Ficheiros e Parsing

Laboratórios de Informática III Guião #2-2

Departamento de Informática Universidade do Minho

2024/25

Contents

1	Con	eitos	
	1.1	Leitura e escrita de ficheiros	
		1.1.1 fopen e fclose	
		1.1.2 fgets	
		1.1.3 getline	
		1.1.4 fputs e fprintf	
		1.1.5 fseek	
	1.2	Paths	
		Streams	
	1.4	Ferramentas de parsing simples	
2	Exe	rícios	1

1 Conceitos

1.1 Leitura e escrita de ficheiros

1.1.1 fopen e fclose

Para a leitura e escrita de ficheiros em C, é comum usar a função fopen do módulo <stdio.h>. Esta função tem a seguinte declaração:

```
FILE* fopen(const char* filename, const char* mode);
```

- filename path do ficheiro a abrir. O caminho pode ser absoluto (e.g., /tmp/test.txt) ou relativo (e.g., test.txt).
- mode modo de abertura do ficheiro:
 - r abertura do ficheiro em modo de leitura. O ficheiro deve existir;
 - r+ abertura do ficheiro para leitura e escrita. O ficheiro deve existir;
 - w abertura do ficheiro para escrita. Se o ficheiro não existir, é automaticamente criado. Se o ficheiro existir, é truncado;
 - w+ tal como r+ mas o ficheiro é criado se não existe e truncado se existe;
 - a abertura do ficheiro em modo de escrita para acrescentar dados aos já existentes. O ficheiro é criado se não existir;
 - a+ tal como a mas o ficheiro também pode ser lido.

Os modos podem ainda ter o prefixo b para diferenciar entre formato textual e binário. Contudo, em sistemas Unix essa flag não tem efeito.

• Retorna um apontador para uma stream de um ficheiro.

Para fechar a stream associada a um ficheiro, usamos a função fclose:

```
int fclose(FILE* stream);
```

1.1.2 fgets

A função fgets permite ler caracteres de uma stream para uma string:

```
char* fgets (char* str, int num, FILE* stream);
```

- str string de destino (memória tem que ser previamente alocada);
- num máximo número de caracteres a ler da stream;
- stream stream de onde ler;
- Retorna um apontador para str, ou NULL se tiver chegado ao fim do ficheiro.

A string de destino poderá ser preenchida com menos do que num caracteres caso a stream termine (e.g., o fim de um ficheiro seja alcançado), ou caso a linha termine (\n). Por exemplo, considere o seguinte programa catCaps, que imprime para a consola o conteúdo dum ficheiro, convertendo todas as letras em maiúsculas:

```
#include <stdio.h>
    #include <stdlib.h>
2
    #include <ctype.h> // toupper
    void toUpperStr(char* str) {
        while (*str) {
            *str = toupper(*str);
            str++;
        }
    }
10
11
    int main(int argc, char **argv) {
12
        if (argc < 2) {
13
            fprintf(stderr, "Missing filename.\n");
14
            return 1;
15
16
17
        char* filename = argv[1];
18
        FILE* fp = fopen(filename, "r");
        if (!fp) {
20
            perror("Error"); // imprime "Error: No such file or directory" caso o ficheiro não exista
21
22
        }
23
24
        char str[100];
25
        while(fgets(str, 100, fp)) {
            toUpperStr(str); // notar que str pode não conter a linha toda de uma vez se esta for maior
27
        que 100. O resto da linha estará nos próximos fgets
            printf("%s", str);
28
29
30
        fclose(fp);
31
        return 0;
32
    }
33
```

Também é de salientar a função perror, que imprime para a consola uma mensagem descrevendo o último erro encontrado ao executar uma função do sistema ou da biblioteca standard do C, justamente com um prefixo passado por argumento. No exemplo em cima, esta irá imprimir Error: No such file or directory caso o ficheiro a abrir não exista. É por isso bastante útil para determinar o porquê uma dada função não ter executado com sucesso.

1.1.3 getline

getline é uma função que permite ler um ficheiro de texto linha a linha, tendo a seguinte assinatura: 1

```
ssize_t getline(char** lineptr, size_t* n, FILE* stream);
```

- lineptr apontador para o apontador de uma string onde é guardada a linha lida;
- n tamanho alocado em memória heap à string lineptr;
- stream apontador para o descritor do ficheiro a ler;
- Retorna o número de caracteres lidos, ou -1 se não conseguiu ler mais.

¹Notar que não está disponível em sistemas Windows. Para poder ser usada aí, o programa terá que ser compilado e corrido num ambiente como WSL: https://learn.microsoft.com/en-us/windows/wsl/install. Notar ainda que em Windows, por defeito, muitos editores de texto guardam novas linhas recorrendo a \r\n, enquanto que em Unix usa-se \n. Visto que hoje em dia a maioria dos programas em Windows são compatíveis com \n, deverá ser este o formato utilizado para maximizar compatibilidade.

Não é necessário alocar com antecedência a string onde será guardada a linha. Quando o valor de *lineptr é NULL, a função getline aloca em memória heap (i.e., através de malloc) espaço suficiente para a linha a ser lida, e ajusta o valor de n de acordo com o tamanho alocado. É também possível passar como valor inicial de *lineptr um apontador alocado em memória heap pelo programador, devendo nesse caso o valor de n ser definido de acordo com o espaço alocado. Caso a função necessite de mais espaço do que o alocado, para chamadas consecutivas, é automaticamente aplicada a operação realloc para aumentar o tamanho da string, e o valor de n é novamente ajustado. É devido a esta gestão automática do espaço alocado à string que é necessário um apontador para apontador de string.

Por exemplo, considere o seguinte programa catCaps2, tal como catCaps que processa garantidamente uma linha de cada vez:

1.1.4 fputs e fprintf

A função fputs simplesmente escreve uma string para uma stream. Já a função fprintf funciona de forma semelhante à função printf, recebendo uma string e formatando-a com os argumentos recebidos, contudo escreve o output para a stream recebida. Ambas a funções retornam 0 se executaram com sucesso:

```
int fputs(const char* str, FILE* stream);
int fprintf(FILE* stream, const char* format, ...);
```

Por exemplo, considere o programa log, que adiciona ao ficheiro log.log a string recebida por argumento, bem como a data atual:

```
#include <stdio.h>
    #include <time.h>
   int main(int argc, char **argv) {
        FILE* fp = fopen("log.log", "a");
        time_t t = time(NULL);
        struct tm timeInfo = *localtime(&t);
        fputs("The time is: \n", fp);
        fprintf(fp, "[%d-%02d-%02d %02d:%02d:%02d] - %s\n",
10
            timeInfo.tm_year + 1900, timeInfo.tm_mon + 1, timeInfo.tm_mday,
11
            timeInfo.tm_hour, timeInfo.tm_min, timeInfo.tm_sec, argv[1]);
12
13
14
        fclose(fp);
        return 0;
15
   }
16
```

```
$ ./log "First entry"
$ ./log "Second entry"
$ ./log ""
$ ./log End

$ cat log.log
[2022-10-12 09:17:21] - First entry
[2022-10-12 10:20:03] - Second entry
[2022-10-12 12:02:34] -
[2022-10-12 13:11:26] - (null)
[2022-10-12 18:07:44] - End
```

1.1.5 fseek

A última função sobre processamento de ficheiros que vale a pena referir é a fseek, que permite alterar a posição leitura ou escrita numa stream. Para que seja possível navegar ao longo de uma stream, e.g., uma stream de um ficheiro, o programa mantém a posição atual em que se encontra no ficheiro, sendo escritas e leituras feitas a partir dessa posição. Por omissão, operações de leitura ou escrita avançam sequencialmente a posição dentro de uma stream. A função fseek permite definir uma posição arbitrária para a próxima operação de acesso ao conteúdo dessa stream (acesso "direto"):

```
int fseek(FILE* stream, long int offset, int origin);
```

- stream stream alvo;
- offset número de bytes a mover a posição na stream em relação a origin;
- origin posição de referência para o offset. Existem 3 valores pré-definidos:
 - SEEK_SET aponta para o início da stream;
 - SEEK_CUR aponta para a posição atual da stream;
 - SEEK_END aponta para o fim da stream;
- Retorna 0 se executou com sucesso.

No exemplo seguinte, lemos uma linha e depois colocamos a posição da stream novamente no início dessa linha:

```
read = getline(&line, &len, fp);
printf("%s", line); // hello world\n
fseek(fp, -read, SEEK_CUR);
read = getline(&line, &len, fp); // hello world\n
printf("%s", line);
```

1.2 Paths

Quando lemos ou escrevemos ficheiros, usamos sempre o seu caminho – ou *path* – para os identificar. Um caminho pode ser absoluto ou relativo:

- absoluto refere-se ao mesmo ficheiro independentemente da diretoria atual. Em sistemas Unix começa sempre por / (e.g., /tmp/test.txt), enquanto que em Windows começa pela unidade onde está o ficheiro (e.g., C:\test.txt);
- relativo o ficheiro a que se refere depende da diretoria atual. Notar que quando passamos um caminho relativo a programa C, este será relativo à diretoria onde o programa é chamado, e não a diretoria onde se encontra o ficheiro executável (programa):

```
user@pc:/home/user/work $ ls
prog
user@pc:/home/user/work $ 1s prog
catCaps.c\ catCaps\ test.txt
user@pc:/home/user/work $ cat prog/test.txt
Hello world
This is another line
Lucky numbers: 4 8 15 16 23 42
user@pc:/home/user/work $ ./prog/catCaps test.txt
File not found: test.txt
user@pc:/home/user/work $ ./prog/catCaps prog/test.txt
HELLO WORLD
THIS IS ANOTHER LINE
LUCKY NUMBERS: 4 8 15 16 23 42
user@pc:/home/user/work $ ./prog/catCaps /home/user/prog/test.txt
HELLO WORLD
THIS IS ANOTHER LINE
LUCKY NUMBERS: 4 8 15 16 23 42
```

1.3 Streams

Na secção anterior, foram usadas streams para ler e escrever de um ficheiro, representados por FILE*, nomeadamente nas funções getline, fputs, e fprintf. Estas streams apontavam para ficheiros que foram abertos pelo sistema operativo. Para além das streams de ficheiros, temos ainda 3 streams especiais:

- stdin de onde o programa recebe o input por defeito. Pode ser acedido diretamente, através da consola, ou indiretamente, ao redirecionar um ficheiro (usando <) ou fazendo *pipe* do output de outro programa (usando |);
- stdout para onde o programa envia o output por defeito (e.g., fazendo printf). Pode ser enviado para a consola, redirecionado para um ficheiro (usando >) ou para outro programa (usando |);
- stderr tal como stdout mas reservado tipicamente para mensagens de erro. Pode ser redirecionado para um ficheiro através de 2>.

Considere o programa catCaps3, que faz o mesmo que catCaps2 mas também pode processar o input da consola se não receber um ficheiro por argumento:

```
2
    int main(int argc, char **argv) {
        FILE* fp = NULL;
        if (argc < 2) { // se não recebermos um ficheiro por argumento lemos do stdin (stdio.h)
5
            fp = stdin;
        }
        else {
            char* filename = argv[1];
            fp = fopen(filename, "r");
11
            if (!fp) {
                perror("Error");
12
                return 2;
13
            }
14
        }
15
```

```
17 ...
18 }
```

Para terminar o programa (i.e., enviar EOF (end-of-file)) podemos usar o comando Ctrl-D em Unix ou Ctrl-Z em Windows:

```
ler do stdin
$ ./catCaps3
first line
FIRST LINE
second line
SECOND LINE
end
END
(Ctrl-D)
ler de um ficheiro
$ ./catCaps3 test.txt
HELLO WORLD
THIS IS ANOTHER LINE
LUCKY NUMBERS: 4 8 15 16 23 42
ler do stdin (redirecionado de um ficheiro)
$ ./catCaps3 < test.txt</pre>
HELLO WORLD
THIS IS ANOTHER LINE
LUCKY NUMBERS: 4 8 15 16 23 42
ler do stdin (recebendo o output de outro programa)
$ ls | ./catCaps3
CATCAPS3
CATCAPS3.C
TEST. TXT
```

1.4 Ferramentas de parsing simples

Uma das ferramentas mais simples para o parsing de strings é a função strsep ² (e alternativas como strtok_r ³), que permite iterar sobre as substrings dado um delimitador:

```
char* strsep(char** stringp, const char* delim);
```

Esta simplesmente substitui os delimitadores na string original por NULL, devendo por isso haver o cuidado de fazer uma cópia desta caso seja precisa posteriormente. Por exemplo, considere o programa wc, que conta o número de palavras num ficheiro ou a partir do stdin:

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

int countWords(char* str) {
   int count = 0;
   char* token = strsep(&str, " ");

while (token) {
   if (strlen(token) > 0) { // ignorar espaços consecutivos}
```

 $^{^2\}mbox{N\~{a}}\mbox{o}$ disponível em todos os sistemas, mas é facilmente implementada.

³O equivalente em Windows é strtok₋s .

```
11
                  count++;
             }
12
             token = strsep(&str, " "); // obter próximo token
13
14
15
         return count;
16
    }
17
18
    int main(int argc, char **argv) {
19
         FILE* fp = NULL;
20
21
         // mesmo código que catCaps3
22
23
         while ((read = getline(&line, &len, fp)) != -1) {
24
             line[read-1] = ' \setminus 0'; // ignorar new line ( \setminus n )
25
              count += countWords(line);
26
27
28
         printf("%d\n", count);
29
31
    }
32
```

```
$ cat test.txt
hello world
this is another line
lucky numbers: 4 8 15 16 23 42
$ ./wc test.txt
14
```

Note-se que o uso da função alternativa strtok deve ser evitado.⁴

Outra função para processamento de strings é a sscanf, que pode ser vista como o inverso da printf: em vez de formatar uma string com argumentos, "desformata" uma string nos seus argumentos e retorna o número de placeholders preenchidos:

```
int sscanf(const char* str, const char* format, ...);
```

Considere o programa verboseLog, que lê um ficheiro de log tal como especificado na Secção 1.1.4 e imprime para a consola uma versão estendida:

```
char* intMonthToStr(int monthInt) {
    char* monthStr = malloc(10 * sizeof(char));

switch (monthInt) {
    case 1:
        strcpy(monthStr, "January");
        break;
    ...
}

return monthStr;
}
```

 $^{^4}$ Não usar, visto que não permite execução intercalada, podendo causar erros com bibliotecas externas.

```
void sayEntry(char* str) {
       int year, month, day, hour, minute, second;
17
       char* text = malloc((strlen(str) - 24) * sizeof(char));
18
19
       // %[^{n}] significa "ler até encontrar um \n"
20
       21
   \hookrightarrow text);
22
       char* monthFull = intMonthToStr(month);
23
       printf("On %s %d of %d, at %d:%d:%d, the log says: %s\n", monthFull, day, year, hour, minute,
24
      second, text);
25
       free(monthFull);
       free(text);
27
28
29
   int main(int argc, char **argv) {
30
31
       while ((read = getline(&line, &len, fp)) != -1) {
32
           sayEntry(line);
       }
35
   }
36
```

\$./verboseLog log.log On October 12 of 2022, at 9:17:21, the log says: First entry On October 12 of 2022, at 10:20:3, the log says: Second entry On October 12 of 2022, at 12:2:34, the log says: On October 12 of 2022, at 13:11:26, the log says: (null) On October 12 of 2022, at 18:7:44, the log says: End

Da mesma forma também existe a função scanf, semelhante a sscanf, que lê o input do stdin:

```
int scanf(const char* format, ...);
```

```
printf("Enter a small line (<= 20 chars): ");
char str[20];
scanf("%s", &str);
printf("Size of the input: %d", strlen(str));</pre>
```

Cuidado que scanf não previne *buffer overflows* visto que não recebe um tamanho máximo, ou seja, pode tentar preencher uma string com mais caracteres do que esta consegue suportar. Sendo assim, deve ser evitada.

2 Exercícios

Considere novamente a estrutura Deque do Guião 1. Pretende-se criar um programa que possa ler, de um ficheiro ou do stdin, uma sequência de comandos para modificar uma Deque (comandos são sempre válidos). Considere que a Deque armazena apenas inteiros. Crie um novo módulo command_parser, composto pelos ficheiros command_parser.c e command_parser.h, que irá implementar as seguintes operações:

- PUSH e1 [e2, ...] fazer o push de um ou mais elementos para a Deque. Ex: PUSH 5 10 15. A operação deverá ser aplicada elemento a elemento, e pela ordem em que os elementos são passados, i.e., a começar pelo elemento e1;
- PUSH_FRONT e1 [e2, ...] fazer o push_front de um ou mais elementos para a Deque. A operação deverá ser aplicada elemento a elemento, e pela ordem em que os elementos são passados, i.e., a começar pelo elemento e1.
- POP fazer o pop da Deque. Deverá também imprimir o elemento retirado para a consola (ou a string EMPTY, caso a Deque esteja vazia);
- POP_FRONT fazer o pop_front da Deque . Deverá também imprimir o elemento retirado para a consola (ou a string EMPTY, caso a Deque esteja vazia);
- SIZE imprimir para a consola o tamanho da Deque;
- REVERSE inverter os elementos da Deque;
- PRINT imprimir a Deque para a consola na forma [e1 -> e2 -> ...] . Se a Deque estiver vazia, a função deverá imprimir [] .

Considere ainda a seguinte estrutura auxiliar para armazenar os comandos:

```
typedef struct cmd {
char* command;
int* args; // NULL se não houver
int nargs; // número de argumentos
} Cmd;
```

- Crie a função void processCommand(Deque* deque, Cmd* cmd) no módulo command_parser, que recebe a Deque e um comando e executa-o tal como especificado em cima;
- Crie a função Cmd* parseLine(char* line) no módulo command_parser, que recebe uma linha e constrói um Cmd;
- 3. Crie uma função main, num ficheiro main.c que lê o input de um ficheiro passado por argumento, ou do stdin caso não tenha sido passado nenhum, e chama as funções parseLine e processCommand linha a linha. Para isso, o ficheiro no módulo main.c deverá fazer uso dos módulos command_parser e Deque (através de includes). Certifique-se que toda a memória alocada é libertada antes do programa terminar.
- 4. Crie um ficheiro Makefile para compilar o seu programa.

Exemplo de uma execução:

```
$ cat commands.txt
PRINT
PUSH 10
POP
POP_FRONT
PUSH 20 30
PUSH_FRONT 40 50 60
POP_FRONT
```

```
REVERSE
PUSH 10
SIZE
PRINT

$ ./main commands.txt
[]
10
EMPTY
60
5
[30 -> 20 -> 40 -> 50 -> 10]
```