LabSO

Fondamenti linguaggio C e processi

C: caratteristiche

- **Struttura minimale**
- **Pochissime parole chiave (meno del Pascal ad esempio)**
- **Facile da apprendere**
- **Unix compliant**
- **Alla base di Unix (nato per scrivere Unix)**
- **Molto portabile “sintatticamente”, anche se poco portabile “logicamente“**
- **Organizzazione a passi: sorgente → intermedio (es. Assembly) → oggetto → eseguibile**
- **Disponibilità librerie (standard + extra)**
- **Nessuna struttura di alt(issim)o livello (tipo “decorators” o altro)**

C: compilazione

- **Il testo del programma viene elaborato dal compilatore**
- **Il compilatore legge il testo una sola volta.**
- **Commenti multlinea o semplici**
- **Comandi per il pre-processore: “direttive” (DIPENDENTI DAL TOOL di compilazione)**
- **Un programma C:**
  - eseguibile ha una funzione “main” dove inizia
  - Impostato come libreria non ha un “main”
- **Componenti:**
  - Preprocessore [gcc -E]:
    - elabora i comandi a lui riservati (direttive “#”) e GENERA il “sorgente effettivo”
  - Compilatore (propriamente detto) [gcc -S]:
    - traduce da “C” ad Assembly (“.a”)
    - genera i file “oggetto” (“.o”)
  - Linker [gcc]:
    - combina i file generati allo step precedente con le librerie (generazione eseguibile)

C: direttive

- **Il preprocessore manipola “letteralmente” il codice: da sorgente a sorgente**
- **“Elabora” le righe che iniziano con il carattere “hash” (valido anche per i commenti) e un “comando” senza spazi accanto: sono commenti validi, ma se riconosciuti possono essere elaborati**
- **#include <file> oppure #include “file” (inclusione da percorso di sistema o relativo)**
- **#define NOME TESTO (crea una sorta di “alias” per sostituzione letterale)**
- **#define NOME(A,B,...) TESTO A e B (crea una sorta di “alias” per sostituzione con “parametri”)**
  - la sostituzione letterale implica che bisogna tener conto di eventuali “precedenze”
  - (es. #define MULT(X,Y) X*Y OPPURE #define MULT(X,Y) X*Y e poi utilizzo con printf(“%d\n”, MULT(1+1,3))
  - la traduzione avviene a tempo di compliazione, NON di esecuzione (diversamente da una funzione)
  - il termine del TESTO è la fine della riga (usare eventualmente “backslash” per multilinea)
- **#if CONDIZIONE - #ifdef NOME - #ifndef NOME - #else - #endif per controlli condizionali** (ad esempio per “ignorare temporaneamente” un blocco anche con commenti, considerando che i commenti annidati NON sono consentiti, si può circondarlo con #if 0 - #endif)
- **Esempio per debugging mediante #define DEBUG - #ifdef DEBUG - #endif**
- **Uso di gcc -DNOME=VAL per definizione con direttive dalla shell (es. gcc -DDEBUG)**

Processi

- **esempio:**
  - watch -n1 ‘date’& (vedere l’<id> in output)
  - ps -o pid,ppid,pgid,sid,uid,euid <id>
  - kill -9 <id>
- **Identificativi di processo:**
  - PID - Process ID
  - PPID - Parent Process ID
  - PGID - Process Group ID
  - SID - Session ID
  - RUID - Real User ID / EUID - Effective User ID

Processi: PID

- **Attributo di identificazione univoco**

Processi: PPID

- **Attributo di identificazione del processo genitore.**

  Il processo genitore (o padre) è quello che l'ha creato fino a che esiste, poi eventualmente può essere “ereditato” da un altro in base al metodo di implementazione (tipicamente “init”)

Processi: PGID

- **Attributo di identificazione del gruppo di appartenenza.**

  Un gruppo è un insieme di processi che possono interagire in maniera stretta tra loro. Al momento della creazione un processo entra nel gruppo del genitore.

Processi: SID

- **Attributo di identificazione della sessione.**

  Una sessione è un insieme di gruppi di processi definito per poterli gestire. Ogni gruppo appartiene a una sessione e alla creazione un processo è associato alla sessione del gruppo cui appartiene. Può modificare l'appartenenza volontariamente (funzione `setsid()`)

Processi: RUID/EUID

- **Attributo di identificazione dell'utente**

  Ogni processo è associato a un utente con abbinati una serie di permessi.
- RUID: l'utente che lancia il processo
- EUID: l'utente con cui il processo è eseguito

  Possono differire in casi particolari, ad esempio eseguendo il comando `passwd` per modificare la propria password: il bit "`suid`" (set owner user id) impostato (v. `ls -alh /usr/bin/passwd`) lo lancia come "`root`" (EUID) e in esecuzione il processo si preoccupa di limitare la possibilità di azione, entro quanto consentito

C: struttura

```
/*
Multi-line comment
*/

#include <stdio.h> // simple comment
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

- Organizzazione file(s)
- Suddivisione per righe
- Separatore (punto-e-virgola “;”)
- Commenti
- Direttive
- Blocchi (graffe)

C: funzioni/printf, direttive

- int printf(const char *format, ...)
  - format è %[flags][width][.precision][length]specifier
- direttive
  - #include <...> #include “...”
    - es. #include <stdio.h>
    - es. #include “lib.h”
  - #define WHAT description
    - es. #define NUMEROUNO 1

C: tipi

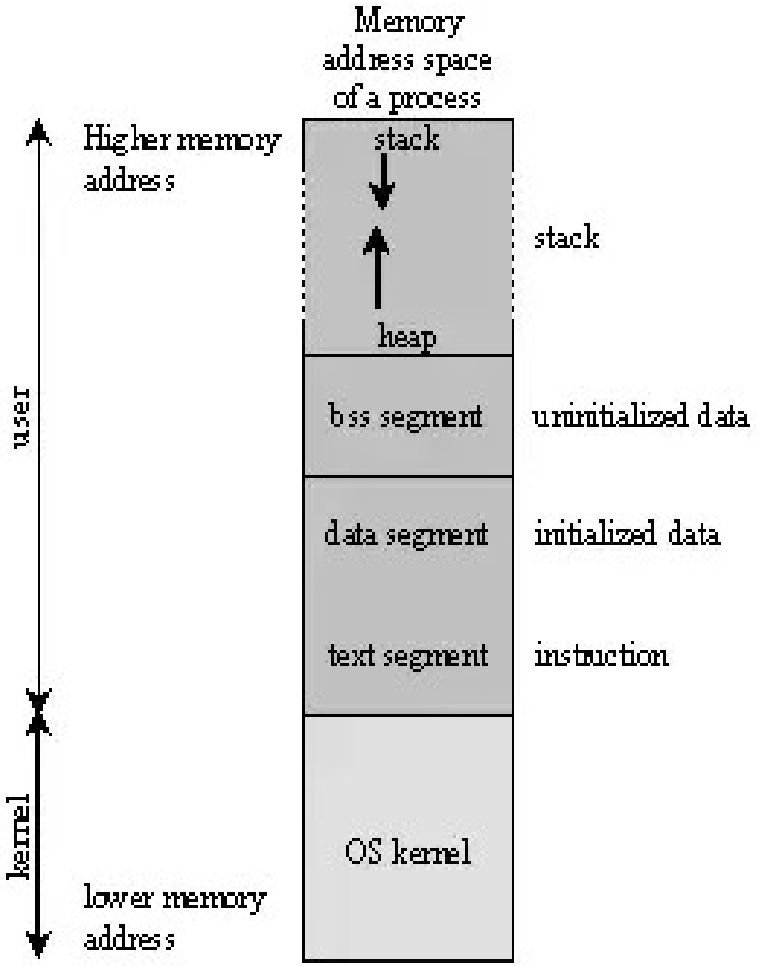
Ci sono diversi tipi con pochi di base dei quali ce ne interessano alcuni:

- Integers
  - char(0:255) “%c”
  - int (−32767:+32767) “%i” o “%d”
  - int (−2147483647:+2147483647) “%li”

- Boolean: NON ESISTE, si può implementare ad esempio con:
  - #define BOOL char
  - #define FALSE 0
  - #define TRUE 1

C: organizzazione memoria I

- **OS Kernel: indirizzi più bassi**
- **Text segment: istruzioni eseguite dalla CPU**
- **Data segment: inizializzazioni (es. int a=5;)**
- **BSS segment: variabili non inizializzate (es. int b; int b[100])**
- **Heap: dynamic custom allocation (es. malloc)**
- **Stack: dynamic automatic allocation (es. funzioni)**



C: organizzazione memoria II

- **Non ci sono primitive ad alto livello**
- **Non c’è garbage collection (!)**
- **Si usano tre tipi di memoria:**
  - Statica (indirizzi immutabili, ad esempio per “int a;”)
  - Dinamica personalizzata (es. malloc)
  - Dinamica automatica (es. dichiarazioni dentro un blocco o chiamate ricorsive)

C: organizzazione memoria III

- **Variabili statiche (dichiarazioni esplicite)**
  - inizializzate vanno nel segmento dati e nel file eseguibile
  - non inizializzate vanno nel segmento dati BSS azzerato a inizio esecuzione
- **Variabili dinamiche (tramite puntatori o new/dispose)**
  - create a run-time
  - se non inizializzate hanno un valore “casuale”
  - ad ogni malloc deve/dovrebbe corrispondere una free
- **Variabili automatiche (locali a una funzione)**
  - se non inizializzate hanno un valore “casuale”

C: stringhe

- **Una stringa è un vettore di caratteri che termina con ‘\0’**
- **possono essere considerati dei puntatori con accesso diretto o indiretto**

```
#include <stdio.h>
#define MAXLEN 80

int main(void) {
  char txt[MAXLEN]="Prova";
  char *c;
  int i;
  for (i=0;i<MAXLEN;i++) {
    c=txt+i;
    printf("%i, %c\n", i, *c);
    if (*c=='\0') break;
  };
  printf("strlen(%s)==%i\n", txt, i);
}
```

C: puntatori

Variabile come puntatore: ad esempio “a” punta ad una cella di memoria

- **& operator: &a è la locazione di memoria assegnata alla variabile “a”**
- *** operator: definisce un puntatore esplicito o accede alla locazione (es. “*b”)**
- `int *myVariable, myVariable2;` **versus** `int* myVariable, myVariable2;` **(equivalenti, ma nel primo è più “chiaro”: si lega alla variabile, non al tipo)**

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int a=7;
  int *b;
  b=&a;
  *b=3;
  printf ("%d (loc. %p, %p)\n", a, &a, b);
  return 0;
}
```

C: typedef

`typedef` **per denominazioni custom di tipi:**

```
#include <stdlib.h>
typedef <type> <alias>
```

Si può anche per puntatori, esempio:

```
#include <stdio.h>
#include <stdlib.h>
typedef char *string; // string è un puntatore a char
int main() {
  string txt="Ciao"; // equivale a char *txt="Ciao";
  printf("%s\n", txt);
  return 0;
}
```

C: struct - I

`struct` permette di costruire aggregati, ad esempio:

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

struct Books book1; // variable book1 of type "struct Books"
```

I campi della variabile sono accessibili con l'operatore infisso " . " (punto), esempio:

```
book1.title
```

C: struct - II

Si può usare `typedef` con `struct`:

```
typedef struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Book;

Book book1;
```

C: struct – III / strcpy

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

int main() {
  typedef struct Books {
    char title[50];
    char author[50];
  } Book;
  Book book;
  strcpy(book.title, "La Divina Commedia");
  strcpy(book.author, "Dante Alighieri");
  printf("%s (%s)\n", book.title, book.author);
  return 0;
}
```

C: union - I

`union` è simile a `struct`, ma anzichè creare un aggregato vero e proprio serve per gestire delle "alternative":

```
union <name> {
member definition;
member definition;
…
member definition;
} <variable>;
```

C: union - II

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;
   // n.b. i campi condividono la memoria, quindi usati insieme si ha un “conflitto”
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

C: puntatori / vettori

Confronto:

```
char *a;
char b[100];
```

- **Osservazione**: `*(a+i)` **e' equivalente a** `a[i]` **e** `i[a]`
  esercizio per casa: dimostrarlo
- **Dichiarazione:** `<type> <name>[<size>]`
- **Inizializzazione:** `<type> <name>[] = { val1, val2, … }`
  (osservazione: `<size>` **è dedotto automaticamente)**
- **Accesso:**
  `<name>[<index>]` **(singolo valore, accesso diretto)**
  `<name>` **(puntatore al primo elemento)**

C: memoria dinamica / casting

Usando `#include<stdlib.h>` si hanno:

`sizeof(<type>)` **per avere la dimensione occupata da un tipo di variabile**

`malloc` **per allocare** `<size>` **bytes di memoria:**

```
void *malloc(size_t size);
```

esempio: `ptr = (int*) malloc(100 * sizeof(int));` **(notare il "casting")**

`calloc` **per allocare lo spazio per** `<num>` **oggetti di dimensione** `<size>` **bytes:**

```
void *calloc(size_t nobj, size_t size);
```

`realloc` **per incrementare lo spazio precedentemente allocato:**

```
void *realloc(void *ptr, size_t newsize);
```

`free` **per liberare lo spazio già allocato:**

```
void free(void *ptr);
```

C: esercizio per casa

Implementare una versione custom della funzione standard `strcpy`:

```
char *strcpy(char *dest, const char *src)
```

che copia la stringa puntata da src in dest e restituisce quest'ultimo riferimento