Introdução à Programação em Linguagem C

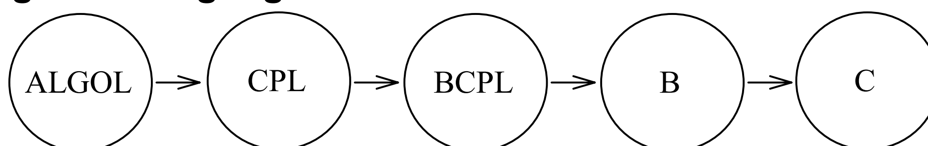
1. Enfoque

Será tratada aqui a linguagem C tradicional, sendo que existem outras variantes tais como ANSI C e GNU C.

2. Objetivos

Familiarização com comandos, tipos de dados básicos, estratégias de fluxo de controle e mecanismos de construção de novos tipos de dados oferecidos pela linguagem C. Uso dessas ferramentas para implementação de algoritmos. Conhecimento de algumas funções da biblioteca de rotinas C no sistema UNIX.

3. Origem da linguagem



4. Noções Básicas sobre a Linguagem C

4.1. Funções

Na linguagem C, todo programa é basicamente um conjunto de *funções*, não havendo distinção entre procedimentos e funções como em PASCAL e FORTRAN. Uma função tem um nome e argumentos associados, e é composta por declarações de variáveis e blocos de comandos, incluindo possivelmente chamadas a outras funções. Um bloco de comandos composto por mais de um comando deve ser delimitado por um par de chaves.

Em um programa C, se um nome não foi previamente declarado, ocorre em uma expressão e é seguido por um parêntese esquerdo, ele é declarado pelo contexto como sendo o nome de uma função. A função de nome especial **main** (principal) contém os pontos de início e término da execução de um programa em C. Neste caso, todo programa em C deve ter pelo menos a função **main**.

Exemplo 1 *O seguinte código em C define uma função denominada* **main** *com uma lista de argumentos vazia (nenhuma variável entre os parênteses que a seguem). A função* **main** *retorna o valor 1 do tipo inteiro. O corpo da função é composto por uma linha de declaração de variáveis e duas linhas de instruções de atribuição (assignment statements). Observe que todas as instruções devem ser terminadas por <;> e os comentários delimitados pelos símbolos </*> e <*/>.*

```
int main()                /* O nome da funcao e' main */
{
  int i, j;               /* 2 variaveis do tipo inteiro */

  i = 10;                 /* atribuir 10 a i */
  j = 30;                 /* atribuir 30 a j */
  return(1);              /* atribuir o valor 1 a funcao main */
}
```

4.2. Variáveis

Uma variável é um nome simbólico dado a uma região da memória que armazena um valor a ser utilizado por uma função. Palavras reservadas da linguagem C (como **int**, **for** e **if**) não podem ser utilizadas como nomes de variáveis. O nome de uma variável pode conter letras e números, mas deve começar com uma letra. Observe que a linguagem C faz distinção entre caracteres maiúsculos e minúsculos.

Toda variável que for utilizada em uma função em C deve ser previamente declarada, ou seja, associada a um dos tipos de dados disponíveis.

4.3. Tipos de dados básicos

C apresenta dois tipos de dados principais: inteiro e caractere, denotados por **int** e **char** respectivamente. Os números inteiros podem ser, portanto, criados através da declaração **int** e eles são manipuláveis através de operações como + (adição), − (subtração), * (multiplicação) e / (divisão). A remoção é feita automaticamente. **Observe que a divisão entre inteiros produz resposta inteira, com a parte fracionária truncada.**

Os tipos de dados ponto flutuante, denotado por **float**, e ponto flutuante de precisão dupla, denotado por **double**, são também oferecidos em C. Como a linguagem C não suporta o tipo de dado booleano, os valores booleanos são representados pelo tipo de dados inteiro, com o <0> denotando <FALSO> e valores diferentes de <0> denotando <VERDADEIRO>.

O tipo de dados inteiro pode ser ainda qualificado com **short**, **long** ou **unsigned** para determinar o domínio de valores representáveis. O domínio efetivo para cada qualificador é **dependente da máquina**.

Os qualificadores **register**, **static** e **extern** são utilizados para definir a classe de armazenamento das variáveis. O qualificador **register** orienta o compilador a armazenar as variáveis preferencialmente em registradores da UCP e o qualificador **static** indica que as variáveis devem ter seus endereços fixos. O qualificador **extern**, por sua vez, mostra que as variáveis devem ser definidas em algum outro lugar. Os tipos de dados não acompanhados por nenhum qualificador explícito são considerados **auto**máticos.

Exemplo 2 *Este exemplo apresenta algumas declarações de variáveis em C:*

```
int i;                 /* variavel do tipo inteiro */
short int z1, z2;      /* variavel do tipo inteiro, porem com
                          metade de numero de palavras para representacao */
char c;                /* variavel do tipo caractere */
unsigned short int j;  /* variavel do tipo inteiro, curto e sem o sinal */
long m;                /* variavel do tipo inteiro, porem com
                          o dobro de numero de palavras para representacao */
register char s;       /* variavel do tipo caractere que deve ser
                          colocado preferencialmente num registrador */
float x;               /* variavel do tipo ponto flutuante com precisao simples */
double x;              /* variavel do tipo ponto flutuante com precisao dupla */
```

Conversões entre os tipos de dados são permitidas. Atribuição como:

```
m = i;
```

onde <m> e <i> são declaradas no Exemplo 2, é considerada válida. Em situações, como passagens de parâmetros entre funções, é comum a prática de conversão de tipos de dados para garantir a compatibilidade. Nestes casos, é recomendado “forçar” um tipo ao outro através do mecanismo de “retipagem” (*casting*), ou seja,

```
m = (long) i;
```

Exemplo 3 *Este exemplo mostra dois usos de “retipagem” para compatibilizar o tipo de dado da variável <i> da função principal* **main** *e o tipo de dado da variável <x> da função* **sub***, que retorna um valor do tipo ponto flutuante de precisão simples. Como a variável <y> do programa principal é do tipo ponto flutuante de precisão dupla, foi utilizado novamente o mecanismo de “retipagem” (***(double)***) para atribuir o valor retornado de* **sub** *à variável <y>. Note que a definição da função* **sub** *segue o padrão da linguagem C tradicional, que difere do padrão ANSI C.*

```
float sub(x)
float x;
{
    float z;

    z = x+20.0;
    return(z);
}

int main()
{
    int i;
    double y;

    y = (double) sub((float) i);
    return(1);
}
```

4.4. Tipos de representação de dados

Em C, números inteiros podem ter representação **decimal** (qualquer seqüência de algarismos entre 0 e 9 que inicie com um algarismo diferente de 0), **octal** (qualquer seqüência de algarismos entre 0 e 7 que inicie com o algarismo 0) ou **hexadecimal** (qualquer seqüência de algarismos entre 0 e F que inicie com o prefixo 0x).

Valores com representação em ponto flutuante utilizam o ponto decimal e/ou a notação exponencial, como em 1.47e-3 (= 0.00147).

Além dos caracteres alfanuméricos e de pontuação, que podem ser representados diretamente pelo símbolo correspondente entre aspas (como em 'A'), são também definidos em C caracteres especiais, como: \n (nova linha), \t (tabulação), \b (retrocesso), \r (*carriage return*), \\ (contrabarra), \' (apóstrofo), \" (aspas), \0 (caractere null), \xxx (qualquer padrão de bits xxx em octal).

4.5. Mecanismos de construção de tipos de dados mais complexos

A linguagem C dispõe de quatro mecanismos para construir tipos de dados mais complexos:

4.5.1. Arranjo ou vetor

É um agregado de elementos (conjunto de variáveis) do mesmo tipo. **Os índices dos elementos de um arranjo começam sempre de 0.** Um arranjo pode ser uni-, bi-, tri- ou *n*-dimensional. **Observe que o compilar C não verifica se o índice para acesso a um elemento de um arranjo está dentro da faixa declarada.**

Exemplo 4 *A declaração*

```
int valores[100];
```

cria um arranjo unidimensional <valores> com 100 números inteiros referenciáveis por valores[0] até valores[99].

Um arranjo bidimensional é na verdade um arranjo unidimensional em que cada elemento é um arranjo. Arranjos *n*-dimensionais, com *n* elevado, não são geralmente utilizados em C, pois além de ocuparem muito espaço de memória, o acesso a seus elementos não ocorre de maneira eficiente.

Um dos tipos de arranjo que mais ocorre em C é o **arranjo de caracteres**, ou *string*. Por convenção, C considera como uma *string* uma seqüência de caracteres armazenada sob a forma de um arranjo do tipo `char` cujo último elemento é o caractere NULL ('\0').

4.5.2. Estrutura (struct) ou registro

É um agregado de elementos não necessariamente do mesmo tipo, mas de tamanho definido. Os elementos de uma estrutura são, muitas vezes, conhecidos como **membros**. Uma operação básica sobre uma estrutura é a referência aos seus membros. Isso é feito pelo operador <.>. Qualquer tipo de dados válido pode estar presente em uma estrutura, até mesmo uma outra estrutura.

Exemplo 5 *A declaração*

```
struct racional {
     int  numerador;
     int  denominador;
} v;
```

OU

```
struct racional {
     int  numerador;
     int  denominador;
};
struct racional v;
```

significa que a variável <v> *é uma estrutura composta de dois membros do tipo inteiro. Para atribuir os valores 2 e 5 a estes membros, recorre-se aos seguintes comandos:*

```
v.numerador = 2;
v.denominador = 5;
```

4.5.3. União (union) ou alternativa

Este mecanismo permite atribuir tipos alternativos de dados (de tamanho fixo) a uma mesma variável.

Exemplo 6 *A declaração*

```
union aluno {
    char nome[10];
    int  ra;
} id;
```

especifica que a variável <id> pode ser um arranjo de 10 caracteres OU um número inteiro. O compilador C aloca sempre um espaço suficientemente grande para alojar o membro de maior tamanho Neste caso particular, se o membro <nome> precisar de 10 bytes e o membro <ra> de 4 bytes, um espaço de memória de 10 bytes será alocado à variável <id> durante a compilação.

4.5.4. Enumeração (enum)

Permite definir tipos de dados por meio dos valores ordenados (ordem crescente) que os dados daquele tipo podem tomar, podendo assim realizar operações lógicas entre eles. Cada valor é denotado por um identificador não-ambíguo.

Exemplo 7 *Depois das declarações*

```
enum mes {jan,fev,mar,abr,mai,jun,jul,ago,set,out,nov,dez};
enum mes valor;
enum status {ERRO,OK};
enum status flag;
```

as seguintes atribuições são válidas:

```
valor = ago;    /* equivalente a valor = 7 */
flag = OK;      /* equivalente a flag = 1 */
```

4.6. Apontadores ou ponteiros

O conceito de apontadores é fundamental para o entendimento e implementação de programas em C. Os apontadores estão associados a mecanismos de referência indireta que permitem atribuir como valores de uma variável os endereços de outras variáveis (os valores "referem" ou "apontam" para outras variáveis). O valor apontado por um apontador p é representado por *p e o endereço de uma variável valor é representado por &valor.

Exemplo 8 *A linha de comando*

```
int i, *pi, a[10], *pa[10], **ppi;
```

declara que a variável <i> é um número inteiro; <pi>, um apontador a um valor inteiro (contém o endereço de um número inteiro); <a>, um arranjo unidimensional de 10 valores inteiros; <pa>, um arranjo unidimensional de 10 apontadores a valores inteiros e <ppi>, o endereço de um apontador a um valor inteiro (apontador a um apontador a um valor inteiro). Para atribuirmos o endereço de uma variável do tipo inteiro <i> à variável <pi>, usamos o operador &, isto é:

```
pi = &i;
```

Observe que uma diferença fundamental entre um apontador e o nome de um arranjo é que o apontador (pi no exemplo 8) é uma variável, enquanto que o nome de um arranjo (a no exemplo 8) é uma constante. No entanto, após a atribuição

```
pi = a;    /* ou pi = &a[0]  */
```

ambos podem ser utilizados para apontar para o início de um arranjo unidimensional de 10 inteiros.

O mecanismo de apontadores é muito útil para processar os elementos de um arranjo de estruturas. Neste caso, pode-se usar o operador → para se ter acesso a um membro da estrutura.

Exemplo 9 *A declaração*

```
struct chave {
      char *palavra_chave;
      int quantidade;
} tabela[100];
```

define um arranjo para 100 palavras reservadas e o seguinte comando

```
struct chave *k;
```

declara que a variável <k> contém o endereço de uma estrutura chave. *Portanto, o comando*

```
k = &tabela[1];
```

implica em atribuir à variável <k> o endereço do segundo elemento do arranjo <tabela>. E se quisermos atribuir o membro <quantidade> do segundo elemento da estrutura <tabela> à variável do tipo inteiro <n>, poder-se-á usar um dos seguintes comandos equivalentes:

```
n = k->quantidade;
```

ou

```
n = *k.quantidade;
```

ou ainda

```
n = tabela[1].quantidade;
```

Os apontadores são também úteis para alocações dinâmicas em linguagem C. Pode-se criar dados do tipo apontador a um determinado tipo de dado e alocar efetivamente o espaço necessário para o valor ou estrutura referenciada somente durante a execução de um programa, como mostra o exemplo 10.

Exemplo 10 *O programa mostra a alocação dinâmica de espaço de memória. Observe que em C quando um arranjo é referenciado pelo seu nome, sem [], estamos acessando de fato o apontador ao primeiro elemento do arranjo (*`&vet[0]≡vet`*).*

```
#include <stdio.h>
#include <malloc.h>          /* inclusao da declaracao das funcoes
                                malloc e free */

int main()
{
     int *vet;               /* vet aponta para um dado
                                do tipo inteiro */
     int i, j;

     printf("entre numero de elementos:");
     scanf("%d", &i);
     vet = (int *)malloc(sizeof(int)*i);
                             /* aloca um espaco igual a i*sizeof(int) bytes
                                sizeof devolve o tamanho de um tipo de
                                dado em bytes */
     for (j=0; j < i; j++)
         vet[j] = j;

     printf("vet[%d] = %d\n", i-1, vet[i-1]);
                             /* imprimir o conteudo do
                                ultimo elemento do arranjo */
     free(vet);
     return(1);
}
```

Utilizando o mecanismo de apontadores, pode-se ainda construir tipos de dados polimórficos em C, porque variáveis do tipo apontador para tipo de dado **void** podem ser utilizadas para armazenar apontadores de qualquer outro tipo.

4.6.1. Apontadores para funções

Apontadores para funções tornam-se interessantes quando o programador não pode determinar qual função deve ser executada em uma dada situação, a não ser durante a execução do programa. É possível referenciar o endereço de uma função pois toda função nada mais é que um conjunto de instruções e dados armazenados na memória.

Exemplo 11 *A forma de se declarar uma variável do tipo apontador para função é ilustrada a seguir:*

```
tipo_1 funcao1(tipo_2,...,tipo_n)  /* declaracao de uma funcao denominada funcao1*/
tipo_1 (*apfunc)()                 /* declaracao de um apontador para funcao */
                                   /* os parenteses sao indispensaveis */
...
apfunc = funcao1;                  /* inicializa apontador */
(*apfunc)(tipo_2,...,tipo_n);      /* invoca funcao1 atraves do apontador apfunc */
```

4.7. Definição de nomes de tipos

Em C é ainda possível atribuir um sinônimo a um tipo de dados, associando a ele uma semântica, através da instrução **typedef**. Embora isso não faça nenhuma diferença ao compilador, a estratégia ajuda muito na clareza dos códigos produzidos.

Exemplo 12 *Os comandos*

```
typedef int semaforo;
typedef unsigned int booleano;
typedef struct chave pc;
```

definem os sinônimos para os tipos de dados **int**, **unsigned int** *e* **struct chave**, *respectivamente. Assim as declarações:*

```
semaforo mutex;
booleano flag;
pc tabela[100];
```

inidicam que `<mutex>`, `<flag>` *e* `<tabela>` *são do tipo* **int**, **unsigned int** *e* **struct chave**, *respectivamente.*

4.8. Comandos de controle

Além da instrução de atribuição, cuja sintaxe é:

```
variavel = valor;
```

a linguagem C suporta os seguintes blocos de instruções:

if...else if...else... :

```
if (condicao){
    bloco de instrucoes;
}
else if (condicao) {
    bloco de instrucoes;
}
else {
    bloco de instrucoes;
}
```

while :

```
while (condicao){
    bloco de instrucoes;
}
```

do...while :

```
do {
    bloco de instrucoes;
} while (condicao);
```

for :

```
for (inicializacao;condicao;expressao) {
    bloco de instrucoes;
}
```

que é completamente equivalente a

```
inicializacao;
while (condicao){
    bloco de instrucoes;
    expressao;
}
```

switch :

```
switch (variavel){
    case padrao1:
        bloco de instrucoes;break;
    case padrao2:
        bloco de instrucoes;break;
    ...
    case ...
        ...
    default:
        bloco de instrucoes;
}
```

O comando **break** interrompe uma execução em laço se ele estiver dentro dos blocos de comando **for** e **while** ou ignora outras instruções subseqüentes dentro do bloco de comando **switch**. Já o comando **continue** interrompe a iteração corrente dentro de um laço de instruções, provocando um desvio incondicional ao início do laço.

O comando **return** desvia incondicionalmente ao programa "chamador" e retorna o valor do tipo de dados da função em que este comando está contido. **Em implementações eficientes procura-se evitar retornar valores de tipo de dados complexos, tais como arranjos e estruturas. No lugar deles retornamos os seus endereços.**

As funções em C podem ter uma lista de argumentos ou não. **Uma boa prática de programação em C é evitar passar argumentos de tipos de dados complexos. Devemos substituí-los pelos seus endereços (passar os seus respectivos apontadores).**

4.9. Entrada/saída

É interessante observar que C não oferece nenhuma instrução específica para entrada/saída. As funções de entrada/saída são feitas através de funções pré-definidas existentes na biblioteca de C, como:

```
printf (controle, [arg1, [arg2, [...]]]);         /* imprimir arg1, arg2, ... conforme
                                                     o formato <controle> */
sprintf (string, controle, [arg1, [arg2, [...]]]);/* concatenar arg1,arg2,... conforme
                                                     o formato <controle> em string */
```

```
scanf (controle, arg1, [arg2, [...]]]);          /* ler arg1,arg2,... conforme o
                                                    formato <controle> */
sscanf (string, controle, arg1, [arg2, [...]]]); /* desconcatenar string conforme o
                                                 formato <controle> em arg1,arg2,... */
```

<`controle`> é de fato uma seqüência de caracteres delimitada por aspas. Qualquer caractere não precedido por % (percentagem) é impresso da forma como ele é. Quando % é encontrado, o próximo argumento <`arg`> é processado (impresso/lido) conforme o formato definido pela letra que segue %, ou seja para:

d : imprimir/ler como um inteiro decimal;
o : imprimir/ler como um inteiro octal;
u : imprimir/ler como um inteiro decimal sem sinal;
x : imprimir/ler como um inteiro hexadecimal;
f : imprimir/ler como um racional no formato **[-]mmm.nnnnnn**;
e: imprimir/ler como um racional no formato **[-]m.nnnnnnE[+ -]xx**;
s : imprimir/ler como uma cadeia de caracteres;
c : imprimir/ler como um caractere simples.

Combinações com <`l`> são permitidas para imprimir "números com precisão maior (*long*)".

Exemplo 13 *O seguinte código em linguagem C ilustra o uso das funções de entrada/saída-padrão ("standard io").*

```
#include <stdio.h>       /* incluir as declaracoes das funcoes
                            de IO - scanf e printf */
void i2a(n,s)            /* converte valor n em caractere. i2a nao retorna nenhum valor.
                            Por isso, ela e' declarada como void */
char s[];
int n;
{
    if (n < 10)
       s[0] = n + '0';
    else
       s[0] = n + 'A' - '0' - 10;
    s[1] = '\0';             /* uma cadeia de caracteres deve ser terminada
                                com '\0' */
    return;
}

int main()
{   int DIGIT;               /* criar um espaco para um valor inteiro */
    char CHAR[2];            /* criar um espaco para um caractere */

    scanf("%d", &DIGIT);     /* ler um valor de tipo inteiro */
    i2a(DIGIT, CHAR);        /* converter o valor em caractere */
    printf("%s \n", CHAR);   /* imprimir a cadeia de caracteres */
    return(1);
}
```

4.10. Expressões

Nesta seção são apresentadas algumas concatenações usuais de operandos e operadores em C. Em C distinguem-se os seguintes operadores;

- **aritméticos unários** : + (positivo) e – (negativo).
- **aritméticos binários** : + (soma), – (subtração), * (multiplicação), / (divisão) e % (resto de uma divisão).
- **de comparação** : < (menor), > (maior), <= (menor ou igual), >= (maior ou igual), == (igual) e != (diferente).
- **lógicos** : && (e), || (ou) e ! (não).
- **de incremento e decremento** : ++ (incremento) e –– (decremento). Se a notação é pré-fixada (antes do operando), o valor do operando é in/decrementado antes de ser utilizado e se a notação é pós-fixada (depois do operando), o valor do operando é in/decrementado depois de ser utilizado. Ressaltamos aqui que o in/decremento depende do tipo da variável. **Tipos inteiro e caractere são sempre in/decrementado de 1, mas os apontadores são in/decrementados de acordo com o tamanho do tipo para o qual eles referenciam.**
- **de manipulação de bits** : & (E bit a bit), | (OU bit a bit), ^ (OU EXCLUSIVO bit a bit), << (deslocamento à esquerda), >> (deslocamento à direita) e ~ (complemento de um).
- **atribuição** : o operador de atribuição é o sinal =. A combinação *op*=, onde *op* é +, –, *, /, %, <<, >>, &, ^ e |), numa expressão <`e1 op= e2;`> é equivalente a <`(e1)=(e1)op(e2);`>.

- **atribuição condicional** : ?. A linha de comando <valor=e1?e2:e3;> significa que <e2> é atribuído a <valor>, se <e1> for verdadeiro; caso contrário, <e3> é atribuído a <valor>.

4.11. Diretrizes para o preprocessador

A linguagem C suporta uma série de diretrizes que facilitam o desenvolvimento de um programa. Estas diretrizes instruem o preprocessador a converter um programa-fonte num programa realmente processável pelo compilador C e elas se distinguem de outras linhas de instruções pelo símbolo # na primeira coluna. **As diretrizes mais importantes são as para inclusão de arquivos e as para definição (e expansão) de macros.**

Uma diretriz da forma

```
#include "myheader.h"
```

indica que o preprocessador deve incluir o conteúdo do arquivo <myheader.h>; enquanto uma diretriz da forma

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

indica que o preprocessador deve acessar o arquivo <stdio.h> no diretório *default* e incluí-lo.

Por exemplo, quando o preprocessador encontra a diretriz

```
#define EPSILON 1e-10
```

ele substitui, a partir daquele ponto do programa, a palavra EPSILON por 1e-10.

4.12. Argumentos na linha de comando

É possível passar argumentos para um programa C a partir da linha de comando do sistema operacional, declarando a função **main** na forma:

```
tipo main(int argc,char *argv[])
```

O primeiro argumento, `argc`, indica o número de *tokens* presente na linha de comando. O segundo argumento, `argv`, é um arranjo de *strings* em que cada elemento do arranjo representa um dos *tokens* da linha de comando.

4.13. Regras de escopo

As variáveis declaradas em uma função C são variáveis locais, privativas à função, não sendo diretamente acessíveis por outras funções. Toda variável local passa a existir somente quando a função é chamada, deixando de existir quando a função termina, sendo por isso denominadas de variáveis automáticas. Esta condição pode ser modificada com a declaração `static`. Outras funções podem ter acesso a variáveis locais de uma determinada função quando recebem como argumento o valor ou o endereço destas variáveis (com o endereço, é possível inclusive alterar o valor destas variáveis).

Como uma alternativa às variáveis locais ou automáticas, é possível definir variáveis externas a todas as funções, denominadas variáveis globais. Qualquer função pode ter acesso direto a variáveis globais, mas é muitas vezes necessário que também haja uma declaração destas variáveis dentro de cada função que a utiliza.

5. Referências Bibliográficas

Bacon, J.W. "The C/Unix Programmer's Guide", Acadix Software & Consulting, 1999.

Kutti, N.S. "C and Unix Programming: A Comprehensive Guide", Lightspeed Books, 2002.

Kernighan, B.W. e Ritchie, D.M. "The C Programming Language", Prentice-Hall, 1978.

Ricarte, I.L.M. DCA/FEEC/Unicamp, Notas de aula do curso EA876, 2001. (ftp://ftp.dca.fee.unicamp.br/pub/docs/ea876/progsist.pdf)

–, Manual de Referência das Funções da Biblioteca C do Sistema UNIX.

–, Manual de Referência do compilador de C (cc e gcc) instalado no UNIX.