LabSO 2025

Laboratorio Sistemi Operativi - A.A. 2024-2025

Michele Grisafi - michele.grisafi@unitn.it

C - introduzione

Perchè C?

- Struttura minimale
- 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** e 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>                          //size.c
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));
}
```

A common standard

La libreria `stdint.h` definisce dei tipi standard dalla dimensione più esplicita:

- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`

Ne vengono definiti molti altri, ma questi sono i più utilizzati.

Structs e Unions

Le structs permettono di aggregare diverse variabili, mentre le unions permettono di creare dei tipi generici che possono ospitare uno tra 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”

Gli enumeratori sono delle variabili che possono avere uno tra una serie di valori statici predefiniti.

```c
#include <stdio.h>

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

Bool

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

enum boolean { FALSE = 0, TRUE = 1 }
void main() {
  bool t = true; // == (int) 1
  bool f = false; // == (int) 0

  enum boolean my_t = TRUE; // == (int) 1
  enum boolean my_f = FALSE; // == (int) 0

  printf("%d %d %d %d", t,f,my_t,my_f);
}
```

Puntatori di variabili

C si sviluppa attorno all'uso di puntatori, ovvero degli alias per zone di memoria 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.

`int * whereIsValore = &valore;`

Esempi:

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

Puntatori di variabili

```
int     i     =     42;
int   *   punt   =   &i;
int    b    =    *(punt);
int   **   p2   =   &punt;
int   ***   p3   =   &p2;
***p3=50;
```

Tipo	Nome	Valore	Indirizzo (stack)
int	i	42	0x602ada38
int *	punt	0x602ada38	0x602ada40
int	b	42	0x602ada3c
int **	p2	0x602ada40	0x602ada48
int ***	p3	0x602ada48	0x602ada50

```
i = 20;
```

Nome	Valore
i	**20**
punt	0x602ada38
b	**?**

Puntatori di funzioni

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

Sintassi simile ma diversa!

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

```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;
}
```

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`

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.

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)

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`

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!

Main.c e conversione tra int e float

```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 0;
}
```

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.

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 - 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).

NB: `arr` è una variabile di tipo puntatore a int (`int *`) che può essere chiamata anche con le parentesi quadre `arr[x]` per indicare l'accesso alla posizione x.

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 firma 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";   // 5 caratteri!!
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*

Read only strings

Sebbene ci siano diversi modi per dichiarare ed inizializzare una stringa, questi hanno comportamenti diversi:

```
char string[] = “ciao”; // writable string in the stack
char * string_read_only = “ciao; // READ-ONLY string
string[2] = ‘a’;
string_read_only[2] = ‘a’; //Segmentation fault!
```

Nel secondo caso, il compilatore salva la stringa in un’area di memoria read only.

C - esempio carattere e argc/argv

```c
#include <stdio.h>                                  //argc.c

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 - 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 * strncat(char *dest, const char *src,size_t n)`
aggiunge al più `n` caratteri di `src` in coda a `dest`, spostando lo `\0` di dest in fondo.

`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`, restituendo 0 se sono uguali, -1 o 1 se `str1` è rispettivamente più grande o più piccola di `str2`

C - funzioni stringhe `<string.h>`

`size_t strlen(const char *str)`

calcola la lunghezza di `str` senza contare il carattere finale `\0`

`char * strncpy(char *dest, const char *src, size_t n)`

copia al più `n` caratteri dalla stringa `src` in `dst`

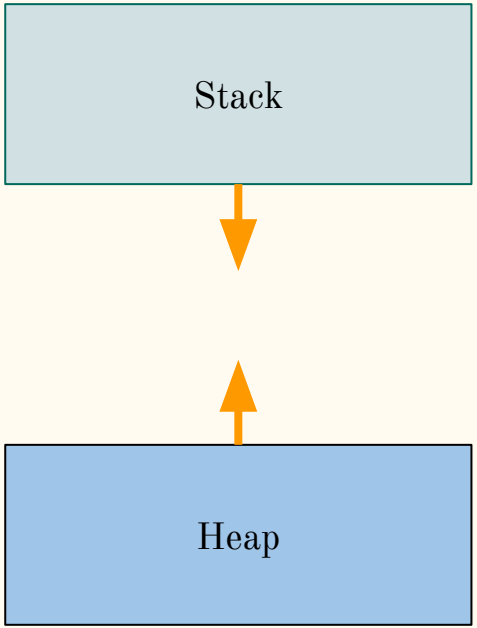
Conversione di stringhe in numeri

```
int atoi(const char * str);
int sscanf(const char *str, const char *format, int * dst);
long int strtol(const char *str, char ** endptr, int base);
```

Atoi è il meno affidabile, sscanf il più versatile.

```
char * ptr; //It will point to the first non convertible char
char * string = “ciao come va”;
long number = atoi(string);
sscanf(string,”%d”,&number);
number = strtol(string,&ptr,10);
```

Memoria dinamica



La memoria di un processo viene divisa in diversi segmenti, ognuno dedicato ad dei dati differenti. Due segmenti molto importanti sono lo **stack** e la **heap**.

Stack: contiene tutte le variabili locali delle varie funzioni. Ogni volta che una funzione viene chiamata, lo stack viene incrementato.

Heap: contiene le variabili dinamiche, allocate a run-time. Può crescere o diminuire a seconda delle operazioni che l'utente richiede.

Quando ci serve la heap?

Originariamente era necessaria per ogni allocazione dinamica, come per esempio:

```
int var_size = 30;
int var_array[var_size]; //It used to throw error!
```

Adesso, grazie alla VLA (Variable Length Array), queste operazioni sono supportate. Tuttavia, quando usciamo dalla funzione tutte le variabili locali vengono distrutte → la heap ci consente di condividere variabili tra più funzioni, senza abusare delle variabili globali.

Malloc e free

`void * malloc(size_t size)`

Alloca size bytes in memoria e restituisce il puntatore a quella zona di memoria.

`void free(void * pnt)`

Libera la zona puntata da `pnt`

NB: è importante deallocare (free) tutte le variabili che non servono più, una ed una sola volta! Ci sono delle vulnerabilità che nascono dall'uso improprio di queste istruzioni (double-free, use-after-free)

Heap e globals

La variabile viene allocata in una funzione, ed il puntatore ha validità fino a che non viene espressamente distrutta con `free`

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

int global_var = 3;
int * get_heap_var();

int main(void) {
  printf("Global: %d\n", global_var);
  int * pnt_int = get_heap_var();
  printf("Heap: %d\n", *pnt_int);
  free(pnt_int);
}
int * get_heap_var(){
  int * new_pnt = (int *)malloc(sizeof(int));
  *new_pnt = 5;
  return new_pnt;
}
```

C - esercizi

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 senza usare strlen.

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)