LabSO 2022

Laboratorio Sistemi Operativi - A.A. 2021-2022

dr. Andrea Naimoli	Informatica LT andrea.naimoli@unitn.it
dr. Michele Grisafi	Ingegneria informatica, delle comunicazioni ed elettronica (LT) michele.grisafi@unitn.it

C - introduzione

Perchè C?

- Struttura minimale
- Poche parole chiave (con i suoi pro e contro!)
- Unix compliant, alla base di Unix, nato per scrivere Unix
- Organizzato a passi, con sorgente, file intermedi ed eseguibile finale
- Disponibilità di librerie conosciute e standard
- Efficiente perché di basso livello
- Pieno controllo del programma e delle sue risorse
- Ottimo per interagire con il sistema operativo

Direttive e istruzioni fondamentali

- `#include … / #define … /`
- `char / int / … / enum (v. esempio seguente)`
- `for ( initialization ; test; increment ) { … ; }`
- `break / continue`
- `switch (expression){ case val: … [break;] [default: …] }`
- `while (expression) { … } / do { … } while (expression)`
- `if (expression) { … } [else { … }]`
- `struct / union`

(consultare una documentazione standard ed esercitarsi)

Tipi e Casting

C è un linguaggio debolmente tipizzato che utilizza 8 tipi fondamentali. È possibile fare il casting tra tipi differenti:

```
float a = 3.5;
int b = (int)a;
```

La grandezza delle variabili è **dipendente dall' architettura di riferimento** i valori massimi per ogni tipo cambiano a seconda se la variabile è *signed* o *unsigned*.

- void (0 byte)
- char (1 byte)
- short (2 bytes)
- int (4 bytes)
- float (4 bytes)
- long (8 bytes)
- double (8 bytes)
- long double (8 bytes)

NB: non esiste il tipo boolean, ma viene spesso emulato con un char.

sizeof *(operatore)*

`sizeof (type) / sizeof expression`

Si tratta di un operatore che elabora il tipo passato come argomento (tra parentesi) o quello dell'espressione e restituisce il numero di bytes occupati in memoria.

```c
#include <stdio.h>
void main() {
  int x = 10;
  printf("variable x      : %lu\n", sizeof x);
  printf("expression 1/2  : %lu\n", sizeof 1/2);
  printf("int type        : %lu\n", sizeof(int));
  printf("char type       : %lu\n", sizeof(char));
  printf("float type      : %lu\n", sizeof(float));
  printf("double type     : %lu\n", sizeof(double));
}
```

Puntatori di variabili

C si evolve attorno all'uso di puntatori, ovvero degli alias per zone di memorie condivise tra diverse variabili/funzioni. L'uso di puntatori è abilitato da due operatori: '*' ed '&'.

'*' ha significati diversi a seconda se usato in una dichiarazione o in un'assegnazione:

`int *punt;` → crea un puntatore ad intero

`int valore = *(punt);` → ottiene valore puntato

'&' ottiene l'indirizzo di memoria in cui è collocata una certa variabile.

`long whereIsValore = &valore;`

Esempi:

```
float   pie   =   3.4;
float  *pPie  =  &pie;
pie       *=       2;
float pie4 = *pPie * 2;

char  *array  =  "str";
*array     =      's';
array[1]   =      't';
*(array + 2) = 'r';
```

Puntatori di variabili

```
int      i      =      42
int   *   punt   =   &i;
int b = *(punt);
```

Tipo	Nome	Valore	Indirizzo
int	i	42	0xaaaabbbb
int *	punt	0xaaaabbbb	0xccccdddd
int	b	42	0x11112222

```
i = 20;
```

Nome	Valore
i	**20**
punt	0xaaaabbbb
b	**?**

Puntatori di funzioni

```c
#include <stdio.h>
float xdiv(float a, float b) {
  return a/b;
}
float xmul(float a, float b) {
  return a*b;
}
int main() {
  float (*punt)(float,float);
  punt = xdiv;
  float res = punt(10,10);
  punt = &xmul; //& opzionale
  res = (*punt)(10,10);
  printf("%f\n", res);
  return;
}
```

C consente anche di creare dei puntatori a delle funzioni: puntatori che possono contenere l'indirizzo di funzioni differenti.

Sintassi simile ma diversa!

```c
float (*punt)(float,float);
```

```
ret_type (* pntName)(argType,argType,...)
```

main.c

- A parte casi particolari (es. sviluppo moduli per kernel) l'applicazione deve avere una funzione "`main`" che è utilizzata come punto di ingresso.
- Il valore di ritorno è **int**, un intero che rappresenta il codice di uscita dell'applicazione (variabile `$?` in bash) ed è 0 di default se omesso. Può essere usato anche void, ma non è standard.
- Quando la funzione è invocata riceve normalmente in input il numero di argomenti (`int argc`), con incluso il nome dell'eseguibile, e la lista degli argomenti come "vettore di stringhe" (`char * argv[]`) (*)

(*) in C una stringa è in effetti un vettore di caratteri, quindi un vettore di stringhe è un vettore di vettori di caratteri, inoltre i vettori in C sono sostanzialmente puntatori (al primo elemento del vettore) → lista di argomenti spesso indicata con "`char ** argv`"
Utile riferimento generale: `https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`

main.c

```c
#include <stdlib.h>
#include <stdio.h>
#define DIVIDENDO 3

int division(int var1, int var2, int * result){
    *result = var1/var2;
    return 0;
}

int main(int argc, char * argv[]){
    float var1 = atof(argv[1]);
    float result = 0;
    division((int)var1,DIVIDENDO,(int *)&result);
    printf("%d \n",(int)result);
    return;
}
```

Esecuzione - esempio

Compilazione:

```
gcc main.c -o main
```

Esecuzione:

```
./main arg1 arg2
```

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("%d\n", argc);
  printf("%s\n", argv[0]);
  return 0;
}
```

In output si ha “3” (numero argomenti incluso il file eseguito) e “./main” (primo degli argomenti).

In generale quindi `argc` è sempre maggiore di zero.

Direttive

Il compilatore, nella fase di preprocessing, elabora tutte le direttive presenti nel sorgente. Ogni direttiva viene introdotta con '#' e può essere di vari tipi:

`#include <`*`lib`*`>`	copia il contenuto del file *lib* (cercando nelle cartelle delle librerie) nel file corrente
`#include "`*`lib`*`"`	come sopra ma cerca prima nella cartella corrente
`#define `*`VAR VAL`	crea una costante VAR con il contenuto VAL, e sostituisce ogni occorrenza di VAR con VAL
`#define MUL(A,B) A*B`	dichiara una funzione con parametri A e B. Queste funzioni hanno una sintassi limitata!
`#ifdef, #ifndef, #if, #else, #endif`	rende l'inclusione di parte di codice dipendente da una condizione.

Macro possono essere passate a GCC con `-D NAME=VALUE`

Librerie standard

Librerie possono essere usate attraverso la direttiva `#include`.
Tra le più importanti vi sono:

- stdio.h: FILE, EOF, stderr, stdin, stdout, fclose(), etc...
- stdlib.h: atof(), atoi(), malloc(), free(), exit(), system(), rand(), etc...
- string.h: memset(), memcpy(), strncat(), strcmp(), strlen(), etc...
- math.h: sin(), cos(), sqrt(), floor(), etc...
- unistd.h: STDOUT_FILENO, read(), write(), fork(), pipe(), etc...
- fcntl.h: creat(), open(), etc...

...e ce ne sono molte altre.

Direttive - esempi

```c
#include <stdio.h>
#define ITER 5
#define POW(A) A*A

int main(int argc, char **argv) {
#ifdef DEBUG
    printf("%d\n", argc);
    printf("%s\n", argv[0]);
#endif
    int res = 1;
    for (int i = 0; i < ITER; i++){
        res *= POW(argc);
    }
    return res;
}
```

```
gcc main.c -o main.out -D DEBUG=0
gcc main.c -o main.out -D DEBUG=1
```

```
./main.out 1 2 3 4
```

Stesso risultato!

Structs e Unions

Structs permettono di aggregare diverse variabili, mentre le unions permettono di creare dei tipi generici che possono ospitare uno di vari tipi specificati.

```
struct Books{
    char author[50];
    char title[50];
    int bookID;
} book1, book2;

struct Books book3 =
{“Rowling”,”Harry Potter”,2};
strcpy(book1.title,“Moby Dick”);
book2.bookID = 3;
```

```
union Result{
    int intero;
    float decimale;
} result1, result2;

union Result result3;
result3.intero = 22;
result3.decimale = 11.5;
```

Typedef

Typedef consente la definizione di nuovi tipi di variabili o funzioni.

```
typedef unsigned int intero;

typedef struct Books{
    …
} bookType;

intero var = 22; //= unsigned int var = 22;
bookType book1; //= struct Books book1;
```

C - esempio “enum”

```
#include <stdio.h>

enum State {Undef = 9, Working = 1, Failed = 0};
void main() {
    enum State state=Undef;
    printf("%d\n", state); // output è “9”
}
```

C - vettori e stringhe

C - vettori I

I vettori sono sequenze di elementi omogenei (tipicamente liste di dati dello stesso tipo, ad esempio liste di interi o di caratteri).

I vettori si realizzano con un puntatore al primo elemento della lista.

Ad esempio con `int arr[4] = {2, 0, 2, 1}` si dichiara un vettore di 4 interi inizializzandolo: sono riservate 4 aree di memoria consecutive di dimensione pari a quella richiesta per ogni singolo intero (tipicamente 2 bytes, quindi 4*2=8 in tutto)

C - vettori II

`char str[7] = {‘c’, ‘i’, ‘a’, ‘o’, 56,57,0}` : 7*1 = 7 bytes

`str` è dunque un puntatore a `char` (al primo elemento) e si ha che:

`str[n]` corrisponde a `*(str+n)`

e in particolare `str[0]` corrisponde a `*(str+0)`=`*(str)`=`*str`

C - stringhe

Le stringhe in C sono vettori di caratteri, ossia puntatori a sequenze di bytes, la cui terminazione è definita dal valore convenzionale 0 (zero).

Un carattere tra **apici singoli** equivale all'intero del codice corrispondente.

In particolare un vettore di stringhe è un vettore di vettore di caratteri e dunque:

```
char c;        #carattere
char *str;     #vettore di caratteri / stringa
char **strarr; #vettore di vettore di caratteri / vettore di stringhe
```

Si comprende quindi la segnatura della funzione main con `**argv`.

Array e stringhe

C supporta l'uso di stringhe che, tuttavia, corrispondono a degli array di caratteri.

```
int nome[DIM];
long nome[] = {1,2,3,4};
char string[] = “ciao”;
char string2[] = {‘c’,’i’,’a’,’o’};
nome[0] = 22;
```

Gli array sono generalmente di dimensione statica e non possono essere ingranditi durante l'esecuzione del programma. Per array dinamici dovranno essere usati costrutti particolari (come malloc).

Le stringhe, quando acquisite in input o dichiarate con la sintassi `"stringa"`, terminano con il carattere '`\0`' e sono dunque di grandezza *str_len+1*

C - esempio carattere e argc/argv

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int code=0;
  if (argc<2) {
    printf("Usage: %s <carattere>\n", argv[0]);
    code=2;
  } else {
    printf("%c == %d\n", argv[1][0], argv[1][0]);
  };
  return code;
};
```

C - argomenti da CLI

- Per il parsing degli argomenti da CLI la libreria `getopt.h` mette a disposizione `getopt` e `getopt_long`.

  (v. https://www.gnu.org/software/libc/manual/html_node/Getopt.html)

- Si può effettuare un parsing manuale scorrendo gli argomenti.

C - parsing manuale argomenti: esempio

```
#define MAXOPTL 64
#define MAXOPTS 10
#include <stdio.h>
#include <string.h>
// arrays of options and of values
char opt[MAXOPTS][MAXOPTL];
char val[MAXOPTS][MAXOPTL];

int main(int argc, char **argv) {
  int a=0, o=0;
  // loop into arguments:
  while (++a<argc && o<MAXOPTS) {
     if (strcmp("-h", argv[a])==0)
          strcpy(opt[o++], "help");
```

```
…

      if (strcmp("-k", argv[a])==0) {
          strcpy(opt[o++], "key");
         if (a+1<argc)
           strcpy(val[o-1], argv[++a]);
      }
   }
   // dump options (keys/values):
   for (a=0; a<o; a++) {
     printf("opt[%d]: %s,%s\n",a,opt[a],val[a]);
   }
   return 0;
}
```

C - funzioni stringhe `<string.h>`

Dato che le stringhe sono riferite con un puntatore al primo carattere non ha senso fare assegnamenti e confronti diretti, ma si devono usare delle funzioni. La libreria standard string.h ne definisce alcune come ad esempio:

`char * strcat(char *dest, const char *src)` aggiunge src in coda a dest
`char * strchr(const char *str, int c)` cerca la prima occorrenza di c in `str`
`int strcmp(const char *str1, const char *str2)` confronta `str1` con `str2`
`size_t strlen(const char *str)` calcola la lunghezza di `str`
`char * strcpy(char *dest, const char *src)` copia la stringa src in dst
`char * strncpy(char *dest, const char *src, size_t n)` copia n caratteri dalla stringa src in dst

C - esercizi per casa

1. Scrivere un'applicazione che data una stringa come argomento ne stampa a video la lunghezza, ad esempio:
   `./lengthof "Questa frase ha 28 caratteri"`
   deve restituire a video il numero 28.

2. Scrivere un'applicazione che definisce una lista di argomenti validi e legge quelli passati alla chiamata verificandoli e memorizzando le opzioni corrette, restituendo un errore in caso di un'opzione non valida.

3. Realizzare funzioni per stringhe `char *stringrev(*char str)` (inverte ordine caratteri) e `int stringpos(*char str, char chr)` (cerca *`chr`* in *`str`* e restituisce la posizione)

(In tutti i casi si può completare l'esercizio gestendo gli eventuali errori di immissione da parte dell'utente come parametri errati o altro)

C - files

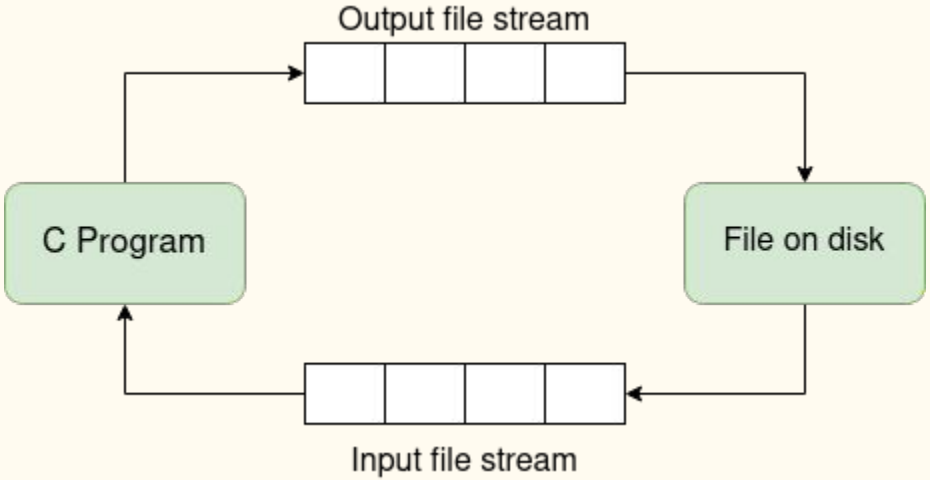
Interazione con i file

In UNIX ci sono due modi per interagire con i file: **streams** e **file descriptors.**

- **Streams**: forniscono strumenti come la formattazione dei dati, bufferizzazione, ecc...
- **File descriptors**: interfaccia di basso livello costituita dalle system call messe a disposizione dal kernel.

Interazione con i file - Streams

Utilizzando gli streams, un file è descritto da un puntatore a una struttura di tipo FILE (definita in stdio.h). I dati possono essere letti e scritti in vari modi (un carattere alla volta, una linea alla volta, ecc.) ed essere interpretati di conseguenza.



Interazione con i file - Stream

```c
#include <stdio.h>

FILE *ptr;  //Declare stream file
ptr = fopen("filename.txt","r+"); //Open

int id;
char str1[10], str2[10];
while (!feof(ptr)){ //Check end of file
    //Read int, word and word
    fscanf(ptr,"%d %s %s", &id, str1, str2);
    printf("%d %s %s\n",id,str1,str2);
}

printf("End of file\n");

fclose(ptr); //Close file
```

filename.txt:

```
1 Nome1 Cognome1
2 Nome2 Cognome2
3 Nome3 Cognome3
```

Modes:

- r: read
- w: write or overwrite (create)
- r+: read and write
- w+: read and write. Create or overwrite
- a: write at end (create)
- a+: read and write at end (create)

Interazione con i file - Stream

```c
#include <stdio.h>
#define N 10

FILE *ptr;
ptr = fopen("fileToWrite.txt","w+");
fprintf(ptr,"Content to write"); //Write content to file
rewind(ptr); // Reset pointer to begin of file
char chAr[N], inC;
fgets(chAr,N,ptr); // store the next N-1 chars from ptr in chAr
printf("'%c' '%s'",chAr[N-1], chAr);
do{
    inC = fgetc(ptr); // return next available char or EOF
    printf("%c",inC);
}while(inC != EOF); printf("\n");
fclose(ptr);
```

File Descriptors

Un file è descritto da un semplice **intero** (file descriptor) che punta alla rispettiva entry nella file table del sistema operativo. I dati possono essere letti e scritti soltanto un buffer alla volta di cui spetta al programmatore stabilire la dimensione.

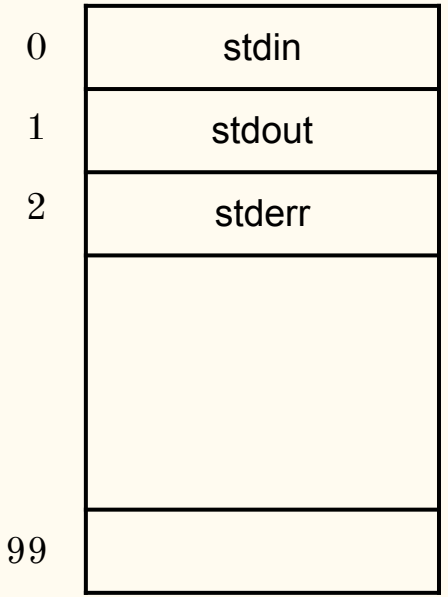
Un insieme di system call permette di effettuare le operazioni di input e output mantenendo un controllo maggiore su quanto sta accadendo a prezzo di un'interfaccia meno amichevole.

File Descriptors

Per accedere al contenuto di un file bisogna creare un canale di comunicazione con il kernel, aprendo il file con la system call open la quale localizza l'i-node del file e aggiorna la *file table* del processo.

A ogni processo è associata una tabella dei file aperti di dimensione limitata (circa 100 elementi), dove ogni elemento della tabella rappresenta un file aperto dal processo ed è individuato da un indice intero (il "file descriptor")

I file descriptor 0, 1 e 2 individuano normalmente standard input, output ed error (aperti automaticamente)

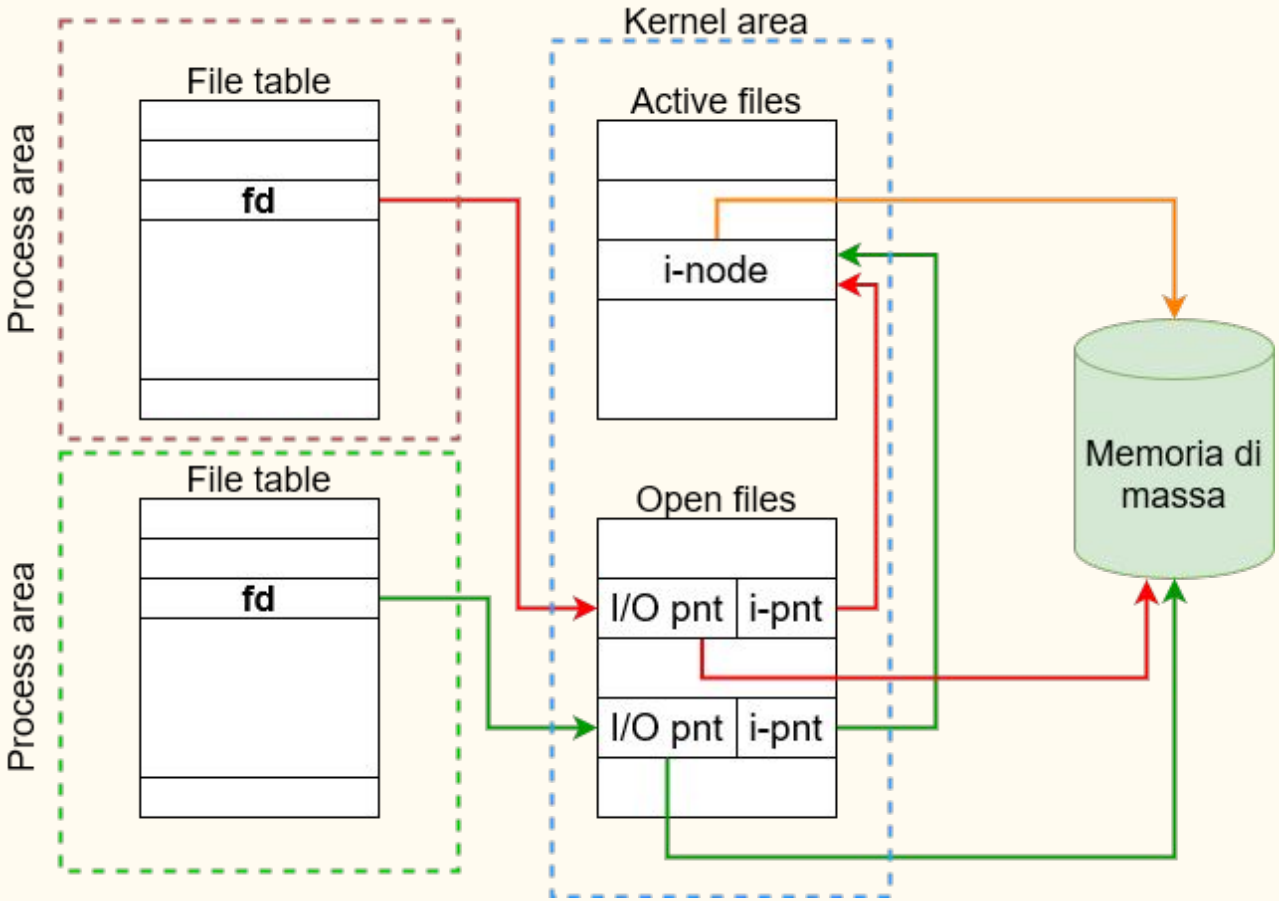


File Descriptors

Il kernel gestisce l'accesso ai files attraverso due strutture dati: **la tabella dei files attivi e la tabella dei files aperti**. La prima contiene una copia dell'inode di ogni file aperto (per efficienza), mentre la seconda contiene un elemento per ogni file aperto e non ancora chiuso. Questo elemento contiene:

- I/O pointer: posizione corrente nel file
- i-node pointer: Puntatore a inode corrispondente

La tabella dei file aperti può avere più elementi corrispondenti allo stesso file!


Kernel area
Process area
File table
fd
Active files
i-node
Memoria di massa
Process area
File table
fd
Open files
I/O pnt
i-pnt
I/O pnt
i-pnt

Interazione files - File Descriptors

L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close.**

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

//Open new file in Read only
int openedFile = open("filename.txt", O_RDONLY);
char content[10]; int canRead;
do{
     bytesRead = read(openedFile,content,9); //Read 9B from openedFile to buffer content
     content[bytesRead]=0;
     printf("%s",content);
} while(bytesRead > 0);
close(openedFile);
```

Interazione files - File Descriptors

L'Input/Output Unix è basato essenzialmente su cinque funzioni: **open**, **read**, **write**, **lseek** e **close.**

```
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
//Open file (create it with user R and W permissions)
int openFile = open("name.txt", O_CREAT|O_RDWR, S_IRUSR|S_IWUSR);
char toWrite[] = "Professor";

write(openFile, "hello  world\n", strlen("hello world\n")); //Write to file
lseek(openFile, 6, SEEK_SET); // riposiziona l'I/O pointer
write(openFile, toWrite, strlen(toWrite)); //Write to file

close(openFile);
```

Open() flags - File Descriptors

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Flags: ORed interi che definiscono l'apertura del file

- Deve contenere uno tra O_RDONLY, O_WRONLY, or O_RDWR
- O_CREAT: crea il file se non esistente
- O_APPEND: apri il file in append mode (lseek automatico con ogni write)
- O_TRUNC: cancella il contenuto del file (se aperto con W)
- O_EXCL: se usata con O_CREAT, fallisce se il file esiste già

Mode: definiscono i privilegi da dare al file creato: S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXU, S_IRGRP, ..., S_IROTH

Creat() e lseek()- File Descriptors

```
int creat(const char *pathname, mode_t mode);
```

Alias di open(file,O_CREAT|O_WRONLY|O_TRUNC, mode)

```
off_t lseek(int fd, off_t offset, int whence);
```

Muove la “testina” del file di un certo offset a partire da una certa posizione:

- SEEK_SET = da inizio file,
- SEEK_CUR = dalla posizione corrente
- SEEK_END = dalla fine del file.

C - files: canali standard I

- I canali standard (in/out/err che hanno indici 0/1/2 rispettivamente) sono rappresentati con strutture “stream” (`stdin`, `stdout`, `stderr`) e macro (`STDIN_FILENO`, `STDOUT_FILENO`, `STDERR_FILENO`).

- La funzione `fileno` restituisce l’indice di uno “stream”, per cui si ha:
  - `fileno(stdin)=STDIN_FILENO   // = 0`
  - `fileno(stdout)=STDOUT_FILENO // = 1`
  - `fileno(stderr)=STDERR_FILENO // = 2`
- `isatty(stdin) == 1` (se l’esecuzione è interattiva) OPPURE 0 (altrimenti)

`printf("ciao");` e `fprintf(stdout, "ciao");` sono equivalenti!

C - files: canali standard II

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

void main() {
  printf("stdin:  stdin ->_flags = %hd, STDIN_FILENO  = %d\n",
    stdin->_flags,  STDIN_FILENO
  );
  printf("stdout: stdout->_flags = %hd, STDOUT_FILENO = %d\n",
    stdout->_flags, STDOUT_FILENO
  );
  printf("stderr: stderr->_flags = %hd, STDOUT_FILENO = %d\n",
    stderr->_flags, STDERR_FILENO
  );
}
```

C - funzioni/operatori

generali e di uso comune

printf / fprintf

```
int printf(const char *format, ...)
int fprintf(FILE *stream, const char *format, ...)
```

Inviano dati sul canale stdout (`printf`) o su quello specificato (`fprintf`) secondo il formato indicato.

Il formato è una stringa contenente contenuti stampabili (testo, a capo, ...) ed eventuali segnaposto identificabili dal formato generale:

```
%[flags][width][.precision][length]specifier
```

Ad esempio: `%d` (intero con segno), `%c` (carattere), `%s` (stringa), ...

Ad ogni segnaposto deve corrispondere un ulteriore argomento del tipo corretto.
(rivedere esempi precedenti)

exit

`void exit(int status)`

Il processo è terminato restituendo il valore `status` come codice di uscita. Si ottiene lo stesso effetto se all'interno della funzione `main` si ha `return status`.

La funzione non ha un valore di ritorno proprio perché non sono eseguite ulteriori istruzioni dopo di essa.

Il processo chiamante è informato della terminazione tramite un "segnale" apposito. I segnali sono trattati più avanti nel corso.

C - piping via bash

C - piping via bash

- In condizioni normali l'applicazione richiamata da bash ha accesso ai canali stdin, stdout e stderr comuni (tastiera/video).
- Se l'applicazione è inserita via bash in un "piping" (come in `ls | wc -l`) allora:
  - Accede all'output del comando a sinistra da stdin
  - Invia il suo output al comando di destra su stdout

```
#define MAXBUF 10
#include <stdio.h>
#include <string.h>

int main() {
    char buf[MAXBUF];
    fgets(buf, sizeof(buf), stdin); // may truncate!
    printf("%s\n", buf);
    return 0;
}
```

```
$ gcc src.c -o pip.out
$ echo "ciao come stai" | ./pip.out
```

C - esempio piping da bash

Esempio di una semplice applicazione che legge da `stdin` e stampa su `stdout` invertendo minuscole [a-z] con maiuscole [A-Z]

```c
#include <stdio.h>

int main() {
    int c, d;
    // loop into stdin until EOF (as CTRL+D)
    while ((c = getchar()) != EOF) { // read from stdin
        d = c;
        if (c >= 'a' && c <= 'z') d -= 32;
        if (c >= 'A' && c <= 'Z') d += 32;
        putchar(d);         // write to stdout
    };
    return (0);
}
```

CONCLUSIONI

Comprendendo il funzionamento dei vari tipi di variabili, in particolare la gestione dei puntatori e dei vettori, e sfruttando poi le funzioni illustrate è possibile realizzare delle applicazioni che manipolano argomenti passati via CLI o anche interagire con processi terzi (in particolare attraverso il file-system o via bash con il piping).