C

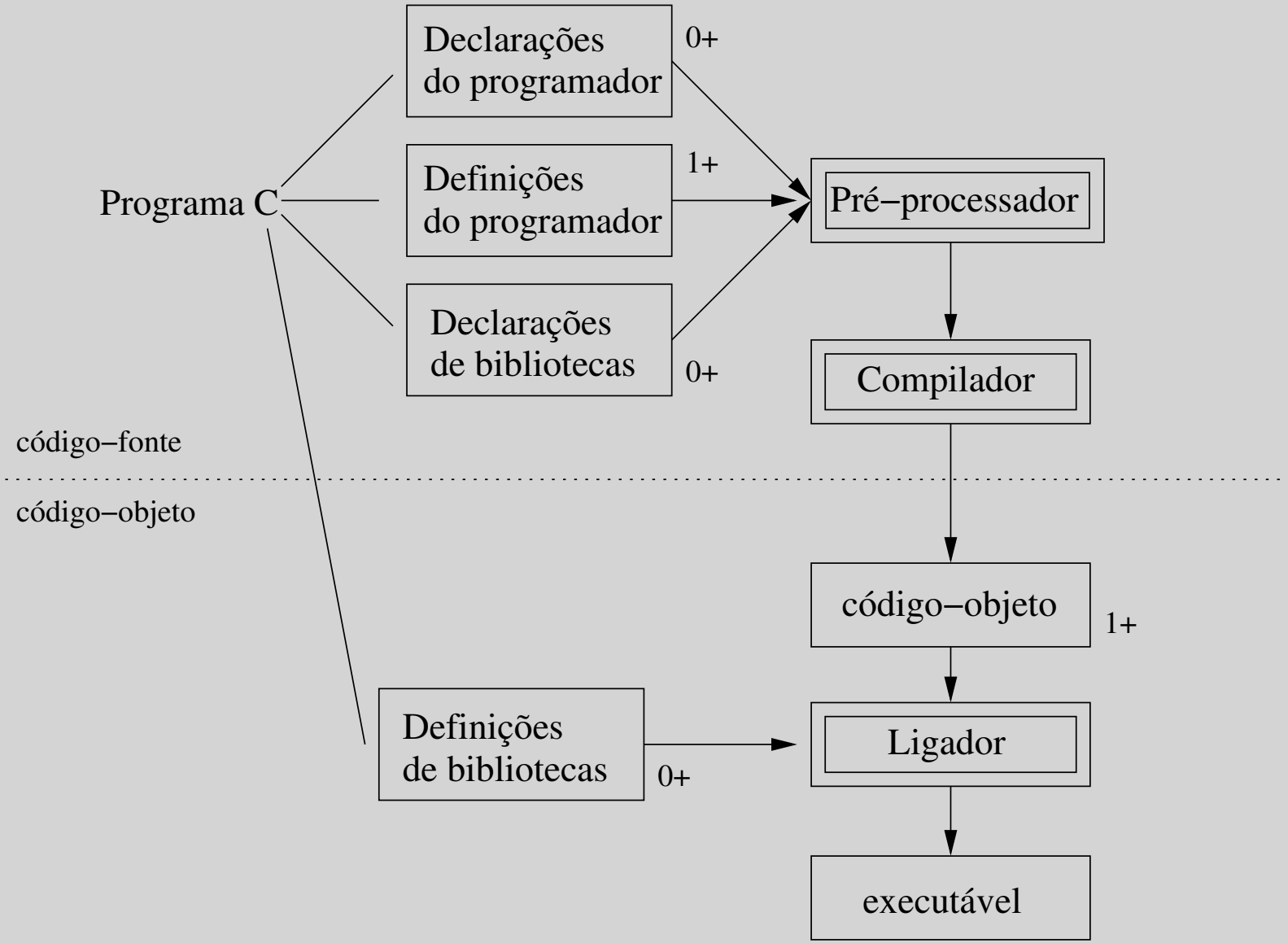
Guilherme P. Telles

IC

13 de abril de 2023

Pré-processador

Compilação


Declarações
do programador
0+
Definições
do programador
1+
Programa C
Declarações
de bibliotecas
0+
Pré−processador
Compilador
código−fonte
código−objeto
código−objeto
1+
Definições
de bibliotecas
0+
Ligador
executável

Diretivas do pré-processador

- Uma linha que começa com # é uma diretiva de pré-processador. (pode ser precedida por espaços).
- A sintaxe é independente da sitaxe da linguagem C.
- O efeito de uma diretiva começa onde ela aparece em um arquivo e continua até o fim do arquivo ou até que a diretiva seja desativada.

#include

- `#include <arquivo>`
  `#include "arquivo"`

- O pré-processador substitui a linha pelo conteúdo do arquivo.
- O arquivo é procurado em diretórios que dependem do sistema.
- A forma com aspas inclui o diretório corrente na busca.
- Não há restrição ao conteúdo do arquivo. Pode conter inclusive outras diretivas, que serão expandidas também.

#define

- `#define identificador [tokens]`

- Uma definição feita com `#define` é chamada de constante simbólica ou macro.
- O pré-processador substitui cada ocorrência do `identificador` por `tokens`, exceto dentro da definição de constantes string.
- `tokens` vai até o fim da linha e pode ser omitido. Neste caso ocorrências do identificado serão substituídas pela cadeia vazia.

- A definição de uma macro pode ser quebrada em várias linhas

```
#define macro linha_1 \
              linha_2 \
               ...
```

#undef

- Faz com que a definição seja desativada daquele ponto em diante.

```
#define pi 3.14
...
#undef pi
```

Constantes simbólicas

- Constantes simbólicas são usadas para melhorar a clareza, legibilidade e facilitar a manutenção.

```
#define MAX 100

int V[MAX], i;

for (i=1; i<MAX; i++)
  V[i] = 1;
```

- Podem ter desempenho melhor.

```
#define TAXA 0.3
...
imposto = preco * TAXA;
```

potencialmente é mais eficiente que

```
float taxa = 0.3;
...
imposto = preco * taxa;
```

pois não é preciso ler taxa da memória.

- Já foi mais importante do que é hoje.

Macros

- `#define id(id,...,id) tokens`

```
#define quad(x) ((x)*(x))

quad(2)    // ((2)*(2))

quad(2+2)    // ((2+2)*(2+2))

quad(quad(2)) // ((((2)*(2)))*(((2)*(2))))
```

- Macros são usadas no lugar de funções para evitar a chamada da função e gerar código mais eficiente (ainda é verdade, mas já foi mais importante).
- São independentes do tipo e permitem definir "funções" genéricas.

```
#define min(x,y) (((x<y))?(x):(y))
```

- Uma definição de macro pode usar tanto macros quanto funções em seu corpo.

```
#define min(a,b,c,d) min(min(a,b),min(c,d))
```

- Problema potencial:

```
#define quad(x) ((x)*(x))

int c=2;
quad(c++);
```

Compilação condicional

- Diretivas que fazem com que o pré-processador não envie trechos do programa para o compilador.

```
#if expressao-integral-constante
#elif expressao-integral-constante
#else
#endif

#ifdef identificador
#endif

#ifndef identificador
#endif
```

- `defined identificador` ou `defined(identificador)` podem ser usados com `#if` para testar se um nome está definido. Retorna 0 ou 1.

- Útil para
  - Setar variáveis que dependem da plataforma.
  - Ativar/desativar código de depuração.
  - Comentar trechos de código já comentados.

```
#ifdef UNIX
const int nice = 10;
#elif defined(MSDOS)
const int nice = 0;
#else
const int nice = 5;
#endif
```

exm-debug-1.c

```c
#include <stdio.h>


int somar(int V[], int n) {

  int soma = 0;

  for (n-=1; n>=0; n--) {
    soma += V[n];
    #ifdef DEBUG
    printf("%d\n",soma);
    #endif
  }

  return soma;
}


int main(void) {

  int A[10], i;

  for (i=0; i<10; i++)
    A[i] = i;

  printf("Soma: %d\n",somar(A,10));
}
```

- A constante DEBUG (ou qualquer outra) pode ser definida no arquivo ou na linha-de-comandos para o GCC.

```
#define DEBUG 1

#ifdef DEBUG
printf(...)
#endif
```

```
gcc -DDEBUG=1 prog.c -o prog
gcc -DDEBUG prog.c -o prog
```

Algumas outras coisas

Qualificadores

Qualificadores de classe de armazenamento

- Modificam a forma de armazenar uma variável.
  - `auto`, `static` e `register`

auto

- Variáveis definidas dentro de uma função ou de um bloco são da classe de armazenamento auto.
- O sistema aloca memória para as variáveis automáticas quando entra em um bloco e libera memória quando sai do bloco.
- Essas variáveis são locais ao bloco.
- O corpo de uma função que tem declarações é um bloco.
- Raramente vemos a palavra auto ser usada em um programa.

extern

- Uma variável definida fora de uma função é da classe de armazenamento extern.
- O sistema aloca memória permanentemente para uma variável extern.
- Essa variável é global a todas as funções declaradas depois dela.

global.c

```c
#include <stdio.h>

int a=1, b=2, c=3;

int f(void) {
  int a, b;

  a = 4;
  b = 5;
  c = 6;

  return a+b+c;
}

int main(void) {
  printf("%d %d %d\n",a,b,c);
  printf("f %d\n",f());
  printf("%d %d %d\n",a,b,c);
}
```

- Quando uma variável é qualificada com extern na declaração, isso indica ao compilador que aquela variável vai ser definida em outro lugar, naquele arquivo ou em outro arquivo.
- Isso também extende a visibilidade da variável para a união de todos os arquivos que compõem o programa.

globalf.c

```c
int a=1, b=2, c=3;

int f(void) {
  int a, b;

  a = 4;
  b = 5;
  c = 6;

  return a+b+c;
}
```

global-main.c

```c
int f(void);

extern int a, b;

int main(void) {

  extern int c;

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

Qualificador `register`

- Define uma variável que na medida do possível vai ser mantida em um registrador da CPU.

```
register int pivot;
```

- Usada para melhorar o desempenho de operações realizadas várias vezes usando um mesmo operando.
- Já foi mais importante no passado.

static

- Aplicado a uma variável local, faz com que a variável não seja recriada a cada chamada da função.
- A variável local é armazenada na área de dados do programa e seu conteúdo preservado durante todas as execuções da função.
- A inicialização é feita apenas uma vez.
- O escopo continua local.

static1.c

```c
#include <stdio.h>

int f() {
  static int n_exec = 0;

  n_exec++;
  printf("%d\n",n_exec);

  return 0;
}


int main(void) {

  for (int i=0; i<10; i++)
    f();
}
```

- Aplicado a uma variável ou função definida fora de uma função (portanto externas), especifica que o escopo da definição é o arquivo.
- É uma forma de definir variáveis que são compartilhadas por um conjunto de funções, mas é privatida do arquivo.

staticf.c

```c
#include <stdio.h>

static void staticf(void) {
  printf("Inside staticf\n");
}
```

static-main.c

```c
#include <stdio.h>

void staticf(void);

int main(void) {
   staticf();
   return 0;
}
```

- `gcc staticf.c static-main.c` não funciona.

Qualificadores de tipo

- Restringem a forma como um identificador pode ser usado.
  - `const`, `volatile`

Qualificador const

- Indica que a variável pode ser inicializada mas não pode ser alterada.

```
const int i=7;
```

- Não é possível usar um apontador para uma constante para alterar o valor dela.

```
const int c = 13;

const int* p;  // apontador para constante int.

p = &c;
*p = 17;  // erro
```

- Mas apontador constante vale:

```
int i = 17;
int* const q = &i;  // apontador constante para int.

*q = 10;
q = q + 1;  // erro
```

Qualificador `volatile`

- Indica que a variável pode ser alterada pelo hardware.

```
extern volatile int clock;
```

- Combinada com `const` indica uma variável que pode ser alterada pelo hardware mas não pelo programador.

```
extern const volatile int clock;
```

void*

`void*`

- Podemos considerar `void*` como um tipo de apontador genérico.
- Conversões entre apontadores de tipos são permitidas quando um dos apontadores é `void*` (e somente nessa situação).
- Útil para a criação de funções que podem receber ou devolver um dado de tipo não especificado.

exm-apont-3.c

```c
#include <stdio.h>

int main(void) {

  float f = 3.14;

  void *v;
  int *pi;
  float *pf;

  pf = &f;
  v = pf;
  pi = v;

  printf("%f %f %d\n", *pf, *((float*) pi), *pi);

  // pi = pf; // nao vale.
}
```

Funções como parâmetros

Funções como parâmetros

- É possível passar uma função como parâmetro para outra.
- O protótipo deve coincidir com a definição do parâmetro.

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

qsorti.c

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

int cmpi(const void *a, const void *b) {
  int* p = (int*) a;
  int* q = (int*) b;

  return *p == *q ? 0 : (*p > *q ? 1 : -1);
}

int cmpi2(const void *a, const void *b) {
  return *((int*)a) == *((int*)b) ?
    0 :
    (*((int*)a)) > (*((int*)b)) ? 1 : -1;
}


int main() {

  int V[] = { 101, 51, 42, 97, 13, -7, 3 };
  int n = 7;

  qsort(V, n, sizeof(int), cmpi);

  for (int i=0; i<n; i++)
    printf("%d ",V[i]);
  printf("\n");
}
```

qsorts.c

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

int cmps(const void *a, const void *b) {
  return strcmp(*(const char**) a, *(const char**) b);
}


int main() {

  char* V[] = { "Peter","Piper","picked","a","peck","of","pickled","peppers" };
  int n = 8;

  qsort(V, n, sizeof(char*), cmps);

  for (int i=0; i<n; i++)
    printf("%s ",V[i]);
  printf("\n");

  return 0;
}
```

errno.h

errno.h

- Define a variável inteira `errno` que é setada por algumas funções das bibliotecas e por chamadas de sistema quando ocorre algum erro para indicar a causa.
- O valor de `errno` só tem significado quando o retorno da função indica erro.
- Quando o programa começa, errno tem valor 0.
- Nenhuma biblioteca ou chamada de função seta o valor de `errno` para 0.
- `perror()` imprime uma mensagem descrevendo o erro, `strerror()` produz uma string descrevendo o erro.

exm-errno-1.c

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


int main(void) {

  errno = 0;
  FILE* fp = fopen("teste000.txt", "r");
  if (!fp) {
    printf("errno=%d\n",errno);
    perror("Uma mensagem");
  }

  return 0;
}
```

limits.h, float.h

limits.h

- Definições de vários limites integrais. P.ex.

```
#define CHAR_BIT   8
#define CHAR_MAX   127
#define CHAR_MIN   -128

#define SHRT_MAX   32767
#define SHRT_MIN   -32768

#define INT_MAX    2147483647
#define INT_MIN    -2147483648
```

e muitas outras.

float.h

- Definições de vários limites fracionários. P.ex.

```
#define DBL_MAX   1.7976931348623157e+308
#define FLT_MAX   3.40282347e+38F
#define LDBL_MAX  1.7976931348623157e+308

#define DBL_MIN   2.2250738585072014e-308
#define FLT_MIN   1.17549435e-38F
#define LDBL_MIN  2.2250738585072014e-308

#define DBL_EPSILON   2.2204460492503131e-16
#define FLT_EPSILON   1.19209290e-07F
#define LDBL_EPSILON  2.2204460492503131e-16
```

e muitas outras.

math.h

- Várias constantes e funções matemáticas estão definidas em math.h. A lista é longa.
- abs(int) é uma exceção, está em stdlib.
- A compilação com gcc precisa incluir -lm

Número variável de parâmetros

Funções com número variável de parâmetros

- C permite declarar funções com número variável de parâmetros, representados pelas reticências na definição da função.

  ```
  nome(par1, par2, ... );
  ```

- Deve haver pelo menos um parâmetro fixo.
- `stdarg.h` define tipos e macros para acesso aos parâmetros.

stdarg.h

- `va_start(va_list p, ultimo_par)`

  Inicia o apontador `p` para o primeiro parâmetro da lista variável, que é o primeiro parâmetro depois de `ultimo_par`. `ultimo_par` é o nome do último parâmetro nomeado no declaração da função. Não fornece uma indicação do tamanho da lista de parâmetros.

- `t va_arg(va_list p, tipo t)`

  Retorna o valor do parâmetro apontado por `p`, convertido para o tipo `t`, e faz `p` apontar para o próximo parâmetro da lista variável. Não retorna uma indicação de que a lista acabou.

- `void va_end(va_list p)`

  Encerra o acesso à lista de parâmetros. **Precisa** ser chamada no final do processamento.

- Para determinar o número de parâmetros na lista `...` há três métodos típicos:
  - Usar uma cadeia de formato:
    `printf("%d %f\n", i, f);`
  - Especificar o número de parâmetros:
    `soma(4, -4, 9, 6, 5);`
  - Especificar um terminador:
    `soma(dummy, -4, 9, 6, 5, INT_MIN);`

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


int soma(int tamanho, ... ) {

  double result = 0;
  int i;

  va_list ap;
  va_start(ap, tamanho);

  for (i=1; i<=tamanho; i++)
    result += va_arg(ap, double);

  va_end(ap);

  return result;
}


int main() {
  printf("%d\n",soma(5, 1.1, 2.1, 3.7, 4.5, 5.4));
  return 0;
}
```

Parâmetros para main

Parâmetros para main

- É possível passar parâmetros diretamente para a função main.
- Os parâmetros são passados na forma de um vetor de strings.

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

  - argc é o número de parâmetros
  - argv é o array de parâmetros
- `argv[0]` é o nome do programa e `argv[argc]` é um apontador nulo.

exm-parmain-1.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {

  int i;

  for (i=0; i<=argc; i++)
    printf("%d %s\n", i, argv[i]);

  return 0;
}
```

Tipos de tamanho fixo

stdint.h

- `stdint.h` define tipos inteiros de tamanho fixo:

```
int8_t  uint8_t
int16_t  uint16_t
int32_t  uint32_t
int64_t  uint64_t
```

- Nem todos precisam estar disponíveis em um certo sistema.
- Define macros para os limites:

  INTN_MIN INTN_MAX UINTN_MAX

- Define tipos inteiros com pelo menons N bits e limites:

  `int_leastN_t  uint_leastN_t`
  INT_LEASTN_MIN INT_LEASTN_MAX UINT_LEASTN_MAX

- Define tipos inteiros rápidos com pelo menons N bits e limites:

  `int_fastN_t uint_fastN_t`
  INT_FASTN_MIN INT_FASTN_MAX UINT_FASTN_MAX

- Define tipos inteiros com número máximo de bits e limites:

  `intmax_t and uintmax_t`
  INTMAX_MIN INTMAX_MAX UINTMAX_MAX

- Opcionalemnte define tipos inteiros grandes o bastante para conter um endereço válido e limites:

  `intptr_t and uintptr_t`
  INTPTR_MIN INTPTR_MAX UINTPTR_MAX

Especificadores para printf e scanf

- `inttypes.h` define constantes para usar na cadeia de formato com os tipos definidos em `stdint.h`.
- São definidos de forma regular.
- Os primeiros três caracteres são

PRI	para usar com saída (printf, fprintf, wprintf etc.)
SCN	para usar com entrada(scanf, fwscanf etc.)

- O quarto caractere é

d	formatação decimal
x	formatação hexadecimal
o	formatação octal
u	formatação unsigned int
i	formatação inteira

- Os demais caracteres são

N	número de bits
PTR	para os tipos PTR
MAX	para os tipos MAX
FAST	para os tipos FAST

```
printf("got %"PRIu64"\n", ua57_get(2,UA));
printf("%3"PRIu32, i);
printf("LCP[%zu]=%"PRIu32" != %zu\n" ,i ,LCP[i] ,l);
```

Arquivos

Arquivos

- De forma abstrata, podemos considerar que um arquivo é uma cadeia de caracteres ou registros que pode crescer à direita, e que está armazenada em alguma memória não-volátil.
- Do ponto de vista do sistema operacional, um arquivo:
  1. tem nome,
  2. tem tamanho,
  3. está em algum sistema de arquivos (localização lógica),
  4. está aberto ou fechado,
  5. pode ser lido, escrito ou atualizado.

- O sistema operacional provê funções básicas para manipulação de arquivos. (v.g. `open`, `creat`, `read` no unix.)
- As linguagens de programação oferecem funções para manipulação de arquivos que usam as funções básicas do SO.
- Normalmente as funções de manipulação de arquivos usam buffers.

Buffers

- Buffers são porções de memória principal usadas para armazenar dados em trânsito entre arquivos e memória.
- Dispositivos não-voláteis são muitas ordens de grandeza mais lentos que a memória principal.
- Buffers aumentam a eficiência das operações, pois permitem transferir dados em blocos e reduzem o número de acessos ao dispositivo.
- O uso de buffers Implica que o conteúdo do arquivo pode estar desatualizado em relação às modificações feitas por programas que escrevem no arquivo.
- Normalmente existe pelo menos um buffer de escrita e pelo menos um de leitura.

Arquivos binários e textuais

- Alguns sistemas operacionais diferenciam entre arquivos textuais e binários, p.ex. o MS-DOS.
  - No MS-DOS o ctrl-Z significa fim-de-arquivo texto. A tentativa de ler um arquivo binário como um arquivo textual pode impedir a leitura de todo o conteúdo do arquivo se o ctrl-Z aparecer antes do fim do arquivo.
- A maioria dos sistemas diferencia entre arquivos textuais e binários apenas logicamente, para tratar adequadamente os fins-de-linha.

Índice

- Um arquivo define um índice para a posição corrente de leitura ou escrita do arquivo.
- O índice normalmente pode ser manipulado diretamente e as funções de arquivo alteram seu valor.
- As funções de C para arquivos sempre realizam uma operação na posição indicada pelo índice.

Arquivos em C

- Os arquivos são manipulados através de funções que escrevem e lêem de **streams**.
- Uma stream é uma variável associada com um arquivo e do tipo `FILE*`.
- O tipo `FILE` é o nome de uma estrutura cujos membros definem o estado de um arquivo.
- `FILE*` também é o tipo das streams `stdin`, `stdout` e `stderr`.

stdio

- Define, dentre outras coisas,
  - O tipo `FILE`,
  - `stdin`, `stdout` e `stderr`,
  - funções para abrir, fechar, ler e escrever,
  - funções para posicionar o índice,
  - constantes.

Abertura

- `FILE* fopen(const char* nome,`
  `const char* modo)`

  Abre o arquivo `nome` em um `modo`. O modo determina que espécie de operações podem ser feitas com o arquivo. O nome deve ser válido no sistema operacional.

  Retorna `NULL` se não for possível abrir o arquivo no modo especificado.

Modos

- `r`: Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto.
- `w`: Abre um arquivo texto para gravação. Se o arquivo não existir, ele é criado. Se já existir, o conteúdo anterior é destruído.
- `a`: Abre um arquivo texto para adicionar dados. Se o arquivo já existir, o índice do arquivo é posicionado no fim dele. Se não existir, ele é criado.

Modos

- `b`: Indica um arquivo binário. Pode ser combinado com `r`, `w` ou `a`. Por exemplo, `rb` para leitura de arquivo binário.
- `+`: Abre o arquivo para atualização. Pode ser combinado com `r`, `w`, `a`, `rb`, `wb` ou `ab`. Por exemplo, `w+b` para escrita e leitura de arquivo binário.

```c
FILE* fp;

fp = fopen("exemplo.txt", "r");
if (!fp)
  printf("Erro na abertura do arquivo.");

fp = fopen("exemplo.bin", "wb");
if (!fp)
  printf("Erro na abertura do arquivo.");
```

Abertura

- ```
  FILE* freopen(const char* nome,
                const char* modo,
                FILE* fp)
  ```

  Fecha o arquivo associado a `fp` e associa `fp` ao arquivo `nome` com o `modo` especificado.

  Retorna `NULL` em caso de falha.

- Usada principalmente para associar stdin, stdout ou stderr com um arquivo.
  ```

Fechamento

- `int fclose(FILE* fp)`

  Grava o buffer associado ao arquivo `fp` em disco e fecha-o.

  Retorna zero no caso de sucesso e EOF caso contrário.

```c
FILE* fp;

fp = fopen("exemplo.txt", "r");
if (!fp)
  printf("Erro na abertura do arquivo.");
fclose(fp);

fp = fopen("exemplo.bin", "wb");
if (!fp)
 printf("Erro na abertura do arquivo.");
fclose(fp);
```

Indicadores de status

- ```
  int feof(FILE *stream);
  int ferror(FILE *stream);
  int fileno(FILE *stream);
  void clearerr(FILE *stream);
  ```

  feof indica se o fim-de-arquivo foi alcançado, ferror indica se houve erro, fileno devolve o descritor do arquivo e clearerr reseta os indicadores de fim-de-arquivo e de erro.

  Outra forma de verificar se o final do arquivo foi atingido é testando o retorno das funções de entrada.
  ```

Entrada não-formatada

- `int fgetc(FILE* fp)`

  Retorna um caractere do arquivo apontado por `fp`.

  Retorna `EOF` se o fim-do-arquivo for encontrado ou se ocorrer um erro de leitura.

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

char c;

FILE* fp = fopen("dados.txt","r");
if (!fp) {
  // houve erro na abertura.
}

errno = 0;
while ((c = fgetc(fp)) != EOF) {
  printf("%c", c);
}

if (ferror(fp) != 0) {
  // houve erro durante a leitura.
}

clearerr(fp);
// Se ferror(fp) for usado de novo, clearerr deve ser chamada.

fclose(fp);
```

- `char* fgets(char* str, int n, FILE* fp)`

  Lê no máximo `n-1` caracteres do arquivo e armazena na memória a partir da posição apontada por `str`, até encontrar `\n` ou o fim-do-arquivo. Se `\n` for lido, ele é acrescentado à string. O caractere `\0` é acrescentado ao fim da string.

  Retorna a cadeia lida ou `NULL` se houver um erro de leitura ou se o fim-de-arquivo é encontrado sem nenhum caractere ter sido lido.

```c
char s[100];

FILE* fp = fopen("dados.txt","r");
if (!fp) {
  // houve erro na abertura.
}


while (fgets(s,100,fp)) != NULL)
  printf("%s",s);
fclose(fp);

if (ferror(fp) != 0) {
  // houve erro durante a leitura.
}
```

Entrada formatada

- `int fscanf(FILE* fp, const char* fmt, ...)`

  Similar a `scanf`, lê texto de `fp` e processa de acordo com as diretrizes na cadeia `fmt`.

  Retorna o número de conversões bem sucedidas ou `EOF` se fim-do-arquivo for alcançado ou se houve um erro.

```c
int i;
float f;

FILE* fp = fopen("dados.txt","r");
if (fp == NULL) {
{
  // houve erro na abertura.
}

while (fscanf(fp,"%d %f\n",&i,&f)) != EOF)
  printf("%d %f\n",i,f);

if (ferror(fp) != 0) {
  // houve erro durante a leitura.
}

fclose(fp);
```

Saída não-formatada

- `int fputc(int c, FILE* fp)`

  Converte `c` para `unsigned char` e escreve no arquivo `fp`.

  Retorna cconvertido para `int` se for bem sucedida ou `EOF` no caso de erro.

```c
char string[100];
int i;

FILE* fp = fopen("dados.txt","w");
if (fp == NULL) {
  // houve erro na abertura.
}

printf("Digite uma string:");
fgets(string,99,stdin);

for (i=0; string[i]; i++) {
  if (fputc(string[i], fp) == EOF) {
    //houve erro de escrita.
  }
}

fclose(fp);
```

Saída não-formatada

- `int* fputs(char* s, FILE* fp)`

  Copia `s` no arquivo `fp`, exceto pelo caractere nulo.

  Retorna um valor não-negativo se for bem sucedida ou `EOF` em caso de erro.

```c
char string[100];
int i;

FILE* fp = fopen("dados.txt","w");
if (fp == NULL) {
  // houve erro na abertura.
}

printf("Digite uma string:");
fgets(string,99,stdin);

if (fputs(string, fp) == EOF) {
  //houve erro de escrita.
}

fclose(fp);
```

Saída formatada

- `int fprintf(FILE* fp, const char* fmt, ...)`

  Similar a `printf`, escreve texto formatado no arquivo `fp`.

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

```c
char string[100];
int i;

FILE* fp = fopen("dados.txt","w");
if (!fp) {
  // houve erro na abertura.
}

if (fp == NULL) { ... }

printf("Digite uma string:");
fgets(string,99,stdin);

if (fprintf(fp, "%s\n", string) < 0) {
  //houve erro de escrita.
}

fclose(fp);
```

Buffers

- `int fflush(FILE* fp)`

  Grava os buffers (no espaço do programa) pendentes em disco.
  Retorna zero se for bem sucedida ou EOF caso contrário.

Índice

- `int fseek(FILE* fp, long n, int pos)`

  Move o índice `n` bytes a partir de pos.

  Retorna zero se for bem sucedida, ou não-zero caso contrário.

  A origem do deslocamento `pos` pode ser:

Nome	Valor	Significado
SEEK_SET	0	Início do arquivo
SEEK_CUR	1	Posição atual
SEEK_END	2	Fim do arquivo

- `long ftell(FILE* fp)`

  Retorna a posição atual do indicador de posição do arquivo ou -1 em caso de erro.

  Para arquivos binários a posição atual é o número de bytes. Para arquivos texto, depende do sistema operacional.

- `void rewind(FILE* fp)`

  Move o indicador de posição para o início do arquivo.

  Retorna -1 em caso de falha.

Entrada de blocos

- `unsigned fread(void* p, int t,`
  `int n, FILE* fp)`

  Lê `n` itens de tamanho `t` do arquivo apontado por `fp` e armazena na posição de memória apontada por `p`.

  Retorna o número de itens lidos ou `EOF` se fim-do-arquivo for alcançado ou se houve um erro.

Saída de blocos

- `unsigned fwrite(void* p, int t,`
  `int n, FILE* fp)`

  Escreve, no arquivo `fp`, `n` itens de tamanho `t` que estão na região da memória apontada por `p`.

  Retorna o número de itens escritos ou um número menor do que deveria ser escrito em caso de erro.

```c
struct aluno {
  char nome[20];
  short idade;
} aluno a1;

FILE* fp = fopen("alunos.dat", "wb");
if (!fp) {
  // houve erro na abertura.
}

strcpy(a1.nome,"Nome do aluno");
a1.idade = 20;

if (fwrite(&a1, sizeof(struct aluno), 1, fp) < 1) {
  //houve erro de escrita.
}

fclose(fp);
```

```c
struct aluno {
  char nome[20];
  short idade;
} aluno t[100];

FILE* fp = fopen("turma2.dat", "rb");
if (!fp) {
  // houve erro na abertura.
}

if (fread(t, sizeof(struct aluno), 100, fp) < 100) {

  if (feof(fp) {
    // fim-de-arquivo.
  }
  else {
    // houve erro durante a leitura.
  }
}

fclose(fp);
```

Truncamento

- `int ftruncate(int fd, off_t n)`

  Trunca o arquivo `fd` deixando-o com `n` bytes. Se o tamanho do arquivo é menor que `n`, os dados além de `n` são perdidos. Se é menor que `n`, o arquivo é aumentado e as novas posições preenchidas com zeros.

  Retorna 0 se for bem sucedido e -1 caso contrário.

  O parâmetro `fd` é o descritor do arquivo, que pode ser obtido da stream usando `fileno`.
  `int fileno(FILE* stream)`

Remoção

- `int remove(const char* nome)`

  Remove o arquivo.

  Retorna 0 se for bem sucedida ou -1 caso contrário.

Mudança de nome

- `int rename(const char* velho,`
  `const char* novo)`

  Troca o nome de um arquivo.

  Se já existir um arquivo com o novo nome, o comportamento depende do sistema operacional.