



Laboratorio di sistemi operativi – T4

Nozioni integrative sulla programmazione in linguaggio C

Il programma

1. Introduzione a UNIX
2. **Nozioni integrative del linguaggio C**
3. controllo dei processi;
4. pipe e fifo;
5. code di messaggi;
6. memoria condivisa;
7. semafori;
8. segnali;
9. introduzione alla programmazione bash



Dal programma C al file eseguibile

Le opzioni di gcc

File sorgenti C



Sorgenti C dopo il preprocessing

1. C-preprocessor



```
#define PI 3.14
#include <stdio.h>
int main(){
    printf("Hello
World!\n");
}
```

```
int stdin=0;
int stdout=1
int stderr=2;
int main(){
    printf("Hello
World!\n");
}
```

File in linguaggio assemblativo

2. C-compiler



```
MOV EAX
JUMP main
LOAD
...
```

File oggetto



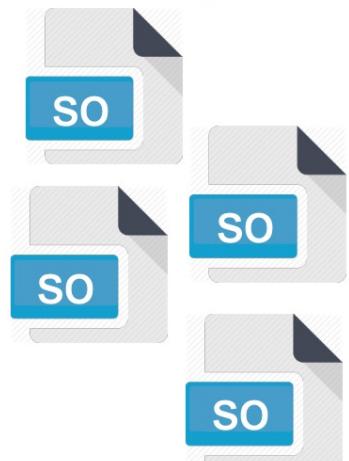
```
0100101001001
0010010010010
1011010101000
11011011
...
```

File eseguibile



```
0100101001001
0010010010010
1011010101000
11011011
...
```

4. linker



```
0100101001001
0010010010010
1011010101000
11011011
...
```

Librerie

1-Preprocessore

- Riceve in input un programma scritto in C (in realtà, non è un obbligo)
- Restituisce in output un altro file di testo dove ogni direttiva presente è stata interpretata e rimpiazzata/espansa con il testo corrispondente
- Direttive
 - `#define`
 - `#include`
 - `#if, #ifdef, #ifndef`

Direttiva #include

- `#include <nomefile.h>`
 - Inserisce nel punto in cui si trova la direttiva il contenuto del file nomefile.h
 - La ricerca avviene solo nelle cartelle di sistema (es: /usr/include), e non nella directory corrente
- `#include "filename.h"`
 - La ricerca del fine avviene prima nella directory corrente e poi nelle cartelle di sistema

```
gcc -E filename.c  
(per vedere l'output del preprocessore)
```

```
gcc -I /home/schi/mylibs  
(per aggiungere una cartella di ricerca)
```

Direttiva #include

- La libreria standard (glibc) offre diverse librerie che useremo nel corso di questo corso
 - stdio.h
 - string.h
 - math.h
 - errno.h
 - limits.h
 - stdlib.h
 - ctype.h
 - ...

`ldd eseguibile`

Permette di conoscere le librerie utilizzate da un eseguibile

```
cp /usr/lib/aarch64-linux-gnu/libc.a .
ar x libc.a
ls -l | grep printf
```

Contenuto della libreria libc (dipende dall'architettura)

Direttiva #define

- Permette di definire un nome simbolico per la definizione di una costante o una macro
- Per convenzione, i nomi simbolici si definiscono con le lettere maiuscole dell'alfabeto
- `#define` è usata anche per definire macro parametriche

```
#define VEC_LEN 80 int  
v[VEC_LEN], i;  
  
for (i=0; i<VEC_LEN; i++) {  
    /* something */  
}
```

```
#define SQUARE(x) x*x  
a = SQUARE(2)+SQUARE(3);
```

```
#define SUM(x,y) x+y  
a = SUM(1,2)*SUM(1,2);
```

```
#define SUM(x,y) ((x)+(y))  
a = SUM(1,2)*SUM(1,2);
```

Direttiva #define

- Le direttive solitamente sono composte da una sola linea
- E' possibile definirne anche con più linee
- I parametri di una macro, se indicati con #, vengono sostituiti con la stringa del nome stesso e non con il valore

```
#define EXCHANGE(type,a,b) { \  
    type aux ; \  
    aux = a; \ a = b; \  
    b = aux; }
```

```
#define PRINT_INTV(v) printf("%s=%i\n",#v,v);  
  
PRINT_INTV(var1);
```

Direttiva #define

- Una macro può essere anche definita a runtime quando si invoca il compilatore gcc

```
gcc -D PI=3.14  
(equivalente a #define PI 3.14 nel file)
```

- E' possibile definire una costante senza valore (utile per le direttive condizionali)

```
#define DEBUG
```

#define : macro predefinite

- __LINE__
 - A decimal constant representing the current line number.
- __FILE__
 - A string representing the current name of the source code file.
- __DATE__
 - A string representing the current date when compiling began for the current source file. It is in the format "mmm dd yyyy", the same as what is generated by the asctime function.
- __TIME__
 - A string literal representing the current time when compiling began for the current source file. It is in the format "hh:mm:ss", the same as what is generated by the asctime function.

Inclusioni condizionali con #if, #ifdef e #ifndef

- E' possibile includere porzioni di codice in base
 - Alla valutazione di una espressione che contiene costanti definite prima

```
#if CONST > 10
    /* codice incluso se vero*/
#else
    /* codice incluso se falso*/
#endif
```

- Alla definizione (o meno) di una macro
 - Usato anche per evitare inclusioni cicliche (vedere le slides successive)

```
#ifdef PLUTO
    /* codice inserito se
    PIPPO è definito*/
#endif
```

```
#ifndef PLUTO
    /* codice inserito se
    PIPPO non è definito*/
#endif
```

Inclusioni condizionali con #if, #ifdef e #ifndef

```
gcc -D DEBUG=1
```

```
#include <assert.h>

...
list add_node(list list_ptr, int value) {
#ifdef DEBUG
    assert(list_ptr != NULL);
#endif
node* new_elem = malloc(sizeof(node));
...
return list_ptr;
}
```

2-Compilazione

- Il compilatore si occupa di tradurre l'output del preprocessore in un file di testo contenente una sequenza di istruzioni assembly
- Per fermare l'esecuzione dopo la compilazione:

```
gcc -S filename.c
```

Opzioni di compilazione

- ci sono moltissime opzioni:
 - `-g` aggiunge le info di debug (usato per `gdb`)
 - `-O2 -Os -O0` per i vari livelli di ottimizzazione
 - `-std=c89` seleziona lo standard ANSI C (1989)
 - Variabili dichiarate solo all'inizio di un blocco (ad esempio, `for (int i=0; i<10; i++)`, non rispetta lo standard
 - I commenti inline non sono ammessi (solo `/* */`)
 - `-Wall` visualizza tutti i warnings
 - `-pedantic` non compila programmi non conformi con lo standard ANSI C
 - `-m32 -marm` per compilare su altre architetture
 - `-lm` per utilizzare le librerie matematiche

```
gcc -S -g -O0 filename.c
```

3-Assemblatore

- Prende in input il file testo del compilatore
- Produce il file binario oggetto
- Per fermare la compilazione dopo l'assemblatore

```
gcc -c filename.c
```

- Per vedere il contenuto del file oggetto

```
hexdump -C filename.o
```

4-Linking

- Prende in input uno o più file oggetto
 - Un solo file in input deve avere definita la funzione main ()
- Restituisce un file eseguibile chiamato a.out
 - Con l'opzione -o il nome può essere modificato

```
gcc filename.o filename2.o -o eseguibile
```

```
gcc filename.c -o eseguibile
```



Classi di memorizzazione

Classi di memorizzazione

- tutte le variabili e le funzioni del C hanno 2 attributi
 - la classe di memorizzazione
 - il tipo
- le 4 classi di memorizzazione possibili sono le seguenti
 - auto
 - extern
 - register
 - static

Classi di memorizzazione: auto

- È la classe di memorizzazione di default e quindi può essere omessa
- all'ingresso in un blocco il sistema alloca memoria per le variabili automatiche; pertanto tali variabili sono considerate locali al blocco
- all'uscita dal blocco, il sistema libera la memoria assegnata alle variabili automatiche, causando la perdita dei loro valori
- al rientro nel blocco, il sistema alloca nuovamente la memoria senza però recuperare i valori precedenti

Classi di memorizzazione: extern

- Variabili definite e allocate in altri file
 - Un altro programma
 - Il Sistema Operativo
 - ...
- Anche le funzioni possono essere definite extern
 - `extern int variabile`
- Il compilatore assume che questa variabile esiste (e non alloca spazio per essa)
- Il linker può generare un errore

Classi di memorizzazione: register

- classe di memorizzazione che ha come obiettivo l'aumento della velocità di esecuzione
- segnala al compilatore che la variabile corrispondente dovrebbe essere memorizzata in registri di memoria ad alta velocità
- qualora il compilatore non possa allocare un registro fisico, viene utilizzata la classe automatica (il compilatore dispone solo di una parte dei registri, che possono invece essere utilizzati dal sistema)
- quando la velocità è importante, il programmatore può scegliere poche variabili alle quali viene fatto più frequentemente accesso (per esempio, variabili di ciclo e parametri delle funzioni)
- la variabile register è di norma dichiarata nel punto più vicino possibile al punto in cui viene utilizzata, per consentire la massima disponibilità di registri fisici, utilizzati solo quando necessario

```
register int i;  
for(i = 0; i < LIMIT; ++i)  
    ...
```

Classi di memorizzazione: static

- variabili static servono per permettere a una variabile locale di mantenere il valore precedente al rientro in un blocco

```
void f(void) {  
    static int count = 0;  
    ++count;  
    ...  
}
```

- alla prima chiamata della funzione `count` viene inizializzata a zero;
- alle chiamate successive non viene più inizializzata, ma mantiene il valore che aveva alla precedente chiamata di funzione

Classi di memorizzazione: static extern

- questo tipo di classe di memorizzazione fornisce meccanismo di ‘privatezza’ (insieme di restrizioni sulla visibilità di variabili o funzioni che sarebbero altrimenti accessibili) fondamentale per la modularità dei programmi
- le variabili statiche esterne sono variabili esterne con visibilità ristretta: sono accessibili dal resto del file in cui sono dichiarate
- non sono disponibili
 - alle funzioni precedentemente definite all’interno del file
 - alle funzioni definite all’interno di file differenti

```
void f(void) {  
    ... // v non disponibile  
}  
  
static int v;  
  
void g(void) {  
    ... // v disponibile  
}
```

l’idea è cioè di disporre di una variabile globale per una famiglia di funzioni, e al contempo privata per il file

Classi di memorizzazione e valore di default

- sia le variabili **statiche** sia quelle **esterne** non inizializzate esplicitamente vengono **inizializzate a zero** dal sistema
- anche array, stringhe, puntatori, strutture e union subiscono lo stesso trattamento: per array e stringhe ogni elemento viene posto a zero
- al contrario, normalmente le variabili **automatiche** e **registro** non vengono inizializzate dal sistema, e quindi contengono inizialmente valori «sporchi»

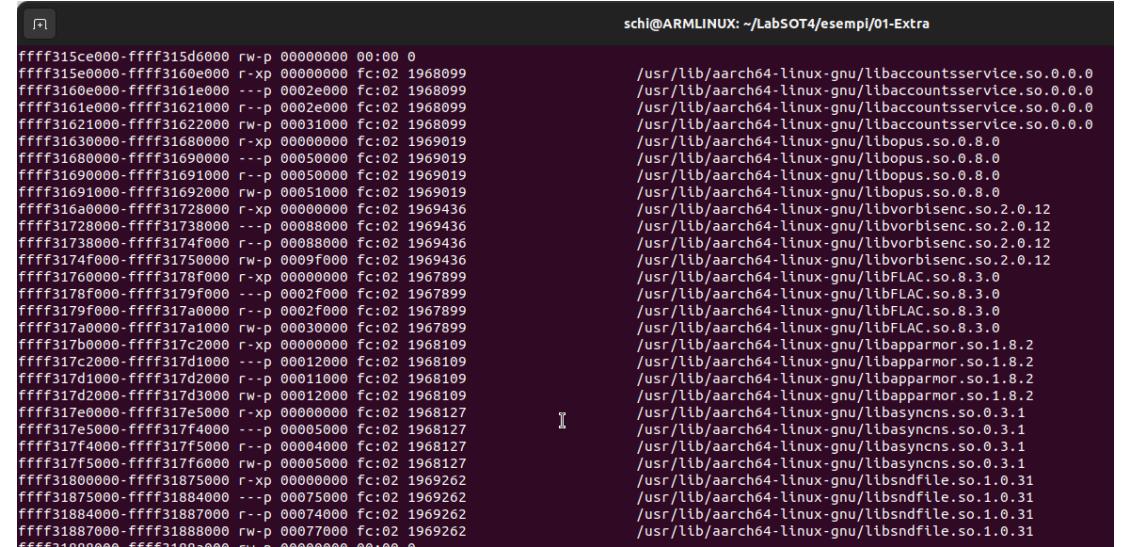
L'allocazione delle variabili

- Normalmente, le variabili sono salvate in memoria
 - BSS (Block Standard by Symbol)
 - Stack
 - Heap
- Inoltre, possono essere salvate nei registri

I segmenti di memoria

- Il sistema operativo è responsabile dell'assegnamento di alcuni segmenti di memoria ad ogni processo
- I segmenti di memoria sono mappati nello spazio di indirizzamento del processo
- Ogni segmento ha
 - Indirizzo di inizio e di fine
 - Flags che determinano le modalità di accesso
 - Read, write, execute (contiene il codice), privato/condiviso

```
ps -aux | grep <nome_processo>
cat /proc/<pid>/maps
```



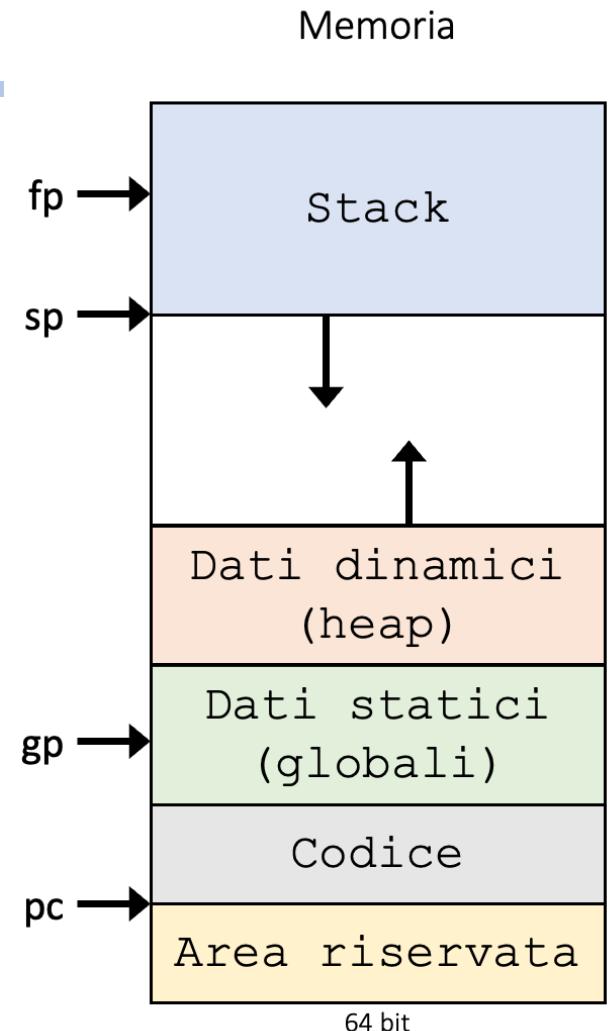
```
schl@ARMLINUX: ~/LabSOT4/esempi/01-Extra
ffff315ce000-ffff315d6000 rw-p 00000000 00:00 0
ffff315e0000-ffff3160e000 r-xp 00000000 fc:02 1968099
ffff3160e000-ffff3161e000 ---p 0002e000 fc:02 1968099
ffff3161e000-ffff31621000 r--p 0002e000 fc:02 1968099
ffff31621000-ffff31622000 rw-p 00031000 fc:02 1968099
ffff31630000-ffff31680000 r-xp 00000000 fc:02 1969019
ffff31680000-ffff31690000 ---p 00050000 fc:02 1969019
ffff31690000-ffff31691000 r--p 00050000 fc:02 1969019
ffff31691000-ffff31692000 rw-p 00051000 fc:02 1969019
ffff316a0000-ffff31728000 r-xp 00000000 fc:02 1969436
ffff31728000-ffff31738000 ---p 00088000 fc:02 1969436
ffff31738000-ffff3174f000 r--p 00088000 fc:02 1969436
ffff3174f000-ffff31750000 rw-p 0009f000 fc:02 1969436
ffff31760000-ffff3178f000 r-xp 00000000 fc:02 1967899
ffff3178f000-ffff3179f000 ---p 0002f000 fc:02 1967899
ffff3179f000-ffff317a0000 r--p 0002f000 fc:02 1967899
ffff317a0000-ffff317a1000 rw-p 00030000 fc:02 1967899
ffff317b0000-ffff317c2000 r-xp 00000000 fc:02 1968109
ffff317c2000-ffff317d1000 ---p 00012000 fc:02 1968109
ffff317d1000-ffff317d2000 r--p 00011000 fc:02 1968109
ffff317d2000-ffff317d3000 rw-p 00012000 fc:02 1968109
ffff317e0000-ffff317e5000 r-xp 00000000 fc:02 1968127
ffff317e5000-ffff317f4000 ---p 00005000 fc:02 1968127
ffff317f4000-ffff317f5000 r--p 00004000 fc:02 1968127
ffff317f5000-ffff317f6000 rw-p 00005000 fc:02 1968127
ffff31800000-ffff31875000 r-xp 00000000 fc:02 1969262
ffff31875000-ffff31884000 ---p 00075000 fc:02 1969262
ffff31884000-ffff31887000 r--p 00074000 fc:02 1969262
ffff31887000-ffff31888000 rw-p 00077000 fc:02 1969262
ffff31888000-ffff31889000 r--p 00000000 00:00 0
```

Variabili nel BSS

- Segmento di memoria read/write
- Dimensione decisa a tempo di compilazione
- Vi sono memorizzate
 - Variabili globali
 - Variabili locali static
- Allocate all'inizio del programma
- Deallocate alla fine del programma

Variabili nello stack

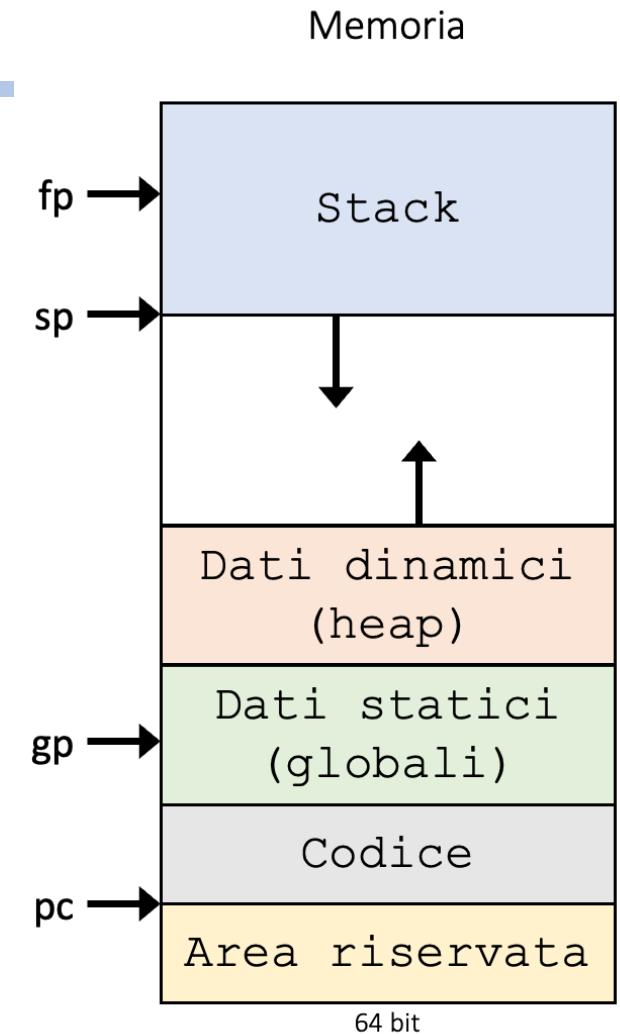
- Variabili dichiarate all'interno di una funzione (compresi i parametri)
- Viene modificato lo stack pointer
- Allocate all'inizio della funzione
- Deallocate quando la funzione termina



Dal corso di Architetture...

Variabili nello heap

- Allocazione dinamica, decisa a runtime
- Utilizzato quando si richiede uno spazio di memoria per una variabile attraverso le primitive come
 - `void * malloc(size_t size);`
 - `void * calloc(size_t nmemb , size_t size);`
 - `void * realloc(void *ptr, size_t size);`
- NB: la memoria allocata dinamicamente va deallocata!
 - `void free(void *ptr);`



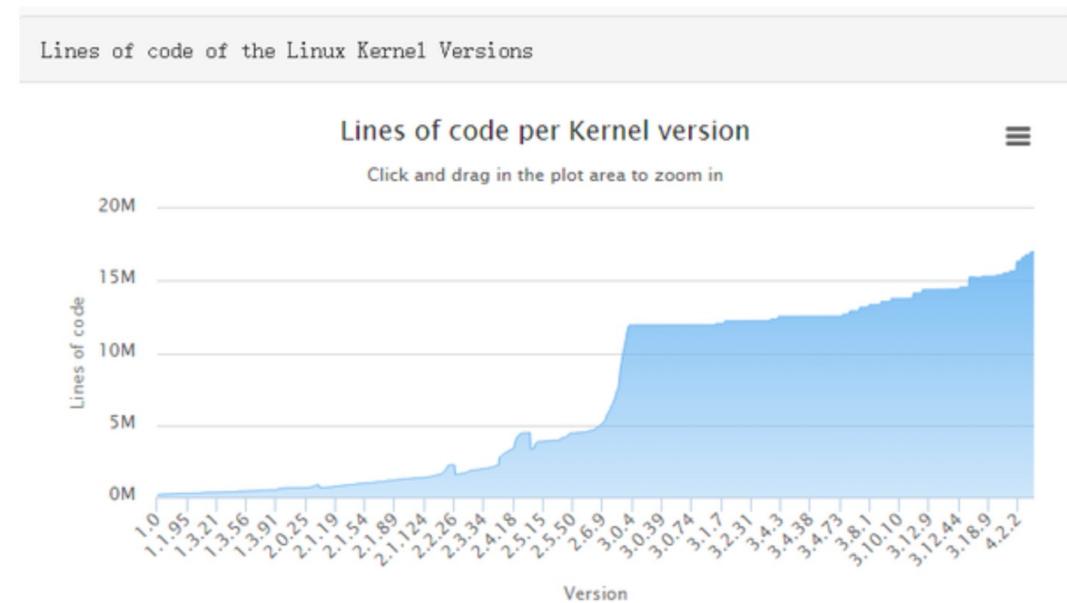
Dal corso di Architetture...



Programmazione modulare

Programmazione modulare

- Al giorno d'oggi, i progetti software
 - Sono complessi e composti da moltissime righe di codice
 - Si basano sul riuso di funzionalità (library) scritte da altri programmati
- Avere quindi il codice di un programma scritto su un unico file è impraticabile
- Una piccola modifica in un unico punto non può obbligare ad una ricompilazione di tutto il progetto



Programmazione modulare

- Raggruppare le funzioni che si occupano di una specifica funzionalità in un unico file sorgente → un modulo
- Un modulo è composto da
 - Interfaccia (header file)
 - Lista delle funzioni e dei tipi di dato del modulo
 - Viene incluso in ogni modulo/applicazione che lo usa (#include)
 - Non deve essere mai compilato
 - Implementazione
 - Codice che implementa le funzioni contenute nell'header file
 - Codice che implementa le funzioni interne che non sono esposte nell'interfaccia
 - Non contiene la funzione main()
 - Compilato con gcc –c e produce un file oggetto .o
- Applicazione
 - Utilizza uno o più moduli includendo i corrispondenti header file
 - Deve contenere la funzione main()
 - Compilato e linkato con il file oggetto dei moduli

Programmazione modulare

```
#include "operazioni.h"

int main(void) {
    int i = 3;
    int j = 5;

    printf("i+j = %d\n", somma(i,j) );
    printf("i*j = %d\n", moltiplica(i,j) );
}
```

applicazione.c

```
int moltiplica(int primo, int secondo);
int somma(int primo, int secondo);
```

operazioni.h

```
#include "operazioni.h"

int moltiplica(int primo, int secondo) {
    return primo*secondo;
}

int somma(int primo, int secondo) {
    return primo+secondo;
}
```

operazioni.c

Programmazione modulare: esercizio

- Scrivere un programma che dati due numeri interi passati come argomenti, restituisca la loro somma e la loro moltiplicazione
- Organizzare il codice in tre componenti separati
 - Modulo che contiene la funzione per la somma
 - Modulo che contiene la funzione per la moltiplicazione
 - Applicazione che contiene il main e sfrutta le funzioni dei primi due moduli

Programmazione modulare

```
#include "somma.h"
int somma(int primo, int secondo) {
    return (primo+secondo);
}
```

somma.c

```
#include "prodotto.h"
int moltiplica(int primo, int secondo) {
    return (primo*secondo);
}
```

prodotto.c

```
int somma(int primo, int secondo);
```

somma.h

```
int moltiplica(int primo, int secondo);
```

prodotto.h

```
#include "somma.h"
#include "prodotto.h"

int main() {
    int i = 3;
    int j = 5;
    printf("i+j = %d\n", somma(i,j));
    printf("i*j = %d\n", moltiplica(i,j));
}
```

Compilazione dei moduli

- i singoli moduli, cioè i file .c possono essere compilati in file oggetto specificando il parametro -c
- esiste una dipendenza fra i file .h e il file .c che li include: se un file di intestazione cambia, allora i moduli dipendenti devono essere ricompilati

```
gcc -c somma.c
gcc -c prodotto.c
gcc -c applicazione.c

$ gcc somma.o prodotto.o applicazione.o -o applicazione
```

Ciclicità delle dipendenze

```
#include "pippo.h"  
  
type pluto(arg1,arg2);  
  
pluto.h
```



```
#include "pluto.h"  
  
type pippo(arg1,arg2);  
  
pippo.h
```

```
gcc -o applicazione.c
```

Loop!



```
#include "pippo.h"  
  
int main() {  
...  
}  
  
applicazione.c
```

Ciclicità delle dipendenze

```
#ifndef __PLUTO_H__  
#define __PLUTO_H__  
  
#include "pippo.h"  
  
type pluto(arg1,arg2);  
#endif
```

pluto.h

```
#ifndef __PIPPO_H__  
#define __PIPPO_H__  
  
#include "pluto.h"  
  
type pippo(arg1,arg2);  
#endif
```

pippo.h

Soluzione!

```
gcc -o applicazione.c
```

```
#include "pippo.h"  
  
int main() {  
...  
}
```

applicazione.c

Compilazione dei moduli

- se il progetto consiste di centinaia di files, può tuttavia essere utile la compilazione separata: ma come determinare quali moduli ricompilare?
- dimenticando di ricompilarne qualcuno, può capitare che il linker generi errori nel tentare di utilizzare un file oggetto obsoleto;
- peggio, se la signature non cambia, l'errore non sarà segnalato preventivamente, provocando errori a runtime
- possiamo risolvere tali problemi con l'utility [make](#).



Le variabili di ambiente ed il linguaggio C

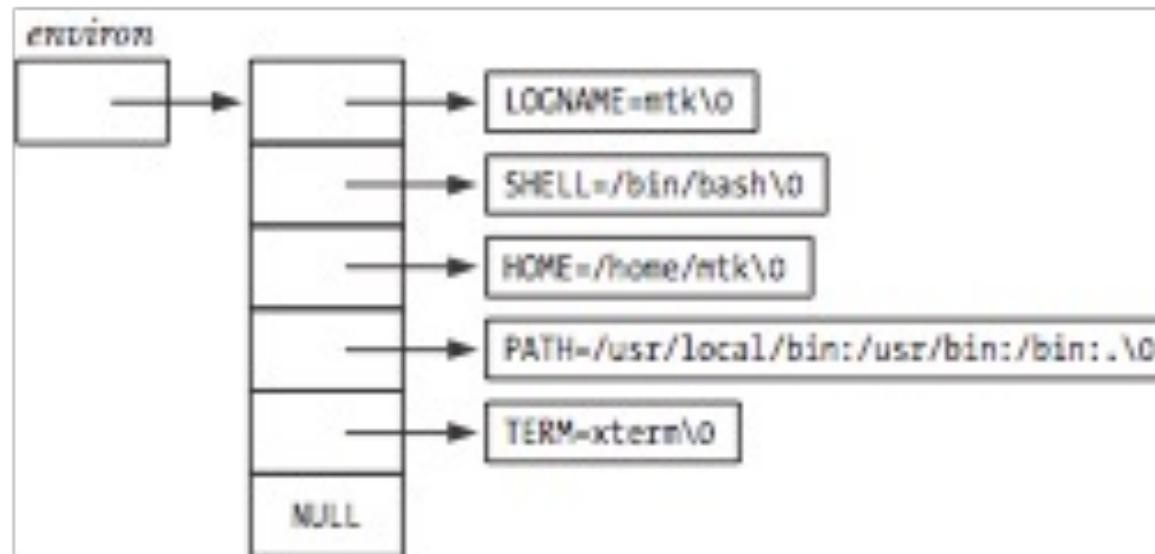
Variabili di ambiente

- Ogni processo ha associato un array di stringhe che contiene l'elenco delle variabili di ambiente
- Ogni elemento dell'array ha la forma nome=valore (esempi: HOME, LOGNAME, PATH, ...)
- Ogni volta che un processo viene creato, eredita le variabili di ambiente del processo creatore

```
export USERVER=value  
printenv  
echo $USERVER
```

Accesso alle variabili di ambiente

- All'interno di un programma C, la lista delle variabili di ambiente può essere consultata usando la variabile globale
 - `char **environ`
 - La variabile `environ` è un NULL-terminated array



Accesso alle variabili di ambiente

- La funzione
 - `char * getenv (const char *name)`
 - Restituisce la stringa della variabile di ambiente passata come parametro

```
#include <unistd.h>

extern char **environ;
// di qui in poi è possibile utilizzare
// environ
```

Make

- la make-utility è uno strumento che può essere utilizzato per automatizzare il processo di compilazione
- in generale è più flessibile degli ambienti integrati (IDE, integrated development environments)
- si basa sull'utilizzo di un file (chiamato `makefile`) che descrive le dipendenze presenti nel progetto
- è quindi possibile utilizzare il comando `make` che si avvale della marcatura temporale dei files e ricompila i target file che sono più vecchi dei sorgenti

Make – regole implicite

- Regole implicite
 - make test
 - «test» è il target
- Se non vi sono regole esplicite definite (vedi slide successive), make tenta di creare un eseguibile cercando il file «test.c» oppure «test.o»
 - gcc test.c -o test
- Possibile anche solamente la compilazione senza il linking
 - make test.o

Make – regole esplicite

- Il makefile elenca un insieme di target rules: regole per la compilazione e l'istruzione da eseguire per compilare a partire dai sorgenti

```
nome_target: file1.o file2.o filen.o
    gcc -o nome_target file1.o file2.o filen.o
file1.o: file1.c file1.h
    gcc -c file1.c
...
filen.o: filen.c filen.h
    gcc -c filen.c
clean:
    rm -f *.o
```

make nome_target

Make

make nome_target

make nome_target -f miomakefile

- nell'invocare make da linea di comando è possibile specificare quale target compilare
 - make file1.o
- Se viene invocato senza opzione dalla linea di comando, utilizza il target di default, il primo specificato nel makefile
 - make
- Nell'esempio qua sotto, per cancellare tutti i file oggetto
 - make clean

```
nome_target: file1.o file2.o filen.o  
    gcc -o nome_target file1.o file2.o filen.o  
...  
clean:  
    rm -f *.o  
run:  
    ./nome_target
```

makefile

Make – variabili di ambiente

- Si possono definire alcune variabili di ambiente (anche internamente al makefile) per definire:

- CC
 - Il comando da usare per il compilatore
- CFLAGS
 - I flags da usare durante la compilazione
- LDFLAGS
 - I flags per la fase di linking

```
export CC="gcc"
```

```
export CFLAGS= "-std=c89 -pedantic"
```

```
export LDFLAGS= "-lm"
```

Make: esercizio

- Creare un makefile per l'esercizio precedente
- Verificare che su invocazioni consecutive di `make`, vengono compilati solamente i file effettivamente modificati
- Per modificare un file in maniera «fittizia» usare il comando
 - `touch <filename>`

Esercizio

- Creare un programma che
 - Stampa la lista delle variabili di ambiente definite
 - prende in input il nome di una variabile di ambiente e stampa a video il suo valore
 - Il programma deve anche controllare la presenza del parametro prima di procedere con la ricerca della variabile di ambiente