C secondo sCagnetto

indice

concetti basilari (tipi, sintassi, ecc...)

compilazione

```
$clang programma.c #genera a.out
$ ./a.out #esegue
$clang file.c -o nome #rinomica a.out in nome

$clang -s file.c #crea file.s in assembly
$clang -c file.c #genera il file binario file.o
$clang file.o -o file #genera eseguibile
```

pre-processing

```
//precedute da #
#include <file.h> //include header e funzioni
#define NOME valore //definisce costante

#ifdef //se costante definita codice eseguito
#ifndef //se costante definita codice non eseguito
#else //possibile specificare ramo else
#endif //termina
```

```
eevitare inclusione molteplice header:
//NOTA: codice scritto in add.h!!
#ifndef ADD_H__ //non definito add.h
#define ADD_H__ //definisci
int add(int, int);
//resto del corpo header
#endif
```

sintassi di base e costrutti

```
costrutti(if, for, while, switch);
operatori artimetici: + - * / %(resto)
operatori logici: && || !
confronti: < > <= >= ==
speciali: += *=
```

tipi semplici e strutture

```
char, short, int, long , float, double // in ordine di promozione, (tipo) val casting

x = (int)5.0; //casting

const: dichiarazione costante interno funzione //const tipo nome = valore;
const int SIZE = 12;
```

```
booleano non esiste, espressioni intere
espressione FALSA = 0;
espressione VERA != 0;
if (x) , while(1) //valide
```

```
struct tipo_struct{ //es. struct
    tipo componente1; //es int key;
    tipo componente2; //es int* next;
    ...
};

struct tipo_struct nome { datocomponente1, datocomponente2, ...};

accesso alle componenti:
(puntatore).componente
puntatorestruttura->componente
```

```
// ESEMPIO
//dichiarazione di una struttura
struct point{
    float x;
    float y;
};
//dichiarazione di una variabile
struct point p = {3, 4};

//operazioni
printf("%f, %f\n", p.x, p.y);
scanf("{ %f, %f }", &p.x, &p.y);

//esempio funzione
float abs(struct point p){
    return sqrt(p.x * p.x + p.y * p.y); }

//puntatore a strutture e accesso
struct point *pp = &p;

printf("%f %f\n", pp->x, pp->y); //Equivalente alla seguente
printf("%f %f\n", (*pp).x, (*pp).y);
```

funzioni

```
//dichiarazione obbligatoria per ogni funzione
//tranne dichiarazioni presenti in un header
tipo_ritornato nome_funz (tipo_par1, tipo_par2,...)

int factorial (int); //esempio
//scope variabili locali è dentro funzione
```

array

```
//passati per RIFERIMENTO e non valore
tipo nome[N] = {valori}; //array di N valori
tipo nome[N] = { }; //array di N zeri
tipo nome[] = {elemento1, elemento2, ..., elemento n} //array n elementi
```

```
//esempio:
int array[5] = {1,2,3,4,5}; //dichiarazione
int *pa = &a[0]; //puntatore array
printf("%d", *pa); //stampa 1
printf("%d", *(pa+1)); //stampa 2
```

```
void fill(int *begin, int size, int value) {
 for(int *p=begin;p<begin+size;++p)
 *p=value;
 }
```

```
//è possibile anche dichiarare array multidimensionali:
float matrix[4][3] = {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
matrix[1][1] = 2;
```

puntatori

```
tipo *nome_puntatore = valore iniziale;

&: ottiene un puntatore alla variabile (=indirizzo memoria)
*: accesso alla variabile del puntatore (= valore)

p+i: puntatore avanza di i*sizeof(T) byte
*(p+i): p[i]

NOTA:
int a[10][20]; //alloca spazio per 200 interi (10 x 20)
int *b[10]; //alloca spazio per 10 puntatori a intero

a[i][j] e b[i][j] denotano due int, ma b[i] può puntare a un array di lunghezza diversa
```

```
//COMPORTAMENTI UTILI
#define SIZE 4
int a[SIZE]={10,20,30,40}
int *p = &a[0]; // analogo: int *p = a

printf("%d",*p); -> ritorna 10
printf("%d",*p+1); -> ritorna 11
printf("%d", *(p+1)); -> ritorna 20
//NOTA: p+2 = puntatore ad elemento a[2]

printf("%d", p); -> ritorna un errato
printf("%d", &p); -> ritorna un valore errato
```

input/output standard

```
#include <stdio.h>

char getchar(); // legge carattere dallo standard input
void putchar(c); // stampa carattere nello standard output
```

operazioni su stringhe

```
#include <string.h>

dichiarazione:
char message[] = "testo";
char message[] = {'t', 'e', 's', 't', 'o', 0} //0 carattere terminatore
//NOTA: '0' char 0; 0 oppure '\0' carattere terminatore
```

```
int strlen(char *s); // restituisce la lunghezza di una stringa
int strncmp(char *s1, char *s2, unsigned len); //confronto lessicografico (0 uguali, -1 altrimenti)
int strcmp(char *s1, char *s2); // versione meno sicura senza lunghezza
char *strncpy(char *dest, char *source, unsigned len);
 //copia i primi len char di source in dest
char *strncat(char *dest, char *source, unsigned len);
 //concatena i primi len char di source a dest
long long strtoll(const char *str, char **endptr, int base);
 // converte stringa in long long
extern char *strchr(const char *formato, int __c);
//trova la prima occorrenza di c (puoi usare 'c') in s


NON FARE: char *stringa = "testo della stringa";
```

stampa

```
// Stampa su standard input/outpt, rito
int printf(const char *formato, var1, var2, ...); //stampa in stdout, sostituendo % con arg1, ...
//esempio const char *formato: "ciao %s!\n", %s sarà sostituita con stringa

int scanf("formato", &var1, &var2, ...);
//legge stdin e memorizza nelle variabili passate come argomento

NOTA: posso usarla in un ciclo
while (scanf("%d", &array[read])==1) //se scanf ha funzionato
```

```
sscanf(char *source, "formato", &var1, &var2, ...); // legge da stringa invece che sdin
//esempio
char msg[] = "43 23 25";
sscanf(msg, "%d %d %d", &x, &y, &z); //x=43, ...

sprintf(char *dest, const char *formato, var1, var2, ...);
// stampa su una stringa invece che su standard output

snprintf(char *dest, size_t maxlen, const char *formato, var1, var2, ...);
// ulteriore argomento lunghezza massima stringa (consigliata)
```

memoria dinamica

```
void *malloc(unsigned n); // alloca n byte contigui, void * puntatore qualsiasi tipo
void free(void *ptr); // libera memoria allocata con malloc (buona norma usarlo sempre)
void *realloc(void *ptr, unsigned new_size); //ritorna puntatore a nuova memoria
void *calloc(unsigned count, unsigned size); // alloca memoria azzerandola precedentemente
```

```
//esempi di uso:
int *elementi = malloc(n * sizeof(int));
//...
free(elementi);

int *array = malloc(size * sizeof(int));
while (scanf()==1){
    //se ho raggiunto fine array
    size *= 2; //raddoppio size
    array = realloc(array, size * sizeof(int));
}
```

accesso ai file

```
FILE *stdout; // standard output (FILE in scrittura)
FILE *stdin; // standard input (FILE in lettura)
FILE *stderr; // standard error (FILE in scrittura)
```

```
FILE *fopen(char *name, char *mode); //puntatore del file su cui operare
mode può essere: "r", "w", "a", "rb", "wb"

int fclose(FILE *fp); // chiude il file pointer (buona norma utilizzarlo sempre)

int fprintf(FILE *fp, char *format, ...); // analogo a printf() con stdout
int fscanf(FILE *fp, char *format, ...); // analogo a scanf() con stdin
int fgetc(FILE *fp); // analogo a getchar()
int fputc(int c, FILE *fp); // analogo a putchar()

size_t fread(void *ptr, size_t size, size_t nitems, FILE *file);
// legge (size * nitems) byte da file e scrive su *ptr

size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *file);
 // scrive (size * nitems) byte da *ptr su file

int feof(FILE *fp); // restituisce vero (n > 0) se lettura è arrivata alla fine
int ferror(FILE *fp); // restituisce vero (n > 0)

char *strerror(int errno); //ritorna stringa errore
void perror(const char *__s); // stampa messaggio
//uso
perror("fork()"); //out  fork(): descrizione errore
strerror(eerno)); //out: descrizione errore

int fseek(FILE *fp, long offset, int whence);
// imposta la posizione attuale a _offset_ byte da posizione _whence_
// whence = SEEK_SET (inizio file), SEEK_CUR (posizione corrente), SEEK_END (fine file)

int ftell(FILE *file); // restituisce posizione attuale
```

chiamate POSIX

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

// Standard file descriptors
- 0: standard input
- 1: standard output
- 2: standard error
```

```
int open(const char *pathname, int openflags); // restituisce il file descriptor
//flags:
- O_RDONLY: open for reading only
- O_WRONLY: write only
- O_RDWR: reading and writing
- O_APPEND: append on each write
- O_CREAT: create file if it does not exist
- O_TRUNC: truncate size to 0
- O_EXCL: error if O_CREAT and the file exist

int creat(const char *pathname, mode_t mode);
// equivalente a open() con flag O_CREAT | O_WRONLY | O_TRUNC

int close(int fd); // chiude il file descriptor

ssize_t read(int fd, void *buffer, size_t nbytes); // legge
ssize_t write(int fd, const void *buffer, size_t n); // scrive
off_t lseek(int fd, off_t offset, int whence); // imposta la posizione
// whence = SEEK_SET (inizio file), SEEK_CUR (posizione corrente), SEEK_END (fine file)

int unlink(const char *pathname); // rimuove un file
int fcntl(int fd, int cmd, ... /* arg */); // manipola il file descriptor
int chmod(const char *pathname, mode_t mode); // cambia i permessi del file
```

```
#include <sys/types.h>
#include <sys/stat.h>

int stat(const char *pathname, struct stat *out); // ottiene informazioni sul file
int fstat(int filedes, struct stat *out); // ottiene informazioni sul file descriptor

struct stat {
    dev_t st_dev;       // device id
    ino_t st_ino;       // inode number
    mode_t st_mode;     // tipo di file e permessi
    nlink_t st_nlink;   // numero di link non simbolici
    uid_t st_uid;       // UID
    gid_t st_gid;       // GID
    dev_t st_rdev;      // device type
    off_t st_size;      // dimensione del file
    time_t st_atime;    // tempo di ultimo accesso
    time_t st_mtime;    // tempo di ultima modifica
    time_t st_ctime;    // tempo di creazione
    long st_blksize;    // dimensione del blocco
    long st_blocks;     // numero di blocchi
};
```

processi e fork

```c
#include <sys/types.h>

pid_t getpid(void); // restituisce il PID
pid_t getppid(void); // restituisce il PPID
pid_t getpgrp(void); // restituisce il gruppo

pid_t fork(void); // duplica processo, PID > 0 padre, PID = 0 figlio, -1 errore

//es:
pid_t pid=fork();
if (pid == 0)
    printf("figlio pid %d", (int)getpid());
else if (pid > 0)
    printf("padre pid %d",pid);
else
    fprintf(stderr,"errore");
```

```c
pid_t wait(int *stat_loc); // aspetta la terminazione di un processo figlio

void exit(int status); // termina il processo

char *getenv(const char *name); // ottiene il valore di una variabile d'ambiente

int setenv(const char *name, const char *value, int overwrite);
// setta una nuova variabile d'ambiente, overwrite: 1 sovrascrive, 0 non sovrascrive
```

```c
//famiglia exec, lancia processo che esegue programma diverso

//argomenti riga comando passati alla funzione senza enviroment
int execl(const char *name, const char *arg0, ..., NULL);

//argomenti riga comando passati come array, senza enviroment
int execv(const char *name, const char **argv);

//argomento riga comando passati alla funzione, senza enviroment, cerca nel path
int execlp(const char *name, ...);

//argomento riga di comando passati come array, senza enviroment, nel path
int execvp(const char *name, cons char **argv);

//argomento riga di comando passati alla funzione, con enviroment
int execle(const char *name, ..., /* envp */);

//argomenti riga di comando, come array, con enviroment
int execve(const char *name, const char **argv, const char *envp);
```

```c
#include <unistd.h>

int chdir (const char *path); //cambia current working directory
int chroot (const char *path); //raramente utile
```

```c
uid_t getuid(); //conoscere user id
gid_t getgid(); //consocere group id

setuid(); //per abbassare i propri privilegi
setgid();
```

pipe

```c
//un capo scrive e uno legge mediante FIFO

#include <unistd.h>

pipefd[2] = {int fd_reader, int fd_writer}

int pipe(int pipefd[2]); /* crea una pipe
pipefd è un array di due int, filedescriptor
pipefd[0] è il fd che legge dalla pip
pipefd[1] è il fd che scrive nella pipe */

int dup2(int oldfd, int newfd); //redirige newfd alla stessa risorsa di oldfd

//scrivere dalla pipe, uso POSIX
// close(pipefd[0]); nel caso non serva
write(pipefd[1], "testo messaggio!", MSGSIZE);

//leggere dalla pipe, uso POSIX
//close(pipefd[1]); nel caso non serva
read(pipefd[0], msg, MSGSIZE); //nota: msg puntatore in cui registrare messaggio
```

```c
//esempio
int fds[2] = {0,0} //posix {stdin, stdin}
pipe(fds); //stdin in lettura, stdin in scrittura
dup2(fds[1],1); //ridirigo in scrittura sdout
```

segnali

```c
#include <signal.h>

int kill(pid_t pid, int sig); // invia il segnale _sig_ al processo con PID _pid_
int raise(int sig); // invia il segnale _sig_ a se stessi
unsigned int alarm(unsigned int secs); // ricezione di SIGALRM dopo int secs
```

```
Segnali:
- identificati da un int
- anomali, non possono ignorati: dal 3 al 12 inclusi, 17, 24,25
- altri int possono essere ignorati e gestiti

SIGINT (2): CTRL-c
SIGSTP(18): CTRL-Z
SIGCHLD(20): cambio status child
SIGKILL(9): non può essere ignorato
```

```c
per gestire segnale
typedef void (*sighandler_t)(int);
sighandler_t (int sig, sighandler_t handler);
//esempio:

signal(SIGINT, ahahha); //nel corpo del main

void ahahha (int x){
    printf("hai premuto CTRL-C e io stampo questo");
}
```

socket

```c
#include <sys/socket.h>
#include <sys/types.h>

//creazione socket, devono farlo sia client che server
int socket(int domain, int type, int protocol); /*
- domain:  AF_LOCAL (locale)
- type: SOCK_STREAM, UDP (non affidabili, non nostro caso)
- protocol: 0 sceglie il S.O. (meglio nei nostri casi) */

//apro il socket e ottengo il suo file descriptor
int sockfd = socket(AF_LOCAL, SOCK_STREAM,0); //file descriptor

//operazioni server
//apro con socket() e ottengo file descriptor

//associo un indirizzo al socket con bind (creo socket file)
int bind(int sockfd, const struct sockaddr *addr, size_t addrlen);

//metto il scoket in ascolto, sockfd
int listen(int sockfd, int queue_size); //queue_size max client in attesa

//quando ricevo una connessione accetto, ottenendo nuovo file descriptor
int accept(int sockfd, struct sockaddr *addr, size_t *addrlen);
//nota: blocca processo finché client non si connette

//operazioni client
//apro con socket() e ottengo nuovo file descriptor

//provo a connettermi al socket
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);

//per comunicare dati:

ssize_t send(int sockfd, const void *buf, size_t len_buf, int flags); // invia dati
//*buf: è il dato;

ssize_t recv(int sockfd, void *buf, size_t len_buf, int flags); // riceve dati
//*buf: dove registro dato

//nota: con flags = 0 posso usare write() e read() POSIX

int close(int sockfd); // chiude socket, usato sia in client che in server!
```

```c
//sockaddr
struct sockaddr_un{
    short sa_family; // = AF_LOCAL
    char sun_path[108]; //indirizzo del socket, va bene /temp/...
};
```

multithreading

```
$ clang -lpthread programma.c -o programma #bisogna linkare nella shell
```

```c
#include <pthread.h>

//creo il thread
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
 void *(*start_routine)(void *), void *arg); /*

thread: puntatore che funge da handler
attr: puntatore a struttura con opzioni e configurazioni aggiuntive
*start_routine: nome_funzione che verrà eseguita nel nuovo thread
    deve essere così: void *nome_funzione (void *ptr){...}
arg: puntatore a void che verrà passato come argomento di nome_funzione
    NOTA: arg può essere un qualsiasi dato, importante che in pthread_create
            sia castato in (void *) : (void *)msg - con msg stringa va bene

 */

// crea un nuovo thread
int pthread_join(pthread_t thread, void **retval); // aspetta la terminazione di un thread
int pthread_mutex_lock(pthread_mutex_t *mutex); // blocca un mutex
int pthread_mutex_unlock(pthread_mutex_t *mutex); // sblocca un mutex
```