C SHEET

compilazione semplice

il file deve avere estensione .c

`$clang programma.c` genera file a.out
`$ .\a.out` esegue il file

altrimenti posso rinominare il file a.out con
`$ clang file.c -o nome`
`$ .\nome`

4 fasi: pre-processing, compilazione (C → assembler), assemblaggio (assembler → binario), linking

visualizzare il codice assembly (post compilazione):
`$ clang -s file.c`
`$ cat file.s`

visualizzare file oggetto (post assembklaggio):

`$ clang -c file.c`
si genera un file `file.o`, non apribile con un editor di testo

posso compilare per stadi, per esempio:

`$ clang add.c -c`
`$ clang main.c -c`
`$ clang add.o main.o -o add`

progetto c e compilazione separata

Un unico file dovrà contenere la funzione `main()`.
Per chiamare una funzione definita in un file diverso la *dichiarazione* deve essere visibile. Analogamente anche per costanti, tipi, ecc
Di norma le dichiarazioni vengono inserite negli header che:

- contengono esclusivamente le dichiarazioni
- file incluso con `include <file.h>` o `include "file.h"`
- il file header deve essere incluso sia nel file.c contenente le definizioni delle funzioni, sia negli altri file
- il file.c di definizione se include l'header corrispondente non ha bisogno della dichiarazione
- poiché più inclusioni dello stesso header non vanno bene, per evitare ripetizioni tra file:
  - `#ifdef`: if defined, codice incluso se prima è stata definita la costante
  - `#ifndef`: if not defined, codice escluso se prima è stata definita la costante

esempio ifdef

```
#define SIMBOLO
#ifdef SIMBOLO //se definito
    codice...
#endif
```

esempio ifndef (header che si accorga nel caso venga definito più volte, buona pratica)

```
#ifndef ADD_H__ //se non è definito, in questo
#define ADD_H__ //definisci
int add(int x, int y);
//resto del corpo del file header
#endif
```

grafo dipendenze e makefile

`make` si occupa di gestire compilazione dei programmi c. Il grafo delle dipendenze viene codificato in un file di testo chiamato `Makefile`

sintassi:

```
target : source file(s)
    command
```

nota, command deve essere preceduto da <tab>. In caso di source più recente i target (dipendenti) vengono aggiornati eseguendo i comandi specificati nelle regole del Makefile
esempio di makefile:

```
add: add.o main.o
    clang add.o main.o -o add
add.o: add.h add.c
    clang -c add.c
main.o: add.h main.c
    clang -c main.c
```

inovcando `make` viene interpretato il Makefile e compilato il programma.

direttive pre processing

`#`: direttive del processore, a *inizio* del programma

`#define`: serve a definire una costante o una funzione
`#include`: serve a preprocessare un file con determinate funzioni

Es: `#include <stdio.h>`

- definire una costante `#define NOME valore`
- includere file `#include <nome.h>`

tipi presenti in c

char, short int, int, long int, float, double, unsigned int (anche long / short)

conversioni mediante:

- promozioni: char → short → int → long int → float → double
- cast: (tipo) variabile

particolarità:

- le stringhe sono considerate array di char
- non esiste il tipo bool, costrutti si aspettano espressioni intere (0 = falso)
- struct...

sintassi di base

- dichiarazione variabili:
- dichiarazione funzione (prototipo): `tipo_ritornato nome_funzione(lista_parametri);`
- dichiarazione di tipo:
- costrutti di controllo e cicli (if,for,while, switch)
- operatori aritmetici: +, -, *, / (parte intera), % (modulo), ++, --
- operatori di confronto: < > <= >= == !=
- operatori logici: && (and), || (or), ! (not)
- costanti

costrutti
for(dichiarazione e inizio indice; condizione fine; passo): termina quando condizione fine = 0
while(condizione): se condizione uguale a 0 termina, altrimenti continua

specificatori di formato:

- `%d` : intero
- `%f` : float
- `%c` : char
- `%s` : stringa
- `%p` : puntatore
- `%x` : esadecimale
- `%2.3f` : float con almeno 2 cifre intere e 3cifre decimali (formato)

nota %% quoting del carattere % in printf su visual studio

sequenze di escape:

- \n newline
- \t Tab
- \\ singola backslash '\'
- " doppi apici

struttura del programma

```
#include ...

int main(){
    blocco di codice
    return 0;
}
```

funzioni

Le funzioni vengono prima dichiarate e poi definite tramite l'implementazione

- dichiarazione: "intestazione", tipo nome(argomenti)
- implementazione: corpo

dichiarazione deve precedere la chiamata, se

variabili e passaggio per valore

le funzioni operano su **copie** dei valori degli argomenti, non sugli argomenti stessi. L'unico modo per operare sugli argomenti è passare il loro indirizzo di memoria, tramite *puntatori*.

- scope: visibilità delle variabili

array

dichiarazione `tipo nome_array[dim] = {valori}`

generalmente uso dei puntatori per accedere agli elementi dell'array o per passare l'array ad una funzione
gli array non vengono passati per valore, ma per riferimento

esempio:

```
int array[5] = {1,2,3,4,5};
int *pa = &a[0];
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

variabile che contiene l'**indirizzo di memoria** di un'altra variabile

`tipo *nome_puntatore = valore iniziale;`

ci sono due operatori:

- adress of `&`: ottine un puntatore alla variabile (= indirizzo di memoria)
- dereferenziazione `*`: accesso alla variabile puntata dal puntatore (= valore)

tramite aritmetica dei puntatori posso accedere agli elementi di un array:
*p + i : puntatore avanza di i * sizeof(T) byte
*(p + i) : p[i]

in quanto variabili, anche i puntatori possono essere elementi di un array
`char *line[42] = {};` array di 42 puntatori a char *

`int i = 2;`
`int *p = 0;`: un puntatore di valore 0
`&i` : indirizzo della variabile i
`&p` : indirizzo del puntatore p
`int *p = &1;`: il puntatore p = indirizzo di i
`int *p = i;`: il puntatore assume valore i(=2);
`int x = *p;`: x è uguale a contenuto nell' indirizzo di p, se p non è un registro

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
```

puntatory e array

alcuni comportamenti utili:

```
#define SIZE 4
int a[SIZE]={10,20,30,40}
int *p = &a[0]; //importanza dichiarazione puntatore con presenza &variabile

printf("%d",*p); -> ritorna 10
printf("%d",*p+1); -> ritorna 11
printf("%d", *(p+1)); -> ritorna 20

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

stringhe

le stringhe sono array di char, terminate da un carattere nullo '\0'
esempio:
`char s[] = "ciao";`
`char s[] = {'c','i','a','o','\0'};`

nota: `char *stringa = "Ciao mondo";` compila ma **NON** è corretto

librerie viste a lezione

- stdio.h: standard input/output
- string.h: funzioni per la manipolazione di stringhe

argomenti da riga di comando

un programma C può ricevere argomenti da riga di comando, tramite la funzione
`int main(int argc, char **argv)`

`$ nome_programma argv[1] argv[2] ...`
`$ somma -s`

argc: numero di argomenti passati (il primo è la chiamata, non serve contarli)
argv: array di puntatori a carattere, che puntano alle stringhe degli argomenti
primo parametro: nome programma
ultimo elemento (argv[argc]) è NULL

scanf e sscanf

Di default gli spazi bianchi tra due valori in input vengono ignorati

```
int x = 0, y = 0;
scanf("%d %d", &x, &y);
```

posso anche richiedere altri caratteri in input, che vengono richiesti ma ignorati in lettura

```
float real = 0, float imag = 0;
scanf("( %f , %f )", &real, &imag);
```

`$ "(3.14, 0)"`

leggere e convertire il valore secondo tipo specificato e ignorarlo:
`scanf("(%f %*c %f)", &real, &imag);`

allocazione dinamica

funzione malloc `void *malloc(unsigned n);` argomentto numero di byte da allocare, ritorna il puntatore all'inizio dell'area di memoria (di qualsiasi tipo)

funzione sizeof() `sizeof(tipo)` ritorna il numero di memoria dedicata per una singola istanza del tipo (es. un byte per un char e via dicendo)

la funzione free() `free(*puntatore)` serve a liberare la memoria allocata dinamicamente con malloc, in quanto la memoria allocata dinamicamente non ha uno scope preciso, rimanendo allocata

strutture

Sono un **tipo di dato** aggregato, che raggruppa variabili di tipo diverso in un'unica identità. Analoghe ai tipi base (quindi possono essere contenute in un array e possono esserci puntatori del tipo struttura)

dichiarazione di una struttura

```
struct name{
    tipo nome1;//istanza 1
    tipo nome2;//istanza 2
    ...
}

essendo come
struct name p = { componente1, componente2, ...} //varaibile di tipo struct
```

L'accesso alle componenti delle struct può avvenire in due modi:

- tramite puntatore alle componenti: `(puntatore).componente`
- stessa cosa ma tramite *operatore dedicato*: `puntatorestruttura->componente`, (s->var)

Esempio:

```c
//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);
```

PROGRAMMAZIONE DI SISTEMA

Un processo interagisce con sistema operativo tramite chiamate di sistema. Al programmatore sono fornite funzioni, di tipo:

- ISO C
- POSIX

accesso ai file

prima di leggere/scrivere bisogna *aprire* un file:

```c
FILE *fopen(char *name, char *mode);
```

*name = nome del file, *mode = stringa modalità "r" (reading), "w" (writing), "a" (append)

ottenuto il puntatore di tipo FILE*, possiamo usarlo per operare sul file con fprintf(),fscanf(),fgetc(),fputc()...

per *chiudere* un file:

```c
int fclose(FILE *fp);
```

NOTA: esistono tre file pointer standard già aperti e pronti all'uso:

- `stdout` : standard output → *printf("str")* equivale a *fprintf(stdout, "str")*
- `stdin` : standard input → *scanf(*char, "formato", ...)* equivale a *fscanf(stdin, "formato", ...)*
- `stderr` : standard error

gestione degli errori

tramite feof() e ferror() distinguo i casi in cui c'è un errore in lettura rispetto al caso in cui il file è terminato. Per distinguere quale errore, le funzioni impostano la variabile globale errno, in errno.h si trovano le costanti corrispondenti ai possibili errori riportati dalle funzioni stnadard (EACCES E EISDIR)

tramite la funzione strerror() posso ottenere una stringa di descrizione dell'errore

```
fprintf(stderr," descrizione errore:%s\n", strerror(errno));
```

anche la funzione pererror() agisce analogamente

posizionamento in lettura/scrittura

lettura e scrittura avvengono in modo sequenziale (inizio → fine)

```
int fseek( FILE *file, long offset, int whence)
```

es. `fseek(file, 0, SEEK_SET)` ; fa tornare all'inizio del file

la funzione `ftell()` restituisce posizione corrente

input/output binario

fopen(char* nome_file, "rb"): reading binary, apre il file in lettura binaria (output)
fopen(char *nome_file, "wb"): writing binary, apre il file in scrittura binaria (input)

per I/O binario si usano funzioni apposite:

lettura:

```
size_t fread(void *ptr, size_t size, size_t nitems, FILE *file );
```

legge size*nitems byte dal file e scrive nella memoria puntata da ptr

scrittura:

```
size_t fwrite(void *ptr, size_t size, size_t nitems, FILE *file);
```

scrive sul file size*nitems byte dalla memoria puntata da ptr

FUNZIONI POSIX

#include <unistd.h>

- permettono di scrivere/leggere da fonti diverse: pipe,socket
- funzioni POSIX effettuano direttamente le system call, ISO adottano un buffer interno
- permettono di gestire permessi, link e altri attributi dei file

aprire un file

```
int open(const char *path, int openflags);
```

apre il file path nel modo specificato da openflags:

- 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

restituisce un file descriptor che:

- rappresenta il file aperto
- può essere associato a file ma anche a pipes, socket, ecc
- tre file descriptor default:
  - standard input: (0)
  - standard output: (1)
  - standard error: (2)

lettura e scrittura tramite read() and write() che sono analoghe alle rispettive ISO fread() e fwrite()

```
ssize_t read(int fd, void *buffer, size_t nbytes);
```

```
ssize_t write(int fd, void *buffer, size_t nbytes);
```

chiamate stat e fstat

stat() e fstat() permettono di accedere in lettura alle informazioni e proprietà di un file

stat(): primo argomento pathname

```
int stat(const char *pathname, struct stat *out);
```

fstat() opera su un descrittore di un file già aperto

```
fstat( int filedes, struct stat *out);
```

risultato è un puntatore di una struttura struct stat:

```
struct stat{
...
}
```

GESTIONE DEI PROCESSI

<sys/types.h>

processo: unità base di esecuzione di un sistema UNIX, isolato rispetto agli altri processi

system call per processi:

getpid(): restituisce il PID
getppid(): restituisce il PPID
getpgrp(): restituisce il gruppo

fork(): `pif_t fork()` duplica il processo, restituisce PID con padre, 0 con figlio;

exec():

wait():

exit():

fork

```
pid_t fork();
```

crea una copia del processo chiamante e restituisce il PID del figlio al genitore, 0 al figlio
esempio importante

system call exec

famgilia di macro delle funzioni `exec()` servono a lanciare processi che eseguano programmi diversi rispetto al chiamante, sostituisce il processo attuale con esecuzione di un altro file eseguibile

```
int execl(const char *path, cons char *arg0, ...);
```

- `path` eseguibile che si vuole lanciare
- `arg0, ...` sono gli argomenti da riga di comando, ultimo argomento NULL

sovrascrive il programma originale con quello passato da parametro, pertanto le istruzioni successive a execl() verranno eseguite in caso di errore di esecuzione di execl() con ritorno del controllo al chiamante

```
int main(){
    print("ciao")
    execl ("/bin/ls", "ls","-l", NULL; )
}
```

si combina molto bene con fork(): fork() crea il nuovo processo ed exec() esegue un nuovo programma nel figlio

ambiente di un processo

ambiente di un processo: insieme dei valori che processo eredita dal padre:

- array di puntatore a stringhe, del tipo *NOME=valore*, terminato da puntatore nullo
- ambiente viene passato attraverso un terzo parametro alla funzione main()

l'array è presente anche nella variabile globale environ:

```
extern char **environ;
```

tuttavia si preferisce operare con

getenv(): recupera enviroment dato il pathname
setenv): `int setenv(char *name, char *value, int overwrite)`

l'ambiente è un dato privato del processo
di default coincide con quello del padre, che però può decidere arbitrariamente l'ambiente del figlio prima dell'exec() tramite:

execle(): `int execle(const char *path, arg1,...,argn,NULL, const char **envp);`
execve(): `int execve(const char *path, const char **argv, const char **envp);`

`envp` ambiente desiderato

PATH

PATH: usata per individuare eseguibile corrispondente al comando, execv/execl non cercano nel path

due versioni cercano invece nel path

- execvp() 😳)
- execlp() 😳)

famiglia exec

```
int execl(const char *name, ...); argomenti riga comando passati alla funzione senza enviroment
int execv(const char *name, const char **argv); argomenti riga comando passati come array, senza enviroment
int execlp(const char *name, ...); argomento riga comando passati alla funzione, senza enviroment, cerca ne
int execvp(const char *name, cons char **argv); argomento riga di comando passati come array, senza enviro
int execle(const char *name, ..., /* envp */); argomento riga di comando passati alla funzione, con enviro
int execve(const char *name, const char **argv, const char *envp); argomenti riga di comando, come array, c
```

current working directory e root directory

current working directory: directory corrente
root directory: directory che il processo vede come directory radice (/.)

possono essere cambiate con

#include<unistd.h>

`int chdir (const char *path);`
`int chroot(const char *path);` utile in rari casi

User ID e Group ID

sistema di permessi: associazione ID utente/gruppo a un processo

processo con uid 0: privilegi di root
ogni file ha proprio UID e GID e propri permessi accesso
se UID e GID corrispondono con permessi di accesso file, processo può accedere

per consocere proprio UID/GID:

uid_t getuid();
gid_t getgid();

per abbassare i propri privilegi:

setuid();
setgid();

ogni processo associa real UID e real GID che coincidono con quelli dell'utente che ha lanciato il processo, tuttavia esistono effective UID e effective GID (EUID/EGID) che determinano effettivamente i privilegi. Solitamente coincidono.

Se bit Set UID o Set GID attivo, effective UID/GID sarà quello del proprietario del file e non quello dell'utente che lo ha lanciato

pipe

sistema di comunicazione tra due processi, in cui uno invia dati tramite `write` e l'altro legge tramite `read` (in modo FIFO)

per creare una pipe

#include <unistd.h>

pipe(): `int pipe(int *filedes);` filedes punta a un array a due interi (filedescriptor):

- filedes[0] legge dalla pipe
- filedes[1] scrive nella pipe.

uno dei due capi deve essere un processo figlio

- file descriptor aperti dopo fork()
- possibile redirigere un file descriptor aperto su un altro. La redirezione resta in piedi dopo execv()

per redirigere
dup2(): `int dup2(int old_fd, int new_fd);`

dopo la chiamata new_fd punterà alla stessa risorsa puntata da old_fd (che doveva essere già aperto). Se new_fd già in uso, chiuso e riaperto

es

```
int fds[2]={ }; //nota: è l'array {0,0}
pipe(fds);      //nota: fds[0] = (stdin), fds[1] = 0 (stdin)
dup2(fds[1], 1);//nota:
```

segnali

meccanismo per inviare interrupt ai processi, possono essere gestiti tramite:

- funzione (signal handling)
- blocco del signale
- invio a un altro processo

system call principale

kill():

segnale può essere lanciato solo a processi dello stesso utente (salvo root)
si può mandare segnali a se stessi con raise() oppure alarm(secs) che causa ricezione di SIGALARM dopo intervallo di tempo in secs

signal handling

un segnale si può gestire, eseguendo una funzione ogni volta che viene ricevuto mediante:

```
typedef void (*sighandler_t)(int); //definizione tipo puntatore a funzione
sighandler_t signal(int sig, sighandler_t handler);
```

signal(): registra funzione puntata da handler come gestore del segnale *sig*, versione semplificata di sigaction()

Socket

Sono un meccanismo di processo bidirezionale:

- operano su file descriptor
- filosofia client/server
- server ascolta un indirizzo, processi client si connettono
- stesso modello di comunicazioni in rete
- più domini: UNIX-domain (locale), Internet-domain (IPV4/IPV6)

socket(): crea file descriptor di un capo della connessione (server e client)

client
connect(): connette un socket a un altro in ascolto

server
bind(): lega il socket a un indirizzo
listen() marca il socket come passivo (per accettare connessioni)
accept(): accetta connessione in arrivo

funzione socket

#include <sys/socket.h>

`int socket(int domani, int type, int protocol);` dominio, tipo, protocollo

nel caso di dominio UNIX:

```
int fd = socket(AF_LOCAL, SOCK_STREAM, 0);
```

AF_LOCAL: no rete, SOCK_STREAM: affidabile, 0 protocollo scelto dal SO

la funzione bind() lega un socket a un indirizzo, mentre listen() abilita l'ascolto

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

struct sockaddr_un{
    short sa_family;
    char sun_path[108];
};

int bind (int  sockfd, const struct sockaddr *addr, size_t addr_len);
int listen(int sockfd, int queue_size);
```

sockfd: file descriptor del socket
sockaddr: indirizzo del socket, ovvero il nome di un socket file creato da bind()

queue_size: numero massimo di client che possono rimanere in attesa

accept e connect

per connettersi come client su cui esista un processo in ascolto:

connect: `int connect(int sockfd, const struct sockaddr *adress, size_t add_len)`

la connessione si instaura quando il server chiama accept():

accept: `int accept(int sockfd, struct sockaddr *address, size_t *addr_len);` restituisce un nuovo file descriptor, il vecchio è utilizzabile per accettare un'altra connessione (solitamente forkando il processo)

per comunicare, o come tutti gli altri file, oppure:

send: `ssize_t sen(int fd, const void *buffer, size_t length, int flags);`
recv: `ssend_t recv(int fd, void *bufferm, size_t length, int flags);`

fd: id del socket
*buffer: messaggio o puntatore
length: lunghezza del buffer
flags: 0, protocollo scelto dal sistema

simili a write() e read() ma con parametro aggiuntivo flags.

multithreading

ogni processo può contenere uno o più thread
eseguiti in modo indipendente ma condividono gran parte delle risorse del processo, ma separate risorse legate al proprio flusso
l'accesso alle risorse condivise avviene in race condition

#include pthread (POSIX thread)

NOTA (serve linkare la libreria):
`$ clang -lpthread programma.c -o programma`

creare un thread

pthread_create():
`int pthread_create(pthread_t *thread, pthread_attr_t *attr, void * (*start_routine) (void*), void *arg);`

ritorna 0 se non ci sono errori

*thread: puntatore a variabile che funge da handler per rappresentare thread
*attr: puntatore a struttura di opzioni e configurazioni aggiuntive (NULL deafult)
*start_routine (void *): puntatore a una funzione che verrà eseguita dal nuovo thread (deve accettare e restituire un void*)
arg: puntatore a void passato come argomento alla funzione

thread attribute object: struct pthread_attr_t

```
struct pthread_attr_t attr;
pthread_attr_init(&attr);

pthread_create(&thread, &attr, func, arg);
pthread_attr_destroy(&attr);
```

il valore di ogni singolo attributo X si imposta con l'apposita funzione `pthread_attr_setX()`

- `schedpolicy`: politiche di scheduling
- `inheritsched`: se il thread eredita le policy di scheduling del thread padre
- `scope`: specifica
- `schedparam`: specifica la priorità associata al thread

terminazione thread

thread termina quando la sua esecuzione principale ritorna oppure viene chiamato pthread_exit

```
void pthread_exit(void *retval);
```

mentre la funzione pthread_join() permette un thread di aspettare la fine di un altro di cui abbia l'handler

`int pthread_join(pthread_t th, void **value_ptr);`

la funzione blocca il thread in attesa del thread th, il valore di ritorno viene scritto in *value_ptr

sincronizzazione di thread

i thread possono comunicare direttamente, ma devono devono essere sincronizzati causa race condition.
la sincronizzazione è spesso necessaria anche della gestione dei segnali

mutex

se thread prova a bloccare mutex già bloccato viene sospeso, riprende al liberarsi del mutex

`pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;` inizializza il mutex

in seguito
pthread_mutex_lock(&mutex);
sezione critica che può essere eseguita un thread alla volta
pthreaf_mutex_unlock(&mutex);

condition variable

condition variable segnala un evento, un thread può sospendersi in attesa di qualcosa sulla condition variable oppure un thread può svegliare thread in attesa per segnalare

`pthread_cond_t var = PTHREAD_COND_INITIALIZER;` inizializza la condition variable

successivamente posso:
`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`

cond: puntatore alla variabile
mutex: puntatore al mutex