6

Bibliotecas estandardizadas

Sumário:

Este capítulo objectiva fazer:

- revisão e aprofundamento da linguagem C;
- aquisição de competências básicas na utilização de bibliotecas de C/Unix.

- NOTA: Este capítulo não será alvo de debate ou prática nas aulas. Só serve para revisões de matéria supostamente leccionada em semestres anteriores.

Introdução

Existe uma relação estreita entre a linguagem C e o sistema Unix/Linux . O sistema operativo Unix/Linux está escrito em grande parte em C e a história do SO Unix é também a história da linguagem C.

A utilização das bibliotecas estandardizadas aquando da unificação (linkage) do código objecto requer a inclusão prévia e criteriosa dos seguintes ficheiros (.h) nos ficheiros fonte:

1. **`string.h`** (strings)
2. **`stdlib.h`** (ordenação)
3. **`math.h`** (matemática)
4. **`stdio.h`** (I/O)
5. **`memory.h`** (memória)
6. **`time.h`** (tempo)
7. **`unistd.h`** (acesso a ficheiros e directorias)

A inclusão dum ficheiro (.h) faz-se através da directiva `#include` ao pré-processador.
Por exemplo, `#include <stdio.h>` faz a inclusão dos protótipos das funções de entrada/saída cujo código objecto se encontra na biblioteca **libC**, que é a biblioteca estandardizada da linguagem C.

A biblioteca **libC** é a única que não precisa ser explicitamente especificada durante a compilação. O código objecto de **libC** é automaticamente unificado com o código objecto de qualquer programa escrito em C. Normalmente o código está contido numa biblioteca estática **libc.a** e outro dinâmica **libc.so**

A utilização de qualquer outra função pertencente a outra biblioteca torna obrigatória a especificação da respectiva biblioteca no acto da compilação. Por exemplo, se um programa chamado `myp.c` usa funções matemáticas, então há que fazer:

- a inclusão do ficheiro `math.h` neste programa através da directiva *#include* , i.e. a linha de código `#include <math.h>` tem de ser escrita no início de `myp.c`, e depois
- explicitar a biblioteca libm no comando de compilação, i.e.
  $ cc myp.c –o myp –lm
  onde **–lm** é uma indicação para o unificador (linker) fazer a unificação do código objecto do programa com a bliblioteca **libm**.

Existe um manual on-line para as funções da linguagem C. Por exemplo, para saber a informação disponível sobre a função `rand`, só é necessário escrever o seguinte na linha de comando do Bash :

$ man rand

Algumas funções existe duas ou mais vezes nas paginas de manual por exemplo write (bash shell) e write (low-level I/O da linguagem C). As vezes é necessário especificar o manual que pretende pesquisar. Compare os seguintes por exemplo

$ man write (bash shell)
$ man 2 write (low level sytem calls)
$ man 3 fwrite (c standard library) será equivalente a **man fwrite** porque **fwrite** ocorre apenas uma vez nas paginas manual.

Ou mesmo acontece com printf

$ man printf (bash shell)
$ man 2 printf (não existe)
$ man 3 printf (c standard library

Strings <string.h>

O código objecto das funções declaradas em `string.h` encontra-se na biblioteca **libC**.

Uma string é uma sequência de zero ou mais caracteres que termina com o carácter NULL ('\0'). Uma string é representado por um vector (array) unidimensional de caracteres, um apontador para uma zona de memória que contém caracteres ASCII.

Funções básicas:

- Compara `string1` com `string2`:
  `int strcmp(const char *string1, const char *string2)`
- Compara n caracteres do `string1` com `string2`:
  `int strncmp(const char *string1, const char *string2, int n)`
- Copia `string2` para `string1`:
  `char *strcpy(const char *string1, const char *string2)`
- Devolve mensagem de erro correspondente ao número `errnum`:
  `char *strerror(int errnum)`
- Determina o comprimento duma string:
  `int strlen(const char *string)`
- Concatena n caracteres da `string2` à `string1`:
  `char *strncat(const char *string1, char *string2, size_t n)`

Funções de pesquisa:

- Determina a primeira ocorrência do caráter `c` na string:
  `char * strchr(const char *string, int c)`
- Determina a última ocorrência do caráter `c` na string:
  `char * strrchr(const char *string, int c)`
- Localiza a primeira ocorrência da substring `s2` na string `s1`:
  `char *strstr(const char *s1, const char *s2)`

Funções de Repartição

- String Tokenizer . Divide uma string numa sequencia de sub-strings denominados tokens. A divisão é feita usando qualquer dos caracteres dos delimiters

  ```
  char * strtok ( char * str, const char * delimiters );
  ```

```
Exemplo
char *str1 = "ola esta tudo bem?";
char *t1;
for   ( t1 = strtok(str1," ");  t1 != NULL;  t1 = strtok(NULL, " ") )
      printf("token:  %s\n",t1);
```

Explicação do ciclo *for* :

- Inicialização chamada da função strtok `()` e carregamento com o string `str1`
- Terminação do ciclo quando t1 é igual a `NULL`
- Continuação : os *tokens* do str1 são atribuídos ao apontador t1 com uma chamada a strtok() com o primeiro argumento NULL

Exercício 6.1 Faça uma listagem do ficheiro <string.h>

Exercício 6.2 Imprima o ficheiro <string.h>

Exercício 6.3 Faça um programa que:

- leia duas strings x e y;
- faça a saída dos respectivos comprimentos;
- faça a saída da string "x está dentro de y", no caso de x ser uma sub-string de y;
- faça a saída da sub-string de y que antecede x, no caso de x ser uma sub-string de y.

Exemplo:

Input:	x=abc
	y=lisboaabc123
Output:	comprimento x=4 comprimento y=12
	x está dentro de y sub-string que anteced: lisboa

Ordenação <stdlib.h>

O código objecto das funções declaradas em `stdlib.h` encontra-se na biblioteca **libC**, a qual implementa algoritmos de pesquisa e ordenação tais como, por exemplo, o *quicksort* e o *bublesort*.

O algoritmo quicksort tem o seguinte protótipo:

```
void qsort(void *base, size_t nelem, size_t width,
                int (_USERENTRY *fcmp)(const void *elem1, const void *elem2));
```

onde:

- base aponta para o elemento base (0-ésimo) da tabela a ordenar;
- nelem é o número de elementos da tabela;
- width é o tamanho em bytes de cada elemento da tabela;
- fcmp é a função de comparação que é obrigatoriamente usada com a convenção _USERENTRY. Fcmp aceita dois argumentos, elem1 e elem2, cada um dos quais é um aponatdor para um elemento da tabela. A função de comparação compara os dois elementos apontados e devolve um inteiro como resultado, a saber:

*elem1 < *elem2	⇒	fcmp devolve um inteiro < 0
*elem1 == *elem2	⇒	fcmp devolve 0
*elem1 > *elem2	⇒	fcmp devolve um inteiro > 0

Exemplo da Utilização:
Seja x um vector de 30 inteiros e cmp uma função de comparisão, o simples <. Para chamar o qsort para ordenar os dez elementos do vector entre as posições 10..20
int x [30]
int cmp(void *a, void *b) { `return ( (int *)a < (int *)b ); }`

```
qsort( &x[10], 10, sizeof(int), cmp);
```

Exemplo 6.1:

Escreva e execute o pequeno programa a seguinte para mostrar o funcionamento do *quicksort*.

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

int sort_function(const void *a, const void *b);
char list[5][4]={"cat", "car", "cab", "cap", "can"};


  int main(void)
  {
     int x;

     for (x=0; x<5; x++) printf("%s\n", list[x]);          /* antes */

     qsort((void*)list,
            5,
            sizeof(list[0]),
            sort_function
          );

     for (x=0; x<5; x++) printf("%s\n", list[x]);         /* depois */

     return 0;
  }

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

Exercício 6.4 Faça um programa que:

- leia dinamicamente um vector de *n* inteiros;
- preencha este vector com números aleatórios, para o que deve usar a função `rand`;
- ordene o vector usando a função `qsort`;
- e, finalmente, mostre o vector ordenado no écran.

Matemática <math.h>

A utilização de qualquer função matemática da biblioteca **libm** num dado programa `mpg.c` requer a declaração da directiva `#include <math.h>` em `mpg.c`.

Além disso, é necessário incluir explicitamente a biblioteca **libm** na compilação do programa `mpg.c` de modo a que a unificação (linkage) das funções matemáticas usadas em `mpg.o` e o seu código existente em **libm** se concretize. Isto é, na linha de comando do Unix deve escrever-se o seguinte:

$ cc –o mpg mpg.c –lm

É absolutamente essencial não esquecer a inclusão do ficheiro `<math.h>` no programa `mpg.c`. Caso contrário, o compilador não servirá de grande ajuda.

Algumas funções:

- Calcula o coseno dum ângulo em radianos:
  `double cos(double x)`
- Calcula o ângulo do coseno de x:
  `double acos(double x)`
- Calcula o ângulo da tangente de y/x:
  `double atan2(double y, double x)`
- Calcula o valor inteiro mais pequeno que excede x:
  `double ceil(double x)`

Algumas constantes pré-definidas:

HUGE	O valor máximo dum número de vírgula flutuante com precisão simples.
M_E	A base do logaritmo natural (e).
M_LOG2E	O logaritmo de base 2 de e.
M_LOG10E	O logaritmo de base 10 de e.
M_LN2	O logaritmo natural de 2.
M_LN10	O logaritmo natural de 10.
M_PI	Valor de π.

Exercício 6.5
Edite, compile e execute um programa que utilize algumas funções e constantes matemáticas.

I/O de alto-nível <stdio.h>

Bibliografia utilizada:
Cap.12 (P. Darnell e P. Margolis. *C: a software engineering approach*)
Cap. 5 (W. Stevens. *Advanced Programming in the Unix environment*)
Caps. 7, 13 (B. Forouzan, R. Gilberg. *Computer Science: a structured programming approach using C*)

As funções descritas nesta subsecção são conhecidas como funções estandardizadas (ou de alto-nível) de entrada/saída. São funções de entrada/saída com entrepósito (ou *buffer*). Isto significa que a escrita/leitura é feita primariamente para/do entrepósito (ou *buffer*), e só depois ocorre a escrita/leitura para/a partir de um ficheiro a partir/para o entrepósito (ou buffer).

A entrada/saída directa (ou de baixo nível) de dados para/de um ficheiro é feita por funções de entrada/saída sem entrepósito. Cada função directa de entrada/saída (e.g. `read` e `write`) invoca uma chamada ao sistema.

O ficheiro <stdio.h>

A utilização de qualquer função estandardizada de I/O obriga à inclusão do ficheiro `stdio.h` no ficheiro (.c) onde a função está a ser usada.

O ficheiro `stdio.h` contém:

- Os cabeçalhos (protótipos ou declarações) de todas as funções estandardizadas de I/O.
- A declaração da estrutura FILE.
- Várias macros, entre as quais se conta:
  a) `stdin` (dispositivo estandardizado de entrada)
  b) `stout` (dispositivo estandardizado de saída)
  c) `stderr` (dispositivo estandardizado de erro)
  d) `EOF` (marcador de end-of-file)

Entreposição (*buffering*)

Um entrepósito (ou *buffer*) é uma área de memória temporária para ajudar a transferência de dados entre dispositivos ou programas que operam a diferentes velocidades. Além disso, qualquer entrepósito usado pelas funções estandardizada de I/O permite-nos usar um número mínimo de chamadas `read` e `write`.

Ou seja, um entrepósito não é mais do que uma área de memória onde os dados são armazenados temporariamente antes de serem enviados para o seu destino. Entreposição é um mecanismo mais eficiente de transferência de dados porque permite a um sistema operativo minimizar o número de acessos ao dispositivos de I/O.

De facto, é estremamente importante reduzir o mais possível o número de operações físicas de escrita e leitura, dado que, por contraposição à memória, os dispositivos de memória secundária (e.g. discos rígidos e cassetes de fita magnética) são bastante lentos.

Todos os sistemas operativos usam entrepósitos (buffers) para ler/escrever de/para dispositivos de I/O. Isto significa que qualquer sistema operativo acede a dispositivos de I/O em fatias (*chunks*) de tamanho fixo, chamados blocos (*blocks*). Normalmente, um bloco tem 512 ou 1024 octetos (*bytes*). Portanto, mesmo se nós quisermos ler só um carácter a partir dum ficheiro, o sistema operativo lê o bloco inteiro no qual o carácter se encontra. Isto parece não ser muito eficiente, mas imagine-se que se pretendia ler 1000 caracteres dum ficheiro. No caso da I/O sem entrepósito, o sistema terá de realizar 1000 operações de procura e leitura. Em contrapartida, com I/O com entrepósito, o sistema lê um bloco inteiro para a memória, e depois procura cada carácter em memória se for necessário. Isto poupa 999 operações de I/O.

Streams

A linguagem C não faz qualquer distinção entre dispositivos tais como um terminal (monitor e teclado) ou um controlador de fita magnética e ficheiros lógicos no disco rígido, represente tudo como um ficheiro (ver directório /proc num sistema Linux)

A independência relativamente ao dispositivo de I/O, portanto a virtualização do I/O, consegue-se usando *streams*. Cada *stream* está associado a um ficheiro ou dispositivo. Um *stream* consiste numa sequência ordenada de bytes. Um *stream* pode ser visto como um array unidimensional de caracteres. Ler/escrever de/para um ficheiro ou dispositivo faz-se lendo/escrevendo de/para a corrente que lhe está associada.

Para realizar operações estandardizadas de I/O, há que associar um *stream* a um ficheiro ou a um dispositivo. Isto faz-se através da declaração dum ponteiro para um estrutura do tipo FILE. A estrutura FILE contém vários campos:

- nome do ficheiro,
- descritor do ficheiro
- modo de acesso,
- bloco de memoria (buffer)
- ponteiro para o próximo carácter na corrente.

Streams estandardizados e redireccionamentos

Há três *streams* que são abertos automaticamente para qualquer programa:

- `stdin` (entrada estandardizada, que é por defeito o teclado)
- `stout` (saída estandardizada, que é por defeito o écran)
- `stderr`(saída estandardizada de erros, que é por defeito o écran)

Os *streams* estandardizados podem ser redireccionados para outros ficheiros ou dispositivos.
Há duas formas de fazer um redireccionamento dum *stream*:

- através de opções em comandos Unix,
- através dos operadores <, <<, >, >> em comandos Unix.

Por exemplo:

a) `comando > fich` redirecciona a saída estandardizada para o ficheiro `fich`
b) `comando >> fich` redirecciona e concatena a saída estandardizada ao o ficheiro `fich`
c) `comando >& fich` redirecciona a saída estandardizada de erros para o ficheiro `fich`
d) `comando >>& fich` redirecciona e concatena a saída estandardizada de erros ao o ficheiro `fich`
e) `comando < fich` redirecciona a entrada estandardizada para o ficheiro `fich`

Funções básicas de I/O:

- Lê um carácter do `stdin`:
```
int getchar(void)
```

- Escreve um carácter para o `stdout`:
```
int putchar(char ch)
```

- Lê uma string de caracteres (terminada por um newline) a partir do `stdin` e coloca-a em s, substituindo o newline por um carácter nulo (\0):
```
char *gets(char *s)
```

- Funções de saída e entrada formatadas:
```
int printf(const char *format,…)
int scanf(const char *format,…)
```

Funções de I/O para/de ficheiros:

Para usar um ficheiro há que primeiro abri-lo com a função:

- Abre um ficheiro:
  `FILE *fopen(char *name, char *mode)`

O argumento `name` é o nome do ficheiro em disco que se pretende aceder. O argumento `mode` indica o tipo de acesso ao ficheiro.

Há dois conjuntos de modos de acesso. O primeiro serve para *streams* de texto, ao passo que o segundo é adequado para *streams* binárias.

Os modos básicos para *streams* textuais são os seguintes:
- "r" (read) leitura
- "w" (write) escrita
- "a" (append) concatenação

Os modos binários são exactamente os mesmos, excepto que um b tem de ser concatenado à direita do nome do modo. Temos assim:
- "rb" (read) leitura
- "wb" (write) escrita
- "ab" (append) concatenação

Exemplo 6.2:

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

int main(void)
{
   FILE *stream;

   stream = fopen("test.txt","r");
   if (stream == NULL)
      printf("Erro na abertura do ficheiro test.txt\n");
   exit(1);
}
```

Outras funções de I/O para/de ficheiros:

- **fclose()** Fecha um *stream* associado a um ficheiro;
- **fgetc()** Lê um carácter a partir dum *stream*;
- **fgets()** Lê uma string a partir dum *stream*;
- **fputc()** Escreve um carácter para um *stream*;
- **fputs()** Escreve uma *string* para um *stream*;
- **fscanf()** O mesmo que `scanf`(), mas agora os dados são lidos a partir dum dado ficheiro;
- **fprintf()** O mesmo que `printf`(), mas agora os dados são escritos para um dado ficheiro;
- **fflush()** Transcreve os dados para o ficheiro associado com um dado *stream*, esvaziando-o;
- **fread()** Lê um bloco de dados binários a partir dum *stream*.

Inquirições ao estado duma stream

Existem algumas funções para saber o estado dum ficheiro, nomeadamente:

- Devolve o valor `true` se a corrente está na posição EOF:
  `int feof(FILE *stream)`
- Devolve o valor `true` se um erro ocorreu:
  `int ferror(FILE *stream)`
- Limpa a indicação de erro que tenha ocorrido anteriormente:
  `int clearerr(FILE *stream)`
- Devolve o descritor inteiro do ficheiro associado com o `stream`:
  `int fileno(FILE *stream)`

Granularidade da I/O :

Uma vez aberto um *stream*, pode escolher-se entre três tipos diferentes de I/O sem formatação:

- **I/O carácter-a-carácter**. Um carácter de cada vez é lido ou escrito, tal que as funções estandardizadas de I/O manipulam a entreposição (ou *buffering*), no caso de o *stream* ser entreposto (ou *buffered*).
- **I/O linha-a-linha**. Se quisermos ler ou escrever uma linha de cada vez, então usamos as funções `fgets` e `fputs`. Cada linha termina com um carácter `newline`. Além disso, temos que especificar o comprimento máximo da linha quando a função `fgets` é chamada.
- **I/O directa**. É suportada pelas funções `fread` e `fwrite`. Uma operação de I/O directa permite ler ou escrever um conjunto de objectos, cada um dos quais tem um tamanho que deve ser especificado. Estas duas funções são muitas vezes aplicadas a ficheiros binários, tal que cada operação de I/O realiza a leitura ou a escrita duma estrutura.

Memória <memory.h>

As operações de memória estão implementadas através de funções cujos cabeçalhos ou protótipos estão declarados no ficheiro string.h. É, no entanto, conveniente considerar o ficheiro memory.h onde tradicionalmente as funções seguintes foram especificadas..

Algumas funções de memória:

- Procura :
  `void *memchr(void *s, int c, size_t n)`
- Compara dois blocos de memória com n bytes de tamanho :
  `int memcmp(void *s1, void *s2, size_t n)`
- Copia um bloco de memória com n bytes de tamanho :
  `Void *memcpy(void *dest, void *src, size_t n)`
- Move um bloco de memória com n bytes de tamanho :
  `int memmove(void *dest, void *src, size_t n)`
- Inicializa um bloco de memória de n bytes com o carácter c :
  `int memset(void *s, int c, size_t n)`

Exemplo 6.3:
O seguinte exemplo ilustra a utilização da função memcpy.

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

int x[]={5,4,3,2,1};
int y[10]={1,2,3,4,5,6,7,8,9,10};
int *z;
int i;

main()
{
    for (i=0; i<10; i++)                                    /*  a situacao ANTES  */
            printf("%d",y[i]);
    putchar('\n');
    z=(int *)memcpy(y+1, x, sizeof(x));

    z=y;
    for (i=0; i<10; i++)                                    /*  a situacao DEPOIS  */
            printf("%d",*z++);
    putchar('\n');
    for (i=0; i<10; i++)
            printf("%d",y[i]);
    putchar('\n');
    return 0;
}
```

Exercício 6.5:
Escreva uma função que inverta o conteúdo dum bloco de n octetos de memória. Isto significa existe uma troca de valores entre o 0-ésimo e o n-ésimo octeto, entre o 1-ésimo e o (n-1)-ésimo octetos, etc. De seguida, escreva um programa que usa aquela função para inverter uma cadeia de caracteres.

Temporização <time.h>

As funções de temporização são úteis por várias razões, nomeadamente para saber a data e a hora correntes, medir o tempo de execução duma operação, inicializar geradores de números aleatórios, etc.

Funções básicas:

- Devolve o tempo em segundos desde 00:00:00 GMT, Jan 1, 1970 :
  `time_t time(time_t *tloc)`

- Preenche uma estrutura apontada por tp de acordo com a definição em `<sys/timeb.h>` :
  `int ftime(struct timeb *tp)`

- Converte um long integer referente ao tempo do relógio numa string de 26 caracteres na forma Sun Sep 16 01:03:52 1999 :
  `ctime()`

`time_t` é provavelmente um `typedef` para um unsigned `long int`. A definição encontra-se no ficheiro <time.h>

A estrutura `timeb` tem 4 campos:

```
struct timeb {
    time_t time;                                                    /*  em segundos  */
    unsigned short millitm;     /*  ate 1000 milisegundos para intervalos mais precisos  */
    short timezone;                                    /*  em minutos a oeste de Greenwich  */
    short dstflag;         /*  se flag nao-nula, indica que a mudança de hora é aplicável  */
}
```

Exemplo 6.4:
Um programa para medir o tempo duma operação.

```
#include <stdio.h>
#include <sys/types..h>
#include <time.h>

main()
{
    long i;
    time_t t1, t2;
    time(&t1);
    for (i=1; i<=100000; i++)
                printf("%ld %ld %ld %f\n",i,i*i, i*i*i, (float)i*i*i);
    time(&t2);
    printf("Tempo para 1000 quadrados, cubos e duplos quadrados=%ld segundos\n",(long)(t2-t1));
}
```

Exemplo 6.5:
Um programa para inicializar um gerador de números aleatórios.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types..h>
#include <time.h>

main()
{
    int i;
    time_t  t1=0;
    printf("5 numeros aleatorios sem seed\n", (int)t1);

    for (i=0;i<5;++i)
      printf("%d",rand());
    printf("\n\n");

    time(&t1);
    srand((long)t1);  /* use tempo em segundos para activar seed */
    printf("5 numeros aleatorios (Seed = %d):\n", (int)t1);
    for (i=0;i<5;++i)
      printf("%d",rand());

    printf("\n\n Agora corre o programa outra vez\n");
}
```

Exercício 6.6:
Altere o penúltimo programa (6.5) para determinar o tempo utilizado pela CPU (veja `clock` no manual on-line).

Exercício 6.7:
Veja também o ficheiro time.h. Escreva um programa que devolva a data e hora actuais, assim como o tempo que falta até ao início do próximo fim-de-semana (sexta-feira, 19.00).