação a ser aplicada na ocorrência de determinado sinal. Cada sinal tem a sua ação *default* especificada.

Como cada sinal tem uma ação *default* configurada e, na maioria dos casos, esta ação é padronizada para os diversos sistemas; deve-se verificar a ação *default* configurada para o sinal na documentação do sistema operacional. As possíveis ações padrões a serem aplicadas sobre os sinais são:

- o O processo é terminado – O processo que recebe o sinal é terminado. As rotinas de `cleanup` são executadas normalmente.

- o O processo é terminal com geração de `core dump` – Além de o processo ser terminado, será feita a geração do arquivo de `core dump` representando a imagem do processo salva em disco. O `core dump` é gravado em um arquivo chamado `core`.
- o Não é colocada a ação `SIG_DFL` depois da chamada da função de `signal handler` – O sistema operacional, quando desvia a execução do programa para um `signal handler` volta a colocar a ação `default` para o sinal ocorrido. Devido a isto, sempre coloca-se como primeiro comando do `signal handler` a função para reinstalar o `signal handler` novamente. Acontece que alguns sinais do *kernel* não desligam a ação do `signal handler` na ocorrência do sinal, ficando o mesmo instalado no processo.
- o O sinal não pode ser ignorado – Alguns sinais não podem ser ignorados pelo processo. Isto é feito para que sempre se tenha a possibilidade de cancelar um processo, indiferente do mesmo tentar ignorar o sinal. Como exemplo de sinais que não podem ser ignorados temos `SIGKILL` e `SIGSTOP`.
- o Não se pode instalar um `signal handler` – Não se pode instalar uma função `signal handler` para o sinal. Geralmente estes sinais também não podem ser ignorados pelo sistema.
- o Para processos parados, o sinal é perdido – Nesta ação, caso o processo esteja parado, o sinal enviado para o mesmo não será guardado para posterior processamento.
- o O sinal é ignorado pelo sistema – Esta ação indica que naturalmente o sinal já é ignorado pelo sistema.
- o O processo é paralisado (*stopped*) – A ação deste tipo de sinal é deixar o processo em estado parado, ou seja, o processo continua ativo mas não está rodando. O processo só voltará a rodar caso receba um sinal indicando o reinício do mesmo.

## 21.2 Alguns Sinais

- o `SIGKILL` – Este sinal não pode ser ignorado nem se pode instalar uma função `signal handler` para o mesmo. Como a ação `default` deste processo é terminar o processo, ele disponibiliza uma maneira segura de se cancelar qualquer processo.
- o `SIGTERM` – Este sinal existe para implementar a morte programada de um processo. O sinal pode ser interceptado e/ou ignorado. Ele é o sinal `default` enviado pelo comando `kill`. Geralmente é colocada uma rotina de finalização neste sinal, onde o processo realiza o seu término de maneira organizada. A ação `default` para o sinal é terminar o processo.

- o `SIGSTOP` – Quando o processo recebe este sinal, ele ficará parado no sistema, em situação de *stopped*. Este sinal não pode ser ignorado pelo processo, nem é possível instalar uma função `signal handler` para o mesmo.
- o `SIGCONT` – Este sinal indica para um processo parado voltar a rodar no sistema, mudando o processo para a situação de *running*. A ação *default* para este sinal é ser ignorado pelo processo que o recebe, a menos que o mesmo esteja em situação *stopped*.

Para maiores informações sobre outros sinais, consulte o manual *on-line* do sistema para ver o comando `kill`. Para maiores informações sobre *status* de processos, veja o comando `ps`.

## 21.3 Função `signal`

Sintaxe:

```
void (*signal(int sig, void(*action)(int)))(int);
```

Para instalar uma função `signal handler` ou indicar para o processo ignorar um sinal, deve-se usar a função `signal`. A função irá receber no primeiro parâmetro o nome do sinal sobre o qual será efetuado o processamento.

Para instalar um `signal handler`, deve-se indicar, no segundo parâmetro, a função que deverá ser executada na ocorrência do sinal. Esta função deve ser definida como retornando `void` e tendo um parâmetro `int`. Quando ocorre um sinal que possua `signal handler` instalado, na ocorrência deste sinal, a função é executada, sendo colocado o número do sinal no parâmetro da função. Devido a isto, é possível instalar uma única função para diversos sinais, bastando verificar o número do mesmo no parâmetro para se tomar as devidas providências.

A função `signal` retorna o endereço da função previamente instalada, para o caso de precisar restaurar a função anterior. Caso não seja possível instalar um `signal handler` para um sinal, a função irá retornar um erro (valor `-1`). Para que não haja problemas no teste deste valor, deve-se sempre testar contra a constante `SIG_ERR` definida no arquivo `sys/signal.h`.

Para ignorar um sinal, deve-se passar como segundo parâmetro a constante `SIG_IGN` e para voltar o processamento default para um sinal, deve-se informar a constante `SIG_DFL`. Estas constantes estão definidas no arquivo `sys/signal.h`.

Veja o exemplo:

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

void signal_handler_function(int iSignal )
{
    fprintf(stderr, "\nRecebido o sinal %d.", iSignal);
}

void main(void)
{
    int i;

    for (i = 1; i <= SIGMAX; i++)
    {
        if (signal (i, signal_handler_function) == SIG_ERR)
        {
            fprintf(stderr, "\nErro na instalacao da funcao signal han-
            dler\n");
            fprintf(stderr, "Sinal : %d.\n", i);
        }
    }

    while(1)
        sleep(1);

    exit (0);
}
```

Função que vai tratar os sinais. Para tratar cada sinal, basta utilizar as constantes definidas.

Número máximo de sinais que o sistema operacional tem disponível.

Registro da chamada da função para cada sinal.

Loop infinito. Aguardando envio de sinais...

### Programa 21.1

#### Resultado do Programa 21.1

```
Erro na instalacao da funcao signal handler
Sinal : 9.

Erro na instalacao da funcao signal handler
Sinal : 17.

Recebido o sinal 4.
Recebido o sinal 7.
Recebido o sinal 1.
Recebido o sinal 15.Killed
```

Estes sinais não podem ser tratados pelo programa.

Programa recebeu sinal 9 (KILL).

#### o Verificando PID do programa (ps -u laureano)

```
UID      PID     TTY     TIME CMD
10004    75168 pts/10   0:00 p21_1
10004    291286 pts/11   0:00 bash
```



```
10004 323240 pts/10  0:00 bash
10004 411648 pts/11  0:00 ps
```

o Enviando sinais para o programa.

```
$> kill -4 75168
$> kill -7 75168
$> kill -HUP 75168
$> kill -TERM 75168
$> kill -9 75168
```

Matando o programa...

## 21.4 Função kill

Sintaxe:

```
int kill(pid_t pid, int sig);
```

A função `kill`, apesar do nome, serve para enviar um sinal para um determinado processo ou para um grupo de processos. O sinal só será entregue no processo destino caso exista permissão para isto. Geralmente um processo com permissão de `root` pode enviar qualquer sinal para qualquer processo do sistema. Processos de usuários normais poderão enviar sinais somente para processos pertencentes ao mesmo usuário.

Na função deve-se indicar o nome do sinal, definido no arquivo `sys/signal.h` e o número do processo (`PID`) que irá receber o sinal.

Caso o primeiro parâmetro seja:

- o `> 0` – O sinal é enviado para o processo cujo `PID` é indicado no parâmetro.
- o `= 0` – O sinal é enviado para todos os processos cujo `process group ID` seja igual ao do processo que chama a função.
- o `< -1` – O sinal é enviado para todos os processos cujo `process group ID` seja idêntico ao valor absoluto do parâmetro.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>
```

## 194 ♦ Programando em C para Linux, Unix e Windows

```
void signal_handler_function(int iSignal)
{
    if( iSignal == SIGUSR1)
    {
        if (signal (iSignal, signal_handler_function) == SIG_ERR)
        {
            fprintf (stderr, "\nErro na reinstalacao de signal han-
dler\n");
            fprintf (stderr, "Sinal : %d.\n", iSignal);
        }
        printf ("\nO processo filho digitou um numero par\n");
    }
    else if( iSignal == SIGCHLD )
    {
        int iStatus;
        printf ("\nO processo filho terminou o seu processamento\n");

        wait (&iStatus);
        if(WIFEXITED(iStatus))
        {
            printf("\nTermino normal : |%d|\n", WEXITSTATUS(iStatus));
        }
        else if (WIFSIGNALED(iStatus))
        {
            printf("\nCancelado por sinal : |%d|\n", WTERMSIG(iStatus));
            exit(0);
        }
    }
}

void main(int argc, char **argv )
{
    int iPid;

    if (signal(SIGUSR1, signal_handler_function) == SIG_ERR ||
        signal(SIGCHLD, signal_handler_function) == SIG_ERR)
    {
        perror (argv[0]);
        exit (errno);
    }

    printf("\nCriando um processo filho\n");
    iPid = fork();
    if(iPid < 0)
    {
        perror (argv[0]);
        exit (errno);
    }
    if (iPid != 0)
    {
        while (1)
        {
            sleep (2);
        }
    }
}
```

Se for sinal de usuário...

Reinstala a função para as próximas chamadas...

O pai verifica o filho....

...o término pode ser digitando 0 no filho....

...ou filho recebe um sinal.

Instala a função para tratar sinais. Pode ser utilizada uma função para cada sinal.

Pai em loop infinito....

```

if (iPid == 0)
{
    int iVlr;
    printf("\nExecutando o processo filho\n");
    while(1)
    {
        printf ("Digite um numero (0 p/ terminar) : ");
        scanf ("%d", &iVlr);
        if (iVlr == 0)
            break;
        if ((iVlr % 2) == 0)
        {
            if (kill(getppid(), SIGUSR1) < 0)
            {
                perror(argv[0]);
                exit(errno);
            }
        }
    }
    exit(0);
}

```

Pegando o PID do pai e enviando um sinal para ele...

## Programa 21.2

### Resultado do Programa 21.2

```

$> p21_2
Criando um processo filho
Executando o processo filho

Digite um numero (0 p/ terminar) : 2
Digite um numero (0 p/ terminar) :
O processo filho digitou um numero par
3
Digite um numero (0 p/ terminar) : 3
Digite um numero (0 p/ terminar) : 4
Digite um numero (0 p/ terminar) :
O processo filho digitou um numero par
5
Digite um numero (0 p/ terminar) : 6
Digite um numero (0 p/ terminar) :
O processo filho digitou um numero par
O processo filho terminou o seu processamento
Cancelado por sinal : |15|

```

### o Verificando PID e PPID do programa (ps -fu laureano)

UID	PID	PPID	C	STIME	TTY	TIME	CMD
laureano	26836	323240	0	10:20:19	pts/10	0:00	p21_2
laureano	194340	291286	11	10:21:06	pts/11	0:00	ps -fu laureano
laureano	197000	26836	0	10:20:19	pts/10	0:00	p21_2

## 196 ♦ Programando em C para Linux, Unix e Windows

```
laureano 291286 236008 1 08:36:15 pts/11 0:00 bash
laureano 323240 114400 0 08:36:04 pts/10 0:00 bash
```

o Enviando sinal 15 para o programa

```
$> kill -TERM 197000
```

## 21.5 Função raise

Sintaxe:

```
int raise(int sig);
```

A função `raise` serve para enviar um sinal para o próprio processo. Indica-se no parâmetro o número ou o nome do sinal a ser enviado para o processo. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/wait.h>
#include <signal.h>

void signal_handler_function(int iSignal)
{
    if( iSignal == SIGUSR1)
    {
        if(signal(iSignal, signal_handler_function) == SIG_ERR)
        {
            fprintf(stderr, "\nErro na reinstalacao de signal handler\n");
            fprintf(stderr, "Sinal : %d.\n", iSignal);
        }
        printf("\nO processo digitou um numero par\n");
    }
    else
    {
        printf("\nO processo terminou o seu processamento\n");
        printf("\nSinal: |%d|\n", iSignal );
        exit(0);
    }
}

void main(int argc, char **argv )
{
    int iVlr;
    if (signal(SIGUSR1, signal_handler_function) == SIG_ERR ||
        signal(SIGTERM, signal_handler_function) == SIG_ERR)
    {
```

```

        perror(argv[0]);
        exit(errno);
    }

    while(1)
    {
        printf ("Digite um numero (0 p/ terminar) : ");
        scanf ("%d", &iVlr);
        if (iVlr == 0)
            break;
        if ((iVlr % 2) == 0)
        {
            if( raise(SIGUSR1) < 0)
            {
                perror(argv[0]);
                exit(errno);
            }
        }
    }
    exit(0);
}

```

Envio do sinal SIGUSR1 para o próprio processo.

### Programa 21.3

#### Resultado do Programa 21.3

```

Digite um numero (0 p/ terminar) : 4

O processo digitou um numero par
Digite um numero (0 p/ terminar) : 5
Digite um numero (0 p/ terminar) : 1
Digite um numero (0 p/ terminar) :
O processo terminou o seu processamento
Sinal: |15|

```

#### o Verificando o PID (ps -u laureano)

UID	PID	TTY	TIME	CMD
10004	110578	pts/10	0:00	ps
10004	233598	pts/11	0:00	p21_3
10004	291286	pts/11	0:00	bash
10004	323240	pts/10	0:00	bash

#### o Enviado o sinal 15 para o processo

```
$> kill -TERM 233598
```

## 21.6 Função sleep

Sintaxe:

```
unsigned int sleep(unsigned int seconds);
```

A função `sleep` deixa o processo suspenso durante a quantidade de segundos informada no parâmetro. Este tempo que o processo permanece parado, devido ao escalonamento do *kernel*, pode ser um pouco maior que o informado no parâmetro.

Caso o processo receba um sinal enquanto esteja suspenso pela função `sleep`, o processo volta a rodar e é realizada a execução da função `signal handler` correspondente ao sinal ocorrido. No término da função, caso a mesma não termine o processo, a função `sleep` termina, devolvendo em seu parâmetro a quantidade de segundos ainda restantes no despertador.

Veja o exemplo:

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

int main (void)
{
    time_t hora;
    hora = time(NULL);

    printf ("Numero de segundos antes  : %d\n", hora);
    printf ("Dormindo 5 segundos\n");

    sleep(5);

    time(&hora);

    printf ("Numero de segundos depois : %d\n", hora);
    exit(0);
}
```

### Programa 21.4

#### Resultado do Programa 21.4

```
Numero de segundos antes  : 1116856612
Dormindo 5 segundos
Numero de segundos depois : 1116856617
```

## 21.7 Cuidados com algumas funções (funções reentrantes)

O processamento de uma função `signal handler` ocorre de maneira assíncrona e a interrupção no fluxo de execução pode acontecer durante a execução de alguma função do próprio kernel, portanto, pode-se ter problemas de reentrância caso a própria função `signal handler` execute a mesma função do *kernel* que foi interrompida e a mesma faça uso de alguma variável estática para controle interno. Por exemplo, a função `signal handler` foi executada durante um comando `strcpy` e a função `signal handler` executa outro comando `strcpy`.

Então, deve-se sempre verificar se a situação descrita não ocorre e se as funções não reentrantes não são utilizadas dentro de funções `signal handler`.

Segue uma relação de funções que podem ser usadas dentro de funções `signal handlers` sem problemas, pois possuem a característica de reentrância necessária.

```
_exit(), access(), alarm(), chdir(), chmod(), chown(), close(),  
creat(), dup(), dup2(), execl(), execve(), exit(), fcntl(), fork(),  
fstat(), getegid(), geteuid(), getgid(), getpid(), getppid(),  
getuid(), kill(), link(), lseek(), mkdir(), mkfifo(), open(),  
pause(), pipe(), read(), rename(), rmdir(), setuid(), setgid(),  
sleep(), stat(), sysconf(), time(), times(), umask(), uname(),  
unlink(), utime(), wait(), waitpid(), write().
```



22

## Daemons (Serviços) em Linux e Unix

*Eu viverei para sempre ou morrerei tentando.*  
Spider Robinson, escritor canadense

### 22.1 Conceito de daemon

Em várias situações é preciso que um processo fique rodando continuamente ("eternamente") em uma máquina. A estes processos dá-se o nome de *daemons* (ou *serviços* no Windows).

Os *daemons* apresentam as seguintes características:

- o Geralmente são programas que devem ser iniciados assim que o sistema operacional entra no ar. Coloca-se a chamada dos mesmos nos arquivos de configuração para que eles entrem no ar automaticamente durante o processo de boot do sistema.
- o Um daemon só deve ser cancelado quando o sistema operacional está encerrando o seu processamento. O daemon fica rodando enquanto o sistema estiver no ar.
- o São processos que rodam em *background* e não devem ter um terminal associado a eles.

### 22.2 Regras para codificação de um daemon

Para se codificar um *daemon* deve-se realizar uma série de tarefas e chamadas de funções para que o processo se comporte como um *daemon*.



Por questão de facilidade de programação e buscando a modularidade, esses passos são geralmente colocados em uma função chamada `daemon_init` criada dentro do programa e chamada no início do programa na função `main`.

Os passos a serem realizados são:

- o A primeira coisa a fazer no processo é chamar a função `fork` para duplicar o processo atual e terminar o processo pai. Esta duplicação causa uma série de efeitos colaterais desejáveis em um *daemon*. Inicialmente, o término do processo pai libera o *shell*, pois o mesmo acha que o comando terminou. Segundo, o processo filho herda o `Process Group ID` do pai mas cria um novo `Process ID`, garantindo que este processo não será um processo líder de grupo.
- o Deve-se chamar a função `setsid` para criar uma nova sessão. Com a criação de uma nova sessão, o processo filho torna-se o líder da sessão, torna-se o líder do grupo de processos e não irá possuir um terminal de controle.
- o Deve-se trocar o diretório atual para o diretório raiz ou para um diretório específico. Este diretório preferencialmente não deve ser um diretório montado depois do processo de *boot*.
- o Deve-se mudar a máscara de criação do processo para 0 usando a função `umask`. Isto possibilita o processo *daemon* criar arquivos com a permissão desejada. Caso não se chame esta função, pode ser que o processo *daemon* herde alguma máscara que esteja desabilitando alguma permissão necessária para o funcionamento do *daemon*.
- o Todos os descritores de arquivos que não serão utilizados pelo processo *daemon* devem ser fechados. Isto previne que o *daemon* segure algum descritor herdado aberto. O processo *daemon* deve selecionar quais descritores devem ser fechados de acordo com a lógica do mesmo. Geralmente o *daemon* fecha todos os descritores antes de abrir qualquer arquivo, garantindo assim que somente os arquivos necessários ficarão abertos durante o tempo de vida do *daemon*.
- o Caso, durante a vida do processo *daemon*, ele precise ler algum arquivo de configuração ou mudar sua configuração interna, deve-se instalar um `signal handler` para o sinal `SIGHUP`, pois como o processo está desconectado de qualquer terminal, ele naturalmente nunca receberá este sinal do sistema operacional.

## 22.3 Função `setsid`

Sintaxe:

```
pid_t setsid(void);
```

Cada processo possui um `Process ID` que identifica unicamente o mesmo no sistema. Adicionalmente, um processo possui um `Process Group ID` que indica a qual grupo de processo ele pertence.

Um grupo de processos é uma coleção de um ou mais processos com um único `Process Group ID`. Cada grupo de processo possui um processo chamado de processo líder. O processo líder é identificado como sendo o processo que possui o `Process ID` igual ao `Process Group ID`. Por exemplo, todos os processos ligados por um *pipeline* irão pertencer ao mesmo grupo de processos.

Pode-se agrupar um ou mais grupos de processos em uma sessão. Tipicamente todos os processos iniciados durante uma sessão de um *shell* irão pertencer à mesma sessão de processos.

O conceito de grupo de processo pode ser usado na função `signal`, que permite que se mande um sinal para todos os processos pertencentes a um grupo e na função `waitpid` que permite que o processo pai receba o código de retorno de qualquer processo filho pertencendo a um grupo de processos.

A função `setsid`, quando chamada de um processo, irá realizar a seguinte tarefa em relação à sessão e ao grupo de processos:

- o É criada uma nova sessão de processos. O processo que chamou a função é o único integrante desta sessão. O processo, portanto, se torna líder da sessão.
- o O processo torna-se também líder do grupo de processos. O `Process Group ID` do grupo fica sendo o `Process ID` do processo.
- o O processo não terá um terminal de controle associado a ele. Esta característica é importante para o processo *daemon*.

Veja o exemplo:

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

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
```

} Headers ou includes Necessários para o *daemon*

```
int daemon_init (void)
{
    pid_t iPid;
    long iMaxFd;
    int i;

    if ((iPid = fork()) < 0)
        return -1;

    if (iPid != 0)
        exit(0);

    setsid();

    chdir ("/");

    umask (0);

    iMaxFd = sysconf (_SC_OPEN_MAX);
    for (i=0; i < iMaxFd; i++)
        close (i);

    return 0;
}
```

Rotina usada para transformar o processo em um processo *daemon*. O nome *daemon\_init* é uma convenção adotada, sua função pode ter qualquer nome.

1º Passo – Duplicar o processo usando *fork*. O processo pai é encerrado.

2º Passo – Chamar a função *setsid* para criar uma nova sessão de processo, ficando o processo filho como líder da sessão e sem um terminal de controle associado ao processo.

3º Passo – Troca-se o diretório atual para o diretório raiz (*root*) ou para um diretório próprio do *daemon*.

4º Passo – Inicializa a máscara padrão de criação de arquivos

5º Passo – Fechando todos os descritores existentes no sistema. Utiliza-se a informação de número máximo de descritores configurado no sistema e obtido com a função *sysconf*. Para outras opções da função *sysconf*, veja o manual *on-line* do sistema.

```
void main (int argc, char *argv[])
{
    int iFd;
    char szBuffer[100];
    int i;

    if (daemon_init () < 0)
    {
        perror (argv[0]);
        exit (errno);
    }
}
```

Chamando a função para transformar o processo em *daemon*.

## 204 ♦ Programando em C para Linux, Unix e Windows

```
sprintf (szBuffer, "/tmp/daemon%d.arq", getpid());
iFd = open (szBuffer, O_CREAT | O_WRONLY, 0700);

i = 1;
while (1)
{
    sleep(3);
    sprintf(szBuffer, "Esta eh a linha de numero %04d\n", i++);
    write(iFd, szBuffer, strlen (szBuffer));
}

exit (0);
}
```

Para fins de testes, o programa abre um arquivo e de 3 em 3 segundos grava uma linha no arquivo.

### Programa 22.1

#### Resultados obtidos após a execução do Programa 22.1

o Resultado do comando `ps -fu laureano`:

UID	PID	PPID	C	STIME	TTY	TIME	CMD
laureano	42576	162230	0	07:41:28	pts/18	0:00	bash
laureano	162230	488506	0	15:16:51	pts/18	0:00	-ksh
laureano	175160	164140	0	07:45:43	pts/5	0:00	vi p22_1.c
laureano	293142	42576	11	08:04:45	pts/18	0:00	ps -fu laureano
laureano	246908	1	0	07:44:06	-	0:00	p22_1

O pai do *daemon* é o processo `init` do sistema operacional.

Nenhum terminal de controle associado ao *daemon*.

o Arquivo gerado:

```
$> ls -l /tmp/daemon246908.arq
-rwx----- 1 laureano prof 527 May 17 07:44 /tmp/daemon246908.arq
```

o Trecho do arquivo gerado:

```
$> cat /tmp/daemon246908.arq
Esta eh a linha de numero 0001
Esta eh a linha de numero 0002
Esta eh a linha de numero 0003
Esta eh a linha de numero 0004
Esta eh a linha de numero 0005
Esta eh a linha de numero 0006
```

## 22.4 Registrando erros com a função syslog

Sintaxe:

```
int syslog(int priority, const char *msg, ...);
```

A função `syslog` entrega uma mensagem para o programa `syslogd` instalado no sistema. A mensagem será entregue conforme a sua prioridade e a configuração do arquivo `/etc/syslog.conf`. Antes de utilizar esta função, consulte a documentação do sistema para entender o funcionamento e a configuração do `syslogd` (`man syslog` e `man syslog.conf`).

Como um processo *daemon* não possui um terminal associado a ele (por necessidade do próprio *daemon*), as mensagens de erro que o *daemon* emitir devem ser gravadas por outros mecanismos, sendo a função `syslog` uma alternativa.

O primeiro parâmetro da função indica o tipo da mensagem (destino) e a prioridade. Devem ser usadas as seguintes constantes:

Destino/Tipo de Mensagem	Descrição
LOG_KERN	Mensagens do kernel do sistema operacional
LOG_USER	Mensagens de usuário
LOG_MAIL	Mensagens do sistema de mail
LOG_DAEMON	Mensagens geradas por daemons
LOG_AUTH	Mensagens de autorização e segurança
LOG_SYSLOG	Mensagens geradas internamente pelo syslog
LOG_LPR	Sistema de Impressão
LOG_NEWS	Novidades no sistema
LOG_UUCP	Mensagens do Sistema UUCP
LOG_CRON	Mensagens do daemon cron
LOG_LOCAL0 até LOG_LOCAL7	Mensagens para uso local e por aplicações

Prioridade	Descrição
LOG_EMERG	Emergência
LOG_ALERT	Condição muito crítica
LOG_CRIT	Condição crítica
LOG_ERR	Condição de erro
LOG_WARNING	Aviso significativo
LOG_NOTICE	Aviso normal
LOG_INFO	Informação
LOG_DEBUG	Mensagem para debug

A combinação para a utilização destas constantes se dá através do OU binário (`|`). Por exemplo, uma mensagem informativa para uso local poderia ser `syslog( LOG_LOCAL0 | LOG_INFO, "teste" )`.

## 206 ♦ Programando em C para Linux, Unix e Windows

Os demais parâmetros da função possuem a mesma característica dos parâmetros da função `printf`, ou seja, coloca-se um formato de mensagem seguido de campos com os valores a serem utilizados neste formato.

O formato aceita a notação “%” seguido de uma letra da mesma maneira que a função `printf`. Adicionalmente aos formatos da `printf`, a função `syslog` aceita o formato “%m” representando diretamente a mensagem de erro acontecida no processo (Similar a `perror`).

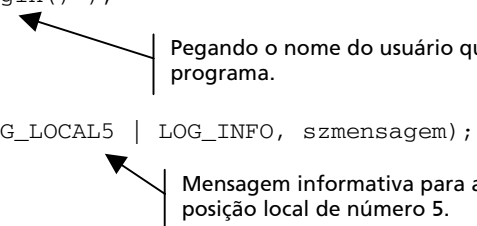
Veja o exemplo:

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

void main(void)
{
    char szmensagem[200];

    sprintf(szmensagem, "O usuario %s executou o programa do sys-
log.", getlogin() );

    syslog(LOG_LOCAL5 | LOG_INFO, szmensagem);
}
```



Pegando o nome do usuário que executou o programa.

Mensagem informativa para a posição local de número 5.

### Programa 22.2

#### Resultado do programa 22.2

o O arquivo `/etc/syslog.conf` foi alterado para incluir a seguinte linha:

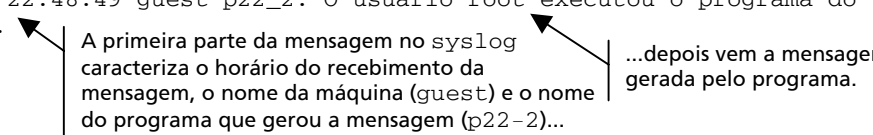
```
local5.* /var/log/teste.log
```

o Após alterar o arquivo `/etc/syslog.conf`, é necessário reiniciar o serviço do `syslog`; uma forma é enviar um sinal para o processo do `syslogd` pedindo a reconfiguração.

```
$> kill -HUP <pid_syslogd>
```

o O arquivo `/var/log/teste.log`, após a execução do programa, conterá a seguinte linha:

```
May 17 22:48:49 guest p22_2: O usuario root executou o programa do
syslog.
```



A primeira parte da mensagem no syslog caracteriza o horário do recebimento da mensagem, o nome da máquina (guest) e o nome do programa que gerou a mensagem (p22-2)...

...depois vem a mensagem gerada pelo programa.



*Quem não se comunica se trumbica.*  
Chacrinha, apresentador brasileiro

## 23.1 Função socket

Sintaxe:

```
int socket(int domain, int type, int protocol);
```

A função `socket` cria um ponto de comunicação e retorna um descritor para um arquivo ou `-1` se houve algum erro (como as funções `creat` e `open`).

Deve-se passar o domínio da comunicação (tipo da comunicação). Normalmente, para comunicações TCP/IP utiliza-se `AF_INET` neste campo. O tipo da comunicação (TCP ou UDP), para comunicação TCP utiliza-se `SOCK_STREAM` e para UDP `SOCK_DGRAM`. O campo protocolo identifica um protocolo em particular que se deseja utilizar. Normalmente é passado 0 (zero) neste campo.

Os tipos e domínios de comunicação estão descritos em `sys/types.h` e `sys/socket.h`.

Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
```

```

void main(void)
{
    int iSock;
    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }
}

```

← Criação do ponto de comunicação...

Programa 23.1

## 23.2 Estrutura sockaddr

Para a programação `socket`, foram definidas estruturas padrão com os parâmetros que devem ser repassados para as demais funções. Para programas TCP/IP utiliza-se a estrutura `sockaddr_in`. Definida da seguinte forma:

```

struct sockaddr_in {
    short int sin_family;
    unsigned short int sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
};

```

← Família do endereço.  
 ← Número da porta.  
 ← Endereço TCP/IP.  
 ← Complemento da estrutura `sockaddr`.

Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
}

```

← Porta de comunicação, normalmente acima de 1024.



```

my_addr.sin_addr.s_addr = INADDR_ANY;
bzero(&(my_addr.sin_zero), 8);
}

```

Programa 23.2

Preenche com o endereço local.

Preenchendo com 0 os *bytes* não utilizados da estrutura.

## 23.3 Funções htonl, htons, ntohl, ntohs

Sintaxe:

```

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);

```

As funções convertem e retornam um endereço passado como parâmetro para um ordenamento de byte significativo. Sendo que as funções `htons` e `htonl` retornam o valor na ordem de *bytes* da rede e as funções `ntohs` e `ntohl` retornam o valor na ordem de *bytes* de um *host*.

- o `htons` – Host to Network Short
- o `htonl` – Host to Network Long
- o `ntohs` – Network to Host Short
- o `ntohl` – Network to Host Long

Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
}

```

## 210 ♦ Programando em C para Linux, Unix e Windows

```
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);
}
```

### Programa 23.3

## 23.4 Função bind

Sintaxe:

```
int bind(int sockfd, const struct sockaddr *my_addr,
socklen_t addrlen );
```

A função `bind` associa o *socket* criado à porta local do sistema operacional. Nesta associação é verificado se a porta já não está sendo utilizada por algum outro processo. Será através desta associação (porta) que o programa irá receber dados (*bytes*) de outros programas. É passada para a função a estrutura criada anteriormente, assim como o *socket* criado. A função `bind` retorna 0 (zero) em caso de sucesso e -1 em caso de erro. Normalmente, a função `bind` é utilizada no lado *server* da aplicação. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if( bind(iSock, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1)
    {
        perror("bind:");
        exit(1);
    }
}
```

```

    }
}

```

Programa 23.4

## 23.5 Funções `inet_aton`, `inet_addr` e `inet_ntoa`

Sintaxe:

```

int inet_aton(const char *cp, struct in_addr *inp);
in_addr_t inet_addr(const char *cp);
char *inet_ntoa(struct in_addr in);

```

A função `inet_aton` converte o endereço passado (inclusive com pontos) para uma estrutura de endereços (binário) válido. Retorna um valor maior que 0 (zero) se a conversão ocorreu ou 0 (zero) se houve algum erro.

A função `inet_addr` converte o endereço passado (inclusive com pontos) para um valor binário (ordenado) em bytes.

A função `inet_ntoa` realiza a operação inversa de `inet_aton`. A partir de um valor binário (estrutura) ela retorna o endereço em formato *string* (inclusive com pontos). Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void main(void)
{
    int iSock;
    struct sockaddr_in dest_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(4950);
    dest_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bzero(&(dest_addr.sin_zero), 8);
}

```

Converte o endereço destino passado para o formato ordenado de *bytes* (binário).

Programa 23.5

## 23.6 Função connect

Sintaxe:

```
int connect(int sockfd, const struct sockaddr *serv_addr,
socklen_t addrlen);
```

A função `connect` inicia uma conexão *socket* do lado do cliente, não sendo necessário associar uma parte no cliente. Na estrutura passada são fornecidas as informações relacionadas ao servidor (destino). A função retorna 0 (zero) se a conexão foi bem sucedida ou -1 se houve erro. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void main(void)
{
    int iSock;
    struct sockaddr_in dest_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(4950);
    dest_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bzero(&(dest_addr.sin_zero), 8);
    if( connect (iSock, (struct sockaddr *)&dest_addr, sizeof(struct
sockaddr)) < 0)
    {
        perror("connect:");
        exit(1);
    }
}
```

← Tenta a conexão no servidor.

Programa 23.6

## 23.7 Função listen

Sintaxe:

```
int listen(int s, int backlog);
```

Após o socket (função `socket`) ter sido criado e uma porta associada (função `bind`) é necessário habilitar o `socket` para receber as conexões. A função `listen` faz justamente este papel, ou seja, habilita que o programa servidor receba conexões de um programa cliente. Deve-se passar o descritor do socket aberto e a quantidade de conexões que podem ficar pendentes até que o programa trate todas as conexões anteriores. A função retorna 0 (zero) em caso de sucesso e -1 em caso de erro. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if( bind(iSock, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1)
    {
        perror("bind:");
        exit(1);
    }

    if( listen( iSock, 10 ) < 0)
    {
        perror("listen:");
        exit(1);
    }
}
```

Habilita o servidor para receber conexões.  
O valor 10 indica que podem existir até 10  
requisições conexões na fila de espera

```
    }
}
```

Programa 23.7

## 23.8 Função `accept`

Sintaxe:

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

Após ter utilizado a função `listen` para habilitar as conexões, é necessário aceitar as conexões. A função `accept` aceita as conexões efetuadas pelos clientes. Deve ser passado para a função o *socket* abertos a estrutura que irá receber os dados do cliente e o tamanho do endereço. A função irá retornar um descritor para a conexão aceita ou `-1` se houve erro. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if( bind(iSock, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1)
    {
        perror("bind:");
        exit(1);
    }

    if( listen( iSock, 10 ) < 0)
```

```

{
    perror("listen:");
    exit(1);
}

while(1)
{
    int iFd;
    struct sockaddr_in client_addr;
    socklen_t sin_size;

    sin_size = sizeof(struct sockaddr_in);

    if( (iFd = accept(iSock, (struct sockaddr *) &client_addr,
    &sin_size)) < 0)
    {
        perror("accept:");
        exit(1);
    }
}

```

Aceita conexões eternamente.  
 Estrutura que terá as informações do programa cliente.  
 Tamanho do endereço.  
 Aceitando conexões...

Programa 23.8

## 23.9 Função send

Sintaxe:

```
ssize_t send(int s, const void *buf, size_t len, int flags);
```

A função `send` é utilizada para enviar uma mensagem para outro *socket*. Para o envio de mensagens, também pode ser utilizada a função `write` (o mesmo que passar o valor 0 em flags da função `send`). A função retorna o número de *bytes* enviados ou `-1` se houve erro.

A função `send` é utilizada em conexões TCP (*stream*) ou orientada à conexão.

Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

```

## 216 ♦ Programando em C para Linux, Unix e Windows

```
void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if( bind(iSock, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1)
    {
        perror("bind:");
        exit(1);
    }

    if( listen( iSock, 10 ) < 0)
    {
        perror("listen:");
        exit(1);
    }
    while(1)
    {
        int iFd;
        struct sockaddr_in client_addr;
        socklen_t sin_size;
        char szMensagem[100];

        sin_size = sizeof(struct sockaddr_in);

        if( (iFd = accept(iSock, (struct sockaddr *) &client_addr,
&sin_size)) < 0)
        {
            perror("accept:");
            exit(1);
        }

        printf("\nServidor recebeu conexao de %s",
inet_ntoa(client_addr.sin_addr));
        memset(szMensagem, 0, sizeof(szMensagem));
        strcpy(szMensagem, "Ola cliente\n");
    }
}
```

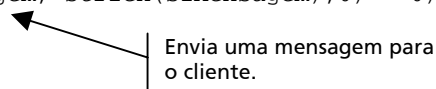
← Identifica a origem da conexão.



```

        if( send( iFd, szMensagem, strlen(szMensagem), 0) < 0)
        {
            perror("send:");
            exit(1);
        }
    }
}

```



Envia uma mensagem para o cliente.

Programa 23.9

## 23.10 Função `recv`

Sintaxe:

```
ssize_t recv(int s, void *buf, size_t len);
```

A função `recv` é utilizada para receber (ler) uma mensagem de um *socket*. Para leitura de mensagens, também pode ser utilizada a função `read`. A função retorna o número de *bytes* lidos ou `-1` se houve erro.

A função `recv` é utilizada em conexões TCP (*stream*) ou orientada à conexão.

Veja o exemplo:

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void main(void)
{
    int iSock;
    int iBytes;
    struct sockaddr_in dest_addr;
    char buffer[100];

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(4950);

```

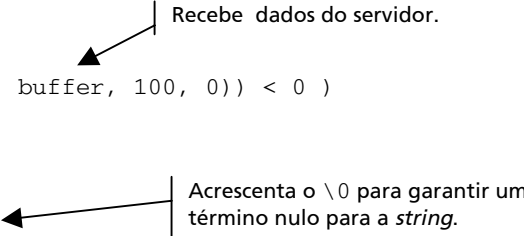
## 218 ♦ Programando em C para Linux, Unix e Windows

```
dest_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
bzero(&(dest_addr.sin_zero), 8);

if( connect(iSock, (struct sockaddr *)&dest_addr, sizeof(struct
sockaddr)) < 0)
{
    perror("connect:");
    exit(1);
}

if ((iBytes=recv(iSock, buffer, 100, 0)) < 0 )
{
    perror("recv");
    exit(1);
}
buffer[iBytes] = '\0';

printf("Recebido: %s",buffer);
}
```



Recebe dados do servidor.

Acrescenta o \0 para garantir um término nulo para a *string*.

Programa 23.10

## 23.11 Funções sendto e recvfrom

Sintaxe:

```
ssize_t sendto(int s, const void *buf, size_t len int flags,
const struct sockaddr *to, socklen_t tolen);
ssize_t recvfrom(int s, void *buf, size_t len int flags,
struct sockaddr *from, socklen_t *fromlen);
```

As funções `sendto` e `recvfrom` têm a mesma função e retorno das funções `send` e `recv`, exceto que são utilizadas para comunicação não orientada a conexões (UDP).

## 23.12 Funções close e shutdown

Sintaxe:

```
int close(int fd);
int shutdown(int s, int how);
```

A função `close` finaliza uma conexão *socket*. A função `shutdown` finaliza toda ou parte de uma conexão full-duplex. As funções retornam 0 (zero) em caso de sucesso ou -1 se houve algum erro. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
```

```

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

void main(void)
{
    int iSock;
    int iBytes;
    struct sockaddr_in dest_addr;
    char buffer[100];

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(4950);
    dest_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bzero(&(dest_addr.sin_zero), 8);

    if( connect(iSock, (struct sockaddr *)&dest_addr, sizeof(struct
sockaddr)) < 0)
    {
        perror("connect:");
        exit(1);
    }

    if ((iBytes=recv(iSock, buffer, 100, 0)) < 0 )
    {
        perror("recv");
        exit(1);
    }
    buffer[iBytes] = '\0';

    printf("Recebido: %s",buffer);

    close(iSock); ← Finaliza a conexão.
}

```

Programa 23.11

## 23.13 Função getpeername

Sintaxe:

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

A função `getpeername` retorna o nome de um cliente que se conectou ao servidor. A função retorna 0 (zero) em caso de sucesso ou -1 se houve algum erro.

## 23.14 Função gethostbyname

Sintaxe:

```
struct hostent *gethostbyname(const char *name);
```

A função `gethostbyname` retorna, a partir de um nome passado, o endereço IP associado ao nome. A função realiza o papel de um DNS – *Domain Name Server*. Ela retorna um ponteiro para uma estrutura ou `NULL` em caso de erro. Veja o exemplo:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    struct hostent *h;

    if (argc != 2)
    {
        printf("Deve-se passar nome da maquina");
        exit(1);
    }

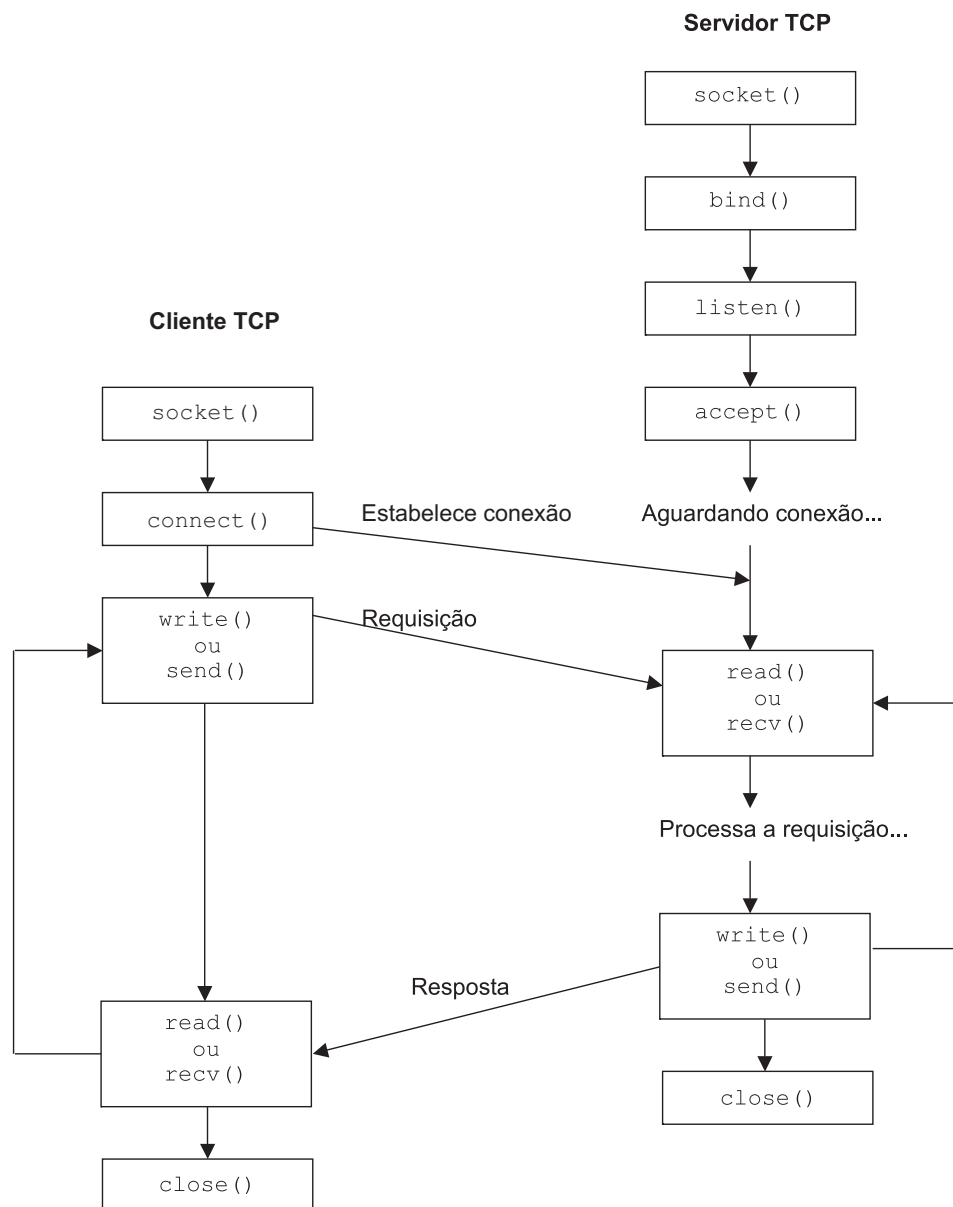
    if ((h=gethostbyname(argv[1])) == NULL)
    {
        perror("gethostbyname:");
        exit(1);
    }

    printf("Nome do Host: %s\n", h->h_name);
    printf("Endereco IP : %s\n", inet_ntoa(*(struct in_addr *)h->h_addr));

    return 0;
}
```

Programa 23.12

## 23.15 Diagrama de servidor/cliente TCP básico



## 23.16 Exemplo completo de um servidor TCP

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

void main(void)
{
    int iSock;
    struct sockaddr_in my_addr;

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if( bind(iSock, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) == -1)
    {
        perror("bind:");
        exit(1);
    }

    if( listen( iSock, 10 ) < 0)
    {
        perror("listen:");
        exit(1);
    }
    while(1)
    {
        int iFd;
        struct sockaddr_in client_addr;
        socklen_t sin_size;
        char szMensagem[100];

        sin_size = sizeof(struct sockaddr_in);

        if( (iFd = accept(iSock, (struct sockaddr *) &client_addr,
&sin_size)) < 0)
```

```

    {
        perror("accept:");
        exit(1);
    }

    printf("\nServidor recebeu conexao de %s",
inet_ntoa(client_addr.sin_addr));

    memset(szMensagem, 0, sizeof(szMensagem));
    strcpy(szMensagem, "Ola cliente\n");
    if( send( iFd, szMensagem, strlen(szMensagem), 0) < 0)
    {
        perror("send:");
        exit(1);
    }
    close(iFd);
}
}

```

#### Programa 23.13

## 23.17 Exemplo completo de um cliente TCP

```

#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>

void main(void)
{
    int iSock;
    int iBytes;
    struct sockaddr_in dest_addr;
    char buffer[100];

    iSock = socket(AF_INET, SOCK_STREAM, 0);
    if( iSock == -1)
    {
        perror("socket:");
        exit(1);
    }

    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(4950);
    dest_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    bzero(&(dest_addr.sin_zero), 8);

```

## 224 ♦ Programando em C para Linux, Unix e Windows

```
if( connect(iSock, (struct sockaddr *)&dest_addr, sizeof(struct
sockaddr)) < 0){
    perror("connect:");
    exit(1);
}
if ((iBytes=recv(iSock, buffer, 100, 0)) < 0 ){
    perror("recv");
    exit(1);
}

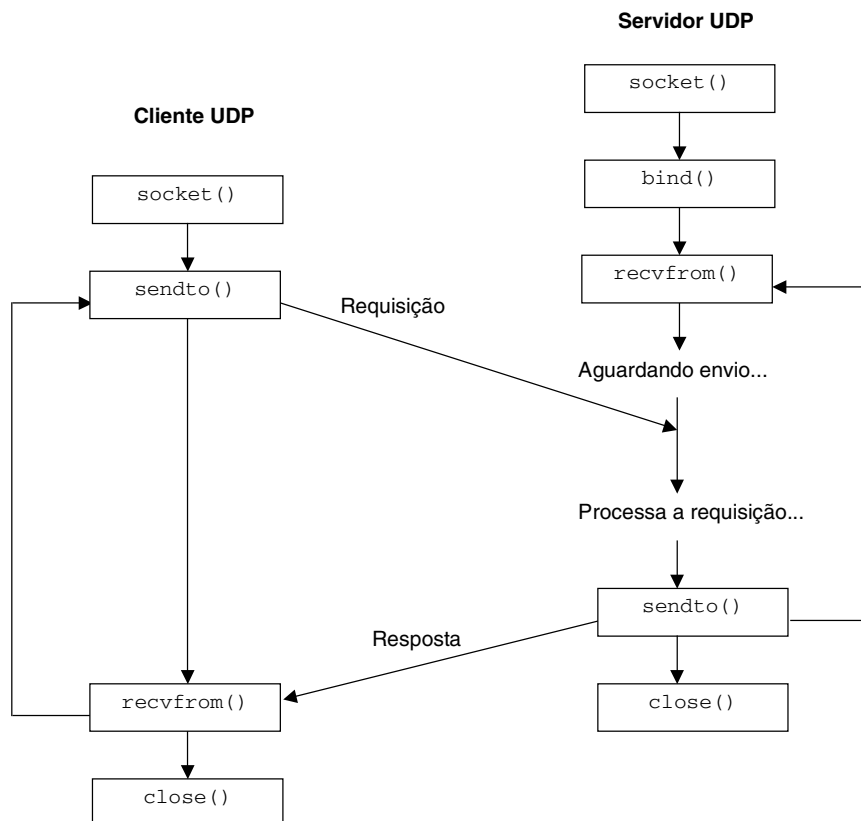
buffer[iBytes] = '\0';

printf("Recebido: %s",buffer);

close(iSock);
}
```

Programa 23.14

## 23.18 Diagrama de servidor/cliente UDP básico





## 23.19 Exemplo completo de um servidor UDP

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>

int main(void)
{
    int iSock;
    struct sockaddr_in my_addr;
    struct sockaddr_in client_addr;
    socklen_t addr_len;
    int numbytes;
    char buffer[100];

    if ((iSock = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(4950);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    if (bind(iSock, (struct sockaddr *)&my_addr, sizeof(struct
sockaddr)) < 0 )
    {
        perror("bind");
        exit(1);
    }
    addr_len = sizeof(struct sockaddr);
    if ((numbytes=recvfrom(iSock, buffer, 100, 0,
        (struct sockaddr *)&client_addr, &addr_len)) < 0)
    {
        perror("recvfrom");
        exit(1);
    }

    printf("Recebendo pacotes de %s\n",inet_ntoa(client_addr.sin_addr));
    printf("o pacote tem %d bytes\n",numbytes);
```

```

    buffer[numbytes] = '\0';
    printf("O conteudo do pacote eh %s\n",buffer);
    close(iSock);
}

```

Programa 23.15

## 23.20 Exemplo completo de um cliente UDP

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>
#include <strings.h>
#include <arpa/inet.h>

int main(int argc, char *argv[])
{
    int iSock;
    struct sockaddr_in server_addr;
    struct hostent *he;
    int numbytes;

    if (argc != 3)
    {
        printf("Passe o nome do servidor e a mensagem");
        exit(1);
    }

    if ((he=gethostbyname(argv[1])) == NULL)
    {
        perror("gethostbyname");
        exit(1);
    }

    if ((iSock = socket(AF_INET, SOCK_DGRAM, 0)) < 0 )
    {
        perror("socket");
        exit(1);
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(4950);
    server_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(server_addr.sin_zero), 8);


```

```
if ((numbytes=sendto(iSock, argv[2], strlen(argv[2]), 0,
    (struct sockaddr *)&server_addr, sizeof(struct sockaddr))) < 0)
{
    perror("sendto");
    exit(1);
}

printf("enviado %d bytes para
%s\n", numbytes, inet_ntoa(server_addr.sin_addr));

close(iSock);
}
```

**Programa 23.16**



## **Técnicas de Programação para Facilitar a Depuração, Documentação, Economia e Execução de Processos**

*É melhor corrigir os nossos próprios erros do que os dos outros.*

Demócrito, filósofo grego

**A**tualmente, um software deve primar pela qualidade extrema, não somente no que diz respeito ao aspecto visual, funcionalidade e precisão nos resultados, mas igualmente importante é a capacidade deste software ser entendido e interpretado por outras pessoas – e não somente o autor.

Através de técnicas simples, este trabalho visa mostrar como obter maior qualidade no desenvolvimento de software, agilidade na depuração, software auto-documentável e de fácil execução.

Depurar um programa – ou "debugar" como é mais conhecido pelos programadores – tornou-se uma tarefa mais fácil em função das milhares de ferramentas existentes, fora as que são disponibilizadas junto com as linguagens.

Entretanto, não é possível depurar um programa se este não obedece a determinadas regras de programação, que facilitam a depuração e o próprio entendimento do programa. Não basta documentar os programas entupindo-os de comentários entre as milhares de linhas de código; deve-se programar de forma simples e objetiva, evitando-se utilizar o último recurso disponível da linguagem – principalmente se o programador não possui total domínio sobre este novo recurso, embora, tomando-me como exemplo, sempre fiquemos tentados a utilizar esta nova funcionalidade para satisfazer nossa "curiosidade", e se funcionar, ótimo, se não, fazemos do jeito tradicional mesmo.

Quando me refiro a programar de forma simples e objetiva, falo em não agrupar muitas operações matemáticas numa mesma linha de código, a colocar múltiplas condições numa cláusula de IF ou WHILE, entre outras situações, que dificultam – e muito – o entendimento do código no momento da manutenção por outros programadores, e até mesmo pelo próprio autor, no momento de uma depuração "emergencial".

Pretendo mostrar, através de exemplos "errados" de programação, analisando estes códigos e mostrando uma alternativa mais simples de codificação, que julgo ser muito importante e que facilita tanto o trabalho de manutenção como depuração do código.

Utilizarei nos meus exemplos a linguagem C, padrão ANSI; todos os exemplos foram compilados e executados numa máquina com sistema operacional UNIX (HP-UX). Começarei falando das tradicionais regras para declaração de variáveis nos códigos dos programas.

A declaração correta de uma variável é a parte mais importante de um programa. Uma variável declarada de forma errada pode causar sérios transtornos, desde um resultado totalmente "maluco" num cálculo matemático até o travamento total da máquina e sistema operacional (se você estiver utilizando uma linguagem como C ou Assembler).

Tenha como premissa somente declarar as variáveis onde você for utilizá-las, não declare uma variável como global se ela somente for utilizada naquela função bem escondida para somar  $2 + 2$ ; além de ocupar memória desnecessariamente, pode ocasionar resultados inesperados.

A maioria das linguagens de programação modernas permite que você crie uma variável em qualquer ponto do programa; utilize-a descartando-a em seguida. Observe o seguinte código:

```
#include <stdio.h>
void main(void)
{
    int a = 10;
    int b = 20;
    int c;
    /* mostrar as variáveis com os valores originais */
    printf( "\nValor de A = %d", a );
    printf( "\nValor de B = %d", b );
    /* trocando os valores entre as variáveis */
    c = a;
    a = b;
    b = c;
    /* mostrar as variáveis com os valores trocados */
```

## 230 ♦ Programando em C para Linux, Unix e Windows

```
    printf( "\nValor de A = %d", a );
    printf( "\nValor de B = %d", b );
}
```

### Programa 24.1

O exemplo anterior é o clássico programa para permuta de conteúdos de variáveis, declarando-se duas variáveis com os valores que devem ser permutados e uma variável auxiliar para realizar a troca. O programa funciona sem problema algum, mas se fosse um programa mais complexo ou se fossem executadas várias instruções antes da troca, a variável que foi declarada para auxiliar a troca de informações ficaria alocada durante toda a execução do programa, ocasionando um consumo de memória. Observe o exemplo a seguir:

```
#include <stdio.h>
void main(void)
{
    int a = 10;
    int b = 20;
    /* mostrar as variáveis com os valores originais */
    printf( "\nValor de A = %d", a );
    printf( "\nValor de B = %d", b );
    /* quero trocar o conteúdo entre as variáveis */
    { /* abro um novo bloco de instruções */
        int c; /* declaro uma variável auxiliar aqui */
        c = a;
        a = b;
        b = c;
    } /* ao ser fechado, todas as variáveis declaradas dentro deste
bloco são limpas da memória */
    /* mostrar as variáveis com os valores trocados */
    printf( "\nValor de A = %d", a );
    printf( "\nValor de B = %d", b );
}
```

### Programa 24.2

Repare que, no exemplo anterior, a variável auxiliar é declarada no momento da sua utilização e logo após é limpa da memória. Esta é uma técnica simples e funcional para declaração de variáveis que são utilizadas somente em alguns pontos do código.

Ao declarar variáveis, tão importante quanto definir o escopo e abrangência da variável – utilização pública, local, estática etc. – é a nomenclatura da declaração de variáveis. Não existe uma regra universal para declaração de variáveis, mas é importante existir um padrão de declarações, de forma que a simples visualização da variável no meio do código do programa identifique o tipo e a abrangência desta variável.

Através de convenções e a experiência de alguns anos como programador e analista de sistemas, cheguei a esta tabela, derivada da notação húngara:

Tipo	Prefixos	Exemplo	Tipo	Prefixos	Exemplo
Char	(ch)	chOpt	structs (definição)	(ST_)	ST_Monit
Int	(i)	iNum	structs	(st)	stFile
Long	(l)	lValor	union (definição)	(U_)	U_Registro
Double	(db)	dbGraus	union	(un)	unBuff
String	(s)	Stela	ponteiros	(p)(tipo)	pchOpt
string c/ "0"	(sz)	szNome	Variáveis Globais	(G_)(tipo)	G_lValor

Não é meu objetivo que você utilize esta tabela como regra imutável de codificação, ela é uma sugestão para auxiliar o programador novato e até mesmo o experiente a adotar um padrão. É possível criar uma tabela para cada linguagem de programação, bastando identificar os tipos de dados possíveis nesta linguagem. Torno a frisar que o importante é você ter um padrão de declaração de variáveis, qualquer que seja, e que este padrão seja comum ao seu local de trabalho e que esteja disponível a todos que irão trabalhar com o código fonte.

Outro item importante a ser observado – como dito anteriormente – é não agrupar uma fórmula matemática complexa em somente uma linha de programa. Tomemos como exemplo uma fórmula hipotética qualquer:

$$A = ((B * C) / 100) * (F * (3 / D))$$

Caso esta fórmula esteja trazendo um valor diferente do esperado, como fazer para descobrir qual parte da fórmula está errada? Você pode depurar o programa, capturar o conteúdo de cada variável – verificando se foram inicializadas com os valores corretos – e realizar o cálculo manualmente. E se, mesmo assim, o resultado achado manualmente for diferente do que o programa está calculando – supondo que o cálculo manual esteja correto – será um erro de arredondamento? Fica muito mais fácil a depuração deste código e conseqüentemente a manutenção, se ele fosse declarado da seguinte forma:

$$\begin{aligned} Y &= (B * C) \\ X &= Y / 100 \\ Z &= (3 / D) \\ W &= F * Z \\ A &= X * W \end{aligned}$$

Repare que, matematicamente, as fórmulas são idênticas.

## 232 ♦ Programando em C para Linux, Unix e Windows

Acredito que neste momento estou levantando uma polêmica entre os programadores que defendem a codificação da fórmula anterior em somente uma linha de programa, alegando que o programa executaria mais rápido – em função de utilizar menos linhas – e consumiria menos memória – em função de não haver a necessidade de se criarem variáveis auxiliares.

Para desmentir os programadores que acreditam que o programa rode mais rapidamente em função de utilizar somente uma linha de código, irei demonstrar como o compilador interpreta a fórmula  $A = ((B * C) / 100) * (F * (3 / D))$ :

```
Passo 1 = B * C
Passo 2 = Passo 1 / 100
Passo 3 = 3 / D
Passo 4 = F * Passo 3
Passo 5 = Passo 4 * Passo 2
```

Ou seja, em vez de você codificar as instruções, quebrando e declarando as variáveis auxiliares, o compilador, no momento da execução, realiza este trabalho, alocando memória, de acordo com o necessário e liberando em seguida. Caso se codifique somente em uma linha, este código fonte torna-se menor, mas em compensação o código pré-compilado (código que será utilizado para a geração do executável) ficará maior. Caso contrário, você terá um código fonte e um código pré-compilado similares. Repare que, em ambos os casos, o tamanho do seu código executável será similar.

E para aplacar a ira dos programadores que julgam estar gastando memória desnecessariamente ao criar variáveis auxiliares, peço para que olhem o exemplo a seguir, onde demonstro como criar variáveis auxiliares e economizando memória do sistema.

No caso, o programa para realizar o cálculo seguindo as premissas de “quebrar” a fórmula matemática em várias subfórmulas, ficaria assim:

```
#include <stdio.h>
void main(void)
{
    float A,B,C, D, F;
    B = 10.23;
    C = 17.87;
    D = 89.34;
    F = 115.01;
    {
        float Y, X, Z, W;
        Y = (B * C);
        X = Y / 100;
        Z = (3 / D);
```



```

        W = F * Z;
        A = X * W;
    }
    /* neste momento, as variáveis auxiliares já não estão mais na
memória */
    printf( "A = %f", A);
}

```

### Programa 24.3

E para provar que o tempo de processamento fica similar, foi executado o código 24.4 – descrito a seguir – e o código 24.3 e comparado o tempo de execução entre ambos – foi utilizado o comando `time` do UNIX para comparar as execuções.

```

#include <stdio.h>
void main(void)
{
    float A,B,C,D, F;
    B = 10.23;
    C = 17.87;
    D = 89.34;
    F = 115.01;
    A = ((B*C)/100)*(F*(3/D));
    printf( "A = %f", A);
}

```

### Programa 24.4

O resultado é mostrado na tabela de comparação:

	Tamanho do fonte em bytes	Tamanho do executável em bytes	Tempo de processamento médio da CPU (milissegundos)	Resultado da operação
Código 24.3	281	46952	0.02 s	7.060104
Código 24.4	191	46776	0.02 s	7.060104

Repare que o tamanho do código 24.3 – com a fórmula matemática quebrada em várias subfórmulas – possui um pequeno e insignificante acréscimo de tamanho no arquivo fonte e no arquivo executável, mas em compensação o tempo de execução – tempo utilizado pelo processador do computador (CPU) para executar as instruções – foram as mesmas, e principalmente, o resultado final não se alterou. Ou seja, tivemos um pequeno prejuízo no tamanho final da aplicação, não perdemos em tempo de processamento, tivemos um grande ganho no tempo de entendimento, manutenção, depuração do código fonte e conseguimos o mesmo resultado matemático.

## 234 ♦ Programando em C para Linux, Unix e Windows

A partir deste momento, os exemplos irão seguir o padrão de “declaração de variáveis” que eu sugeri anteriormente.

Outro tópico importante sobre o trabalho com variáveis é o agrupamento destas em estruturas de dados, de forma a auxiliar a identificar qual a sua finalidade dentro do código fonte. Uma estrutura de dados é similar a um registro de uma tabela qualquer – independentemente do banco de dados utilizado – a única diferença entre eles é que a estrutura de dados fica armazenada na memória do computador durante a execução do programa. Observe o código 24.5:

```
#include <stdio.h>
void main(void)
{
    float fSalario;
    char szNomeFuncionario[40];
    int iNivelCargo;
    char szNomeDepartamento[40];
    fSalario = 7500.00; /* ainda chego lá */
    strcpy( szNomeFuncionario, "Marcos Aurelio Pchek Laureano");
    iNivelCargo = 21;
    strcpy( szNomeDepartamento, "INBR TSV COM" );
    printf("\nFuncionario = %s", szNomeFuncionario);
    printf("\nSalario = %f", fSalario );
    printf("\nNivel Cargo = %d", iNivelCargo );
    printf("\nDepartamento = %s", szNomeDepartamento );
}
```

### Programa 24.5

Observando o código 24.5, você consegue perceber que se trata de um programa para manipular dados de funcionários da empresa. Imagine-se depurando um programa que manipula as informações de funcionários e que este programa trabalhe com vários tipos de informações – documentação, dados de endereço, dados de filiação etc. – e se, num dado momento, a execução deste programa é interrompida. Será necessário depurar o código para localizar a causa do erro. Imagine-se verificando o conteúdo de cada variável, uma a uma. Percebeu o tempo perdido? Observe o código 24.6:

```
#include <stdio.h>
struct ST_FUNCIONARIO
{
    float fSalario;
    char szNomeFuncionario[40];
    int iNivelCargo;
    char szNomeDepartamento[40];
};
```

```

void main(void)
{
    struct ST_FUNCIONARIO stFuncionario;
    stFuncionario.fSalario = 7500.00; /* sonhar não paga imposto */
    strcpy(stFuncionario.szNomeFuncionario, "Marcos Aurelio Pchek
Laureano");
    stFuncionario.iNivelCargo = 21;
    strcpy(stFuncionario.szNomeDepartamento, "INBR TSV COM" );
    printf("\nFuncionario = %s", stFuncionario.szNomeFuncionario);
    printf("\nSalario = %f", stFuncionario.fSalario );
    printf("\nNivel Cargo = %d", stFuncionario.iNivelCargo );
    printf("\nDepartamento = %s", stFuncionario.szNomeDepartamento );
}

```

#### Programa 24.6

O objetivo do código 24.6 é mostrar a utilização de estruturas de dados para agrupar as informações. Você verifica a vantagem na utilização da estrutura de dados no momento de uma manutenção e depuração do programa. Não se perde tempo identificando cada variável e qual a sua finalidade e, principalmente, você consegue ver todo o conteúdo da estrutura de uma vez só. Observe a tabela:

	Tamanho do fonte em bytes	Tamanho do executável em bytes	Tempo de processamento médio da CPU (milissegundos)
Código 24.5	496	36768	0.02 s
Código 24.6	668	36872	0.02 s

Como você pode verificar, o tamanho do código executável é praticamente o mesmo, e o tempo de execução de ambos os programas é igual. Fica evidente que, desta forma, você mantém um código fonte autodocumentável – ao ter mais agilidade na identificação da variável – e não perde em tempo de processamento.

Para finalizar este trabalho, vou comentar o aninhamento de múltiplas condições em cláusulas de IF e WHILE.

Observe o código 24.7:

```

#include <stdio.h>
int main( int iArgc, char ** pszArgv )
{
    int iA, iB, iC;
    if( iArgc < 3 )
    {

```

## 236 ♦ Programando em C para Linux, Unix e Windows

```
    printf("\nEntre com 2 valores de dados" );
    exit(1);
}
iA = iArgc * 10;
iB = atoi( pszArgv[1] ) + iA;
iC = atoi( pszArgv[2] ) * iB;
if ( iA > iB && iC > iA || iC < iB )
    printf("\nOi");
else
    printf("\nTchau");
return(0);
}
```

### Programa 24.7

Afinal, o programa deveria imprimir "Oi" quando  $iA > iB$  e  $iC > iA$  ou  $iC < iB$ ? Ou deveria imprimir "Oi" quando  $iA > iB$  e  $iC > iA$  ou  $iC < iB$ ? Confuso? Vou reformular a pergunta. O programa deveria imprimir "Oi" se somente  $iA > iB$  ou quando  $iC > iA$  ou  $iC > iB$ , ou se  $iA > iB$  e  $iC > iA$  ou quando  $iC < iB$ . Realmente, não é fácil entender o que o programador queria fazer neste caso. Tudo o que sabemos é que o operador && (E) tem precedência sobre o || (OU), e neste caso a condição ficaria igual a  $(iA > iB \ \&\& \ iC > iA) \ || \ iC < iB$ . Este exemplo demonstra a necessidade imperiosa de identificar o que você realmente quer fazer, e não contar simplesmente com a ajuda do compilador. Para tal existem os delimitadores de expressões, que são os mesmos que você utiliza para separar uma fórmula matemática.

$$A = B * C + 10$$
$$A = (B * C) + 10$$

Repare que estas fórmulas são idênticas matematicamente falando, mas a segunda fórmula é bem mais fácil de ser compreendida. Não leva alguém a se perguntar se o programador queria primeiro somar 10 a C e depois multiplicar por B ou multiplicar B por C e somar 10 ao resultado.

Pode parecer exagero comentar situações como a anterior, mas após dar manutenção a diversos programas, construídos nas mais variadas linguagens de programação, vocês não imaginam a quantidade de ocasiões nas quais me deparei com este tipo de situação.

Observe os códigos 24.8 e 24.9:

```
#include <stdio.h>
void main( int iArgc, char ** pszArgv )
{
    int iA[5];
    int iContador;
    int iB;
```

```

iB = atoi(pszArgv[1]);
iA[0]=2;
iA[1]=7;
iA[2]=3;
iA[3]=10;
iA[4]=15;
for( iContador = 0; iContador < 5 && iA[iContador]!= iB; iConta-
dor ++ );
    if( iContador < 5 )
        printf("\nAchei");
    else
        printf("\nNao achei");
}

```

#### Programa 24.8

```

#include <stdio.h>
void main( int iArgc, char ** pszArgv )
{
    int iA[5];
    int iContador;
    int iB;
    iB = atoi(pszArgv[1]);
    iA[0]=2;
    iA[1]=7;
    iA[2]=3;
    iA[3]=10;
    iA[4]=15;
    for( iContador = 0; iContador < 5; iContador ++ )
        if(iA[iContador] == iB ) break;
    if( iContador < 5 )
        printf("\nAchei");
    else
        printf("\nNao achei");
}

```

#### Programa 24.9

Os dois programas fazem a mesma coisa, ou seja, procuram num vetor um elemento qualquer de forma seqüencial, interrompendo tão logo o elemento seja encontrado. Qual dos programas é mais fácil de se entender? A meu ver, o código 24.9 é muito mais simples de entender, embora o código 24.8 demonstre um determinado refinamento na sua concepção.

Nestes momentos cruciais, devemos decidir qual a forma de programação que iremos adotar: o método mais simples, tanto na concepção como na facilidade de entendimento e mais óbvio para a grande maioria dos programadores, ou o método mais refinado, mais trabalhado, que demonstra o domínio do programador sobre os recursos da linguagem utilizada, mas que também requer um maior tempo para análise e compreensão da condição. Observe a tabela:

	Tamanho do fonte em bytes	Tamanho do executável em bytes	Tempo de processamento médio da CPU (milissegundos)
Código 24.8	353	36768	0.02 s
Código 24.9	369	36776	0.03 s

O código 24.8 – mais refinado e trabalhado – é melhor em tudo, seja no tamanho do código fonte, seja no tamanho do código executável e no tempo de processamento, mas leva mais tempo para o entendimento do programa – no momento de uma depuração emergencial, este tempo a mais pode ser crucial para o negócio da empresa – e conseqüentemente mais rápido no processamento. O código 24.9 – mais simples e comum – ocupa mais espaço para armazenamento do código fonte e do código executável, leva mais tempo para ser executado, mas é mais simples de se entender.

Concluindo, pode parecer um despropósito ter realizado todos os comentários anteriores, afinal, todos nós aprendemos na faculdade – pelo menos eu aprendi – que os melhores programas são os que carregam menos variáveis na memória, que fazem as mesmas coisas em menos linhas, que ocupam menos processamento de máquina etc. Concordo com todos estes ensinamentos, mas acredito que na maioria dos casos – deve-se avaliar cada situação – pode e deve ser escrito um programa mais legível e fácil de se manter ou depurar em detrimento de um código mais refinado e “bonito” de se ver.



*Eu posso explicar isso para eles, mas eu não posso entender isso por eles.*

Dan Rather, repórter americano

## A.1 Recursividade

Na linguagem C, as funções podem chamar a si mesmas. A função é *recursiva* se um comando no corpo da função a chama. Recursão é a habilidade que uma função tem de chamar a si mesma, ou seja, é a técnica que consiste simplesmente em aplicar uma função como parte da definição dessa mesma função.

Para uma linguagem de computador ser recursiva, uma função deve poder chamar a si mesma. Um exemplo simples é a função `fatorial`, que calcula o fatorial de um inteiro. O fatorial de um número  $N$  é o produto de todos os números inteiros entre 1 e  $N$ . Por exemplo, 3 fatorial (ou  $3!$ ) é  $1 * 2 * 3 = 6$ .

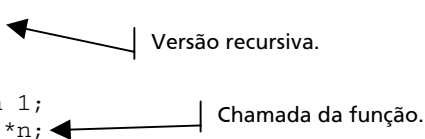
Veja os exemplos:

```
int fatorialc( int n ) ← Versão iterativa.
{
    int t, f;
    f = 1;
    for( t = 1; t<=n; t++ )
        f = f * t;
    return f;
}
```

Programa A.1

## 240 ♦ Programando em C para Linux, Unix e Windows

```
int fatorialr( int n)
{
    int t, f;
    if( n == 1) return 1;
    f = fatorialr(n-1)*n;
    return f;
}
```



### Programa A.2

A versão não-recursiva de fatorial deve ser clara. Ela usa um laço que é executado de 1 a  $n$  e multiplica progressivamente cada número pelo produto móvel.

A operação de fatorial recursiva é um pouco mais complexa. Quando `fatorialr` é chamada com um argumento de 1, a função devolve 1. Caso contrário, ela devolve o produto de `fatorialr(n-1)*n`. Para avaliar essa expressão, `fatorialr` é chamada com  $n-1$ . Isso acontece até que  $n$  se iguale a 1 e as chamadas à função comecem a retornar.

Calculando o fatorial de 2, a primeira chamada a `fatorialr` provoca uma segunda chamada com o argumento 1. Essa chamada retorna 1, que é, então, multiplicado por 2 (o valor original e  $n$ ). A resposta então é 2.

Quando uma função chama a si mesma, novos parâmetros e variáveis locais são alocados na pilha e o código da função é executado com essas novas variáveis. Uma chamada recursiva não faz uma nova cópia da função; apenas os argumentos são novos. Quando cada função recursiva retorna, as variáveis locais e os parâmetros são removidos da pilha e a execução recomeça do ponto da chamada à função dentro da função.

A maioria das funções recursivas não minimiza significativamente o tamanho do código ou melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que suas equivalentes iterativas devido às repetidas chamadas à função. De fato, muitas chamadas recursivas a uma função podem provocar um estouro da pilha. Como o armazenamento para os parâmetros da função e variáveis locais está na pilha e cada nova chamada cria uma nova cópia dessas variáveis, a pilha pode provavelmente escrever sobre outra memória de dados ou de programa. Contudo, não é necessário se preocupar com isso, a menos que uma função recursiva seja executada de forma desenfreada.

A principal vantagem das funções recursivas é ser possível utilizá-las para criar versões mais claras e simples de vários algoritmos.

Ao escrever funções recursivas, deve-se ter um comando `if` em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada.



Se não existir, a função nunca retornará quando chamada (equivalente a um *loop* infinito). Omitir o comando `if` é um erro comum ao escrever funções recursivas.

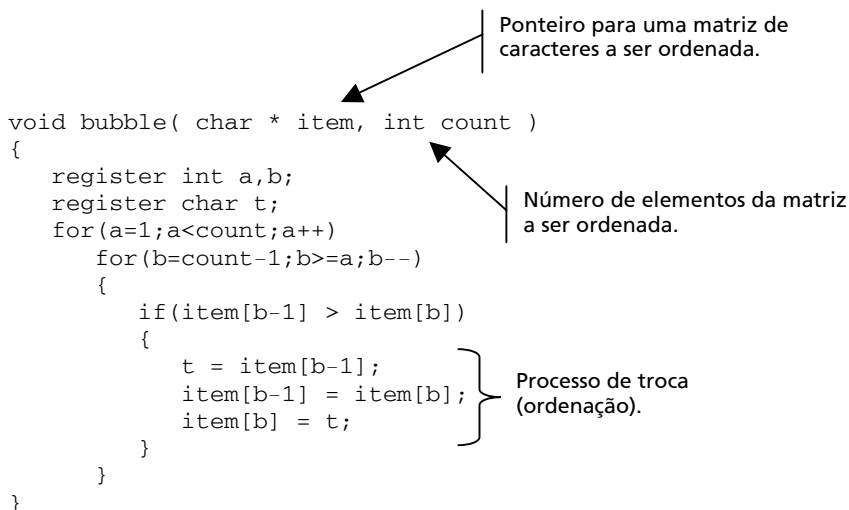
## A.2 Ordenação

### A.2.1 Bolha

A ordenação mais conhecida (e mais difamada) é a ordenação bolha. Sua popularidade vem do seu nome fácil e de sua simplicidade. Porém, é uma das piores ordenações já concebidas.

A ordenação bolha é uma ordenação por trocas. Ela envolve repetidas comparações e, se necessário, a troca de dois elementos adjacentes.

Veja a versão mais simples do algoritmo bolha:



```
void bubble( char * item, int count )
{
    register int a,b;
    register char t;
    for(a=1;a<count;a++)
        for(b=count-1;b>=a;b--)
        {
            if(item[b-1] > item[b])
            {
                t = item[b-1];
                item[b-1] = item[b];
                item[b] = t;
            }
        }
}
```

#### Programa A.3

A ordenação bolha é dirigida por dois *loops*. Dados que existem `count` elementos na matriz, o *loop* mais externo faz a matriz ser varrida `count-1` vezes. Isso garante, na pior hipótese, que todo elemento estará na posição correta quando a função terminar. O *loop* mais interno faz as comparações e as trocas.

Essa versão da ordenação bolha pode ser utilizada para ordenar uma matriz de caracteres em ordem ascendente. Por exemplo, o programa seguinte ordena uma *string*.

## 242 ♦ Programando em C para Linux, Unix e Windows

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

void bubble( char * item, int count );

void main(void)
{
    char vetorb[]="3490bn09685lnv 3-49580bgojfog39458=9ugkj n098=526yh";

    printf("\nAntes = [%s]", vetorb);
    bubble(vetorb,strlen(vetorb)-1);
    printf("\nDepois = [%s]", vetorb);
}
```

### Programa A.4

#### Resultado do Programa A.4

```
Antes = [3490bn09685lnv 3-49580bgojfog39458=9ugkj n098=526yh]
Depois = [ -000023334445555668888999999==bbfggghjjklnnnoouvy]
```

## A.2.2 Quicksort

A Quicksort, inventada e denominada por C.A.R. Hoare, é considerada o melhor algoritmo de ordenação de propósito geral atualmente disponível. É baseada no método de ordenação por trocas (mas muito superior em termos de desempenho à ordenação bolha).

A Quicksort é baseada na idéia de partições. O procedimento geral é selecionar um valor, chamado de *comparando*, e, então, fazer a partição da matriz em duas seções, com todos os elementos maiores ou iguais ao valor da partição de um lado e os menores do outro. Este processo é repetido para cada seção restante até que a matriz esteja ordenada. Por exemplo, dada a matriz *fedacb* e usando o valor *d* para a partição, o primeiro passo da Quicksort rearranja a matriz como segue:

```
Início          f e d a c b
Passo 1         b c a d e f
```

Esse processo é, então, repetido para cada seção – isso é, *bca* e *def*. Assim, o processo é essencialmente recursivo por natureza e, certamente, as implementações mais claras da Quicksort são algoritmos recursivos.

O *comparando central* pode ser selecionado de duas formas. Escolhê-lo aleatoriamente ou selecioná-lo fazendo a média de um pequeno conjunto de valores da matriz. Para uma ordenação ótima, deveria ser selecionado um valor

que estivesse precisamente no centro da faixa de valores. Porém, isso não é fácil para a maioria dos conjuntos de dados. No pior caso, o valor escolhido está em uma extremidade e, mesmo nesse caso, o algoritmo Quicksort ainda tem um bom rendimento. A versão seguinte seleciona o elemento central da matriz. Embora isso nem sempre resulte numa boa escolha, a ordenação ainda é efetuada corretamente.

```
void qs( char *item, int left, int right)
{
    register int i,j;
    char x,y;
    i = left;
    j = right;
    x = item [ (left+right)/2 ];
    do
    {
        while(item[i]<x && i<right) i++;
        while(x<item[j] && j>left) j--;
        if(i<=j)
        {
            y = item[i];
            item[i] = item[j];
            item[j] = y;
            i++;
            j--;
        }
    } while (i<=j);
    if( left<j) qs(item, left, j);
    if( i<right) qs(item, i, right);
}
```

Procedimento de troca (ordenação).

Chamada recursiva...

#### Programa A.5

O próximo programa realiza a chamada da função para ordenação.

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

void qs( char *item, int left, int right);

void main(void)
{
    char vetorq[]="3490bn09685lnv 3-49580bgojfog39458=9ugkj n098=526yh";

    printf("\nAntes = [%s]", vetorq);
    qs(vetorq,0,strlen(vetorq)-1);
    printf("\nDepois = [%s]", vetorq);
}
```

Na primeira chamada, os parâmetros iniciais são os extremos da matriz.

#### Programa A.6

**Resultado do Programa A.6**

```
Antes = [3490bn09685lnv 3-49580bgojfog39458=9ugkj n098=526yh]
Depois = [ -00002333444555566888899999==bbfggghjjklmnnoouv]
```

## A.3 Pesquisa

Bancos de dados existem para que, de tempos em tempos, um usuário possa localizar o dado de um registro simplesmente digitando sua chave. Há apenas um método para se encontrarem informações em um arquivo (matriz) desordenado e um outro para um arquivo (matriz) ordenado.

Encontrar informações em uma matriz desordenada requer uma pesquisa seqüencial começando no primeiro elemento e parando quando o elemento procurado ou o final da matriz é encontrado. Esse método deve ser usado em dados desordenados, mas também pode ser aplicado a dados ordenados. Se os dados foram ordenados, pode ser utilizada uma pesquisa binária, o que ajuda a localizar o dado mais rapidamente.

### A.3.1 Pesquisa Seqüencial

A pesquisa seqüencial é fácil de ser codificada. A função a seguir faz uma pesquisa em uma matriz de caracteres de comprimento conhecido até que seja encontrado, a partir de uma chave específica, o elemento procurado:

```
int sequential_search( char * item, int count, char key )
{
    register int t;
    for( t = 0; t<count; t++)
        if( key == item[t] )
            return t;
    return -1;
}
```

**Programa A.7**

Essa função devolve o índice da entrada encontrada se existir alguma; caso contrário, ela devolve -1.

### A.3.2 Pesquisa Binária

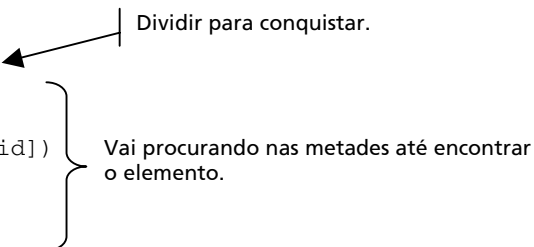
Se o dado a ser encontrado se encontrar de forma ordenada, pode ser utilizado um método muito superior para encontrar o elemento procurado. Esse método é a pesquisa binária que utiliza a abordagem “dividir e conquistar”. Ele primeiro verifica o elemento central. Se esse elemento é maior que a chave, ele testa o elemento central da primeira metade; caso contrário, ele testa o

elemento central da segunda metade. Esse procedimento é repetido até que o elemento seja encontrado ou que não haja mais elementos a testar.

Por exemplo, para encontrar o número 4 na matriz 1 2 3 4 5 6 7 8 9, uma pesquisa binária primeiro testa o elemento médio, nesse caso 5. Visto que é maior que 4, a pesquisa continua com a primeira metade ou 1 2 3 4 5. O elemento central agora é 3, que é menor que 4, então, a primeira metade é descartada. A pesquisa continua com 4 5. Nesse momento o elemento é encontrado.

A seguir, é demonstrada uma pesquisa binária para matrizes de caracteres.

```
int binary( char * item, int count, char key)
{
    int low, high, mid;
    low = 0;
    high = count - 1;
    while( low <= high )
    {
        mid = (low+high)/2;
        if( key<item[mid] )
            high = mid - 1;
        else if( key>item[mid] )
            low = mid + 1;
        else
            return mid;
    }
    return -1;
}
```



Dividir para conquistar.

Vai procurando nas metades até encontrar o elemento.

#### Programa A.8

Este programa pode ser adaptado para realizar pesquisas em qualquer tipo de matriz (inteiros ou estruturas, por exemplo). Exemplo de utilização das funções.

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

int binary( char * item, int count, char key);
int sequential_search( char * item, int count, char key );
void qs( char *item, int left, int right);

void main(void)
{
    int pos;
    char vetors[]="./~2r=-dfx-950]gojftg394a8@ugkj n#26yh";
    char vetorb[]="./~2r=-dfx-950]gojftg394a8@ugkj n#26yh";
```

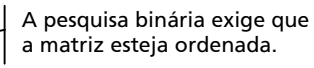
## 246 ♦ Programando em C para Linux, Unix e Windows

```
printf("\nMatriz desordenada [%s]", vetors );
printf("\nProcurando = sequencialmente");
pos = sequential_search(vetors, strlen(vetors), '=');
printf("\n= encontrado na posicao %d", pos );

printf("\nOrganizando a matriz.");
qs(vetorb, 0, strlen(vetorb)-1);
printf("\nMatriz ordenada [%s]", vetorb);

printf("\nProcurando = binariamente");
pos = binary(vetorb, strlen(vetorb), '=');
printf("\n= encontrado na posicao %d\n", pos );

}
```



A pesquisa binária exige que a matriz esteja ordenada.

### Programa A.9

#### Resultado do Programa A.9

```
Matriz desordenada [.,/~2r=-dfx-950]gojftg394a8@ugkj n#26yh]
Procurando = sequencialmente
= encontrado na posicao 6
Organizando a matriz.
Matriz ordenada [ #,--./0223456899=@]adffggghjjknortuxy~]
Procurando = binariamente
= encontrado na posicao 17
```



*Não desanimes. Frequentemente é a última chave  
do molho que abre a fechadura.  
(Anônimo)*

## B.1 Obtendo ajuda no Linux e Unix

O comando `man` é a maneira mais rápida de obter informações sobre um determinado comando. Para visualizar o *help on-line* de todas as funções da linguagem C no ambiente Linux e Unix, basta utilizar este comando para realizar a consulta.

Exemplos:

- o Para ver as opções de compilação do gcc: `man gcc`
- o Para ver como funciona a função `fopen` do C: `man fopen`
- o Para ver como funciona o comando `man`: `man man`

## B.2 Seções do Manual

Toda a documentação do sistema Linux e Unix está dividida em seções. Cada seção é responsável pela documentação de assuntos correlatos. Segue uma breve descrição das seções mais importantes:

- o Seção 1 – Contém os comandos destinados para o usuário comum. São a maioria dos comandos existentes no Linux e Unix.

- o Seção 2 – O sistema Linux e Unix oferece um conjunto de funções que podem ser chamadas dentro de programas C. Esta seção descreve cada uma destas funções disponibilizadas para o programador C.
- o Seção 3 – É a parte do manual que contém a documentação da biblioteca de funções disponibilizadas pelo compilador C padrão da máquina.

É claro que existem outras seções. Atualmente, devido à grande quantidade de softwares e pacotes disponíveis, existem várias seções.

## B.3 Divisão da Documentação

Cada documentação de um comando é dividida em partes. De todas elas podemos destacar como as mais importantes:

- o NAME (NOME) – Contém o nome do comando e uma breve descrição da função do mesmo.
- o SYNOPSIS (SINOPSE) – Contém todas as sintaxes aceitas pelo comando, bem como as opções aceitas e os argumentos esperados.
- o DESCRIPTION (DESCRIÇÃO) – Contém a descrição da função do comando, de forma detalhada. Inclui também a descrição de cada opção e argumento esperado pelo comando.
- o RETURN VALUE (VALOR DE RETORNO) – Indica o código de retorno (ou retornos) de um comando.
- o ERRORS (ERROS) – Esta parte relata as possíveis mensagens de erro que o comando pode emitir durante a sua execução. Pode sugerir correções ou verificações a serem feitas para sanar o problema.
- o SEE ALSO (VEJA TAMBÉM) – Na grande maioria dos casos, um comando está ligado à execução de diversos outros comandos. Nesta parte do manual são colocados todos os comandos citados no texto anterior ou que têm alguma relação com o comando.

## B.4 Exemplo de Utilização

As funções da linguagem C estão todas documentadas no sistema, bastando utilizar o comando `man` para ver o funcionamento das funções. Mas antes é preciso fazer algumas considerações. Utilizando a função `open` como exemplo:

```
$> man open
```

Obtém-se o seguinte trecho (aqui, modificado e comentado) na documentação no sistema Linux:



Seção do manual. No caso, a função `open` é considerada uma chamada de sistema (*system call*).

OPEN(2) System calls

## NAME

`open`, `creat` - open and possibly create a file or device

## SYNOPSIS

Os arquivos necessários (includes) para a função funcionar.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
```

Protótipo da função. Quais os parâmetros que a função recebe e o valor que ela retorna.

## DESCRIPTION

The `open()` system call is used to convert a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with `read`, `write`, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This call creates a new open file, not shared with any other process. (But shared open files may arise via the `fork(2)` system call.) The new file descriptor is set to remain open across `exec` functions (see `fcntl(2)`). The file offset is set to the beginning of the file.

Descrição detalhada a respeito do funcionamento da função.

## RETURN VALUE

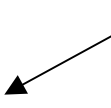
Descrição detalhada sobre o retorno da função.

`open` and `creat` return the new file descriptor, or `-1` if an error occurred (in which case, `errno` is set appropriately). Note that `open` can open device special files, but `creat` cannot create them - use `mknod(2)` instead.

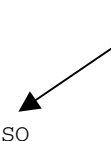
On NFS file systems with UID mapping enabled, `open` may return a file descriptor but e.g. `read(2)` requests are denied with `EACCES`. This is because the client performs `open` by checking the permissions, but UID mapping is performed by the server upon `read` and `write` requests.

## 250 ♦ Programando em C para Linux, Unix e Windows

If the file is newly created, its atime, ctime, mtime fields are set to the current time, and so are the ctime and mtime fields of the parent directory. Otherwise, if the file is modified because of the O\_TRUNC flag, its ctime and mtime fields are set to the current time.

ERRORS  Descrição detalhada sobre os erros ocorridos e dicas de como resolver o problema.

EEXIST pathname already exists and O\_CREAT and O\_EXCL were used.  
EISDIR pathname refers to a directory and the access requested involved writing (that is, O\_WRONLY or O\_RDWR is set).

SEE ALSO  Outras funções relacionadas com esta. No caso, como foi consultada a função open (que trata de abertura de arquivos), o comando man apresenta outras funções que também manipulam arquivos (o número entre parênteses indica o número da seção do manual).

read(2), write(2), fcntl(2), close(2), link(2),  
mknod(2), mount(2), stat(2), umask(2), unlink(2), socket(2),  
fopen(3), fifo(4)



*Não é importante que você entenda o que eu estou  
fazendo ou por que você está me pagando tanto dinheiro.  
O importante é que você continue a fazer assim.*  
(Anônimo)

**C**ompilar é transformar um arquivo legível para o homem (chamado de código-fonte, *source file* em inglês) para um arquivo legível para a máquina (binário, *binary*). Quem faz esse trabalho é o compilador.

O compilador C/C++ padrão no Linux é o `gcc`. Muitas distribuições vêm com o `gcc` incluído. O `gcc` é um dos compiladores mais versáteis e avançados existentes. O `gcc` suporta todos os padrões modernos do C atualmente usados, como o padrão ANSI C, assim como muitas extensões específicas do próprio `gcc`. Utilizar o `gcc` é simples. Vejamos alguns exemplos:

- o Compila o programa `hello.c` e cria o binário `hello` (opção `-o` do `gcc`)  
`$> gcc hello.c -o hello`
- o Compila dois programas (`prog1.c` e `prog2.c`) e cria o binário `programa`.  
`$> gcc prog1.c prog2.c -oprograma`
- o Compila o programa `fat.c`, gera o binário `fat` e indica para o compilador *linkeditar* a biblioteca matemática junto com binário (opção `-l` do `gcc`).  
`$> gcc fat.c -o fat -lm`
- o Compila o programa `def.c`, gera o binário `def` e cria a diretiva `PLATAFORMA` (opção `-D` do `gcc`) com o valor `Linux` (veja o funcionamento no capítulo de Pré-compilação).  
`$> gcc def.c -DPLATAFORMA=Linux -odef`

## 252 ♦ Programando em C para Linux, Unix e Windows

- o Compila o programa `inc.c`, gera o binário `inc`. A opção `-I` indica o caminho para os includes (*headers*) do específicos do projeto e a opção `-L` indica o caminho das bibliotecas específicas do projeto.  
`$> gcc inc.c -I../includes -L../libs -lmylib -o../bin/inc`
- o Compila o `progr1.c` e gera o binário `progr1`. A opção `-O` é para otimização do código gerado.  
`$> gcc -O -oprogr1 progr1.c`

O `libc` (`glibc`) é uma biblioteca usada por quase todos os programas do Linux; o `libjpeg` é uma biblioteca usada em todos os programas que trabalham com o formato JPEG; e assim por diante. No sistema Linux essas bibliotecas são divididas em dois pacotes: um para ser usado por programas já compilados (`glibc` e `libjpeg`, por exemplo), e um para ser usado na compilação de programas que dependem dele (`glibc-devel` e `libjpeg-devel`, por exemplo). Portanto, para compilar programas mais complexos, será necessário ter esses dois pacotes instalados.

Se o programa é constituído por vários arquivos, e normalmente usam bibliotecas e *header-files* externos, será necessário compilar todos eles e juntá-los (*link*) corretamente. Para automatizar esse procedimento, usa-se o comando `make`. Este comando lê um arquivo chamado `Makefile`, onde estará o "roteiro" necessário para a compilação do programa. O objetivo básico do `make` é permitir que seja construído um programa em pequenas etapas. Se muitos arquivos fontes compuserem o executável final, será possível alterar um arquivo e reconstruir o executável sem ter a necessidade de compilar os demais programas. Para tal, é necessário criar um arquivo chamado `Makefile`.

O `make` pode ser composto de várias linhas, cada um indicando como o executável deve ser construído. Normalmente, existem dependências entre as linhas, indicando a ordem de execução das linhas. A disposição das linhas (entradas) dentro do arquivo `Makefile` não importa, pois o `make` irá descobrir qual a ordem correta. O `make` exige alguns cuidados para a criação do arquivo:

- o Sempre colocar uma tabulação no começo de um comando, nunca espaços. Não deve ser utilizada uma tabulação antes de qualquer outra linha.
- o O símbolo `#` (sustenido, tralha, cerquilha ou jogo da velha) indica um comentário na linha.
- o Uma barra invertida no final de uma linha indica que ela irá prosseguir na próxima linha. Ótimo para comandos longos.

Vejamos um exemplo:

```
install: all
        mv manipconfig /usr/local
        mv delbinario /usr/local

all: manipconfig delbinario

manipconfig: cria.o altera.o exclui.o consulta.o editor.o \
             manipula.o principal.o
        gcc -L/home/laureano/libs -o cria.o altera.o exclui.o \
             consulta.o editor.o manipula.o principal.o

delbinario: del.c main.c
            gcc -o del.o main.o

cria.o: cria.c
        gcc -c cria.c

altera.o: altera.o
        gcc -c altera.c

exclui.o: exclui.c
        gcc -c exclui.c

consulta.o: cosulta.c
        gcc -c consulta.c

editor.o: editor.c
        gcc -c editor.c

manipula.o: manipula.c
        gcc -c manipula.c

principal.o: principal.c
            gcc -c principal

del.o: del.c
        gcc -c del.c

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

Para executar os comandos do sistema operacional, é necessário verificar a precedência all...

...que indica que estes arquivos binários devem existir.

Para o binário manipconfig ser criado....

...é necessário que os arquivos objetos existam.

Indica que o comando continua na próxima linha.

Para o objeto ser gerado, é necessário compilar o programa antes. A opção -c do gcc indica que é somente para compilar (gerar o arquivo .o).

Aqui sempre deve vir precedido de uma tabulação (TAB).

## 254 ♦ Programando em C para Linux, Unix e Windows

Os comandos do arquivo Makefile anterior seriam equivalentes aos seguintes comandos (se todos fossem digitados):

```
$> gcc -c main.c
$> gcc -c del.c
$> gcc -c principal
$> gcc -c manipula.c
$> gcc -c editor.c
$> gcc -c consulta.c
$> gcc -c exclui.c
$> gcc -c altera.c
$> gcc -c cria.c
$> gcc del.o main.o -o delbinario
$> gcc -L/home/laureano/libs cria.o altera.o exclui.o consulta.o
editor.o manipula.o principal.o -o manipconfig
$> mv manipconfig /usr/local
$> mv delbinario /usr/local
```

Para maiores informações sobre a utilização e opções dos comandos `gcc` e `make`, veja o manual *on-line (help)* do sistema Linux.

# Programando em C

## Apêndice D

# Utilizando o LCC-Win32

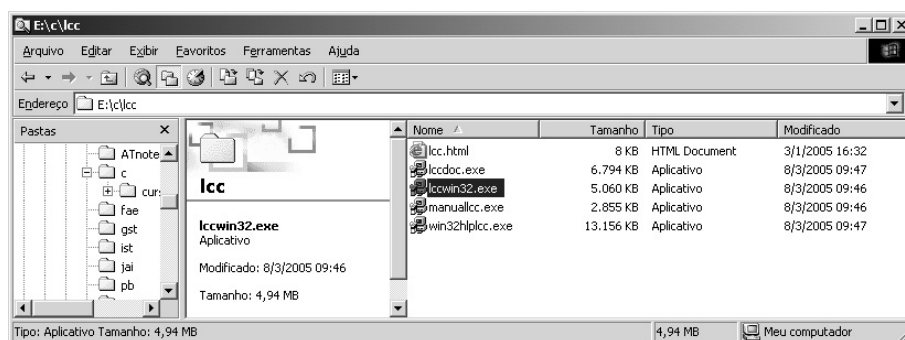
*Quem pode faz. Quem não pode ensina. Quem não sabe ensinar vira chefe.*  
(Anônimo)

O LCC-Win32 é um compilador C para Windows desenvolvido por Jacob Nava, que por sua vez foi baseado no compilador C desenvolvido por Dave Hanson e Chris Fraser.

## D.1 Instalação

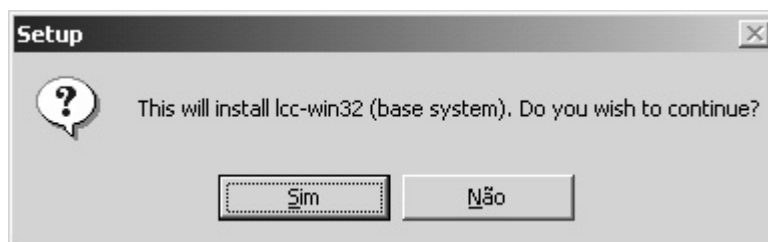
Ele pode ser baixado gratuitamente em <http://www.cs.virginia.edu/~lcc-win32/>. Sua instalação é simples e rápida, bastando dar um duplo clique no executável `lccwin32.exe`. Os próximos passos irão auxiliá-lo a realizar a instalação.

1º Passo – Clicar (duplo clique) no executável do LCC-Win32 (`lccwin32.exe`).

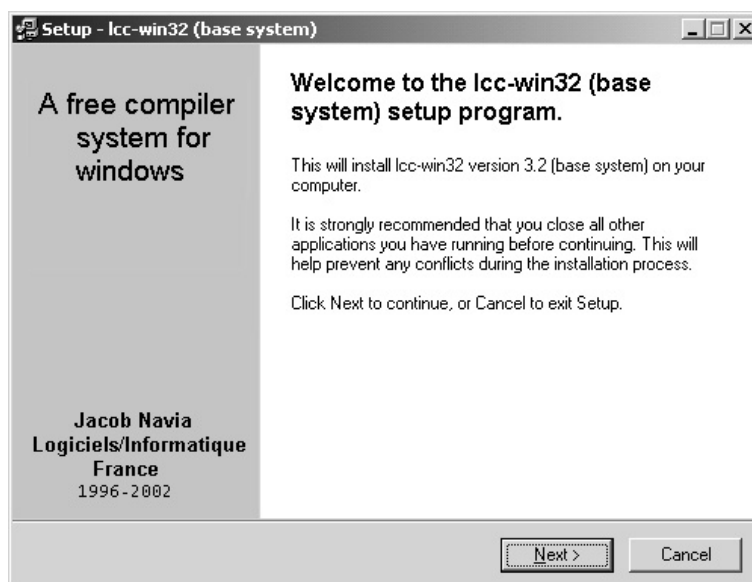


## 256 ♦ Programando em C para Linux, Unix e Windows

2º Passo – Confirmar a instalação. Basta clicar com o mouse no botão *Sim* ou *Yes* (depende da língua utilizada no sistema operacional).

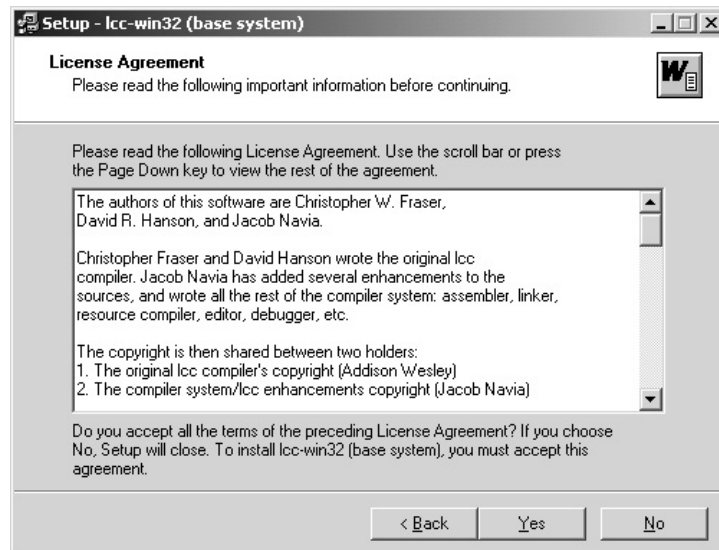


3º Passo – A instalação do LCC-Win32 é do tipo NNF (*next, next and finish*). Basta clicar com o mouse no botão *Next*.

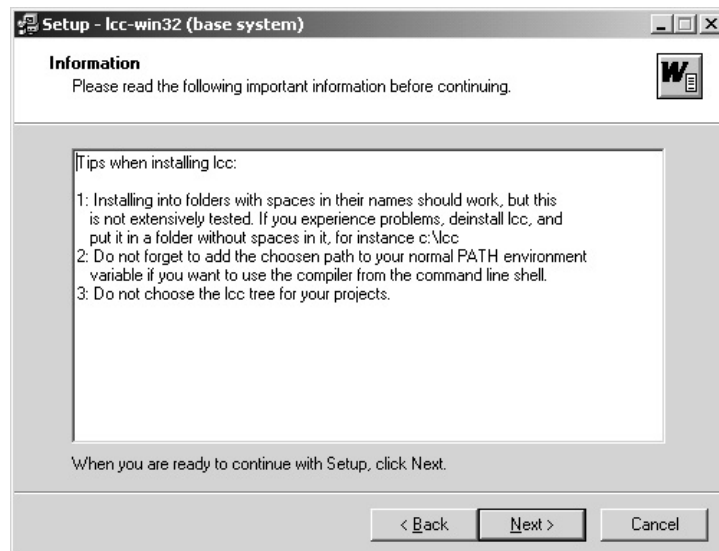




4º Passo – Licença de uso. A licença diz que o software pode ser utilizado para fins pessoais ou didáticos. Clique em **Yes** (Sim) para aceitar os termos de uso e continuar com a instalação.

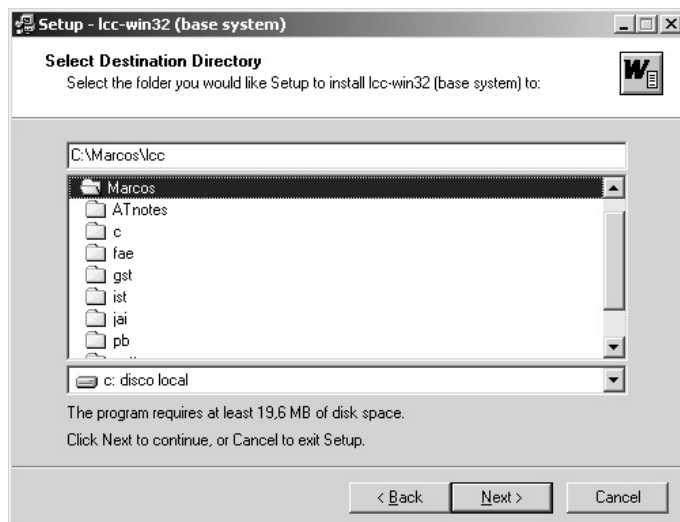


5º Passo – O instalador avisa os procedimentos que serão realizados a seguir. Clique em **Next**.

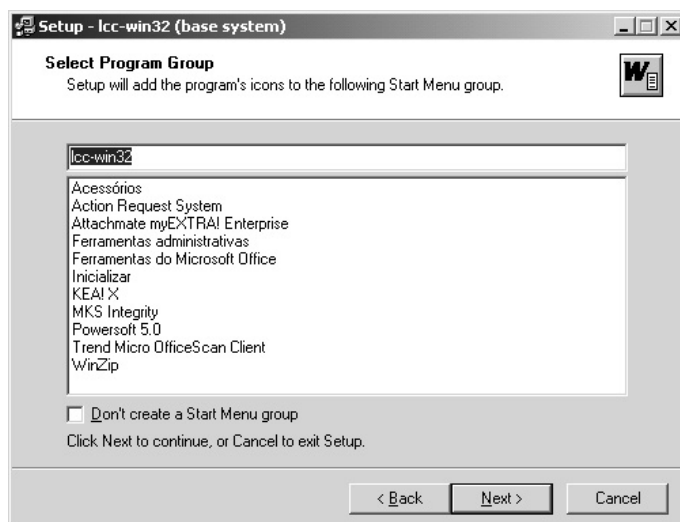


## 258 ♦ Programando em C para Linux, Unix e Windows

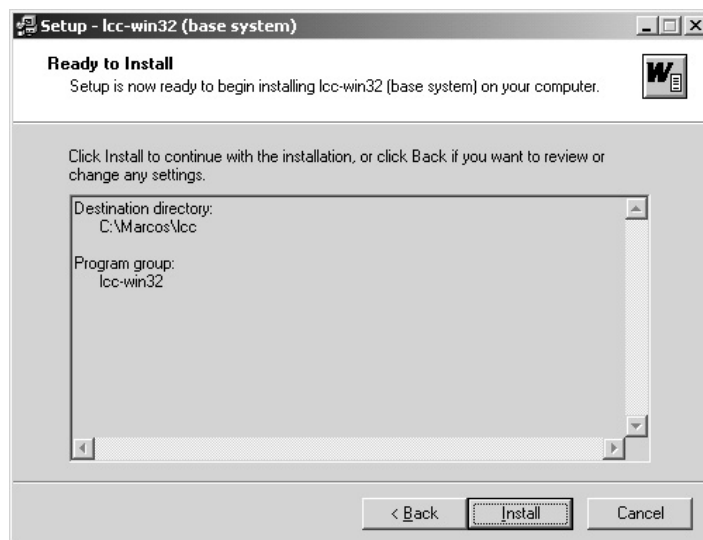
6º Passo – O diretório padrão para instalação é `c:\lcc`. Se quiser escolher outro diretório ou disco rígido (D:, por exemplo) para instalação, basta selecionar no *browser* apresentado. O programa de instalação indica qual o espaço necessário para a instalação. Após a escolher o local da instalação, clique em *Next*.



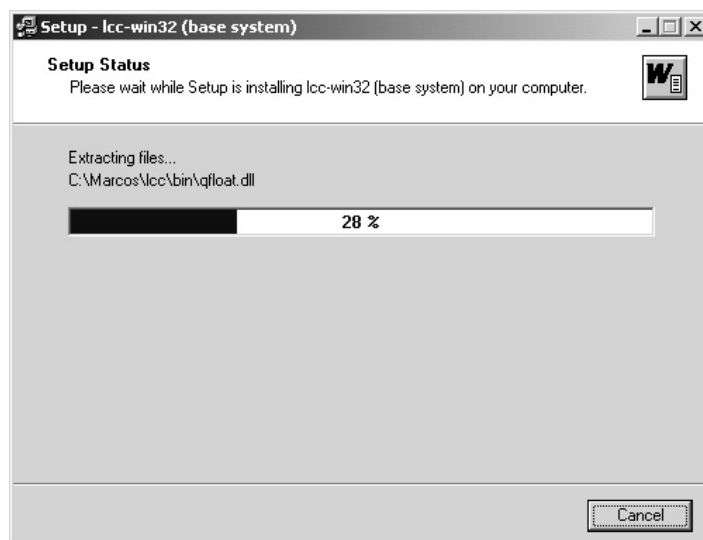
7º Passo – Selecionar em qual grupo do Windows o programa será incluído. A sugestão do programa é criar um grupo novo (`lcc-win32`). Será este grupo que irá aparecer no menu do Windows.



8º Passo – Após a configuração da instalação (conforme passos anteriores), a instalação irá realmente ocorrer após você clicar em *Install*.

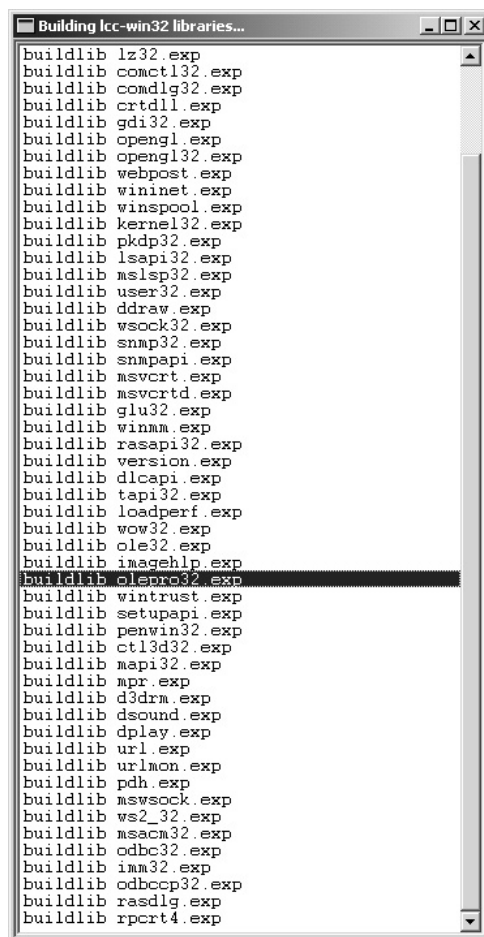


9º Passo – Processo de instalação. Dependendo do seu computador, esta operação pode demorar até 5 minutos... basta aguardar o término...

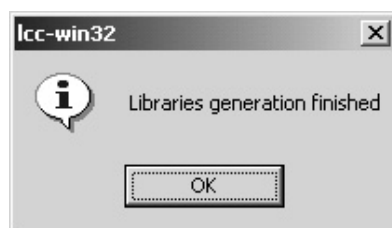


## 260 ♦ Programando em C para Linux, Unix e Windows

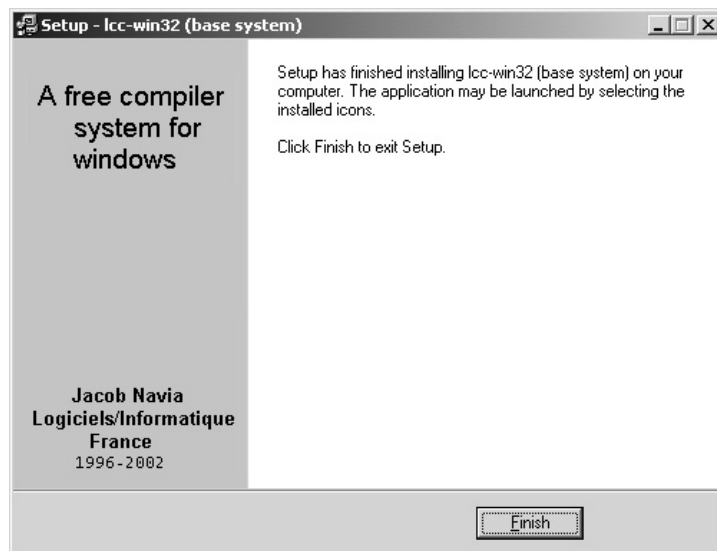
10º Passo – Continuando o processo de instalação. O programa está criando as bibliotecas do sistema.... basta aguardar o término da operação.



11º Passo – Notificação do término da geração das bibliotecas. Clicar em OK.



12º Passo – Programa instalado! Basta clicar em `Finish` para finalizar o programa.



Agora você já pode utilizar o programa para criar, compilar e executar os seus programas.

## D.2 Criando Projetos no LCC-Win32

O que vem a ser um projeto em C? Você provavelmente já deve saber que um compilador é um programa que transforma um arquivo contendo um programa (arquivo fonte) escrito em uma linguagem de alto nível, como o C, para uma linguagem que a máquina é capaz de "entender", ou programa executável.

A linguagem C permite que um programa seja decomposto em diversos módulos, onde cada módulo pode ficar armazenado em arquivos diferentes. Isso permite que, principalmente em projetos grandes, um programa possa ser desenvolvido por uma equipe de programadores, cada um trabalhando em um módulo diferente do programa. O compilador deve aceitar, portanto, diversos arquivos de entrada para gerar um arquivo executável. É comum combinar módulos segundo alguma funcionalidade, criando-se bibliotecas de funções, para que possam ser reutilizados em outros programas. Uma biblioteca do C que vai ser freqüentemente utilizada é a `stdio.h`, que contém várias funções para a entrada e saída de dados.

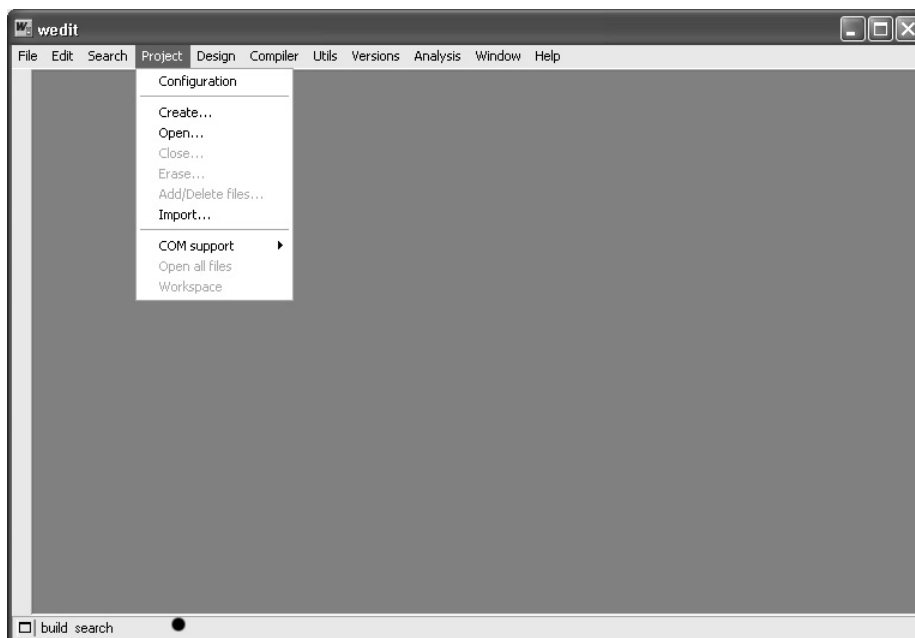
Com a utilização de diversos arquivos (várias centenas ou mais para projetos muito grandes) que podem conter dependências mútuas (partes de um módulo que chamam partes de outro módulo e vice-versa), o compilador deve ser instruído sobre quais arquivos fontes ele deve compilar, em que ordem, qual o nome dos executáveis a serem gerados etc. Essa informação é tipicamente armazenada em outro arquivo, chamado `Makefile`. Makefiles possuem uma sintaxe bem restrita e podem se tornar bastante complicados. Para simplificar esse processo, o ambiente do LCC-Win32 (o WEdit) oferece uma forma mais intuitiva de manter a informação necessária para compilar um programa, que são os projetos.

Toda vez que você for escrever um novo programa no LCC-Win32, você deverá criar um novo projeto. Nesse projeto você poderá incluir arquivos, modificar propriedades da aplicação e até controlar a versão do seu programa, que o ajuda a manter uma documentação sobre as mudanças ocorridas ao longo do desenvolvimento do projeto. Veja o manual do LCC-Win32 para maiores detalhes.

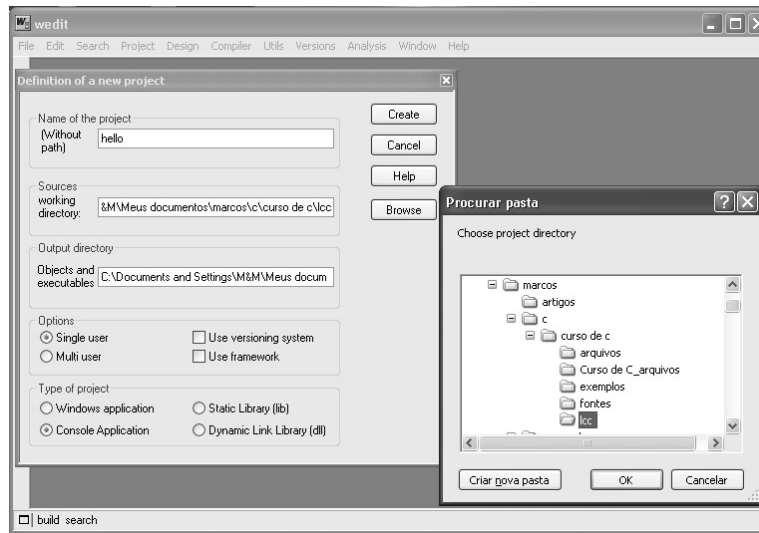
Agora que você sabe o que significa um projeto, você deverá entender melhor o procedimento exigido para a criação de um programa no LCC-Win32.

## D.2.1 Criando um projeto

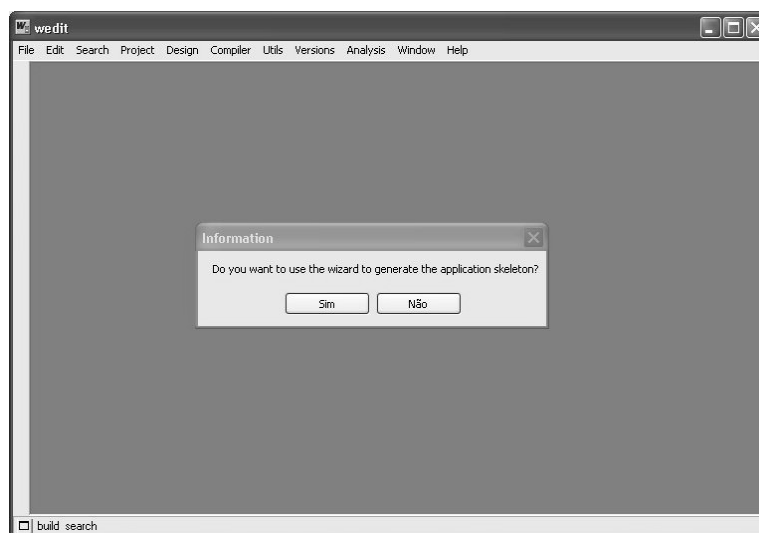
1º Passo – No item “Project” do menu do LCC-Win32 selecione a opção “Create...”



2º Passo – Especifique o nome do projeto e o local onde ele vai ficar armazenado. Depois clique em “Create”. Para fins de aprendizado, vamos trabalhar somente com aplicações de console (“Console Application”), ou seja, que irão em uma tela parecida com a do sistema DOS.

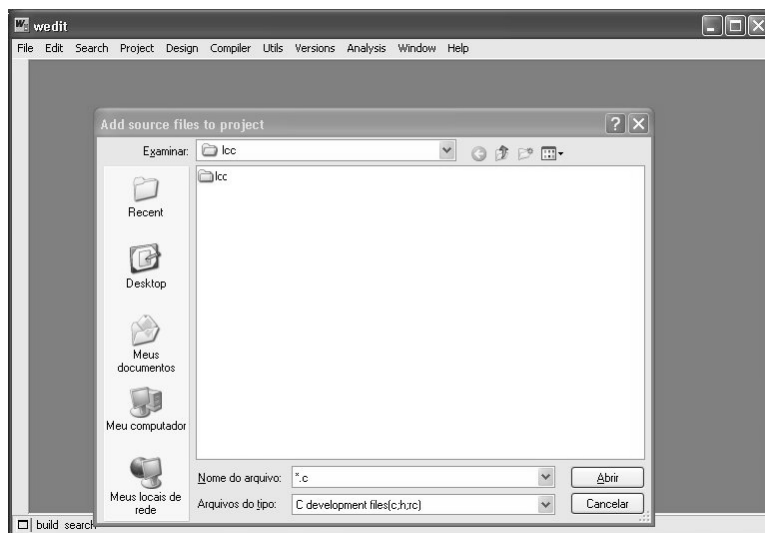


3º Passo – O LCC-Win32 pode criar um esqueleto básico para você de várias aplicações, bastando só complementar as rotinas e funções. No nosso caso, clique em “Não”.

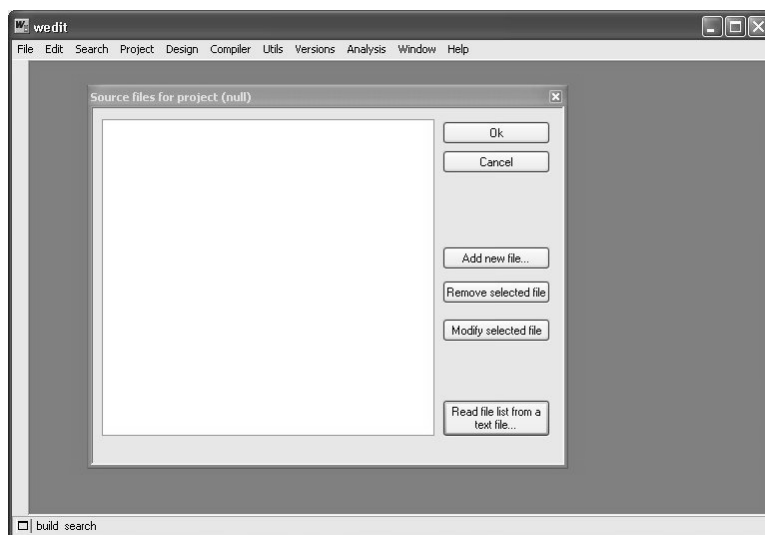


## 264 ♦ Programando em C para Linux, Unix e Windows

4º Passo – Agora temos que selecionar um arquivo que conterá o nosso programa-fonte. Se você não tiver criado este arquivo ainda, clique em “Cancelar”. Este tutorial assume que você ainda não tem um programa pronto.

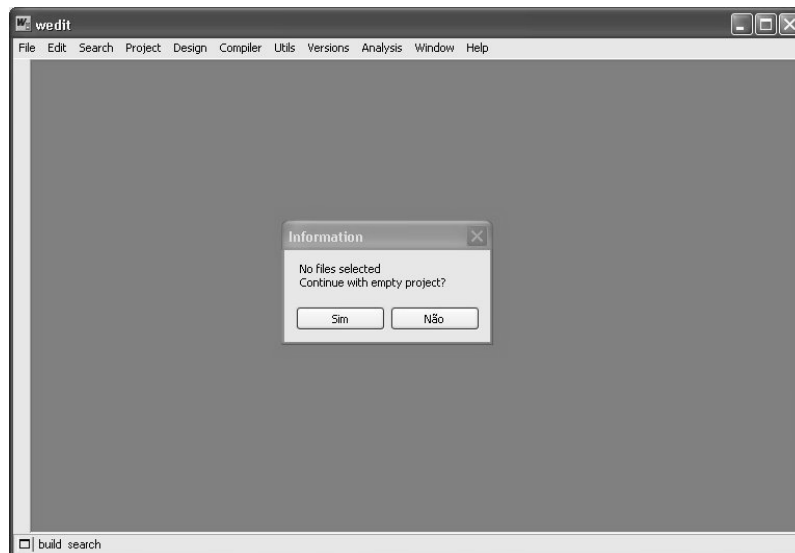


5º Passo – O LCC-Win32 irá demonstrar uma tela de administração de arquivos do seu projeto. Nesta tela você poderá incluir ou excluir arquivos do projeto. Como ainda não temos um arquivo, clique em “Cancel”.

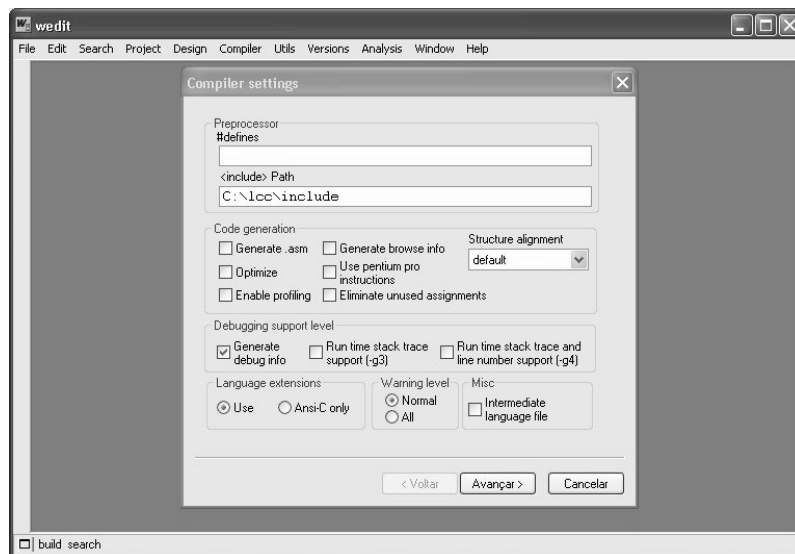




6º Passo – O LCC-Win32 pergunta se você tem certeza que quer gerar um projeto vazio (sem arquivos fontes). Confirme a operação clicando em “Sim”.

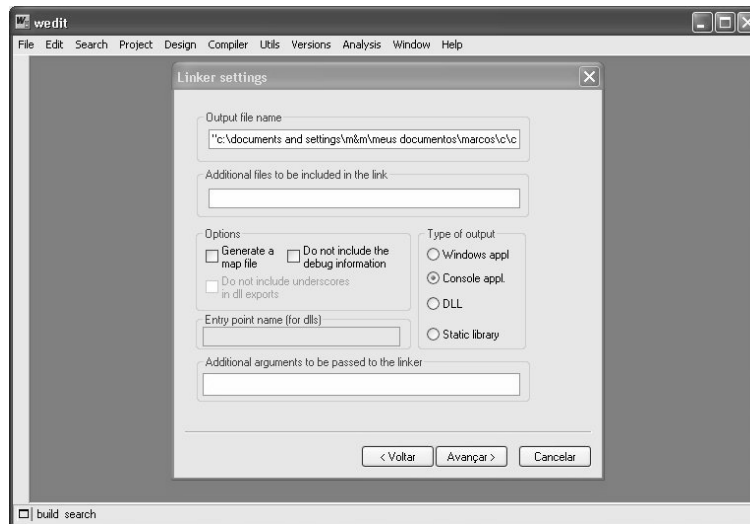


7º Passo – Agora vamos configurar alguns itens da compilação do programa. Esta é a tela onde você irá definir os #defines da aplicação em tempo de execução. Clique em “Avançar”.

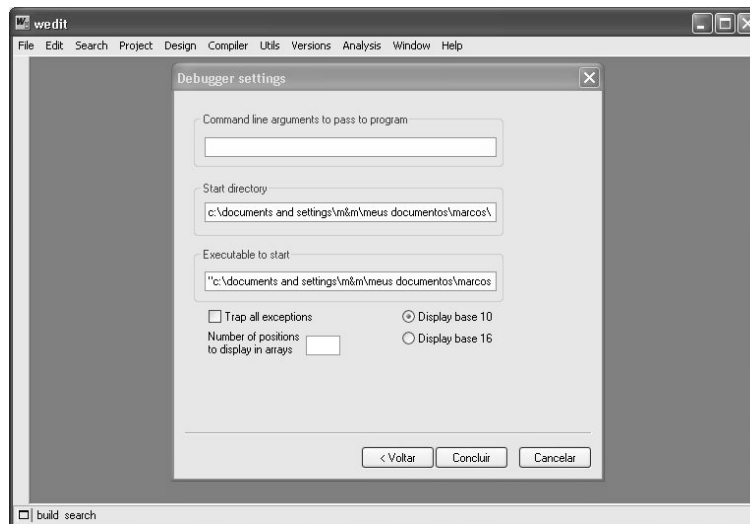


## 266 ♦ Programando em C para Linux, Unix e Windows

8º Passo – Nesta tela você configura onde será gerado o seu programa executável e eventuais arquivos que devem ser linkados ao seu programa. Clique em “Avançar”.



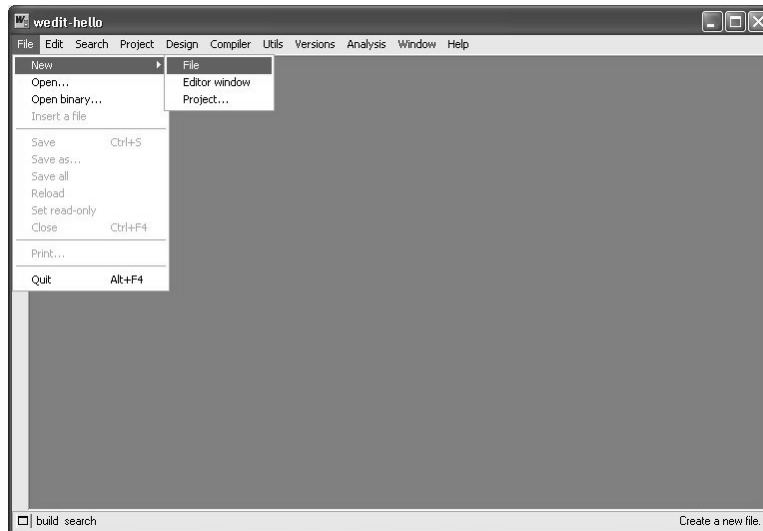
9º Passo – Aqui você configura a fonte das aplicações e os parâmetros que serão passados ao seu programas (recebidos na função `main`, normalmente como `argv` e `argc`). Clique em “Concluir”.



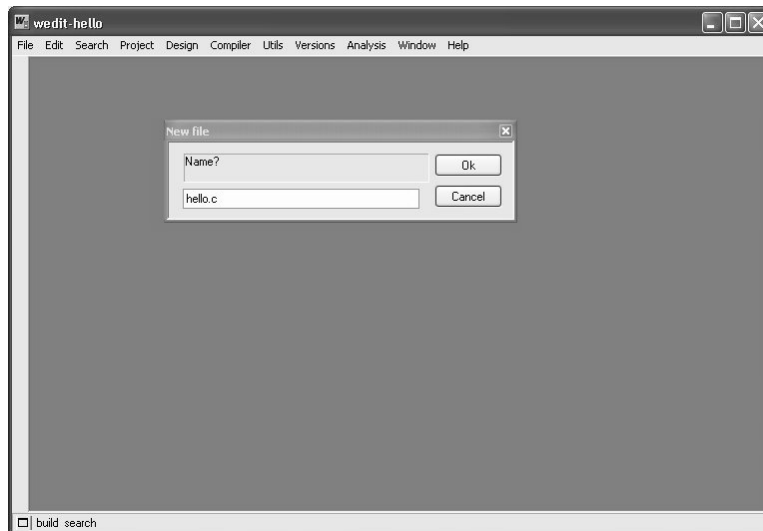
Parabéns. O seu projeto está concluído. Agora temos que criar um programa para incluir no projeto, compilar e executar.

## D.3 Criando um Programa e Compilando

1º Passo – Na opção “File” do menu, selecione “New” e depois “File” para criar um novo programa-fonte em C.

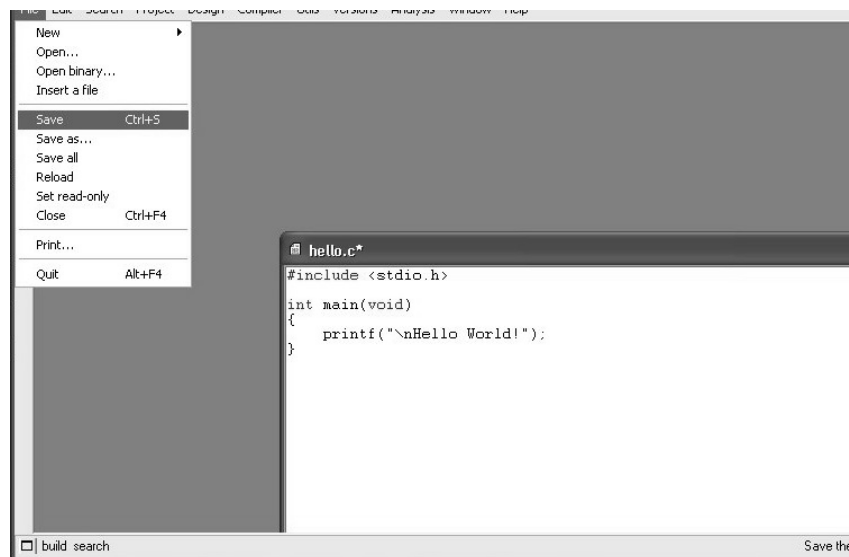


2º Passo – Informe o nome do programa. No nosso caso será o `hello.c`.  
**IMPORTANTE:** Lembre-se sempre de colocar a extensão `.c` nos arquivos-fontes.

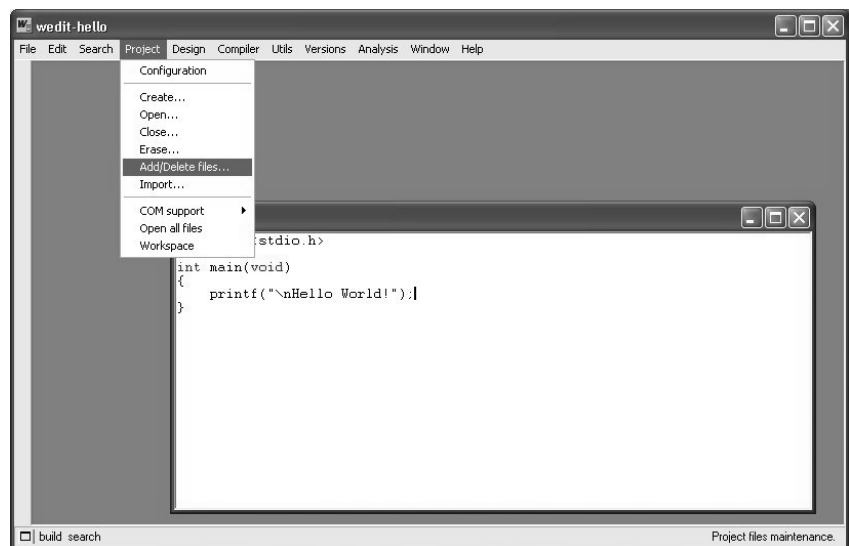


## 268 ♦ Programando em C para Linux, Unix e Windows

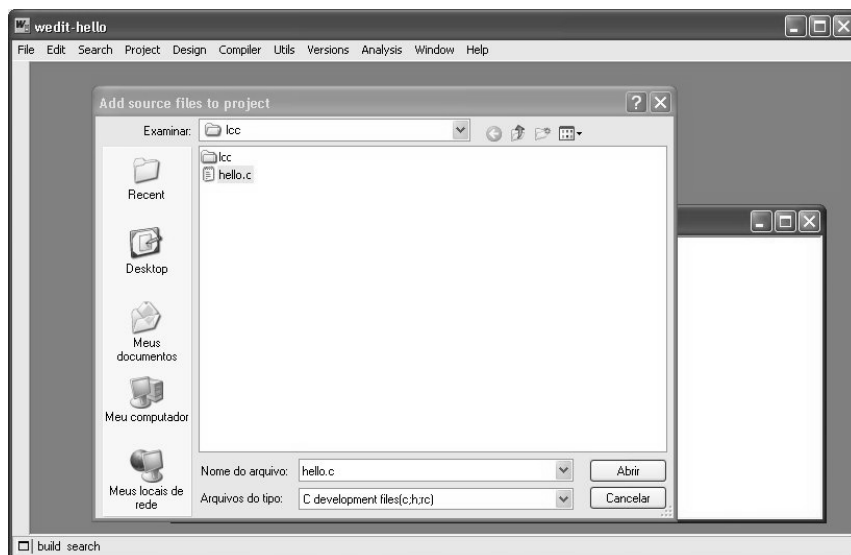
3º Passo – Salve o seu programa. Você pode pressionar **CTRL+S** ou ir até a opção “File” do menu e depois em “Save”.



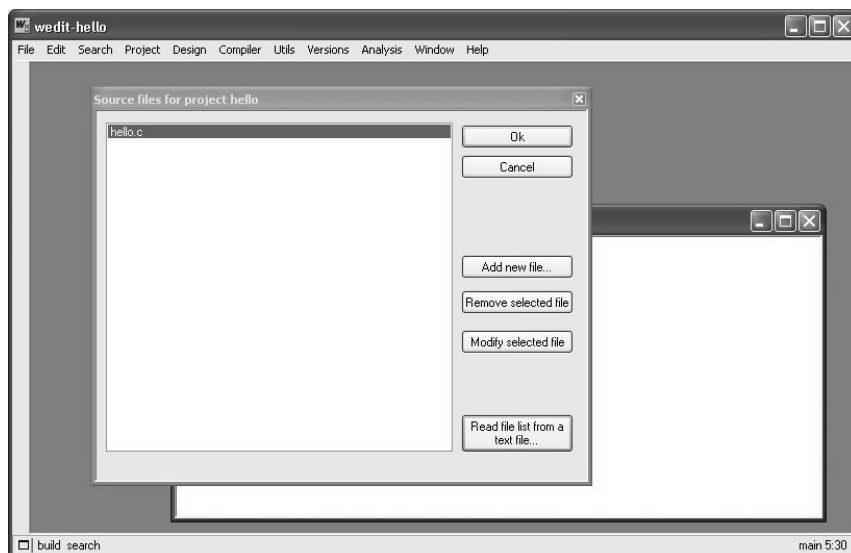
4º Passo – Agora temos que incluir o novo programa no projeto criado anteriormente. Vá até a opção “Project” do menu e selecione “Add/Delete files...”



5º Passo – Selecione o arquivo correspondente ao seu programa e clique em “Abrir”.

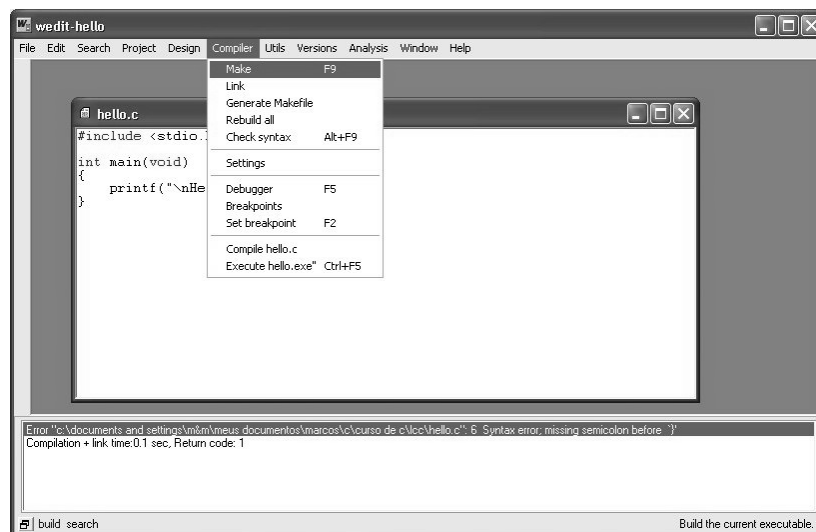


6º Passo – O LCC-Win32 irá demonstrar uma tela de administração de arquivos do seu projeto. Nesta tela você poderá incluir ou excluir arquivos do projeto. Clique em “OK” para confirmar o novo arquivo no projeto.

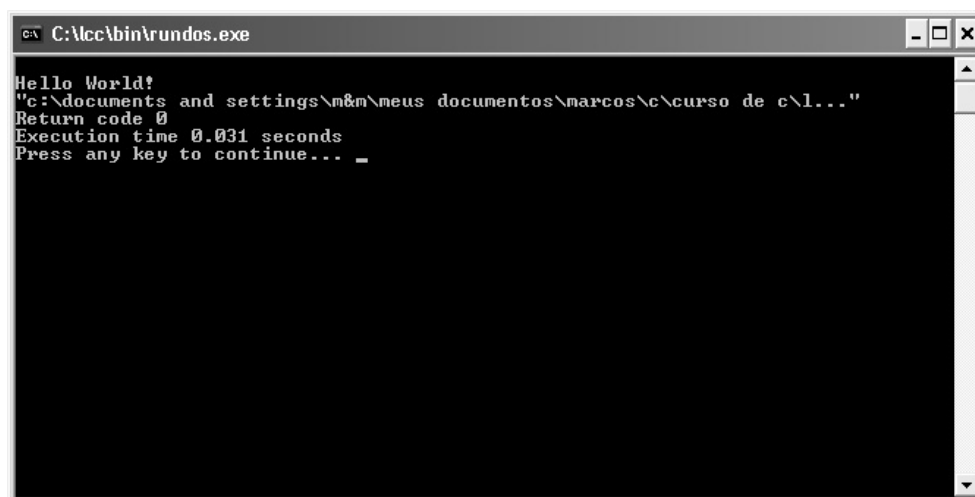


## 270 ♦ Programando em C para Linux, Unix e Windows

7º Passo – Agora é só compilar o programa e, se estiver tudo certo, executá-lo. Para compilar, você pode pressionar **F9** ou ir até a opção “Compiler” do menu e selecione “Make”. Para executar o programa pressione **CTRL+F5** ou vá até a opção “Compiler” do menu e selecione “Execute”. Se o seu programa contiver algum erro de sintaxe, irá aparecer em uma janela abaixo a linha e o erro causado.



8º Passo – O seu programa está executando. Parabéns...





*Autodidata: ignorante por conta própria.*

Mário Quintana, poeta brasileiro

**A** linguagem de programação C possui um enorme conjunto de funções. Muitas das funções disponíveis são específicas de um fabricante ou de um sistema operacional. Este guia apresenta a referência das funções mais utilizadas nos sistemas Linux e Unix (montada a partir do manual do sistema). Ela traz o protótipo da função (sua assinatura) com seus parâmetros, os arquivos cabeçalhos necessários para o funcionamento e o retorno da função. A finalidade desta seção é listar as funções existentes (e seu significado) para facilitar a busca do programador. Para obter maiores informações referentes às funções (forma de uso, exemplos etc.) é necessário consultar o manual do sistema (`man`).

Para a montagem deste apêndice foram consultados os manuais do sistema Unix AIX (IBM) e Linux (Fedora). Algumas chamadas podem ser diferentes de sistema para sistema. Nas pesquisas realizadas foi detectado que algumas chamadas têm a mesma função mas a forma diferente (alguns casos foram listados duplicados). Recomenda-se sempre consultar o manual *on-line* do seu sistema (sistemas Linux são independentes da distribuição, assim como sistemas Unix também o são).

Sempre que uma função resultar em erro, verifique a variável `errno`.

Notações utilizadas nesta seção:

- o > – Maior que
- o < – Menor que
- o – Igual
- o != – Diferente de
- o # – Quantidade

<b>_exit</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>void _exit(int status);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Termina o programa “imediatamente”, ou seja, o programa é terminado e o controle passa automaticamente para o <i>kernel</i> do sistema operacional.	
<b>abort</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void abort(void);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Causa o término de um programa e causa o sinal SIGABRT.	
<b>abs</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int abs(int num);</code>	
<b>Retorno:</b> Número absoluto.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Devolve o número absoluto de um número.	
<b>accept</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int accept(int s, struct sockaddr *addr, socklen_t *addrlen);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/socket.h&gt;</code>
<b>Descrição:</b> Aceita uma conexão em um <i>socket</i> . É usada com 'sockets' baseados em conexão do tipos (SOCK_STREAM, SOCK_SEQPACKET e SOCK_RDM)	
<b>access</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int access(const char *pathname, int mode);</code>	
<b>Retorno:</b> 0 se ok -1 se houve algum erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Verifica as permissões de acesso de um arquivo.	
<b>acos</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double acos(double num);</code>	
<b>Retorno:</b> Arco co-seno ou erro de domínio.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Retorna arco co-seno do número, <i>num</i> deve estar entre 1 e -1.	



<b>alarm</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>unsigned int alarm(unsigned int seconds);</code>	
<b>Retorno:</b> 0 ou nsegundos setados anteriormente.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Envia o sinal SIGALRM para o processo após ter passado os segundos setados.	
<b>alloca</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *alloca(int size);</code>	
<b>Retorno:</b> Ponteiro para a memória ou NULL em caso de erro.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Alocação dinâmica de memória. A memória é liberada automaticamente ao término da função que a chamou, portanto a função <code>free</code> não deve ser chamada para liberar o espaço de memória.	
<b>asctime</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *asctime(const struct tm *tm);</code>	
<b>Retorno:</b> Ponteiro para <i>string</i> contendo a data/hora.	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna a data em formato string especificada pelo conteúdo da <code>struct tm</code> .	
<b>asin</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double asin(double num);</code>	
<b>Retorno:</b> Arco seno ou erro de domínio.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Retorna arco seno do número, <code>num</code> deve estar entre 1 e -1.	
<b>assert</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void assert (int expression);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;assert.h&gt;</code>
<b>Descrição:</b> Abortar a execução do programa se a premissa for falsa.	
<b>atan</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double atan(double num);</code>	
<b>Retorno:</b> Arco tangente.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Retorna arco tangente do número.	
<b>atexit</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int atexit(void (*function)(void));</code>	
<b>Retorno:</b> 0 se OK != 0 se houve erro.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Registra funções para serem executadas após o término normal de um programa.	

**atof****Nível:** 3**Protótipo:** `double atof(const char *str);`**Retorno:** Número real.**Header:** `#include <stdlib.h>`**Descrição:** Converte uma string para um número do tipo `double`.**atoi****Nível:** 3**Protótipo:** `int atoi(const char *str);`**Retorno:** Número inteiro.**Header:** `#include <stdlib.h>`**Descrição:** Converte uma string para um número do tipo `int`.**atol****Nível:** 3**Protótipo:** `long atol(const char *str);`**Retorno:** Número inteiro.**Header:** `#include <stdlib.h>`**Descrição:** Converte uma string para um número do tipo `long`.**atoll****Nível:** 3**Protótipo:** `long long atoll(const char *nptr);`**Retorno:** Número inteiro.**Header:** `#include <stdlib.h>`**Descrição:** Converte uma string para um número do tipo `long long`.**basename****Nível:** 3**Protótipo:** `char *basename(char *path);`**Retorno:** Ponteiro para string**Header:** `#include <libgen.h>`**Descrição:** Manipulação (parse) de nome de arquivos. Captura o nome do arquivo a partir do caminho completo.**bind****Nível:** 2**Protótipo:** `int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);`**Retorno:** 0 se OK

-1 se houve erro

**Header:** `#include <sys/types.h>``#include``<sys/socket.h>`**Descrição:** Associa um nome a um *socket*.**bsearch****Nível:** 3**Protótipo:** `void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));`**Retorno:** Ponteiro para o item**Header:** `#include <stdlib.h>`

NULL se não achar

**Descrição:** Busca um item semelhante ao indicado por *key* num vetor com *nmemb* itens iniciado no endereço indicado por *base*. Os itens têm tamanho *size*.

<b>calloc</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *calloc(size_t nmemb, size_t size);</code>	
<b>Retorno:</b> Ponteiro para a memória ou <code>NULL</code> em caso de erro.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Alocação dinâmica de memória.	
<b>cbrt</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double cbrt( double x );</code>	
<b>Retorno:</b> Raiz cúbica.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Retorna a raiz cúbica.	
<b>ceil</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double ceil(double num);</code>	
<b>Retorno:</b> Menor número possível maior que <code>num</code> .	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Menor número possível que não seja menor que número, por exemplo, <code>ceil(1.03)</code> devolverá <code>2.0</code> .	
<b>chdir</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int chdir(const char *path);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Mudança do diretório de trabalho.	
<b>chmod</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int chmod(const char *path, mode_t mode);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code>
<b>Descrição:</b> Mudança de permissões de um arquivo.	
<b>chown</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int chown(const char *path, uid_t owner, gid_t group);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Muda o usuário ( <i>owner</i> ) de um arquivo.	
<b>chroot</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int chroot(const char *path);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Altera o diretório raiz para aquele especificado.	
<b>clearerr</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void clearerr(FILE *stream);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Limpa os indicadores de erro de um arquivo.	

<b>clearenv</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int clearenv(void);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Elimina/apaga todas as variáveis de ambiente e seu conteúdo.	
<b>clock</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>clock_t clock(void);</code>	
<b>Retorno:</b> O tempo de CPU usado no formato <code>clock_t</code> .	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna uma aproximação do tempo de processamento usado pelo programa. Para obter o número de segundos, divida por <code>CLOCKS_PER_SEC</code> .	
<b>close</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int close(int fd);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Fecha o arquivo apontado por <code>fd</code> (aberto por <code>open</code> ).	
<b>closedir</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int closedir(DIR *dir);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;dirent.h&gt;</code>
<b>Descrição:</b> Fecha um diretório (aberto por <code>opendir</code> )	
<b>closelog</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void closelog(void);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;syslog.h&gt;</code>
<b>Descrição:</b> Fecha o descritor para envio de logs (ver <code>syslog</code> ).	
<b>connect</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/socket.h&gt;</code>
<b>Descrição:</b> Inicializa uma conexão socket.	
<b>copysign</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double copysign(double x, double y);</code>	
<b>Retorno:</b> Número.	<b>Header:</b>
<b>Descrição:</b> Retorna um valor cujo valor absoluto é igual a <code>x</code> , mas cujo sinal é igual ao de <code>y</code> .	

<b>cos</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double cos(double num);</code>	
<b>Retorno:</b> Co-seno.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Devolve o co-seno do número. O número deve estar em radianos.	
<b>creat</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int creat(const char *pathname, mode_t mode);</code>	
<b>Retorno:</b> >0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code> <code>#include &lt;fcntl.h&gt;</code>
<b>Descrição:</b> Cria e abre um arquivo.	
<b>crypt</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *crypt(const char *key, const char *salt);</code>	
<b>Retorno:</b> String criptografada.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> É a função de encriptação de senhas. É baseada no algoritmo DES.	
<b>ctermid</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *ctermid(char *s);</code>	
<b>Retorno:</b> Ponteiro para o terminal.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Obtém o nome do terminal controlador.	
<b>ctime</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *ctime(const time_t *timep);</code>	
<b>Retorno:</b> Ponteiro para <i>string</i> contendo a data/hora.	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna uma <i>string</i> com a data/hora.	
<b>cuserid</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *cuserid(char *string);</code>	
<b>Retorno:</b> Ponteiro para string.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Retorna o nome do usuário do processo.	
<b>daemon</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int daemon (int nochdir, int noclose);</code>	
<b>Retorno:</b> -1 se houver erro.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Desvincula o programa do terminal controlador e o faz executar em <i>background</i> como um <i>daemon</i> .	
<b>difftime</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double difftime(time_t time1, time_t time0);</code>	
<b>Retorno:</b> Diferença em segundos.	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna o número de segundos transcorridos entre <i>time1</i> e <i>time0</i> .	

<b>dirname</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *dirname(char *path);</code>	
<b>Retorno:</b> Ponteiro para string.	<b>Header:</b> <code>#include &lt;libgen.h&gt;</code>
<b>Descrição:</b> Manipulação (parse) de nomes de arquivos. Retorna o nome do diretório de um nome completo de um arquivo.	
<b>div</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>div_t div(int numer, int denom);</code>	
<b>Retorno:</b> Estrutura <code>div_t</code> com os valores calculados.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Calcula o resto e o quociente de uma divisão de inteiros.	
<b>dup</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int dup(int oldfd);</code>	
<b>Retorno:</b> <code>&gt;0</code> se OK <code>-1</code> se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Retorna um novo descritor para o arquivo.	
<b>dup2</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int dup2(int oldfd, int newfd);</code>	
<b>Retorno:</b> <code>&gt;0</code> se OK <code>-1</code> se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Retorna um novo descritor (apontado por <code>newfd</code> ) para o arquivo.	
<b>endgrent</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void endgrent(void);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;grp.h&gt;</code>
<b>Descrição:</b> Fecha o arquivo <code>/etc/group</code> .	
<b>endpwent</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void endpwent(void);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;pwd.h&gt;</code>
<b>Descrição:</b> Fecha o arquivo <code>/etc/passwd</code> .	
<b>execl</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int execl(const char *path, const char *arg, ...);</code>	
<b>Retorno:</b> Se retornar, um <code>int</code> especificando o erro.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Substitui o processo atual por um novo processo a ser carregado (passado como parâmetro).	

#### execle

Nível: 3

**Protótipo:** `int execle(const char *path, const char *arg , ..., char * const envp[]);`

**Retorno:** Se retornar, um `int` especificando o erro. **Header:** `#include <unistd.h>`

**Descrição:** Substitui o processo atual por um novo processo a ser carregado (passado como parâmetro).

#### execlp

Nível: 3

**Protótipo:** `int execlp(const char *file, const char *arg, ...);`

**Retorno:** Se retornar, um `int` especificando o erro. **Header:** `#include <unistd.h>`

**Descrição:** Substitui o processo atual por um novo processo a ser carregado (passado como parâmetro).

#### execv

Nível: 3

**Protótipo:** `int execv(const char *path, char *const argv[]);`

**Retorno:** Se retornar, um `int` especificando o erro. **Header:** `#include <unistd.h>`

**Descrição:** Substitui o processo atual por um novo processo a ser carregado (passado como parâmetro).

#### execve

Nível: 2

**Protótipo:** `int execve(const char *filename, char *const argv [], char *const envp[]);`

**Retorno:** Se retornar, um `int` especificando o erro. **Header:** `#include <unistd.h>`

**Descrição:** Substitui o processo atual por um novo processo a ser carregado (passado como parâmetro).

#### execvp

Nível: 3

**Protótipo:** `int execvp(const char *file, char *const argv[]);`

**Retorno:** Se retornar, um `int` especificando o erro. **Header:** `#include <unistd.h>`

**Descrição:** Substitui o processo atual por um novo processo a ser carregado (passado como parâmetro).

#### exit

Nível: 3

**Protótipo:** `void exit(int status);`

**Retorno:** Não tem retorno. **Header:** `#include <stdlib.h>`

**Descrição:** Termina o programa. Arquivos abertos serão fechados, buffers descarregados e funções cadastradas através da função `atexit` serão executadas. Somente após todos esses passos o controle passa para o *kernel* do sistema operacional.

<b>exp</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double exp(double num);</code>	
<b>Retorno:</b> Logaritmo natural	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Devolve logaritmo natural <i>e</i> elevado à potência de número.	
<b>fabs</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double fabs(double num);</code>	
<b>Retorno:</b> Valor absoluto.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Devolve o valor absoluto de número.	
<b>fchdir</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int fchdir(int fd);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Mudança de diretório, idêntica à função <code>chdir</code> , somente é utilizado um identificador de arquivo para o diretório.	
<b>fchmod</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int fchmod(int fildes, mode_t mode);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code>
<b>Descrição:</b> Mudança de permissões de um arquivo, idêntica à função <code>chmod</code> , somente é utilizado um identificador de arquivo.	
<b>fchown</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int fchown(int fd, uid_t owner, gid_t group);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Mudança de usuário/grupo de um arquivo, idêntica à função <code>chown</code> , somente é utilizado um identificador de arquivo.	
<b>fclose</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int fclose(FILE *stream);</code>	
<b>Retorno:</b> 0 se OK EOF se houve erro.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Fecha um arquivo.	
<b>fcntl</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int fcntl(int fd, int cmd);</code> <code>int fcntl(int fd, int cmd, long arg);</code> <code>int fcntl(int fd, int cmd, struct flock *lock);</code>	
<b>Retorno:</b> -1 se ouve erro Depende de <code>cmd</code> se OK	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;fcntl.h&gt;</code>
<b>Descrição:</b> Realiza várias permissões em um arquivo (indicado pelo descritor do arquivo). As operações serão determinadas por <code>cmd</code> .	



### fdopen

Nível: 3

**Protótipo:** FILE \* fdopen (int FileDescriptor, const char \*type)  
**Retorno:** Ponteiro para FILE (OK)      **Header:** #include <stdio.h>  
 NULL (Erro)  
**Descrição:** Transforma um descritor de arquivo (aberto com as funções de nível 2, como open, por exemplo) em um ponteiro para FILE.

### feof

Nível: 3

**Protótipo:** int feof(FILE \*stream);  
**Retorno:** !=0 se o final do arquivo foi alcançado      **Header:** #include <stdio.h>  
 0 se não  
**Descrição:** Verifica se o final do arquivo foi alcançado.

### ferror

Nível: 3

**Protótipo:** int ferror(FILE \*stream);  
**Retorno:** !=0 se existe erro      **Header:** #include <stdio.h>  
 0 se não  
**Descrição:** Verifica se existe algum erro no arquivo.

### fflush

Nível: 3

**Protótipo:** int fflush(FILE \*stream);  
**Retorno:** 0 se OK      **Header:** #include <stdio.h>  
 EOF se houve erro.  
**Descrição:** Força o descarregamento do *buffer* de gravação de um arquivo.

### fgetc

Nível: 3

**Protótipo:** int fgetc(FILE \*stream);  
**Retorno:** Caractere se OK      **Header:** #include <stdio.h>  
 EOF se houve erro ou o final do arquivo foi alcançado.  
**Descrição:** Lê um caractere do arquivo.

### fgetgrent

Nível: 3

**Protótipo:** struct group \*fgetgrent(FILE \*stream);  
**Retorno:** Ponteiro para estrutura      **Header:** #include <stdio.h>  
 NULL se não existem mais grupos ou se ocorreu um erro.      #include <sys/types.h>  
    #include <grp.h>  
**Descrição:** Ponteiro para a estrutura de grupo (/etc/group).

**fgetpwent****Nível: 3****Protótipo:** `struct passwd *fgetpwent(FILE *stream);`

**Retorno:** Ponteiro para estrutura  
 NULL se não existe mais  
 usuário ou se ocorreu um  
 erro.

**Header:** `#include <stdio.h>`  
`#include <sys/types.h>`  
`#include <pwd.h>`

**Descrição:** Ponteiro para a estrutura de usuários/senhas (`/etc/passwd`).**fgetpos****Nível: 3****Protótipo:** `int fgetpos(FILE *stream, fpos_t *pos);`

**Retorno:** 0 se OK  
 != 0 senão

**Header:** `#include <stdio.h>`

**Descrição:** Retorna a posição atual do arquivo. Equivalente à função `ftell`.**fgets****Nível: 3****Protótipo:** `char *fgets(char *s, int size, FILE *stream);`

**Retorno:** NULL em caso de erro  
 Ponteiro para os dados lidos.

**Header:** `#include <stdio.h>`

**Descrição:** Leitura de dados do arquivo.**fileno****Nível: 3****Protótipo:** `int fileno(FILE *stream);`

**Retorno:** Descritor do arquivo (int)  
 associado ao `stream`.

**Header:** `#include <stdio.h>`

**Descrição:** Retorna o descritor do arquivo (em formato de um inteiro e como é utilizado nas funções de nível 2) associado ao `stream` indicado.**flock****Nível: 2****Protótipo:** `int flock(int fd, int operation);`

**Retorno:** 0 se OK  
 -1 se houve erro.

**Header:** `#include <sys/file.h>`

**Descrição:** Aplica ou remove uma “trava” no arquivo.**floor****Nível: 3****Protótipo:** `double floor(double num);`

**Retorno:** Maior número possível me-  
 nor que `num`.

**Header:** `#include <math.h>`

**Descrição:** Devolve o maior número possível que não seja maior que número, por exemplo, `floor(1.4)` retornará 1.0.**fopen****Nível: 3****Protótipo:** `FILE * fopen(const char *path, const char *type)`

**Retorno:** Ponteiro para FILE (OK)  
 NULL (Erro)

**Header:** `#include <stdio.h>`

**Descrição:** Abre um arquivo e retorna uma `stream` do tipo FILE para o arquivo aberto.

### **fopen64**

**Nível:** 3

**Protótipo:** FILE \* fopen64(const char \*path, const char \*type)

**Retorno:** Ponteiro para FILE (OK)  
NULL (Erro)      **Header:** #include <stdio.h>

**Descrição:** Abre um arquivo e retorna uma stream do tipo FILE para o arquivo aberto.

### **fork**

**Nível:** 2

**Protótipo:** pid\_t fork(void);

**Retorno:** PID se OK  
-1 se houve erro      **Header:** #include <sys/types.h>  
#include <unistd.h>

**Descrição:** Cria um processo filho idêntico ao pai.

### **fpathconf**

**Nível:** 3

**Protótipo:** long fpathconf(int filedес, int name);

**Retorno:** -1 se houve erro      **Header:** #include <unistd.h>  
!=0 se OK (depende de  
name)

**Descrição:** Retorna informações referentes à configuração do arquivo.

### **fprintf**

**Nível:** 3

**Protótipo:** int fprintf (FILE \*stream, const char \*format,  
[value, . . .])

**Retorno:** >0 se OK      **Header:** #include <stdio.h>  
<0 em caso de erro

**Descrição:** Formata uma saída e grava em arquivo apontado por stream.

### **fputc**

**Nível:** 3

**Protótipo:** int fputc(int c, FILE \*stream);

**Retorno:** Próprio caractere se OK      **Header:** #include <stdio.h>  
EOF se houve erro

**Descrição:** Grava um caractere no arquivo.

### **fputs**

**Nível:** 3

**Protótipo:** int fputs(const char \*s, FILE \*stream);

**Retorno:** >0 se OK      **Header:** #include <stdio.h>  
EOF se houve erro.

**Descrição:** Grava uma string no arquivo.

### **fread**

**Nível:** 3

**Protótipo:** size\_t fread ( (const void \*) pointer, size\_t size,  
size\_t NumberOfItems, FILE \* stream)

**Retorno:** Número de bytes lidos.      **Header:** #include <stdio.h>

**Descrição:** Realiza a leitura de bytes (size \* NumberOfItems) de um arquivo.

**free****Nível: 3****Protótipo:** `void free(void *ptr)`**Retorno:** Não tem retorno.**Header:** `#include <stdlib.h>`**Descrição:** Libera espaço previamente alocado com `malloc`, `realloc` ou `calloc`.**freopen****Nível: 3****Protótipo:** `FILE * freopen ( const char *path, const char *type, FILE * stream)`**Retorno:** Ponteiro para `FILE` (OK)**Header:** `#include <stdio.h>`

NULL (Erro)

**Descrição:** Abre outro arquivo e associa a uma `stream` do tipo `FILE` de um arquivo já aberto.**freopen64****Nível: 3****Protótipo:** `FILE * freopen64 ( const char *path, const char *type, FILE * stream)`**Retorno:** Ponteiro para `FILE` (OK)**Header:** `#include <stdio.h>`

NULL (Erro)

**Descrição:** Abre outro arquivo e associa a uma `stream` do tipo `FILE` de um arquivo já aberto.**fscanf****Nível: 3****Protótipo:** `int fscanf(FILE *stream, const char *format, ...);`**Retorno:** Quantidade de dados lidos.**Header:** `#include <stdio.h>`

EOF se erro

**Descrição:** Lê as informações de um arquivo de acordo com os parâmetros passados (como na função `scanf`).**fseek****Nível: 3****Protótipo:** `int fseek(FILE *stream, long offset, int whence);`**Retorno:** 0 se OK**Header:** `#include <stdio.h>`

!=0 se houve erro.

**Descrição:** Posiciona o ponteiro do arquivo dentro do arquivo especificado.**fsetpos****Nível: 3****Protótipo:** `int fsetpos(FILE *stream, fpos_t *pos);`**Retorno:** 0 se OK**Header:** `#include <stdio.h>`

!=0 se houve erro.

**Descrição:** Posiciona o ponteiro do arquivo. Equivalente à função `fseek` passando o parâmetro `SEEK_SET`.

<b>fstat</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int fstat(int filedes, struct stat *buf);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Retorna informações (atributos) a respeito do arquivo.	
<b>fsync</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int fsync(int fd);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Sincroniza um arquivo (totalmente) em disco.	
<b>ftell</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>long ftell(FILE *stream);</code>	
<b>Retorno:</b> Posição atual do arquivo. -1L se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Retorna a posição atual do ponteiro dentro do arquivo em <i>bytes</i> (em forma de um <i>long</i> ).	
<b>ftime</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int ftime(struct timeb *tp);</code>	
<b>Retorno:</b> Sempre 0.	<b>Header:</b> <code>#include &lt;sys/timeb.h&gt;</code>
<b>Descrição:</b> Retorna a data e hora corrente no ponteiro passado.	
<b>ftruncate</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int ftruncate(int fd, off_t length);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Trunca o arquivo para o tamanho especificado.	
<b>fwrite</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>size_t fwrite(const void pointer, size_t size, size_t NumberOfItems, FILE * stream)</code>	
<b>Retorno:</b> Número de bytes gravados.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Realiza a gravação de bytes em um arquivo.	
<b>getc</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int getc(FILE *stream);</code>	
<b>Retorno:</b> Caractere se OK EOF se houve erro ou o final do arquivo foi alcançado.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Lê um caractere do arquivo (equivalente à função <i>fgetc</i> ).	

<b>getchar</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int getchar(void);</code>	
<b>Retorno:</b> Caractere se OK EOF se houve erro ou o final do arquivo foi alcançado.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Lê um caractere da entrada padrão.	
<b>getcwd</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *getcwd(char *buf, size_t size);</code>	
<b>Retorno:</b> Ponteiro para dados lidos NULL em caso de erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Recupera o nome do completo para o diretório de trabalho atual.	
<b>getdate</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>struct tm *getdate (const char *string);</code>	
<b>Retorno:</b> Estrutura para data NULL se houve erro	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna uma estrutura <code>tm</code> para string de data passada.	
<b>getegid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>gid_t getegid (void);</code>	
<b>Retorno:</b> Número do GID	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Retorna o número do GID do grupo efetivo que está executando o processo.	
<b>getenv</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *getenv(const char *name;)</code>	
<b>Retorno:</b> Ponteiro para o conteúdo da variável NULL se a variável não existir	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Retorna o conteúdo de um variável de ambiente.	
<b>geteuid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>uid_t geteuid(void);</code>	
<b>Retorno:</b> Número do UID	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Retorna o número do UID do usuário efetivo que está executando o processo.	
<b>getgid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>gid_t getgid (void);</code>	
<b>Retorno:</b> Número do GID	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Retorna o número do GID do grupo real que está executando o processo.	

### getgrent

Nível: 3

**Protótipo:** `struct group *getgrent(void);`

**Retorno:** Estrutura de grupos

**Header:** `#include <sys/types.h>`  
`#include <grp.h>`

**Descrição:** Retorna as informações do arquivo `/etc/group` em uma estrutura (na forma de um ponteiro).

### getgrgid

Nível: 3

**Protótipo:** `struct group *getgrgid(gid_t GID)`

**Retorno:** Ponteiro para estrutura

**Header:** `#include <sys/types.h>`  
`#include <grp.h>`

NULL se o grupo não existir ou não estiver disponível

**Descrição:** Retorna as informações de um grupo específico a partir do número do GID. As informações são retiradas do arquivo `/etc/group`.

### getgrnam

Nível: 3

**Protótipo:** `struct group *getgrnam(const char * name)`

**Retorno:** Ponteiro para estrutura

**Header:** `#include <sys/types.h>`  
`#include <grp.h>`

NULL se o grupo não existir ou não estiver disponível

**Descrição:** Retorna as informações de um grupo específico a partir do seu nome. As informações são retiradas do arquivo `/etc/group`.

### getgroups

Nível: 2

**Protótipo:** `int getgroups(int ngroups, gid_t GIDSet)`

**Retorno:** # de grupos

**Header:** `#include <sys/types.h>`  
`#include <unistd.h>`

-1 se erro

**Descrição:** Retorna os GIDs (em GIDSet) dos grupos suplementares de um processo.

### gethostid

Nível: 2

**Protótipo:** `long gethostid(void);`

**Retorno:** Indentificador do *host*

**Header:** `#include <unistd.h>`

**Descrição:** Retorna o identificador único do *host*.

### gethostname

Nível: 2

**Protótipo:** `int gethostname(char * name, int length)`

**Retorno:** 0 se OK

**Header:** `#include <unistd.h>`

-1 se erro

**Descrição:** Retorna o nome do *host* da máquina. O tamanho do nome é limitado por `length`.

**getlogin****Nível: 3****Protótipo:** `char *getlogin (void)`**Retorno:** Ponteiro para o login  
NULL se houve erro.**Header:** `#include <sys/types.h>`  
`#include <unistd.h>`  
`#include <limits.h>`**Descrição:** Retorna o *login name* do usuário que está executando o processo.**getlogin\_r****Nível: 3****Protótipo:** `int getlogin_r(char * name, size_t length);`**Retorno:** 0 se OK  
-1 se houve erro.**Header:** `#include <sys/types.h>`  
`#include <unistd.h>`  
`#include <limits.h>`**Descrição:** Retorna o *login name* do usuário a partir do arquivo `/etc/utmp`. Indicado para processos *multi-thread*.**getmsg****Nível: 2****Protótipo:** `int getmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *flags);`**Retorno:** >1 se OK  
-1 se houve erro**Header:** `#include <stropts.h>`**Descrição:** Lê as mensagens da fila de um arquivo.**getopt****Nível: 3****Protótipo:** `int getopt(int argc, char * const argv[], const char *optstring);`**Retorno:** Opção pesquisada  
: se uma opção foi achada  
mas exige parâmetro  
? se a opção for desconhecida  
-1 se o final da lista foi alcançado**Header:** `#include <unistd.h>`**Descrição:** Pega os parâmetros (passados como opções) na linha de comando.**getpeername****Nível: 2****Protótipo:** `int getpeername(int s, struct sockaddr *name, socklen_t *namelen);`**Retorno:** 0 se OK  
-1 se houve erro**Header:** `#include <sys/socket.h>`**Descrição:** Captura o nome do ponto conectado ao *socket*.**getpgid****Nível: 2****Protótipo:** `pid_t getpgid(pid_t pid);`**Retorno:** Process Group ID**Header:** `#include <unistd.h>`**Descrição:** Retorna o número de identificação do grupo para o processo especificado.



### getpgrp

Nível: 2

**Protótipo:** `pid_t getpgrp (void);`

**Retorno:** Número GID.

**Header:** `#include <unistd.h>`

**Descrição:** Retorna o número do GID de grupo do processo.

### getpid

Nível: 2

**Protótipo:** `pid_t getpid(void);`

**Retorno:** Número do PID.

**Header:** `#include <sys/types.h>`  
`#include <unistd.h>`

**Descrição:** Retorna o número do processo.

### getpmsg

Nível: 3

**Protótipo:** `int getpmsg (int fd, struct strbuf *ctlptr, struct strbuf *dataptr, int *bandp, int *flags);`

**Retorno:** >1 se OK

**Header:** `#include <stropts.h>`

-1 se houve erro.

**Descrição:** Lê as mensagens da fila de um arquivo. Idêntico a `getmsg`, exceto que é especificada a prioridade da mensagem que se quer ler.

### getppid

Nível: 2

**Protótipo:** `pid_t getppid (void);`

**Retorno:** Número PID.

**Header:** `#include <sys/types.h>`  
`#include <unistd.h>`

**Descrição:** Retorna o PPID (PID do processo pai) do processo corrente.

### getpwent

Nível: 3

**Protótipo:** `struct passwd *getpwent (void);`

**Retorno:** Ponteiro para estrutura

**Header:** `#include <sys/types.h>`

NULL se houve erro ou o final do arquivo foi alcançado.

`#include <pwd.h>`

**Descrição:** Retorna as informações do próximo usuário (informações são retiradas de `/etc/passwd`)

### getpwnam

Nível: 3

**Protótipo:** `struct passwd *getpwnam(char *name)`

**Retorno:** Ponteiro se OK

**Header:** `#include <sys/types.h>`

NULL se houve erro

`#include <pwd.h>`

**Descrição:** Retorna as informações do usuário especificado em `name` (informações são retiradas de `/etc/passwd`).

**getpwuid****Nível:** 3**Protótipo:** `struct passwd *getpwuid(uid_t uid);`**Retorno:** Ponteiro se OK  
NULL se houve erro**Header:** `#include <sys/types.h>`  
`#include <pwd.h>`**Descrição:** Retorna as informações do usuário especificado em `uid` (informações são retiradas de `/etc/passwd`).**gets****Nível:** 3**Protótipo:** `char *gets(char * string)`**Retorno:** *String* se OK  
NULL se houve erro ou se o final do arquivo foi alcançado.**Header:** `#include <stdio.h>`**Descrição:** Lê os dados da entrada padrão (`stdin`) e guarda na *string*.**getsid****Nível:** 2**Protótipo:** `pid_t getsid(pid_t pid);`**Retorno:** ID se OK  
-1 se houve erro.**Header:** `#include <unistd.h>`**Descrição:** Retorna o ID de seção do processo.**getsockname****Nível:** 2**Protótipo:** `int getsockname(int s, struct sockaddr * name, socklen_t * namelen);`**Retorno:** 0 se OK  
-1 se houve erro**Header:** `#include <sys/socket.h>`**Descrição:** Retorna o nome de um *socket*.**gmtime****Nível:** 3**Protótipo:** `struct tm *gmtime(time_t *time)`**Retorno:** Ponteiro para a estrutura.**Header:** `#include <time.h>`**Descrição:** Retorna uma estrutura com as informações referentes a data/hora a partir da hora informada (hora informada em formato `long`)**gettimeofday****Nível:** 2**Protótipo:** `int gettimeofday(struct timeval *tv, struct timezone *tz);`**Retorno:** 0 se OK  
-1 se houve.**Header:** `#include <sys/time.h>`**Descrição:** Retorna hora do sistema.**getuid****Nível:** 2**Protótipo:** `uid_t getuid(void);`**Retorno:** Número do UID**Header:** `#include <sys/types.h>`  
`#include <unistd.h>`**Descrição:** Retorna o UID do usuário que está executando o processo.

### getumask

Nível: 3

**Protótipo:** `mode_t getumask(void);`

**Retorno:** Máscara padrão.

**Header:** `#include <sys/types.h>`  
`#include <sys/stat.h>`

**Descrição:** Retorna a máscara padrão para criação de arquivos.

### initgroups

Nível: 3

**Protótipo:** `int initgroups(char *user, int GID)`

**Retorno:** 0 se OK

**Header:** `#include <unistd.h>`

-1 se houve erro.

`#include <sys/types.h>`

**Descrição:** A partir das informações passadas, seta as informações de grupo do processo que está sendo executado. Interfere no funcionamento das famílias de funções `getgrent` e `getpwent`.

### isalnum

Nível: 3

**Protótipo:** `int isalnum(int ch);`

**Retorno:** !=0 se `ch` for letra

**Header:** `#include <ctype.h>`

!=0 se `ch` for dígito

0 se `ch` não for alfanumérico

**Descrição:** Verifica se um caractere é alfanumérico ou não.

### isalpha

Nível: 3

**Protótipo:** `int isalpha(int ch);`

**Retorno:** !=0 se for uma letra

**Header:** `#include <ctype.h>`

0 se não for uma letra

**Descrição:** Verifica se um caractere é uma letra. Na língua portuguesa de 'Aa' até 'Zz'.

### iscntrl

Nível: 3

**Protótipo:** `int iscntrl(int ch)`

**Retorno:** !=0 se for um caractere de controle

**Header:** `#include <ctype.h>`

0 se não for um caractere de controle

**Descrição:** Verifica se um caractere é um caractere de controle entre 0 e 0x1F ou 0x7F (DEL).

### isdigit

Nível: 3

**Protótipo:** `int isdigit(int ch);`

**Retorno:** !=0 se for dígito

**Header:** `#include <ctype.h>`

0 se não for dígito

**Descrição:** Verifica se um caractere é dígito (entre 0 e 9).

**isgraph****Nível:** 3**Protótipo:** `int isgraph(int ch);`**Retorno:** !=0 se pode ser impresso  
0 se não pode ser impresso**Header:** `#include <ctype.h>`**Descrição:** Verifica se um caractere pode ser impresso (depende do sistema operacional). Exclui o espaço.**islower****Nível:** 3**Protótipo:** `int islower`**Retorno:** !=0 se minúscula  
0 se não**Header:** `#include <ctype.h>`**Descrição:** Verifica se um caractere (letra) é minúscula ou não.**isprint****Nível:** 3**Protótipo:** `int isprint(int ch);`**Retorno:** !=0 se pode ser impresso  
0 se não pode ser impresso**Header:** `#include <ctype.h>`**Descrição:** Verifica se um caractere pode ser impresso (depende do sistema operacional). Inclui o espaço.**ispunct****Nível:** 3**Protótipo:** `int ispunct(int ch);`**Retorno:** !=0 se for pontuação  
0 se não.**Header:** `#include <ctype.h>`**Descrição:** Verifica se um caractere é um caractere de pontuação.**isspace****Nível:** 3**Protótipo:** `int isspace(int ch);`**Retorno:** !=0 se for um espaço  
0 se não**Header:** `#include <ctype.h>`**Descrição:** Verifica se o caractere é espaço, tabulação, caractere de nova linha ou retorno de carro.**isupper****Nível:** 3**Protótipo:** `int isupper(int ch);`**Retorno:** !=0 se maiúscula  
0 se não**Header:** `#include <ctype.h>`**Descrição:** Verifica se um caractere (letra) é maiúscula.**isxdigit****Nível:** 3**Protótipo:** `int isxdigit(int ch);`**Retorno:** !=0 se for dígito  
0 se não for**Header:** `#include <ctype.h>`**Descrição:** Verifica se o caractere é um dígito hexadecimal (0 até 9 e A até F).

## kill

Nível: 2

**Protótipo:** `int kill(int process, int signal);`

**Retorno:** 0 se OK

**Header:** `#include <sys/types.h>`

-1 se houve erro.

`#include <signal.h>`

**Descrição:** Envia um sinal para o processo.

## killpg

Nível: 2

**Protótipo:** `int killpg(int pgrp, int sig);`

**Retorno:** 0 se OK

**Header:** `#include <signal.h>`

-1 se houve erro.

**Descrição:** Envia um sinal para um grupo de processos. Deve especificar o grupo de processo em `pgrp`.

## labs

Nível: 3

**Protótipo:** `long labs(long num);`

**Retorno:** Número absoluto.

**Header:** `#include <stdlib.h>`

**Descrição:** Devolve o número absoluto de um número.

## llabs

Nível: 3

**Protótipo:** `long long int llabs(long long int j);`

**Retorno:** Número absoluto.

**Header:** `#include <stdlib.h>`

**Descrição:** Devolve o número absoluto de um número.

## lchown

Nível: 2

**Protótipo:** `int lchown (const char *path, uid_t owner, gid_t group);`

**Retorno:** 0 se OK

**Header:** `#include <unistd.h>`

-1 se houve erro

**Descrição:** Muda usuário/grupo de um *link* simbólico (mesmo que `chown`).

## link

Nível: 2

**Protótipo:** `int link(const char *path1, const char *path2);`

**Retorno:** 0 se OK

**Header:** `#include <unistd.h>`

-1 se houve erro

**Descrição:** Cria um *hard link* para o arquivo.

## listen

Nível: 2

**Protótipo:** `int listen(int s, int backlog);`

**Retorno:** 0 se OK

**Header:** `#include`

-1 se houve erro

`<sys/socket.h>`

**Descrição:** Habilita conexões para um *socket*.

<b>localtime</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>Struct tm *localtime(const time_t * time);</code>	
<b>Retorno:</b> Ponteiro para estrutura.	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna uma estrutura com as informações referentes a data/hora a partir da hora informada (hora informada em formato <code>long</code> )	
<b>log</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>Double log(double num);</code>	
<b>Retorno:</b> Logaritmo natural.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Devolve o logaritmo natural do número. O número não pode ser negativo.	
<b>log10</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>Double log10(double num);</code>	
<b>Retorno:</b> Logaritmo base de <code>num</code> .	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Retorna o logaritmo de base 10 do número. O número não pode ser negativo.	
<b>login</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void login(const struct utmp *ut);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;utmp.h&gt;</code>
<b>Descrição:</b> Informa que um “usuário” se conectou (grava informações em <code>/var/run/utmp</code> e <code>/var/log/wtmp</code> ).	
<b>logout</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int logout(const char *ut_line);</code>	
<b>Retorno:</b> 1 se OK 0 se houve erro	<b>Header:</b> <code>#include &lt;utmp.h&gt;</code>
<b>Descrição:</b> Informa que um “usuário” se desconectou (grava informações em <code>/var/run/utmp</code> e <code>/var/log/wtmp</code> ).	
<b>lseek</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>off_t lseek (int fd, off_t offset, int whence)</code>	
<b>Retorno:</b> Nova posição do arquivo -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Posiciona o ponteiro de leitura/gravação dentro do arquivo (equivalente a <code>fseek</code> ).	
<b>lseek64</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>off64_t lseek64 (int fd, off64_t offset, int whence)</code>	
<b>Retorno:</b> Nova posição do arquivo -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Posiciona o ponteiro de leitura/gravação dentro do arquivo (equivalente a <code>fseek</code> ).	

<b>lstat</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int lstat(const char *path, Buffer)</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Obtém informações (atributos, hora de criação etc.) de um <i>link</i> simbólico.	
<b>malloc</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *malloc(size_t size);</code>	
<b>Retorno:</b> Ponteiro para a memória ou NULL em caso de erro.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Alocação dinâmica de memória.	
<b>memchr</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *memchr(const void *s, int c, size_t n);</code>	
<b>Retorno:</b> Ponteiro para o byte encontrado NULL se não foi encontrado.	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Procura um caractere nos primeiros <i>n bytes</i> da <i>string</i> .	
<b>memcmp</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int memcmp(const void *s1, const void *s2, size_t n);</code>	
<b>Retorno:</b> 0 se as <i>strings</i> são iguais <0 se <i>s1</i> < <i>s2</i> >0 se <i>s1</i> > <i>s2</i>	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Compara <i>n bytes</i> entre duas <i>strings</i> .	
<b>memcpy</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *memcpy(void *dest, const void *src, size_t n);</code>	
<b>Retorno:</b> Ponteiro para <i>dest</i>	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Copia <i>n bytes</i> da origem para o destino.	
<b>memmove</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *memmove(void *dest, const void *src, size_t n);</code>	
<b>Retorno:</b> Ponteiro para <i>dest</i>	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Copia <i>n bytes</i> da origem para o destino.	
<b>memset</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *memset(void *s, int c, size_t n);</code>	
<b>Retorno:</b> Ponteiro para <i>s</i>	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Preenche o ponteiro passado (nos <i>n primeiros bytes</i> ) com o caractere informado.	

**mkdtemp****Nível: 3**

**Protótipo:** `char *mkdtemp(char *template);`  
**Retorno:** Ponteiro para string  
 NULL se houve erro.  
**Descrição:** Cria um diretório temporário único no sistema de arquivos.

**mkdir****Nível: 2**

**Protótipo:** `int mkdir(const char *path, mode_t mode)`  
**Retorno:** 0 se OK  
 -1 se houve erro  
**Header:** `#include <sys/stat.h>`  
`#include <sys/types.h>`  
**Descrição:** Cria um diretório no sistema de arquivos.

**mkfifo****Nível: 3**

**Protótipo:** `int mkfifo (const char *path, mode_t mode)`  
**Retorno:** 0 se OK  
 -1 se houve erro  
**Header:** `#include <sys/types.h>`  
`#include <sys/stat.h>`  
**Descrição:** Cria um FIFO (*first-in-first-out*) no sistema de arquivos.

**mknod****Nível: 2**

**Protótipo:** `int mknod(const char *pathname, mode_t mode, dev_t dev);`  
**Retorno:** 0 se OK  
 -1 se houve erro  
**Header:** `#include <sys/types.h>`  
`#include <sys/stat.h>`  
`#include <fcntl.h>`  
`#include <unistd.h>`  
**Descrição:** Cria um arquivo especial (*device, pipe* etc.) no sistema de arquivos.

**mkstemp****Nível: 3**

**Protótipo:** `int mkstemp(char *template);`  
**Retorno:** Descritor do arquivo  
 -1 se houve erro.  
**Header:** `#include <stdlib.h>`  
**Descrição:** Cria um arquivo temporário único.

**mktemp****Nível: 3**

**Protótipo:** `char *mktemp(char *template);`  
**Retorno:** Ponteiro para arquivo  
 NULL se houve erro.  
**Header:** `#include <stdlib.h>`  
**Descrição:** Cria um arquivo temporário único.

**mktime****Nível: 3**

**Protótipo:** `time_t mktime(struct tm *timeptr)`  
**Retorno:** long para tempo  
 -1 se houve erro  
**Header:** `#include <time.h>`  
**Descrição:** A partir da estrutura de data/hora, retorna um novo valor para hora (segundos decorridos desde 01/01/1970).



## mmap

Nível: 2

**Protótipo:** `caddr_t *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`  
`void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`

**Retorno:** Ponteiro para início do endereço  
-1 se houve erro. **Header:** `#include <sys/types.h>`  
`#include <sys/mman.h>`

**Descrição:** Cria um novo mapeamento de memória em disco.

## mount

Nível: 2

**Protótipo:** `int mount(const char *source, const char *target, const char *filesystemtype, unsigned long mountflags, const void *data);`

**Retorno:** 0 se OK  
-1 se houve erro. **Header:** `#include <sys/mount.h>`

**Descrição:** Monta um sistema de arquivos.

## mremap

Nível: 2

**Protótipo:** `void * mremap(void *old_address, size_t old_size , size_t new_size, unsigned long flags);`

**Retorno:** Ponteiro para o novo endereço  
-1 se houve erro. **Header:** `#include <unistd.h>`  
`#include <sys/mman.h>`

**Descrição:** Remapeia a memória mapeada anteriormente.

## msync

Nível: 2

**Protótipo:** `int msync(const void *start, size_t length, int flags);`

**Retorno:** 0 se OK  
-1 se houve erro. **Header:**

**Descrição:** Envia as alterações feitas em um arquivo mapeado em memória usando `mmap` de volta para o disco.

## munmap

Nível: 2

**Protótipo:** `int munmap(void *start, size_t length);`

**Retorno:** 0 se OK  
-1 se houve erro **Header:** `#include <sys/mman.h>`

**Descrição:** Desmapeia um mapeamento de memória em disco.

## nice

Nível: 2

**Protótipo:** `int nice(int inc);`

**Retorno:** 0 se OK  
-1 se houve erro **Header:** `#include <unistd.h>`

**Descrição:** Altera a prioridade do processo.

**open****Nível: 2**

**Protótipo:** `int open(const char *pathname, int flags);`  
`int open(const char *pathname, int flags, mode_t mode);`

**Retorno:** Descritor para arquivo.  
 -1 se houve erro.

**Header:** `#include <sys/types.h>`  
`#include <sys/stat.h>`  
`#include <fcntl.h>`

**Descrição:** Cria/abre um arquivo no sistema de arquivos.

**opendir****Nível: 3**

**Protótipo:** `DIR *opendir(const char *name);`

**Retorno:** Ponteiro para diretório  
 NULL se houve erro

**Header:** `#include <sys/types.h>`  
`#include <dirent.h>`

**Descrição:** Abre um diretório específico.

**openlog****Nível: 3**

**Protótipo:** `void openlog(char *ident, int option, int facility);`

**Retorno:** Não tem retorno.

**Header:** `#include <syslog.h>`

**Descrição:** Abre um arquivo (conexão) para o sistema de logs do sistema operacional. Utilize sempre `syslog`.

**pause****Nível: 2**

**Protótipo:** `int pause(void);`

**Retorno:** -1 sempre

**Header:** `#include <unistd.h>`

**Descrição:** Faz com o que processo que a chama aguarde até que um sinal seja recebido.

**pathconf****Nível: 3**

**Protótipo:** `long pathconf(char *path, int name);`

**Retorno:** Limite se existir  
 -1 se houve erro.

**Header:** `#include <unistd.h>`

**Descrição:** Obtém um valor para a opção de configuração `name` para o nome de arquivo.

**pclose****Nível: 3**

**Protótipo:** `int pclose(FILE *stream);`

**Retorno:** Código de retorno do processo.  
 -1 se houve erro.

**Header:** `#include <stdio.h>`

**Descrição:** Fecha um canal de comunicação *pipeline*.

**perror****Nível: 3**

**Protótipo:** `void perror(const char *s);`

**Retorno:** Não tem retorno.

**Header:** `#include <stdio.h>`

**Descrição:** Mostra a mensagem do erro em função da variável `errno`.

<b>pipe</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int pipe(int filedес[2]);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Cria um par de descritores de arquivo, apontando para um inode pipe, e o coloca no vetor apontado por <code>filedes</code> . <code>filedes[0]</code> é para leitura, <code>filedes[1]</code> é para escrita.	
<b>popen</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>FILE *popen(const char *command, const char *type);</code>	
<b>Retorno:</b> Ponteiro para arquivo NULL se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Abre um processo através de um pipe.	
<b>pow</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double pow(double base, double exp);</code>	
<b>Retorno:</b> $\text{base}^{\text{exp}}$	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Devolve o número base elevada ao número exponencial.	
<b>printf</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int printf(const char *format, [Value, ...]);</code>	
<b>Retorno:</b> >0 se ok <0 em caso de erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Imprime uma mensagem formatada	
<b>psignal</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void psignal(int sig, const char *s);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;signal.h&gt;</code>
<b>Descrição:</b> Mostra uma mensagem na saída de erro padrão contendo o sinal recebido.	
<b>ptrace</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>long ptrace(enum __ptrace_request request, pid_t pid, void *addr);</code>	
<b>Retorno:</b> Dados requisitados ou 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/ptrace.h&gt;</code>
<b>Descrição:</b> Habilita o "acompanhamento" das chamadas de sistemas (veja comando <code>strace</code> ).	
<b>putc</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int putc(int c, FILE *stream);</code>	
<b>Retorno:</b> Caractere gravado.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Grava um caractere no arquivo.	

## 300 ♦ Programando em C para Linux, Unix e Windows

<b>putchar</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int putchar(int c);</code>	
<b>Retorno:</b> Caractere gravado.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Grava um caractere na saída padrão.	
<b>putenv</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int putenv(const char *string);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Cria/modifica uma variável de ambiente.	
<b>putgrent</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int putgrent(const struct group *grp, FILE *fp);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;grp.h&gt;</code>
<b>Descrição:</b> Grava as informações de grupos no arquivo especificado (no formato de <code>/etc/group</code> ).	
<b>putpwent</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int putpwent(const struct passwd *p, FILE *stream);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;pwd.h&gt;</code> <code>#include &lt;stdio.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Grava as informações de usuário no arquivo especificado (no formato de <code>/etc/passwd</code> ).	
<b>puts</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int puts(const char *s);</code>	
<b>Retorno:</b> >0 se OK EOF se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Grava uma string na saída padrão.	
<b>qsort</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void qsort(void *base, size_t nmem, size_t size, int (*compar)(const void *, const void *));</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Classifica/ordena um vetor.	
<b>raise</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int raise (int sig);</code>	
<b>Retorno:</b> 0 se OK <0 se houve erro	<b>Header:</b> <code>#include &lt;signal.h&gt;</code>
<b>Descrição:</b> Envia um sinal para o próprio processo.	

<b>rand</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int rand(void);</code>	
<b>Retorno:</b> Número pseudo-aleatório.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Gera e devolve uma sequência de números pseudo-aleatórios. O valor retornado será um <code>int</code> entre 0 e <code>RAND_MAX</code> .	
<b>read</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>ssize_t read(int fd, void *buf, size_t count);</code>	
<b>Retorno:</b> 0 Final do arquivo # bytes lidos -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Lê bytes de um arquivo.	
<b>readv</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int readv(int fildes, const struct iovec *vector, size_t count);</code>	
<b>Retorno:</b> # bytes lidos -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/uio.h&gt;</code>
<b>Descrição:</b> Lê informações de vários <i>buffers</i> .	
<b>realloc</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void *realloc(void *ptr, size_t);</code>	
<b>Retorno:</b> Ponteiro para a memória.	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Realocação dinâmica de memória.	
<b>readdir</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>struct dirent *readdir(DIR *dp);</code>	
<b>Retorno:</b> Ponteiro para estrutura NULL se o final do diretório foi alcançado.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;dirent.h&gt;</code>
<b>Descrição:</b> Retorna as informações referentes a um diretório.	
<b>readlink</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int readlink(const char *path, char *buf, size_t bufsize);</code>	
<b>Retorno:</b> # bytes lidos se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Lê valor de uma ligação simbólica	
<b>recvfrom</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>ssize_t recvfrom(int s, void *buf, size_t len int flags, struct sockaddr *from, socklen_t *fromlen</code>	
<b>Retorno:</b> # bytes recebidos -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/socket.h&gt;</code>
<b>Descrição:</b> Recebe mensagens de um socket.	

## 302 ♦ Programando em C para Linux, Unix e Windows

<b>remove</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int remove(const char *pathname);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Remove um arquivo do sistema de arquivos.	
<b>rename</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int rename(const char *oldpath, const char *newpath);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Muda de local ou renomeia um arquivo.	
<b>rewind</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void rewind( FILE *stream);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Seta o indicador de posição de arquivo para o início do arquivo.	
<b>rewinddir</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void rewinddir(DIR *dir);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;dirent.h&gt;</code>
<b>Descrição:</b> Posiciona o ponteiro na primeira entrada (início) do diretório.	
<b>rmdir</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int rmdir(const char *pathname);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Apaga um diretório, o qual deve estar vazio.	
<b>scanf</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int scanf(const char *format, ...);</code>	
<b>Retorno:</b> Quantidade de dados lidos. EOF se erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Leitura de dados digitados (entrada padrão) de acordo com os formatos utilizados (int, float, string etc.)	
<b>seekdir</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void seekdir(DIR *dir, off_t offset);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;dirent.h&gt;</code>
<b>Descrição:</b> Pesquisa e posiciona o ponteiro na entrada correspondente.	

## select

Nível: 2

**Protótipo:** `int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);`

**Retorno:** # de descritores  
-1 se houve erro

**Header:** `#include <sys/time.h>`  
`#include <sys/types.h>`  
`#include <unistd.h>`

**Descrição:** Multiplexação síncrona de E/S. Aguarda a mudança de status de um número de descritores de arquivo.

## semctl

Nível: 2

**Protótipo:** `int semctl(int semid, int semnum, int cmd);`

**Retorno:** >0 se OK  
-1 se houve erro

**Header:** `#include <sys/types.h>`  
`#include <sys/ipc.h>`  
`#include <sys/sem.h>`

**Descrição:** Operações de controle de semáforo.

## semget

Nível: 2

**Protótipo:** `int semget(key_t key, int nsems, int semflg);`

**Retorno:** >0 se OK  
-1 se houve erro

**Header:** `#include <sys/types.h>`  
`#include <sys/ipc.h>`  
`#include <sys/sem.h>`

**Descrição:** Lê/obtem um identificador para semáforo.

## semop

Nível: 2

**Protótipo:** `int semop(int semid, struct sembuf *sops, unsigned nsops);`

**Retorno:** 0 se OK  
-1 se houve erro

**Header:** `#include <sys/types.h>`  
`#include <sys/ipc.h>`  
`#include <sys/sem.h>`

**Descrição:** Operações de semáforo.

## send

Nível: 2

**Protótipo:** `ssize_t send(int s, const void *buf, size_t len int flags);`

**Retorno:** # bytes enviados  
-1 se houve erro

**Header:** `#include <sys/types.h>`  
`#include <sys/socket.h>`

**Descrição:** Envia mensagem para o socket.

## sendto

Nível: 2

**Protótipo:** `ssize_t sendto(int s, const void *buf, size_t len int flags, const struct sockaddr *to, socklen_t tolen);`

**Retorno:** # bytes enviados  
-1 se houve erro

**Header:** `#include <sys/types.h>`  
`#include <sys/socket.h>`

**Descrição:** Envia mensagem para o socket.

<b>setbuf</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void setbuf(FILE *stream, char *buf);</code>	
<b>Retorno:</b> 0 se OK <0 se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Seta as operações (comportamento) de <i>buffer</i> do arquivo.	
<b>setbuffer</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void setbuffer(FILE *stream, char *buf, size_t size);</code>	
<b>Retorno:</b> 0 se OK <0 se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Seta as operações (comportamento) de <i>buffer</i> do arquivo.	
<b>setenv</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int setenv(const char *name, const char *value, int overwrite);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Cria ou modifica uma variável de ambiente (similar a <code>putenv</code> ).	
<b>seteuid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int seteuid(uid_t euid);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Seta o ID efetivo do usuário no processo.	
<b>setegid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int setegid(gid_t egid);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Seta o GID efetivo do grupo no processo.	
<b>setgid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int setgid(gid_t gid);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Seleciona a identidade de grupo efetiva do processo atual.	
<b>setgrent</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void setgrent(void);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;grp.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Retorna o ponteiro do arquivo para o início de <code>/etc/group</code> .	



### setgroups

Nível: 2

**Protótipo:** `int setgroups(size_t size, const gid_t *list);`

**Retorno:** 0 se OK **Header:** `#include <sys/types.h>`  
 -1 se houve erro. `#include <unistd.h>`

**Descrição:** Seleciona os grupos suplementares para o processo. Somente o superusuário pode usar esta função.

### sethostid

Nível: 2

**Protótipo:** `int sethostid(long hostid);`

**Retorno:** Identificador da máquina **Header:** `#include <unistd.h>`

**Descrição:** Seta o identificador do *host* (armazenado em `/etc/hostid`).

### sethostname

Nível: 2

**Protótipo:** `int sethostname(const char *name, size_t len);`

**Retorno:** 0 se OK **Header:** `#include <unistd.h>`  
 -1 se houve erro

**Descrição:** Altera o nome do *host* do sistema em uso.

### setlinebuf

Nível: 3

**Protótipo:** `void setlinebuf(FILE *stream);`

**Retorno:** 0 se OK **Header:** `#include <stdio.h>`  
 <0 se houve erro

**Descrição:** Seta as operações (comportamento) de *buffer* do arquivo.

### setlocale

Nível: 3

**Protótipo:** `char *setlocale(int category, const char * locale);`

**Retorno:** Ponteiro para string **Header:** `#include <locale.h>`  
 NULL se houve erro

**Descrição:** Seta/obtem as configurações locais do sistema operacional.

### setpgid

Nível: 2

**Protótipo:** `int setpgid(pid_t pid, pid_t pgid);`

**Retorno:** 0 se OK **Header:** `#include <unistd.h>`  
 -1 se houve erro

**Descrição:** Seta o identificador de grupo para o processo.

### setpgrp

Nível: 2

**Protótipo:** `int setpgrp(void);`

**Retorno:** 0 se OK **Header:** `#include <unistd.h>`  
 -1 se houve erro

**Descrição:** Seta o identificador de grupo para o processo. Equivalente a `setpgid(0, 0)`.

<b>setpwent</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>void setpwent(void);</code>	
<b>Retorno:</b> Não tem retorno.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;pwd.h&gt;</code>
<b>Descrição:</b> Reposiciona o ponteiro para o início do arquivo <code>/etc/passwd</code> .	
<b>settimeofday</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int settimeofday(const struct timeval *tv, const struct timezone *tz);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro.	<b>Header:</b> <code>#include &lt;sys/time.h&gt;</code>
<b>Descrição:</b> Seta a hora do sistema.	
<b>setsid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>pid_t setsid(void);</code>	
<b>Retorno:</b> ID da sessão -1 se houve erro.	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Cria uma nova sessão se o processo chamador não é um líder de grupo de processo. Torna o processo líder da sessão.	
<b>setuid</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int setuid(uid_t uid);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Seleciona a ID efetiva de usuário do processo atual.	
<b>setvbuf</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int setvbuf(FILE *stream, char *buf, int mode, size_t size);</code>	
<b>Retorno:</b> 0 se OK <0 se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Seta as operações (comportamento) de <i>buffer</i> do arquivo.	
<b>shutdown</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int shutdown(int s, int how);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/socket.h&gt;</code>
<b>Descrição:</b> Faz todas, ou partes, das conexões <i>full-duplex</i> em um <i>socket</i> , associado com <i>s</i> serem fechados.	
<b>siginterrupt</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int siginterrupt(int sig, int flag);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;signal.h&gt;</code>
<b>Descrição:</b> Habilita um sinal para interromper uma chamada de sistema.	

### sleep

Nível: 3

**Protótipo:** `unsigned int sleep(unsigned int seconds);`

**Retorno:** 0 se terminou o tempo informado  
# segundos que faltavam para o término.

**Header:** `#include <unistd.h>`

**Descrição:** Paralisa o processo (dorme) com a quantidade de segundos informado.

### signal

Nível: 2

**Protótipo:** `typedef void (*sighandler_t)(int);  
sighandler_t signal(int signum, sighandler_t handler);`

**Retorno:** Outros sinais se OK  
SIG\_ERR em caso de erro

**Header:** `#include <signal.h>`

**Descrição:** Instala uma função para tratar os sinais que o processo recebe.

### sin

Nível: 3

**Protótipo:** `double sin(double num);`

**Retorno:** Seno de num.

**Header:** `#include <math.h>`

**Descrição:** Devolve o seno de número. Número deve ser informado em radianos.

### socket

Nível: 2

**Protótipo:** `int socket(int domain, int type, int protocol);`

**Retorno:** Descritor para arquivos  
-1 se houve erro.

**Header:** `#include <sys/types.h>  
#include <sys/socket.h>`

**Descrição:** Cria um *socket* para comunicação.

### sprintf

Nível: 3

**Protótipo:** `int sprintf(char *string, const char * format,  
[Value, ...]);`

**Retorno:** >0 se ok  
<0 em caso de erro

**Header:** `#include <stdio.h>`

**Descrição:** Formata uma saída e coloca em uma *string*.

### sqrt

Nível: 3

**Protótipo:** `double sqrt(double num);`

**Retorno:** Raiz Quadrada ou erro.

**Header:** `#include <math.h>`

**Descrição:** Devolve a raiz quadrada do número ou erro se número for negativo.

### srand

Nível: 3

**Protótipo:** `void srand(int seed);`

**Retorno:** Não tem retorno.

**Header:** `#include <stdlib.h>`

**Descrição:** Estabelece um ponto de partida para uma seqüência gerada pela função *rand*.

<b>stat</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int stat(const char *file_name, struct stat *buf);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code> <code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b> Retorna informações a respeito de um arquivo.	
<b>stime</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int stime(time_t *t);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Ajusta a hora e a data do sistema.	
<b>strcasecmp</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int strcasecmp(const char *s1, const char *s2);</code>	
<b>Retorno:</b> 0 se iguais <0 se s1 < s2 >0 se s1 > s2	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Compara duas strings ignorando se são maiúsculas ou minúsculas.	
<b>strcat</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *strcat(char *dest, const char *src);</code>	
<b>Retorno:</b> Ponteiro para string	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Concatena string origem na string destino.	
<b>strchr</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *strchr(const char *s, int c);</code>	
<b>Retorno:</b> Ponteiro para string NULL se não existir o caractere	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Retorna a primeira ocorrência do caractere na string.	
<b>strcmp</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int strcmp(const char *s1, const char *s2);</code>	
<b>Retorno:</b> 0 se iguais <0 se s1 < s2 >0 se s1 > s2	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Compara duas strings para verificar se são iguais.	
<b>strcoll</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int strcoll(const char *s1, const char *s2);</code>	
<b>Retorno:</b> 0 se iguais <0 se s1 < s2 >0 se s1 > s2	<b>Header:</b> <code>#include &lt;string.h&gt;</code>
<b>Descrição:</b> Compara duas strings para verificar se são iguais (utilizando as configurações locais para a verificação, veja <code>setlocale</code> ).	

### strcpy

Nível: 3

**Protótipo:** `char *strcpy(char *dest, const char *src);`  
**Retorno:** Ponteiro para a *string* destino.  
**Header:** `#include <string.h>`  
**Descrição:** Copia uma *string* (*src*) para outra (*dest*).

### strdup

Nível: 3

**Protótipo:** `char *strdup(const char *s);`  
**Retorno:** Ponteiro para a *string*  
**Header:** `#include <string.h>`  
**Descrição:** Duplica uma *string*.

### strerror

Nível: 3

**Protótipo:** `char *strerror(int errnum);`  
**Retorno:** Ponteiro para a *string*  
**Header:** `#include <string.h>`  
**Descrição:** Retorna uma *string* contendo o erro indicado.

### strftime

Nível: 3

**Protótipo:** `size_t strftime(char *s, size_t max, const char *format, const struct tm *tm);`  
**Retorno:** # caracteres colocados na *string*  
**Header:** `#include <time.h>`  
**Descrição:** Formata uma data/hora em um *string* conforme os parâmetros especificados.

### strlen

Nível: 3

**Protótipo:** `size_t strlen(const char *string)`  
**Retorno:** Tamanho da *string*  
**Header:** `#include <string.h>`  
**Descrição:** Retorna o tamanho de uma *string* terminada com `\0` (NULL)

### strncasecmp

Nível: 3

**Protótipo:** `int strncasecmp(const char *s1, const char *s2, size_t n);`  
**Retorno:** 0 se iguais  
<0 se *s1* < *s2*  
>0 se *s1* > *s2*  
**Header:** `#include <string.h>`  
**Descrição:** Compara duas *strings* (até *n* caracteres) ignorando se são maiúsculas ou minúsculas.

### strncat

Nível: 3

**Protótipo:** `char *strncat(char *dest, const char *src, size_t n);`  
**Retorno:** Ponteiro para *string*  
**Header:** `#include <string.h>`  
**Descrição:** Concatena *string* origem na *string* destino (até *n* bytes).

**strncmp****Nível:** 3**Protótipo:** `int strncmp(const char *s1, const char *s2, size_t n);`**Retorno:** 0 se iguais  
<0 se *s1* < *s2*  
>0 se *s1* > *s2* **Header:** `#include <string.h>`**Descrição:** Compara duas strings (até *n bytes*) para verificar se são iguais.**strncpy****Nível:** 3**Protótipo:** `char *strncpy(char *dest, const char *src, size_t n);`**Retorno:** Ponteiro para a *string* destino. **Header:** `#include <string.h>`**Descrição:** Copia uma *n bytes* de string (*src*) para outra (*dest*)**strlen****Nível:** 3**Protótipo:** `size_t strlen (const char *s, size_t maxlen);`**Retorno:** # de bytes **Header:** `#include <string.h>`**Descrição:** Retorna a quantidade de bytes da string até o máximo indicado.**strpbrk****Nível:** 3**Protótipo:** `char *strpbrk(const char *s, const char *accept);`**Retorno:** Ponteiro para string **Header:** `#include <string.h>`  
NULL se não encontrou**Descrição:** Pesquisa e retorna a primeira ocorrência de *accept* em *s*.**strptime****Nível:****Protótipo:** `char *strptime(const char *s, const char *formato, struct tm *tm);`**Retorno:** Ponteiro para string **Header:** `#include <time.h>`  
NULL se não ocorreu a conversão**Descrição:** Converte uma representação de hora do tipo *string* para uma estrutura de hora *tm*.**strrchr****Nível:** 3**Protótipo:** `char *strrchr(const char *s, int c);`**Retorno:** Ponteiro para string **Header:** `#include <string.h>`  
NULL se não existir o caractere**Descrição:** Retorna a última ocorrência do caractere na string.

### strstr

Nível: 3

**Protótipo:** `char *strstr(const char *haystack, const char *needle);`

**Retorno:** Ponteiro para string  
 NULL se não encontrou

**Header:** `#include <string.h>`

**Descrição:** Procura uma substring dentro de uma string maior.

### strtok

Nível: 3

**Protótipo:** `char *strtok(char *s, const char *delim);`

**Retorno:** Ponteiro para a próxima posição  
 NULL se não existe ou se chegou ao final da string.

**Header:** `#include <string.h>`

**Descrição:** Retorna um token (posição) do delimitador dentro da string.

### sscanf

Nível: 3

**Protótipo:** `int sscanf(const char *str, const char *format, ...);`

**Retorno:** # de dados lidos.  
 EOF se erro

**Header:** `#include <stdio.h>`

**Descrição:** Lê os dados de uma string. A leitura é realizada de forma similar à função `scanf`.

### symlink

Nível: 2

**Protótipo:** `int symlink(const char *oldpath, const char *newpath);`

**Retorno:** 0 se OK  
 -1 se houve erro

**Header:** `#include <unistd.h>`

**Descrição:** Cria um *link* simbólico para o arquivo.

### sync

Nível: 2

**Protótipo:** `int sync(void);`

**Retorno:** 0 sempre.

**Header:** `#include <unistd.h>`

**Descrição:** Sincroniza as informações que estão no cache do sistema com o disco.

### sysconf

Nível: 3

**Protótipo:** `long sysconf(int name);`

**Retorno:** Configuração do ambiente

**Header:** `#include <unistd.h>`

**Descrição:** Obtém informações de configuração do sistema operacional em tempo de execução.

### syslog

Nível: 3

**Protótipo:** `void syslog(int priority, char *format, ...);`

**Retorno:** Não tem retorno.

**Header:** `#include <syslog.h>`

**Descrição:** Gera e envia mensagens para o sistema de logs do sistema operacional.

## 312 ♦ Programando em C para Linux, Unix e Windows

<b>sysctl</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int sysctl(struct __sysctl_args *args);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;linux/unistd.h&gt;</code> <code>#include &lt;linux/sysctl.h&gt;</code>
<b>Descrição:</b> Lê/escreve parâmetros do sistema/kernel. Por exemplo: nome da máquina, número máximo de arquivos abertos.	
<b>system</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int system (const char * string);</code>	
<b>Retorno:</b> >0 se OK 127 se não foi possível executar -1 se houve algum erro	<b>Header:</b> <code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b> Executa um comando no sistema operacional.	
<b>tan</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double tan(double num);</code>	
<b>Retorno:</b> Tangente de num.	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Devolve a tangente do número. Número deve ser informado em radianos.	
<b>telldir</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>off_t telldir(DIR *dir);</code>	
<b>Retorno:</b> Posição atual -1 se houve erro.	<b>Header:</b> <code>#include &lt;dirent.h&gt;</code>
<b>Descrição:</b> Retorna a posição atual dentro da estrutura de diretórios.	
<b>tempnam</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *tempnam(const char *dir, const char *pfx);</code>	
<b>Retorno:</b> Ponteiro para o nome	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Retorna um nome de arquivo temporário único para uso no sistema operacional.	
<b>time</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>time_t time(time_t *t);</code>	
<b>Retorno:</b> Tempo em segundos.	<b>Header:</b> <code>#include &lt;time.h&gt;</code>
<b>Descrição:</b> Retorna o tempo desde 01 de Janeiro de 1970 às 00:00:00 UTC, medido em segundos.	
<b>times</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>clock_t times(struct tms *buf);</code>	
<b>Retorno:</b> Número de tiques do relógio desde que foi ativado.	<b>Header:</b> <code>#include &lt;sys/times.h&gt;</code>
<b>Descrição:</b> Armazena o tempo do processo atual em buf.	



<b>tmpfile</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>FILE *tmpfile (void);</code>	
<b>Retorno:</b> Ponteiro para arquivo NULL se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Cria um arquivo temporário único para uso dentro do sistema operacional.	
<b>tmpnam</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>char *tmpnam(char *s);</code>	
<b>Retorno:</b> Ponteiro para nome NULL se houve erro	<b>Header:</b> <code>#include &lt;stdio.h&gt;</code>
<b>Descrição:</b> Cria um nome para arquivo temporário.	
<b>toascii</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int toascii (int c);</code>	
<b>Retorno:</b> Caractere convertido.	<b>Header:</b> <code>#include &lt;ctype.h&gt;</code>
<b>Descrição:</b> Converte um caractere para o seu código ASCII.	
<b>toupper</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int toupper (int c);</code>	
<b>Retorno:</b> Caractere convertido.	<b>Header:</b> <code>#include &lt;ctype.h&gt;</code>
<b>Descrição:</b> Converte um caractere para maiúscula.	
<b>tolower</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>int tolower (int c);</code>	
<b>Retorno:</b> Caractere convertido.	<b>Header:</b> <code>#include &lt;ctype.h&gt;</code>
<b>Descrição:</b> Converte um caractere para minúscula.	
<b>trunc</b>	<b>Nível: 3</b>
<b>Protótipo:</b> <code>double trunc(double x);</code>	
<b>Retorno:</b> Número	<b>Header:</b> <code>#include &lt;math.h&gt;</code>
<b>Descrição:</b> Trunca o número (arredonda).	
<b>truncate</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>int truncate(const char *path, off_t length);</code>	
<b>Retorno:</b> 0 se OK -1 se houve erro	<b>Header:</b> <code>#include &lt;unistd.h&gt;</code> <code>#include &lt;sys/types.h&gt;</code>
<b>Descrição:</b> Trunca o arquivo conforme o tamanho passado.	
<b>umask</b>	<b>Nível: 2</b>
<b>Protótipo:</b> <code>mode_t umask(mode_t mask);</code>	
<b>Retorno:</b> Máscara anterior.	<b>Header:</b> <code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;sys/stat.h&gt;</code>
<b>Descrição:</b> Seleciona a máscara de criação dos arquivos.	

<b>umount</b>	<b>Nível: 2</b>
<b>Protótipo:</b>	<code>int umount(const char *target);</code> <code>int umount2(const char *target, int flags);</code>
<b>Retorno:</b>	0 se OK -1 se houve erro
<b>Header:</b>	<code>#include &lt;sys/mount.h&gt;</code>
<b>Descrição:</b>	Desmonsta um sistema de arquivos montado através de mount.
<b>uname</b>	<b>Nível: 2</b>
<b>Protótipo:</b>	<code>int uname(struct utsname *buf);</code>
<b>Retorno:</b>	0 se OK -1 se houve erro
<b>Header:</b>	<code>#include &lt;sys/utsname.h&gt;</code>
<b>Descrição:</b>	Obtém o nome e as informações sobre o sistema atual.
<b>unlink</b>	<b>Nível: 2</b>
<b>Protótipo:</b>	<code>int unlink(const char *pathname);</code>
<b>Retorno:</b>	0 se OK -1 se houve erro
<b>Header:</b>	<code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b>	Apaga um nome a partir do sistema de arquivos; se o nome tem a última ligação ( <i>inode</i> ) no sistema de arquivos, o arquivo é removido e o <i>inode</i> liberado.
<b>unsetenv</b>	<b>Nível: 3</b>
<b>Protótipo:</b>	<code>void unsetenv(const char *name);</code>
<b>Retorno:</b>	Não tem retorno.
<b>Header:</b>	<code>#include &lt;stdlib.h&gt;</code>
<b>Descrição:</b>	Remove uma variável de ambiente.
<b>usleep</b>	<b>Nível: 3</b>
<b>Protótipo:</b>	<code>void usleep(unsigned long usec);</code>
<b>Retorno:</b>	Não tem retorno.
<b>Header:</b>	<code>#include &lt;unistd.h&gt;</code>
<b>Descrição:</b>	Suspende a execução do processo em microssegundos.
<b>utime</b>	<b>Nível: 2</b>
<b>Protótipo:</b>	<code>int utime(const char *filename, struct utimbuf *buf);</code>
<b>Retorno:</b>	0 se OK -1 se houve erro
<b>Header:</b>	<code>#include &lt;sys/types.h&gt;</code> <code>#include &lt;utime.h&gt;</code>
<b>Descrição:</b>	Altera a data de acesso ou modificação de um <i>inode</i> .
<b>va_end</b>	<b>Nível: 3</b>
<b>Protótipo:</b>	<code>void va_end( va_list ap);</code>
<b>Retorno:</b>	Não tem retorno.
<b>Header:</b>	<code>#include &lt;stdarg.h&gt;</code>
<b>Descrição:</b>	Finaliza a lista de parâmetros variáveis que uma função pode receber.

### va\_copy

Nível: 3

**Protótipo:** void va\_copy(va\_list dest, va\_list src);

**Retorno:** Não tem retorno.

**Header:** #include <stdarg.h>

**Descrição:** Copia uma lista de parâmetros para outra.

### va\_start

Nível: 3

**Protótipo:** void va\_start( va\_list ap, last);

**Retorno:** Não tem retorno.

**Header:** #include <stdarg.h>

**Descrição:** Inicializa a lista de parâmetros variáveis que uma função pode receber.

### wait

Nível: 2

**Protótipo:** pid\_t wait(int \*status);

**Retorno:** PID do filho

**Header:** #include <sys/types.h>

-1 se houve erro

#include <sys/wait.h>

**Descrição:** Aguarda o término do filho.

### waitpid

Nível: 2

**Protótipo:** pid\_t waitpid(pid\_t pid, int \*status, int options);

**Retorno:** PID do filho

**Header:** #include <sys/types.h>

-1 se houve erro

#include <sys/wait.h>

**Descrição:** Verifica se um processo filho terminou e com qual *status*.

### write

Nível: 2

**Protótipo:** ssize\_t write(int fd, const void \*buf, size\_t count);

**Retorno:** # bytes gravados

**Header:** #include <unistd.h>

-1 se houve erro

**Descrição:** Grava bytes em um arquivo.

### writew

Nível: 3

**Protótipo:** int writew(int filedes, const struct iovec \*vector, size\_t count);

**Retorno:** # bytes gravados

**Header:** #include <sys/uio.h>

-1 se houve erro

**Descrição:** Grava informações em vários *buffers*.





## **Bibliografia**

Advanced Programming in the Unix Environment. Stevens, W. Richard. Editora Addison Wesley Pub.

C Completo e Total. Schildt, Hebert. Editora Makron Books.

Guia Completo ao Teste de Software. Hetzel, William. Editora Campus.

Unix Networking Programming – Networking Apis-Sockets and XTI. Stevens, W. Richard. Editora Prentice Hall.

Unix Networking Programming – Interprocess Communications. Stevens, W. Richard. Editora Prentice Hall.





## Índice Remissivo

---

### #

#define · 58  
#else · 59  
#endif · 59  
#error · 65  
#if · 59  
#ifdef · 62  
#ifndef · 62  
#include · 57  
#undef · 63

---

### ?

? · 35

---

### A

accept · 214  
access · 171  
Ajuda · 247  
ANSI · 95  
asctime · 131  
atexit · 54

---

### B

bind · 210

Bolha · 241  
break · 42

---

### C

C ANSI · 4  
C ISO · 4  
C99 · 5  
calloc · 116  
Case Sensitive · 10  
clearerr · 110  
close · 155, 218  
connect · 212  
Constantes · 6  
continue · 43  
creat · 154  
Criação de processos · 178  
ctime · 131

---

### D

daemon · 200  
do...while · 41

---

### E

errno · 138  
Erros · 138, 205

Estruturas · 120  
 execl · 185  
 exit · 45  
 extern · 52

## F

fclose · 98  
 feof · 104  
 ferrord · 110  
 fflush · 108  
 fopen · 96  
 for · 39  
 fork · 179  
 fprintf · 105  
 fread · 99  
 free · 115  
 fscanf · 106  
 fseek · 103  
 ftell · 109  
 Funções · 47, 48, 271  
 \_exit · Erro! Indicador não definido.  
 abort · Erro! Indicador não definido.  
 abs · Erro! Indicador não definido.  
 accept · Erro! Indicador não definido.  
 access · Erro! Indicador não definido.  
 acos · Erro! Indicador não definido.  
 alarm · Erro! Indicador não definido.  
 alloca · Erro! Indicador não definido.  
 asctime · Erro! Indicador não definido.  
 asin · Erro! Indicador não definido.  
 assert · Erro! Indicador não definido.  
 atan · Erro! Indicador não definido.  
 atexit · Erro! Indicador não definido.  
 atof · Erro! Indicador não definido.  
 atoi · Erro! Indicador não definido.  
 atol · Erro! Indicador não definido.  
 atoll · Erro! Indicador não definido.  
 basename · Erro! Indicador não definido.  
 bind · Erro! Indicador não definido.  
 bsearch · Erro! Indicador não definido.  
 calloc · Erro! Indicador não definido.  
 cbrt · Erro! Indicador não definido.  
 ceil · Erro! Indicador não definido.  
 chdir · Erro! Indicador não definido.  
 chmod · Erro! Indicador não definido.  
 chown · Erro! Indicador não definido.

chroot · Erro! Indicador não definido.  
 clearerr · Erro! Indicador não definido.  
 clearenv · Erro! Indicador não definido.  
 clock · Erro! Indicador não definido.  
 close · Erro! Indicador não definido.  
 closedir · Erro! Indicador não definido.  
 closelog · Erro! Indicador não definido.  
 connect · Erro! Indicador não definido.  
 copysign · Erro! Indicador não definido.  
 cos · Erro! Indicador não definido.  
 creat · Erro! Indicador não definido.  
 crypt · Erro! Indicador não definido.  
 ctermid · Erro! Indicador não definido.  
 ctime · Erro! Indicador não definido.  
 cuserid · Erro! Indicador não definido.  
 daemon · Erro! Indicador não definido.  
 difftime · Erro! Indicador não definido.  
 dirname · Erro! Indicador não definido.  
 div · Erro! Indicador não definido.  
 dup · Erro! Indicador não definido.  
 dup2 · Erro! Indicador não definido.  
 endgrent · Erro! Indicador não definido.  
 endpwent · Erro! Indicador não definido.  
 execl · Erro! Indicador não definido.  
 execl · Erro! Indicador não definido.  
 execlp · Erro! Indicador não definido.  
 execv · Erro! Indicador não definido.  
 execve · Erro! Indicador não definido.  
 execvp · Erro! Indicador não definido.  
 exit · Erro! Indicador não definido.  
 exp · Erro! Indicador não definido.  
 fabs · Erro! Indicador não definido.  
 fchdir · Erro! Indicador não definido.  
 fchmod · Erro! Indicador não definido.  
 fchown · Erro! Indicador não definido.  
 fclose · Erro! Indicador não definido.  
 fcntl · Erro! Indicador não definido.  
 fdopen · Erro! Indicador não definido.  
 feof · Erro! Indicador não definido.  
 ferrord · Erro! Indicador não definido.



fflush · Erro! Indicador não definido.  
 fgetc · Erro! Indicador não definido.  
 fgetgrent · Erro! Indicador não definido.  
 fgetpwent · Erro! Indicador não definido.  
 fgetpos · Erro! Indicador não definido.  
 fgets · Erro! Indicador não definido.  
 fileno · Erro! Indicador não definido.  
 flock · Erro! Indicador não definido.  
 floor · Erro! Indicador não definido.  
 fopen · Erro! Indicador não definido.  
 fopen64 · Erro! Indicador não definido.  
 fork · Erro! Indicador não definido.  
 fpathconf · Erro! Indicador não definido.  
 fprintf · Erro! Indicador não definido.  
 fputc · Erro! Indicador não definido.  
 fputs · Erro! Indicador não definido.  
 fread · Erro! Indicador não definido.  
 free · Erro! Indicador não definido.  
 freopen · Erro! Indicador não definido.  
 freopen64 · Erro! Indicador não definido.  
 fscanf · Erro! Indicador não definido.  
 fseek · Erro! Indicador não definido.  
 fsetpos · Erro! Indicador não definido.  
 fstat · Erro! Indicador não definido.  
 fsync · Erro! Indicador não definido.  
 ftell · Erro! Indicador não definido.  
 ftime · Erro! Indicador não definido.  
 ftruncate · Erro! Indicador não definido.  
 fwrite · Erro! Indicador não definido.  
 getc · Erro! Indicador não definido.  
 getchar · Erro! Indicador não definido.  
 getcwd · Erro! Indicador não definido.  
 getdate · Erro! Indicador não definido.  
 getegid · Erro! Indicador não definido.  
 getenv · Erro! Indicador não definido.  
 geteuid · Erro! Indicador não definido.  
 getgid · Erro! Indicador não definido.  
 getgrent · Erro! Indicador não definido.

getgrgid · Erro! Indicador não definido.  
 getgrnam · Erro! Indicador não definido.  
 getgroups · Erro! Indicador não definido.  
 gethostid · Erro! Indicador não definido.  
 gethostname · Erro! Indicador não definido.  
 getlogin · Erro! Indicador não definido.  
 getlogin\_r · Erro! Indicador não definido.  
 getmsg · Erro! Indicador não definido.  
 getopt · Erro! Indicador não definido.  
 getpeername · Erro! Indicador não definido.  
 getpgid · Erro! Indicador não definido.  
 getpgrp · Erro! Indicador não definido.  
 getpid · Erro! Indicador não definido.  
 getpmsg · Erro! Indicador não definido.  
 getppid · Erro! Indicador não definido.  
 getpwent · Erro! Indicador não definido.  
 getpwnam · Erro! Indicador não definido.  
 getpwuid · Erro! Indicador não definido.  
 gets · Erro! Indicador não definido.  
 getsid · Erro! Indicador não definido.  
 getsockname · Erro! Indicador não definido.  
 gmtime · Erro! Indicador não definido.  
 gettimeofday · Erro! Indicador não definido.  
 getuid · Erro! Indicador não definido.  
 getumask · Erro! Indicador não definido.  
 initgroups · Erro! Indicador não definido.  
 isalnum · Erro! Indicador não definido.  
 isalpha · Erro! Indicador não definido.

iscntrl · Erro! Indicador não definido.  
 isdigit · Erro! Indicador não definido.  
 isgraph · Erro! Indicador não definido.  
 islower · Erro! Indicador não definido.  
 isprint · Erro! Indicador não definido.  
 ispunct · Erro! Indicador não definido.  
 isspace · Erro! Indicador não definido.  
 isupper · Erro! Indicador não definido.  
 isxdigit · Erro! Indicador não definido.  
 kill · Erro! Indicador não definido.  
 killpg · Erro! Indicador não definido.  
 labs · Erro! Indicador não definido.  
 llabs · Erro! Indicador não definido.  
 lchown · Erro! Indicador não definido.  
 link · Erro! Indicador não definido.  
 listen · Erro! Indicador não definido.  
 localtime · Erro! Indicador não definido.  
 log · Erro! Indicador não definido.  
 log10 · Erro! Indicador não definido.  
 login · Erro! Indicador não definido.  
 logout · Erro! Indicador não definido.  
 lseek · Erro! Indicador não definido.  
 lseek64 · Erro! Indicador não definido.  
 lstat · Erro! Indicador não definido.  
 malloc · Erro! Indicador não definido.  
 memchr · Erro! Indicador não definido.  
 memcmp · Erro! Indicador não definido.  
 memcpy · Erro! Indicador não definido.  
 memmove · Erro! Indicador não definido.  
 memset · Erro! Indicador não definido.  
 mkdtemp · Erro! Indicador não definido.  
 mkdir · Erro! Indicador não definido.  
 mkfifo · Erro! Indicador não definido.  
 mknod · Erro! Indicador não definido.  
 mkstemp · Erro! Indicador não definido.  
 mktemp · Erro! Indicador não definido.  
 mktime · Erro! Indicador não definido.  
 mmap · Erro! Indicador não definido.  
 mount · Erro! Indicador não definido.

mremap · Erro! Indicador não definido.  
 msync · Erro! Indicador não definido.  
 munmap · Erro! Indicador não definido.  
 nice · Erro! Indicador não definido.  
 open · Erro! Indicador não definido.  
 opendir · Erro! Indicador não definido.  
 openlog · Erro! Indicador não definido.  
 pause · Erro! Indicador não definido.  
 pathconf · Erro! Indicador não definido.  
 pclose · Erro! Indicador não definido.  
 perror · Erro! Indicador não definido.  
 pipe · Erro! Indicador não definido.  
 popen · Erro! Indicador não definido.  
 pow · Erro! Indicador não definido.  
 printf · Erro! Indicador não definido.  
 psignal · Erro! Indicador não definido.  
 ptrace · Erro! Indicador não definido.  
 putc · Erro! Indicador não definido.  
 putchar · Erro! Indicador não definido.  
 putenv · Erro! Indicador não definido.  
 putgrent · Erro! Indicador não definido.  
 putpwent · Erro! Indicador não definido.  
 puts · Erro! Indicador não definido.  
 qsort · Erro! Indicador não definido.  
 raise · Erro! Indicador não definido.  
 rand · Erro! Indicador não definido.  
 read · Erro! Indicador não definido.  
 readv · Erro! Indicador não definido.  
 realloc · Erro! Indicador não definido.  
 readdir · Erro! Indicador não definido.  
 readlink · Erro! Indicador não definido.  
 recvfrom · Erro! Indicador não definido.  
 remove · Erro! Indicador não definido.  
 rename · Erro! Indicador não definido.  
 rewind · Erro! Indicador não definido.  
 rewinddir · Erro! Indicador não definido.  
 rmdir · Erro! Indicador não definido.  
 scanf · Erro! Indicador não definido.

seekdir · Erro! Indicador não definido.  
 select · Erro! Indicador não definido.  
 semctl · Erro! Indicador não definido.  
 semget · Erro! Indicador não definido.  
 semop · Erro! Indicador não definido.  
 send · Erro! Indicador não definido.  
 sendto · Erro! Indicador não definido.  
 setbuf · Erro! Indicador não definido.  
 setbuffer · Erro! Indicador não definido.  
 setenv · Erro! Indicador não definido.  
 seteuid · Erro! Indicador não definido.  
 setegid · Erro! Indicador não definido.  
 setgid · Erro! Indicador não definido.  
 setgrent · Erro! Indicador não definido.  
 setgroups · Erro! Indicador não definido.  
 sethostid · Erro! Indicador não definido.  
 sethostname · Erro! Indicador não definido.  
 setlinebuf · Erro! Indicador não definido.  
 setlocale · Erro! Indicador não definido.  
 setpgid · Erro! Indicador não definido.  
 setpgrp · Erro! Indicador não definido.  
 setpwent · Erro! Indicador não definido.  
 settimeofday · Erro! Indicador não definido.  
 setsid · Erro! Indicador não definido.  
 setuid · Erro! Indicador não definido.  
 setvbuf · Erro! Indicador não definido.  
 shutdown · Erro! Indicador não definido.  
 siginterrupt · Erro! Indicador não definido.  
 sleep · Erro! Indicador não definido.  
 signal · Erro! Indicador não definido.  
 sin · Erro! Indicador não definido.  
 socket · Erro! Indicador não definido.  
 sprintf · Erro! Indicador não definido.  
 sqrt · Erro! Indicador não definido.  
 srand · Erro! Indicador não definido.  
 stat · Erro! Indicador não definido.  
 stime · Erro! Indicador não definido.

strcasecmp · Erro! Indicador não definido.  
 strcat · Erro! Indicador não definido.  
 strchr · Erro! Indicador não definido.  
 strcmp · Erro! Indicador não definido.  
 strcoll · Erro! Indicador não definido.  
 strcpy · Erro! Indicador não definido.  
 strdup · Erro! Indicador não definido.  
 strerror · Erro! Indicador não definido.  
 strftime · Erro! Indicador não definido.  
 strlen · Erro! Indicador não definido.  
 strncasecmp · Erro! Indicador não definido.  
 strncat · Erro! Indicador não definido.  
 strncmp · Erro! Indicador não definido.  
 strncpy · Erro! Indicador não definido.  
 strnlen · Erro! Indicador não definido.  
 strpbrk · Erro! Indicador não definido.  
 strptime · Erro! Indicador não definido.  
 strrchr · Erro! Indicador não definido.  
 strstr · Erro! Indicador não definido.  
 strtok · Erro! Indicador não definido.  
 sscanf · Erro! Indicador não definido.  
 symlink · Erro! Indicador não definido.  
 sync · Erro! Indicador não definido.  
 sysconf · Erro! Indicador não definido.  
 syslog · Erro! Indicador não definido.  
 sysctl · Erro! Indicador não definido.  
 system · Erro! Indicador não definido.  
 tan · Erro! Indicador não definido.  
 telldir · Erro! Indicador não definido.  
 tempnam · Erro! Indicador não definido.  
 time · Erro! Indicador não definido.  
 times · Erro! Indicador não definido.  
 tmpfile · Erro! Indicador não definido.  
 tmpnam · Erro! Indicador não definido.  
 toascii · Erro! Indicador não definido.  
 toupper · Erro! Indicador não definido.  
 tolower · Erro! Indicador não definido.  
 trunc · Erro! Indicador não definido.  
 truncate · Erro! Indicador não definido.

umask · **Erro! Indicador não definido.**  
 umount · **Erro! Indicador não definido.**  
 uname · **Erro! Indicador não definido.**  
 unlink · **Erro! Indicador não definido.**  
 unsetenv · **Erro! Indicador não definido.**  
 usleep · **Erro! Indicador não definido.**  
 utime · **Erro! Indicador não definido.**  
 va\_end · **Erro! Indicador não definido.**  
 va\_copy · **Erro! Indicador não definido.**  
 va\_start · **Erro! Indicador não definido.**  
 wait · **Erro! Indicador não definido.**  
 waitpid · **Erro! Indicador não definido.**  
 write · **Erro! Indicador não definido.**  
 writev · **Erro! Indicador não definido.**  
 fwrite · 100

## G

getchar · 25  
 geteuid · 168  
 gethostbyname · 220  
 getpeername · 219  
 getpid · 166  
 getppid · 166  
 getuid · 168  
 gmtime · 132  
 goto · 44

## H

htonl · 209  
 htons · 209

## I

if · 33  
 if...else... · 34  
 inet\_addr · 211  
 inet\_aton · 211  
 inet\_ntoa · 211

## K

K&R · 4  
 kill · 193

## L

LCC-Win32 · 255  
 Linux · 166, 188, 247  
 listen · 213  
 localtime · 132  
 lseek · 159

## M

main · 9  
 malloc · 114  
 Matrizes · 70, 72, 89, 126  
 Memória · 113  
 memset · 118  
 mktime · 134

## N

ntohl · 209  
 ntohs · 209

## O

open · 149  
 Ordenação · 241

## P

Palavras Reservadas · 11  
 perror · 140  
 Pesquisa · 244  
 Pointer Member · 127  
 Ponteiros · 32, 86, 127  
 Pré-Compilação · 56  
 printf · 20  
 Protótipo · 47  
 putchar · 24

---

## Q

Quicksort · 242

---

## R

raise · 196  
read · 156  
realloc · 117  
Recursividade · 239  
recv · 217  
recvfrom · 218  
remove · 161  
rename · 164  
return · 49

---

## S

scanf · 24  
send · 215  
sendto · 218  
setuid · 202  
sfrtime · 135  
shutdown · 218  
signal · 191  
Sinais · 188  
sizeof · 18  
sleep · 198  
sockaddr · 208  
socket · 207  
sprintf · 82  
sscanf · 82  
stat · 172  
static · 53  
strcat · 79  
strcmp · 80  
strcpy · 80  
Streams · 111

strerror · 139  
Strings · 75, 90  
strlen · 78  
strncat · 83  
strncmp · 85  
strncpy · 84  
strstr · 92  
strtok · 93  
switch...case · 36  
syslog · 205  
system · 187

---

## T

TCP · 221, 222, 223  
time · 129  
typedef · 17, 123

---

## U

UDP · 224, 225, 226  
umask · 176  
uname · 169  
Uniãos · 142  
Unix · 148, 166, 188, 247  
unlink · 162

---

## V

Variáveis · 14, 50  
Vetores · 69

---

## W

wait · 181  
waitpid · 183  
while · 40  
write · 158





# Sumário

<b>Introdução.....</b>	<b>1</b>
<b>1. Informações Básicas.....</b>	<b>3</b>
1.1 História.....	3
1.2 C de K&R.....	4
1.3 C ANSI e C ISO .....	4
1.4 C99 .....	5
1.5 Comentários .....	6
1.6 Constantes Numéricas .....	6
1.7 Outras Constantes.....	7
1.8 Estrutura de um Programa .....	8
1.9 Função main .....	9
1.10 O que main devolve.....	10
1.11 O C é "Case Sensitive" .....	10
1.12 Palavras Reservadas do C.....	11
<b>2. Tipos de Dados .....</b>	<b>12</b>
2.1 Tipos Básicos .....	12
2.2 Abrangência e Modificadores de Tipo .....	13
2.3 Nomenclatura de Variáveis .....	14
2.4 Definição de Variáveis .....	16
2.5 Atribuição de Valores .....	16
2.6 Definição de Constantes .....	16
2.7 Conversão de Tipos .....	17
2.8 Declaração typedef.....	17
2.9 Operador sizeof .....	18

## **X ♦ Programando em C para Linux, Unix e Windows**

<b>3. Entrada e Saída .....</b>	<b>20</b>
3.1 Função printf .....	20
3.2 Formataadores de Tipos .....	21
3.3 Indicando o Tamanho .....	21
3.4 Função putchar .....	24
3.5 Função scanf .....	24
3.6 Função getchar .....	25
<b>4. Operadores .....</b>	<b>26</b>
4.1 Operadores Aritméticos .....	26
4.2 Operadores Unários .....	26
4.3 Operadores de Atribuição .....	27
4.4 Operadores Relacionais .....	28
4.5 Prioridade de Avaliação .....	28
4.6 Operadores Lógicos .....	29
4.7 Assinalamento de Variáveis .....	30
4.8 Parênteses e Colchetes como Operadores .....	31
4.9 Operador & e * para Ponteiros .....	32
<b>5. Comandos de Seleção .....</b>	<b>33</b>
5.1 Comando if .....	33
5.2 Comando if...else .....	34
5.3 Operador ? : .....	35
5.4 Comando switch...case .....	36
<b>6. Comandos de Repetição .....</b>	<b>39</b>
6.1 Comando for .....	39
6.2 Comando while .....	40
6.3 Comando do...while .....	41
6.4 Comando break .....	42
6.5 Comando continue .....	43
6.6 Comando goto .....	44
6.7 Comando exit .....	45
<b>7. Definições de Funções .....</b>	<b>47</b>
7.1 Criação de Funções .....	47
7.2 Função e Protótipo (assinatura da função) .....	47
7.3 Definindo Funções .....	48
7.4 Comando return .....	49
7.5 Escopo de Variáveis .....	50
7.6 Variáveis Globais .....	50
7.7 Variáveis Locais .....	51
7.8 Definição extern .....	52
7.9 Definição static .....	53
7.10 Função atexit .....	54



<b>8. Pré-Compilação .....</b>	<b>56</b>
8.1 Fases de uma compilação.....	56
8.2 Diretiva #include .....	57
8.3 Diretiva #define.....	58
8.4 Diretivas #if, #else e #endif .....	59
8.5 Diretivas #ifdef e #ifndef .....	62
8.6 Diretiva #undef .....	63
8.7 Diretiva #error .....	65
8.8 Variáveis predefinidas .....	66
<b>9. Vetores e Matrizes.....</b>	<b>69</b>
9.1 Definindo Vetores .....	69
9.2 Definindo Matrizes .....	70
9.3 Matrizes n-Dimensionais .....	71
9.4 Inicializando Matrizes .....	72
9.5 Matrizes como Parâmetros.....	73
<b>10. Strings .....</b>	<b>75</b>
10.1 Implementação de Strings.....	75
10.2 Entrada/Saída de Strings .....	76
10.3 String como vetor .....	77
10.4 Função strlen .....	78
10.5 Função strcat .....	79
10.6 Função strcpy .....	80
10.7 Função strcmp .....	80
10.8 Função sprintf .....	82
10.9 Função sscanf .....	82
10.10 Função strncat .....	83
10.11 Função strncpy .....	84
10.12 Função strncmp .....	85
<b>11. Ponteiros .....</b>	<b>86</b>
11.1 Conceito Básico.....	86
11.2 Definição de Ponteiros .....	86
11.3 Uso de Ponteiros .....	87
11.4 Parâmetros de Saída.....	88
11.5 Operações com Ponteiros .....	89
11.6 Ponteiros e Matrizes .....	89
11.7 Ponteiros e Strings .....	90
11.8 Argumentos de Entrada.....	91
11.9 Função strstr .....	92
11.10 Função strtok .....	93

## **XII ♦ Programando em C para Linux, Unix e Windows**

<b>12. Manipulação de Arquivos (padrão ANSI) .....</b>	<b>95</b>
12.1 Conceitos Importantes.....	95
12.2 Ponteiro para Arquivos .....	96
12.3 Função fopen.....	96
12.4 Função fclose .....	98
12.5 Função fread.....	99
12.6 Função fwrite .....	100
12.7 Função fgets .....	101
12.8 Função fseek .....	103
12.9 Função feof.....	104
12.10 Função fprintf.....	105
12.11 Função fscanf.....	106
12.12 Função fflush .....	108
12.13 Função ftell .....	109
12.14 Função ferror e clearerr.....	110
12.15 Streams Padrão.....	111
<b>13. Alocação de Memória .....</b>	<b>113</b>
13.1 Configuração da Memória .....	113
13.2 Função malloc .....	114
13.3 Função free.....	115
13.4 Função calloc .....	116
13.5 Função realloc .....	117
13.6 Função memset.....	118
<b>14. Estruturas .....</b>	<b>120</b>
14.1 Definição de Estruturas .....	120
14.2 Utilização de Estruturas.....	121
14.3 Definindo mais Estruturas .....	122
14.4 Estruturas e o typedef .....	123
14.5 Estruturas Aninhadas .....	124
14.6 Estruturas e Matrizes.....	126
14.7 Estruturas e Ponteiros .....	127
14.8 Pointer Member .....	127
<b>15. Data e Hora .....</b>	<b>129</b>
15.1 Função time .....	129
15.2 Trabalhando com datas.....	130
15.3 Funções asctime e ctime.....	131
15.4 Funções gmtime e localtime.....	132
15.5 Função mktime .....	134
15.6 Função strftime.....	135

<b>16. Tratamento de Erros .....</b>	<b>138</b>
16.1 Variável errno .....	138
16.2 Função strerror .....	139
16.3 Função perror .....	140
<b>17. Definições Avançadas .....</b>	<b>142</b>
17.1 Definição de Uniões .....	142
17.2 Utilização de Uniões .....	142
17.3 Lista Enumerada .....	143
17.4 Estruturas de Bits .....	144
17.5 Operadores Bit a Bit .....	145
17.6 Deslocamento de Bits .....	146
17.7 Deslocamento de Bits Circular .....	147
<b>18. Manipulação de Arquivos (padrão Linux e Unix) .....</b>	<b>148</b>
18.1 O Sistema de Arquivo Tipo Unix .....	148
18.2 Descritores Pré-alocados .....	149
18.3 Função open .....	149
18.4 Função creat .....	154
18.5 Função close .....	155
18.6 Função read .....	156
18.7 Função write .....	158
18.8 Função lseek .....	159
18.9 Função remove .....	161
18.10 Função unlink .....	162
18.11 Função rename .....	164
<b>19. Buscando Algumas Informações no Linux e Unix .....</b>	<b>166</b>
19.1 Funções getpid e getppid .....	166
19.2 Funções getuid e geteuid .....	168
19.3 Função uname .....	169
19.4 Função access .....	171
19.5 Função stat .....	172
19.6 Função umask .....	176
<b>20. Criando Processos no Linux e Unix .....</b>	<b>178</b>
20.1 Criação de processos no sistema operacional .....	178
20.2 Função fork .....	179
20.3 Função wait .....	181
20.4 Função waitpid .....	183
20.5 Função execl .....	185
20.6 Função system .....	187

## **XIV ♦ Programando em C para Linux, Unix e Windows**

<b>21. Tratamento de Sinais em Linux e Unix .....</b>	<b>188</b>
21.1 Conceito e tratamento de Sinais .....	188
21.2 Alguns Sinais .....	190
21.3 Função signal .....	191
21.4 Função kill.....	193
21.5 Função raise .....	196
21.6 Função sleep .....	198
21.7 Cuidados com algumas funções (funções reentrantes) .....	199
<b>22. Daemons (Serviços) em Linux e Unix.....</b>	<b>200</b>
22.1 Conceito de daemon .....	200
22.2 Regras para codificação de um daemon.....	200
22.3 Função setsid .....	202
22.4 Registrando erros com a função syslog .....	205
<b>23. Programação para Rede .....</b>	<b>207</b>
23.1 Função socket .....	207
23.2 Estrutura sockaddr .....	208
23.3 Funções htonl, htons, ntohl, ntohs .....	209
23.4 Função bind .....	210
23.5 Funções inet_aton, inet_addr e inet_ntoa .....	211
23.6 Função connect .....	212
23.7 Função listen .....	213
23.8 Função accept .....	214
23.9 Função send .....	215
23.10 Função recv .....	217
23.11 Funções sendto e recvfrom .....	218
23.12 Funções close e shutdown .....	218
23.13 Função getpeername .....	219
23.14 Função gethostbyname .....	220
23.15 Diagrama de servidor/cliente TCP básico .....	221
23.16 Exemplo completo de um servidor TCP .....	222
23.17 Exemplo completo de um cliente TCP .....	223
23.18 Diagrama de servidor/cliente UDP básico.....	224
23.19 Exemplo completo de um servidor UDP .....	225
23.20 Exemplo completo de um cliente UDP .....	226
<b>24. Técnicas de Programação para Facilitar a Depuração,     Documentação, Economia e Execução de Processos .....</b>	<b>228</b>
<b>Apêndice A. Programas Avançados .....</b>	<b>239</b>
A.1 Recursividade.....	239
A.2 Ordenação.....	241
A.3 Pesquisa.....	244

<b>Apêndice B. Ajuda .....</b>	<b>247</b>
B.1 Obtendo ajuda no Linux e Unix.....	247
B.2 Seções do Manual .....	247
B.3 Divisão da Documentação .....	248
B.4 Exemplo de Utilização.....	248
<b>Apêndice C. Compilando no Linux.....</b>	<b>251</b>
<b>Apêndice D. Utilizando o LCC-Win32.....</b>	<b>255</b>
D.1 Instalação.....	255
D.2 Criando Projetos no LCC-Win32.....	261
D.2.1 Criando um projeto .....	262
D.3 Criando um Programa e Compilando .....	267
<b>Apêndice E. Guia de Referência das Funções .....</b>	<b>271</b>
<b>Bibliografia .....</b>	<b>317</b>
<b>Índice Remissivo .....</b>	<b>319</b>