Sistemi Operativi, Secondo Modulo, Canale M–Z
Riassunto della lezione del 20/04/2017

Igor Melatti

Il Linguaggio C

- Comandi `java` e `python`
  - i loro argomenti non sono eseguibili, ovvero non contengono istruzioni macchina direttamente eseguibili (o quasi) dalla macchina
- Comando `gcc`: differentemente dal compilatore Java (`javac`), genera eseguibili, che contengono istruzioni macchina
  - il risultato della compilazione di Java non contiene codice direttamente eseguibile dalla macchina, ma solo codice interpretabile dalla Java Virtual Machine
  - quindi, il vero processo è la Java Virtual Machine, cui poi si dà come argomento il `.class` da "interpretare"
  - analogo per python, dove però non c'è neanche la fase di compilazione (almeno, non sempre)
  - vale anche per altri linguaggi, come perl o quelli funzionali
  - invece, il risultato della compilazione con `gcc` è un file che, se lanciato, genera di per sé un processo, senza fare affidamento su nessun altro file eseguibile
- Comando eseguibile in Linux: o è il risultato di `gcc` (o di altro compilatore C/Assembler), o è uno script con lo shabang
  - fino a qualche tempo fa si poteva anche usare `gcc` per creare eseguibili a partire da Java, ma il supporto è finito nel 2016
- Per la storia di C, vedere lezione 1; nel 1983 fu creato lo standard C
  - almeno 2 versioni, con diversità nei dettagli: 1989 e 1999 (C89 o C90 e C99)
  - sarebbe lo *strict C*; `gcc` lo supporta solo se gli viene esplicitamente detto (opzioni `-std=c89`, `-std=c90`, `-std=c99`, che possono essere combinate con `-pedantic`)

  - lo standard C90 c'è solo per compatibilità all'indietro, ma già un file che include `stdio.h` non è compilabile con quello standard
  - se si compila senza precisare quale standard si sta usando, allora si sta usando `gnu90` o `gnu89` (sono uguali), che è un dialetto del C90 che usa anche alcune caratteristiche del C99; faremo riferimento a questo standard negli esempi

- Slides da `https://www.cs.clemson.edu/course/cpsc111/slides/`; vederle tutte, ma qui ci concentriamo su quelle più importanti

- Capitolo 2: importanti slide 6, 7, 8, 9, 16, 20, 23, 24
  - slide 4: vedere gli esempi allegati
    * solo preprocessing di un file: `cpp file.c > filepreprocessato.c`
      · via i commenti
      · tutte le direttive al preprocessore (quelle che iniziano con #) vengono eseguite; in particolare, tutti gli `#include <file>` e gli `#include "file"` vengono sostituiti dal contenuto di `file` (se `file` contiene a sua volta altre direttive di inclusione, vengono anch'esse sostituite dal contenuto del file corrispondente)
      · differenza tra `<>` e `""` per le inclusioni: nel primo caso si tratta di file di sistema (dentro `/usr/include`), nel secondo di file scritti dal programmatore (tipicamente, nella stessa directory del file da preprocessare)
    * solo compilazione: `gcc -c filepreprocessato.c -o file.o`
      · controlla che la sintassi sia a posto
      · per ogni chiamata a funzione, controlla che venga rispettato il corrispettivo header (che quindi deve esistere al momento della chiamata!)
      · il risultato è chiamato *file oggetto*
      · crea effettivamente del codice macchina, ma solo per il contenuto delle funzioni
      · ogni chiamata a funzione ha una destinazione simbolica
    * solo preprocessing + compilazione: `gcc -c file.c -o file.o`
    * solo linking (un solo file): `gcc file.o`
      · risolve tutte le chiamate a funzione: adesso, per ogni funzione chiamata non basta più l'header, ci deve essere anche l'implementazione (blocco di istruzioni)
      · l'implementazione può essere o data dal programmatore (vedere la funzione `f` nell'esempio allegato) o fornita da librerie di sistema (come per la funzione `printf`)

        - sulle librerie ci ritorneremo, per ora è importante notare che alcune librerie sono sempre usate di default per ogni link, altre vanno indicate espressamente
        - la libreria `libc.a`, che contiene l'implementazione di `printf`, è tra quelle che vengono usate automaticamente, senza che occorra passarla esplicitamente al linker
      * solo linking (più file): `gcc file1.o ... file`*n*`.o`
        - si possono anche mischiare file `.c` (sorgenti) e file `.o` (oggetti)
      * tutto (un solo file): `gcc file.c`
      * tutto (più file): `gcc file1.c ... file`*n*`.c`
      * **esercizio**: provare a fare in modo che `hello_world.1.c` sia incluso direttamente dentro `hello_world.2.c`. Si può ancora fare la compilazione separata (ovvero, prima si compilano separatamente `hello_world.1.c` e `hello_world.2.c`, e poi si linkano insieme)? Si può fare anche l'inclusione inversa?
  - slide 6: non necessariamente il main deve essere la prima funzione
  - slide 13: vuol dire avere la riga `#include <math.h>` e usare ad esempio le funzioni `sin` e `cos`
  - slide 16: vedere esempio `include.c` in lezione 16
  - slide 20: questa non è la dichiarazione della `printf`, ma solo un'indicazione di come viene tipicamente usata
  - slide 24: aggiungere anche `\f` (*line feed*): va a capo, ma senza ritornare all'inizio della riga (sotto Windows, invece, `\n` è solo line feed, e infatti l'andata a capo si fa con `\r\n`)
  - **esercizio:** spiegare l'output del seguente codice:

```
#include <stdio.h>

int main()
{
  printf("aaaaa\nbbbbbb\f\rccccc\r\fdddddd\reeeee\ffffff\n");
}
```

- Capitolo 3: tutte importanti, in particolare 8, 9, 10, 11, 12, 13
  - slide 8: in realtà, tutti e 3 gli optional modifiers possono essere presenti contemporaneamente
  - slide 10: vedere anche `https://en.wikipedia.org/wiki/C_data_types#Basic_types`