# Programação Estruturada

Prof. Rodrigo Hausen
http://progest.compscinet.org

## Entrada e Saída
## Parte 2

# FUNÇÕES DE SAÍDA DA BIBLIOTECA PADRÃO

Vamos ver as funções mais comuns da biblioteca para escrever na saída padrão.

- **int** puts(**const char** **s*)

  Escreve a string *s*, seguida de uma quebra de linha, na saída padrão.

  Retorna um número não-negativo caso a operação seja bem sucedida ou a constante EOF em caso de erro.

# FUNÇÕES DE SAÍDA DA BIBLIOTECA PADRÃO

- **int** printf(**const char** *****_formato_, ...);

  Escreve na saída padrão de acordo com a string *formato* e com os parâmetros que a seguem.

  A string *formato* pode ser qualquer coisa. A maioria dos caracteres é impressa como tal, **mas atenção para o caractere %!** Ele será explicado a seguir.

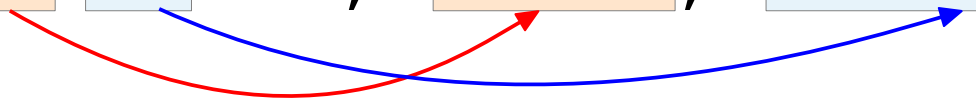
  Retorna o número de caracteres impressos ou um número negativo em caso de erro.

# FUNÇÕES DE SAÍDA DA BIBLIOTECA PADRÃO

Edite novamente o programa alomundo.c

```
#include <stdio.h>

int main(void) {
  printf("%s %s\n", "Alô", "mundo!");
}
```



Compile e execute.

Cada diretiva %… corresponde a um parâmetro na ordem em que aparecem em *formato*.

# DIRETIVAS DA FUNÇÃO PRINTF

Mais comuns:

%**s** – imprime o **char** * (**s**tring) passado como parâmetro. Deve ser um ponteiro **válido**.
%**d** – imprime **int** na base 10 (**d**ecimal)
%**ld** – imprime **long int** na base 10
%**u** – imprime **unsigned int** na base 10
%**f** – imprime **float**[1] na base 10
%**lf** – imprime **double**[1] na base 10 (**l**ong **f**loat)
%**e** – imprime **float** ou **double** em notação científica (constante e **e**xpoente)
%g – escolhe automaticamente a representação mais curta entre %f e %e

[1] Na prática, printf converte float para double, logo %f e %lf têm o mesmo significado.

# LENDO A ENTRADA PADRÃO

Há diversas funções para ler dados da entrada padrão.

Começaremos por:

- **int** scanf(**const char** **formato, ...*);

Lê dados **formatados** da entrada padrão. A string *formato* deve conter **especificadores de conversão**. O resultado da(s) conversão (ões), se houver, serão armazenados nas posições indicadas pelos **ponteiros** passados como argumentos.

Cada ponteiro deve ser de um tipo apropriado ao valor retornado pela conversão especificada.

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

%d – converte para **int**

%u – converte para **unsigned int**

%ld – converte para **long int**

%f – converte para **float**

%lf – converte para **double**

Os especificadores de conversão para scanf são parecidos com as diretivas de printf, **MAS CUIDADO: eles não são idênticos!**

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

Exemplos: (execute no cling)

Lê o próximo inteiro e o próximo float da entrada padrão

```
[cling]$ #include <stdio.h>
[cling]$ int i; float f;
[cling]$ scanf("%d %f", &i, &f)
```

Digite “10 9.9” e veja os valores armazenados em i e f.

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

Exemplos: (execute no cling)

Lê o próximo inteiro e o próximo float da entrada padrão

```
[cling]$ #include <stdio.h>
[cling]$ int i; float f;
[cling]$ scanf("%d %f", &i, &f)
```

Digite “10 9.9” e veja os valores armazenados em i e f.

```
[cling]$ i
(int) 10
[cling]$ f
(float) 9.90000f
```

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

Reinicie o cling e execute novamente

```
[cling]$ #include <stdio.h>
[cling]$ int i; float f;
[cling]$ scanf("%d %f", &i, &f)
```

Desta vez, digite “10 abc 9.9” e veja o resultado.

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

Reinicie o cling e execute novamente

```
[cling]$ #include <stdio.h>
[cling]$ int i; float f;
[cling]$ scanf("%d %f", &i, &f)
```

Desta vez, digite “10 abc 9.9” e veja o resultado.

```
[cling]$ i
(int) 10
[cling]$ f
(float) 0.00000f
```

*Apenas uma das conversões foi bem sucedida. Observe que scanf retornou 1, a quantidade de conversões que foram executadas corretamente.*

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

Para ler strings usando scanf, devem ser usados os especificadores:

%***n***[***xyz…***] - lê, no máximo, ***n*** caracteres **iguais a *x* ou *y* ou *z* ou** …

%***n***[^***xyz…***] - lê, no máximo, ***n*** caracteres **diferentes** de ***x***, ***y***, ***z***, …

Exemplos:

```
char telefone[21];
scanf("%20[-+()0123456789]", telefone);
```

*Um byte a menos que o tamanho do array, para poder armazenar o caractere nulo ao final.*

# ESPECIFICADORES DE CONVERSÃO PARA SCANF

Para ler strings usando scanf, devem ser usados os especificadores:

%***n***[***xyz…***] - lê, no máximo, ***n*** caracteres **iguais a *x* ou *y* ou *z* ou** …

%***n***[^***xyz…***] - lê, no máximo, ***n*** caracteres **diferentes** de ***x***, ***y***, ***z***, …

Exemplos:

```
char telefone[21];
scanf("%20[-+()0123456789]", telefone);
```

*Caracteres permitidos: algarismos, parênteses, + e -*
*Se - for permitido, **sempre** deve vir primeiro.*

# ENTRADA E SAÍDA EM C

A biblioteca stdio.h carrega uma herança de mais de **40 anos** de implementações.

Vantagem: estabilidade.

Desvantagem: as especificações das funções são antigas e não podem ser alteradas. Uma reimplementação poderia eliminar vários problemas.

Exemplo de problemas:

- a função *gets* é insegura e **não deve ser usada nunca**. Mas continua na biblioteca
- entrada/saída de strings é muito primitiva (problemas com acentos, etc.)

# ENTRADA E SAÍDA EM C

A função *scanf* **jamais** deve ser usada para ler dados do usuário, uma vez que o usuário pode fornecer dados errôneos!
(The User is Drunk - youtu.be/r2CbbBLVaPk)

Há problemas ao usar *scanf* caso os dados não sejam bem formatados.

Se tiver de lidar com entrada direta do usuário, **sempre** leia os dados para string, depois converta (funções *strtol*, *strtod*, etc)

Recomenda-se usar **apenas** as funções *getchar* e *fgets* para entrada direta do usuário.

# ENTRADA DO USUÁRIO

Exemplificando os problemas:

Crie o arquivo digitealgo.c e faça um programa que imprima na tela:

**Digite algo para continuar.**

E espere até que o usuário digite algo. Se o usuário pressionar alguma tecla, o programa imprime o caractere digitado e pede para o usuário digitar um número inteiro. Em seguida, imprime o número.

Use a função *getchar* e scanf.

# ENTRADA DO USUÁRIO

```
#include <stdio.h>
int main(void) {
  char c;
  int n;
  puts("Digite algo para continuar.");
  c = getchar();
  printf("Você digitou: %c\n", c);
  puts("Digite um inteiro:");
  scanf("%d", &n);
  printf("O número é %d\n", n);
}
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
```

*Digite um número qualquer, mas não pressione Enter.*

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
1
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
12
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
12
```

*Ô diacho... por que não continua?*

*getchar não deveria pegar só UM caractere?*

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
123
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
1234
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
12345
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
12345
```

*OK... isto não tem graça!*

*Pressione Enter*

# ENTRADA DO USUÁRIO

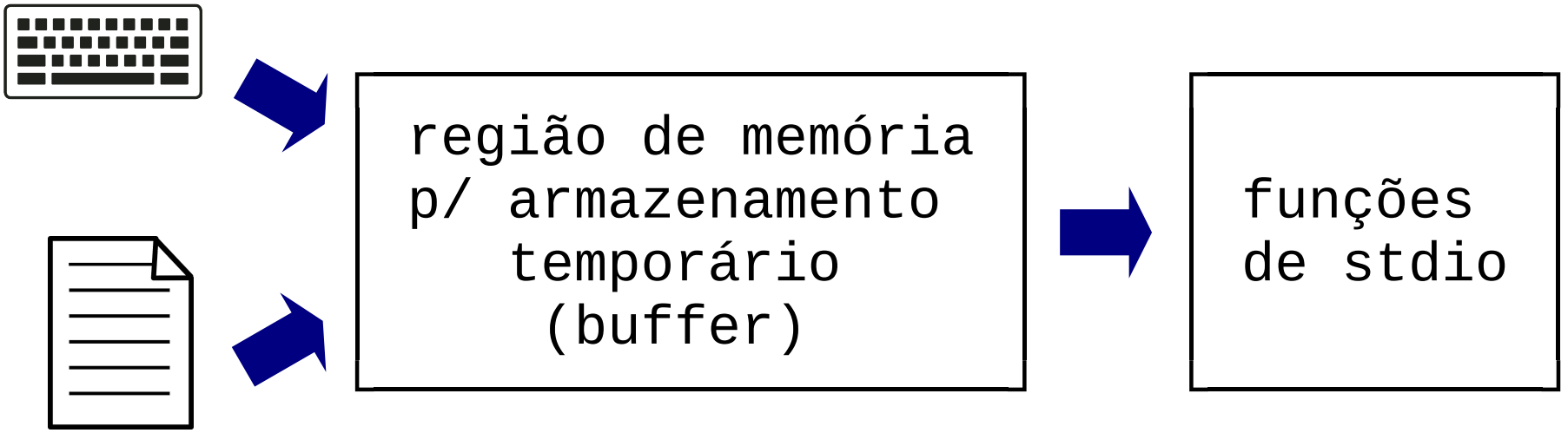
```
$ ./digitealgo
Digite algo para continuar.
12345
Você digitou: 1
Digite um inteiro:
O número é 2345
```

# ENTRADA DO USUÁRIO

```
$ ./digitealgo
Digite algo para continuar.
12345
Você digitou: 1
Digite um inteiro:
O número é 2345
```

*MAS HEIN!? Por que meu programa não esperou que eu digitasse o número inteiro?*

# BUFFERIZAÇÃO




A entrada padrão, venha ela do teclado ou de um arquivo, é **buferizada por linha**:

1. os dados são lidos do teclado/arquivo
2. então são armazenados em uma região da memória até que o usuário pressione Enter (ou até o buffer encher)
3. só então os dados são disponibilizados para as funções de stdio.h

# ENTRADA DO USUÁRIO

```
#include <stdio.h>
int main(void) {
   char c;
   int n;
   puts("Digite Enter para continuar.");
   do {
      c = getchar();
   } while (c != '\n' && c != -1);
   if (c == -1) return 1;
   puts("Digite um inteiro:");
   scanf("%d", &n);
   printf("O número é %d\n", n);
}
```

# BUFERIZAÇÃO NA SAÍDA PADRÃO

A saída padrão também é bufferizada por linha.

```
#include <stdio.h>
#include <unistd.h> // NÃO É PADRÃO!!!

int main(void) {
  printf("Processando");
  for(int i=0; i<10; ++i) {
    sleep(1);
    printf(".");
  }
  printf("\n");
}
```

# BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando
```

# BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando
```

# BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando
Processando.........
$
```

*Após 10 segundos, a mensagem toda aparece, em vez de aparecer um ponto a cada segundo.*

# EVITANDO A BUFFERIZAÇÃO

É possível desativar a bufferização.

(para curiosos: ver função setvbuf)

Mas isto geralmente é **desnecessário** e **desaconselhável**.

Há outras alternativas melhores, as quais serão mencionadas no decorrer do curso.

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

A bufferização por linha pode dificultar a exibição de mensagens de aviso ou de erro na saída padrão.

Solução: não use a saída padrão para mensagens de aviso ou de erro!

Use a **saída de erro**:

```
fprintf(stderr, formato, ...);
```

# EVITANDO A BUFERIZAÇÃO NA SAÍDA PADRÃO

```
#include <stdio.h>
#include <unistd.h> // NÃO É PADRÃO!!!

int main(void) {
  fprintf(stderr, "Processando");
  for(int i=0; i<10; ++i) {
    sleep(1);
    fprintf(stderr, ".");
  }
  fprintf(stderr, "\n");
}
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando..
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando...
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando....
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.....
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.....
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.......
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.........
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.........
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.........
```

# EVITANDO A BUFFERIZAÇÃO NA SAÍDA PADRÃO

```
$ ./processando_fprintf
Processando.........
$ 
```

# A FUNÇÃO fprintf

- **int** fprintf(**FILE** *stream,
  **const char** **formato*, ...);

  Escreve na *stream* de saída de acordo com a string *formato* e com os parâmetros que a seguem.

  A string formato e os parâmetros seguem as mesmas convenções que printf.

  O que é uma *stream*?

## STREAMS

Uma *stream* é uma série de dados (comumente bytes) que podem ser acessados em ordem sequencial.

Em uma *stream de entrada* dados podem ser *lidos.* Em uma *stream de* saída dados podem ser *escritos.*

Uma *stream* é uma abstração que pode descrever a entrada e saída padrões e de erro, arquivos e muito mais! (dispositivos, conexões de rede, ...)

## STREAMS

Há três *streams* padrão definidas em stdio.h:

- **stdin** – representa a entrada padrão
- **stdout** – representa a saída padrão
- **stderr** – representa a saída de erros

A biblioteca stdio também nos permite criar outras *streams*. Veremos hoje como criar *streams* a partir de **arquivos**.

## ABRINDO ARQUIVOS

- **FILE** *fopen(**const char** **caminho*,
  **const char** **modo*)

  Abre o arquivo cujo nome é a string em *caminho* e retorna a *stream* associada a esse arquivo ou NULL em caso de erro.

  Na biblioteca stdio, *streams* são representadas pela estrutura de dados **FILE**.

  A string *modo* controla a forma como o arquivo é aberto: apenas leitura, apenas escrita, leitura/escrita, arquivo de texto ou binário, etc.

## FECHANDO ARQUIVOS

- **int** fclose(**FILE** ***stream***)

  Fecha o arquivo e grava todos os dados pendentes.

  **Sempre** deve ser chamada após terminar de lidar com o arquivo para evitar perda de dados.

# ABRINDO ARQUIVOS

```
#include <stdio.h>


int main(void) {
  FILE *arq = fopen("alomundo.txt", "w");
  // se der erro, termina programa
  if (arq == NULL) return 1;
  fprintf(arq, "Alo mundo!\n");
  fclose(arq);
  return 0;
}
```

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
```

*Compilando...*

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$
```

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
```

*Listando os arquivos no diretório...*

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$
```

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$ ./alomundo
```

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$ ./alomundo
$
```

*O programa não imprime nada na saída padrão nem na saída de erro.*

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$ ./alomundo
$ dir
```

*Listando novamente os arquivos no diretório...*

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$ ./alomundo
$ dir
alomundo  alomundo.c  alomundo.txt
$
```

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$ ./alomundo
$ dir
alomundo  alomundo.c  alomundo.txt
$ cat alomundo.txt
```

*Exibindo o conteúdo do arquivo*
*(no Windows seria*
*type alomundo.txt)*

# ABRINDO ARQUIVOS

```
$ gcc alomundo.c -o alomundo
$ dir
alomundo  alomundo.c
$ ./alomundo
$ dir
alomundo  alomundo.c  alomundo.txt
$ cat alomundo.txt
Alo mundo!
$
```

## A STRING modo DE fopen

- **FILE** *fopen(**const char** **caminho*,
  **const char** **modo*)

  A string modo pode conter as seguintes sequências:

  **r** abre arquivo apenas para leitura, colocando o indicador de posição do arquivo no início

  **r+** abre arquivo para leitura/escrita, colocando o indicador no início

  (obs.: ***r*** de *read*)

## A STRING modo DE fopen

- **FILE** *fopen(**const char** **caminho*,
  **const char** **modo*)

  A string modo pode conter as seguintes sequências:

  **w** cria o arquivo (se não existe) ou trunca-o para comprimento zero (se já existe) e abre-o apenas para escrita. O indicador é colocado no início do arquivo.

  **w+** cria ou trunca o arquivo e abre-o para leitura/escrita, colocando o indicador no início do arquivo.

  (obs.: ***w*** de *write*)

## A STRING modo DE fopen

- **FILE** *fopen(**const char** **caminho*,
  **const char** **modo*)

  A string modo pode conter as seguintes sequências:

  **a** abre para escrita, sem truncar. Se o arquivo não existe, é criado. Indicador é colocado no final do arquivo.

  **a+** abre para escrita e leitura, sem truncar. Se arquivo não existe, é criado. Indicador colocado no final do arquivo.

  (obs.: ***a*** de *append*)

## A STRING modo DE fopen

- **FILE** *fopen(**const char** **caminho*,
  **const char** **modo*)

  Opcionalmente, a letra 'b' pode aparecer como último caractere na string *modo* para indicar que o arquivo deve ser aberto em modo **binário**.

  Por padrão, o arquivo é aberto em modo **texto**, o que pode fazer com que algumas sequências de bytes possam ser tratadas de maneira especial (comumente os caracteres de fim de linha).

## A STRING modo DE fopen

Vamos criar o programa tchaumundo.c que reabre o arquivo alomundo.txt no modo append e adiciona a linha “Tchau mundo!”

## A STRING modo DE fopen

Vamos criar o programa tchaumundo.c que reabre o arquivo alomundo.txt no modo append e adiciona a linha “Tchau mundo!”

```
#include <stdio.h>

int main(void) {
  FILE *arq = fopen("alomundo.txt", "a");
  if (arq == NULL) return 1;
  fprintf(arq, "Tchau mundo!\n");
  fclose(arq);
  return 0;
}
```

# LENDO DE UM ARQUIVO

Assim como usamos fprintf para gravar dados em um arquivo, podemos usar fscanf para ler dados **formatados** de um arquivo.

- **int** fscanf(**FILE** ***stream,**
  **const char** ***formato, ...**)

  Comporta-se de maneira similar a scanf, a única diferença é que o primeiro parâmetro é a referência a uma *stream* de entrada.

  Lembre-se que esta função só pode ser usada para ler dados **estritamente formatados**!

## LENDO DE UM ARQUIVO

- **char** *fgets(**char** *str, **int** tamanho, **FILE** *stream)

  Lê bytes da stream e armazena-os na string str. A leitura termina após ler tamanho-1 bytes, ou ao encontrar o primeiro caractere de quebra de linha '\n', ou o final de arquivo (o que ocorrer primeiro).

  Se um caractere de quebra de linha for lido, ele será armazenado na string.

  O caractere nulo '\0' é armazenado após o último byte lido.

  Retorna str ou NULL em caso de erro ou caso nenhum byte seja lido.

# LENDO ARQUIVO LINHA POR LINHA

```
// lemundo.c
#include <stdio.h>

int main(void) {
  const int tamanho = 256;
  char str[tamanho];
  FILE *arq = fopen("alomundo.txt", "r");
  if (arq == NULL) return 1;
  while (fgets(str, tamanho, arq) != NULL) {
    printf("%s", str);
  }
  fclose(arq);
  return 0;
}
```

## PARA CASA

Usando as funções *getchar*, *strtol* e *isspace*, crie a função:

**int** lelong(**const char** ***prompt,** **long** ***n)

que imprime o prompt na tela e lê um número inteiro do usuário. Se a entrada do usuário for incorreta, repete o processo. Se for correta, coloca o número em n.

A função retorna 1 caso a conversão seja bem sucedida ou 0 em caso de erro.

Use um array com 100 caracteres. Despreze todos os caracteres em branco (use *isspace*) no início da entrada do usuário.

Teste:

```
int main(void) {
  long n;
  int ok = lelong("Digite inteiro: ", &n);
  if (ok) {
    printf("Leu: %ld\n", n);
  } else {
    printf("Erro\n", n);
  }
}
```

Teste:

```
$ ./lelong
```

Teste:

```
$ ./lelong
Digite inteiro:
```

Teste:

```
$ ./lelong
Digite inteiro: inteiro
```

Teste:

```
$ ./lelong
Digite inteiro: inteiro
Digite inteiro: 10
```

Teste:

```
$ ./lelong
Digite inteiro: inteiro
Digite inteiro: 10
Leu: 10
$
```

Teste:

```
$ ./lelong
Digite inteiro: inteiro
Digite inteiro: 10
Leu: 10
$ ./lelong
```

Teste:

```
$ ./lelong
Digite inteiro: inteiro
Digite inteiro: 10
Leu: 10
$ ./lelong
Digite inteiro:
```

*Pressione Ctrl + D (se estiver no Linux/Mac OS) ou Ctrl + Z (no Windows)*

*Estas combinações de teclas sinalizam ao programa que a entrada padrão foi finalizada, ou seja, não há mais dados a serem entrados.*

Teste:

```
$ ./lelong
Digite inteiro: inteiro
Digite inteiro: 10
Leu: 10
$ ./lelong
Digite inteiro: Erro
$
```