Informatica per Statistica
Riassunto della lezione del 10/10/2012

Igor Melatti

Introduzione alla sintassi del linguaggio C

- Caratteristiche fondamentali della *sintassi* del linguaggio C
  - cioè come si fa a scrivere un programma C di modo che poi il compilatore lo accetti senza dare errori (di sintassi, appunto)
- Un programma (scritto in linguaggio) C è costituito da una successione di *definizioni di funzioni*
  - ce ne deve essere esattamente una che si chiami `main`
  - si vedrà in seguito che ci possono essere anche altre cose (ad es. dichiarazioni di variabili globali)
- Per vedere com'è formata, in generale, una definizione di funzione, si consideri il programma C di Figura 1
  - in Figura 1 è presente una sola definizione di funzione, dalla riga 3 alla riga 23
  - in Figura 2 sono presenti tre definizioni di funzione: la prima dalla riga 3 alla riga 6, la seconda dalla riga 8 alla riga 11, e la terza dalla riga 13 alla riga 29
- Per quanto riguarda le funzioni, oltre che alle definizioni di cui ci si occupa adesso, occorre tener presente che esistono anche le *chiamate di funzioni* e le *dichiarazioni di funzioni*, di cui ci si occuperà più avanti
  - nelle Figure 1 e 2 sono presenti dichiarazioni di funzioni all'interno del file `stdio.h` (abbreviazione di *standard input/output*)
  - tipicamente, in un sistema Linux questo file si trova a questo percorso assoluto: `/usr/include/stdio.h`
- Una funzione è concettualmente definita dal suo input, dal suo output, e dalla sequenza di operazioni (codifica di un algoritmo) che trasformano l'input nell'output

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  int primo_operando, secondo_operando;
  char operazione;

  printf("Immettere '+' per addizione, qualsiasi altro carattere per");
  printf(" la sottrazione: ");
  scanf("%c", &operazione);
  printf("Immettere il primo operando: ");
  scanf("%d", &primo_operando);
  printf("Immettere il secondo operando: ");
  scanf("%d", &secondo_operando);
  if (operazione == '+') {
    printf("%d + %d = %d\n", primo_operando, secondo_operando,
           primo_operando + secondo_operando);
  }
  else
    printf("%d - %d = %d\n", primo_operando, secondo_operando,
           primo_operando - secondo_operando);
  return 0;
}
```

Figure 1: Un semplice programma C

  - attenzione: l'output di una funzione *non è* costituito da eventuali `printf` contenute al suo interno
  - e l'input non è costituito da eventuali `scanf` contenute al suo interno

- Una definizione di funzione in C rispecchia tutto ciò, ed è composta come segue:

  `<tipo_output> <nome_funzione> (<argomenti>) {`
  `<blocco_istruzioni> }`

  - tutto ciò che viene scritto tra < e > può essere sostituito con un elemento della relativa *categoria sintattica*
  - tutto il resto (parentesi graffe o tonde, virgole, punti e virgola etc) va messo esattamente com'è
  - <tipo_output> può essere sostituito da un qualsiasi *tipo di dato* (vedere più avanti)
    * nell'esempio di Figura 1, viene sostituito da `int` (inizio della riga 3)
    * indica che tipo di valori può ritornare la funzione
    * nell'esempio di Figura 1, si tratta quindi di un intero (in effetti ritornato alla riga 22)
  - <nome_funzione> deve essere un *identificatore*
    * cioè, qualcosa che identifichi univocamente quella funzione, dandogli un nome
    * nell'esempio di Figura 1, viene sostituito da `main` (riga 3)
    * la regola per scrivere un identificatore è semplice: dev'essere una sequenza di simboli, ciascuno dei quali può essere una lettera (non accentata: dev'essere ASCII...), una cifra o il carattere _ (*underscore*)
    * ma non può cominciare con una cifra
      · identificatori validi: `ciao3`, `ciao`, `c3i1a245o`, `_c3i1a2_45o`
      · identificatori non validi: `3ciao3`, `3_ciao`, `c3i1+a245o`, `_c3i1a2-45o`
  - nota: tra una categoria sintattica e l'altra ci possono essere tutti gli spazi che si vuole
    * per "spazi" si intende lo spazio vero e proprio (tasto centrale in basso di ogni tastiera), il tab (tasto di tabulazione, tipicamente a sinistra della Q sulla tastiera) e l'andata a capo (tasto di invio o return)
    * ad esempio, tra `int` e `main` ci potrebbero anche essere 2 spazi, una tabulazione, un altro spazio e due andate a capo, e il programma sarebbe corretto ugualmente

    - non è invece corretto inserire spazi all'interno delle categorie sintattiche, ad esempio è sbagliato scrivere `ma in`
  - <`argomenti`> è la lista degli input della funzione
    - dev'essere a sua volta così formata: <`tipo_1`> <`nome_input_1`>, ..., <`tipo_`n> <`nome_input_`n>
    - n può anche essere zero (funzione senza argomenti o senza input); occorre comunque aprire e chiudere le parentesi tonde, senza metterci nulla in mezzo
    - se $n = 1$, allora la virgola ovviamente non va messa
    - per i tipi degli input vale lo stesso discorso fatto per il tipo dell'output
    - i nomi degli input devono essere degli identificatori
      - come il nome della funzione, ma devono ovviamente essere nominati diversamente
      - ad esempio, `int main(int main, char **altro)` sarebbe scorretto perché `main` è sia il nome della funzione che di un suo argomento (il primo)
    - nell'esempio di Figura 1, gli argomenti sono 2, uno di tipo `int` e di nome `argc` e uno di tipo `char **` e di nome `argv` (parte finale della riga 3)
  - il blocco delle istruzioni è tipicamente chiamato *corpo* di una (definizione di) funzione
  - tipo di ritorno dell'output, nome della funzione e lista degli argomenti, invece, sono spesso chiamati *intestazione* di una (definizione di) funzione
  - è uno dei modi con i quali si può costruire una *dichiarazione di funzione* (l'unico trattato per adesso)
  - una dichiarazione di funzione, pertanto, è del tipo (notare il punto e virgola alla fine)

    <`tipo_output`> <`nome_funzione`> (<`argomenti`>);

- Ecco com'è fatto il corpo di una funzione, ovvero come può essere formata la categoria sintattica <`blocco_istruzioni`> di cui sopra
- Si compone di due parti (che possono anche essere vuote): le *dichiarazioni (di variabili)* e le *istruzioni*
- Per quanto riguarda le dichiarazioni, sono simili agli argomenti delle funzioni, ma con alcune caratteristiche in più
- Concettualmente, le variabili sono parti della "memoria" nell'architettura di Von Neumann; quando le si dichiara, si dice anche il *tipo* di tali parti
  - concretamente, le variabili sono zone di RAM

  - grazie ai tipi, il compilatore sa (tra le altre cose) quanta RAM dedicare a ciascuna variabile
  - ad esempio, per un `int` vengono tipicamente dedicati 4 bytes (32 bits), per un `double` 8 bytes (64 bits), per un `char` sempre 1 byte (8 bits)

- Ecco la sintassi per le dichiarazioni di variabili:

  <`tipo_1`> <`nome_var_1_1`>, ..., <`nome_var_1_`n_1>;

  $\vdots$

  <`tipo_`m> <`nome_var_`m`_1`>, ..., <`nome_var_`m`_`n_m>;

- In soldoni, sono come le dichiarazioni degli argomenti di una funzione, ma:
  - variabili con lo stesso tipo si possono dichiarare tutte insieme, separando i nomi con le virgole e senza ripetere il tipo comune
    - ad esempio, nella riga 5 di Figura 1 così sono dichiarate le due variabili intere `primo_operando` e `secondo_operando`
  - dichiarazioni di variabili con tipo diverso vanno separate con il punto e virgola
    - ad esempio, tra le righe 5 e 6 di Figura 1

- È venuto il momento di parlare dei *tipi*
  - per i nostri scopi, un tipo è un insieme di valori più un certo numero di operazioni su tali valori
  - tutti i tipi, con l'eccezione del `char`, fanno riferimento ad insiemi concettualmente infiniti, ma praticamente finiti
    - il `char`, invece, è finito anche concettualmente
  - per ora, verranno considerati i seguenti *tipi predefiniti* del C: `int`, `unsigned int` (o anche solo `unsigned`), `float`, `double`, `char`
  - ne esiste un altro, `void`, che però può essere usato direttamente solo come tipo di ritorno (ovvero come tipo di output) per funzioni
  - si vedrà in seguito come sia possibile definirne ed usarne di nuovi; per ora verrà usato un solo *tipo derivato*, ovvero la *stringa*, che viene definita come `char *`
    - fa però eccezione rispetto a quanto detto sopra (almeno per la presente trattazione) nel seguente caso: qualora si vogliano dichiarare più variabili di tipo `char *`, allora o le si scrive ripetendo il tipo e usando il punto e virgola, oppure occorre ripetere l'asterisco
    - ovvero, o così: `char *stringa1; char *stringa2;`

    - oppure così: `char *stringa1, *stringa2;`
    - scrivere invece così: `char *stringa1, stringa2;` equivale a dichiarare (per motivi che saranno chiari più in là) la variabile `stringa1` come di tipo stringa (quindi è ok), mentre `stringa2` risulterebbe solo di tipo carattere
- gli insiemi di valori sono i seguenti: `unsigned` è un sottinsieme di $\mathbb{N}$ (tipicamente l'intervallo $[0, 2^{32} - 1]$), `int` è un sottinsieme di $\mathbb{Z}$ (tipicamente l'intervallo $[-2^{31}, 2^{31} - 1]$), `float` e `double` sono sottinsiemi di $\mathbb{Q}$
- i "tipicamente" di cui sopra si riferiscono al fatto che su diverse macchine quegli intervalli potrebbero variare; esistono comunque dei mezzi per sapere quanto valgano effettivamente tali intervalli
- per alcuni di questi tipi, e per alcuni computer, si possono allargare o restringere tali intervalli usando `long`, `long long`, `short`
    - per esempio, `unsigned long` su alcuni computer corrisponde all'intervallo $[0, 2^{64} - 1]$
- le operazioni definite sui tipi numerici sono quelle aritmetiche usuali
- è però da notare che la divisione ha significato diverso a seconda che sia applicata su interi o su razionali: nel primo caso si parla della *divisione intera*, ovvero dell'operazione di *quoziente*, nel secondo della divisione frazionaria
    - $5/2$ dà come risultato 2, mentre $5.0/2.0$ dà come risultato 2.5
- per ogni tipo, possono essere non solo dichiarate delle variabili, ma anche usate delle *costanti*
    - le costanti, per l'appunto, non cambiano di valore, quindi non hanno riservata nessuna zona di RAM
- il modo di esprimere le costanti cambia a seconda del loro tipo
    - le costanti per i tipi numerici si scrivono semplicemente: `52`, `0`, `-3`, `3.2`, `3.1e-4` (che sta per 3.1×10^{-4}) etc
    - le costanti di tipo carattere vanno racchiuse da apici, e devono contenere un solo carattere: sono costanti di tipo carattere valide `'a'`, `'Z'`, `'9'`, `'+'`, `','`, mentre è sbagliato scrivere `'ab'`
    - alcuni caratteri speciali sono indicati con il carattere backslash, ovvero `\`
        - ad esempio `'\n'` (*newline*, andata a capo), `'\t'` (*tabulation*, tabulazione), `'\\'` (*backslash*, se si vuole scrivere il backslash stesso)
    - le costanti di tipo stringa si esprimono con i doppi apici: `"ciao"`, `"vado a capo\necontinuo a scrivere, poi faccio un tab\te scrivo qualcos'altro"`

```c
#include <stdio.h>

int esegui_op(int primo, char op, int secondo) {
  if (op == '+') return primo + secondo;
  else return primo - secondo;
}

char piu_o_meno(char op) {
  if (op == '+') return '+';
  else return '-';
}

int main(int argc, char **argv) {
  int primo_operando, secondo_operando, risultato;
  char operazione, vera_operazione;

  printf("Immettere '+' per addizione, qualsiasi altro carattere per");
  printf(" la sottrazione: ");
  scanf("%c", &operazione);
  printf("Immettere il primo operando: ");
  scanf("%d", &primo_operando);
  printf("Immettere il secondo operando: ");
  scanf("%d", &secondo_operando);
  vera_operazione = piu_o_meno(operazione);
  risultato = esegui_op(primo_operando, operazione, secondo_operando);
  printf("%d %c %d = %d\n", primo_operando, vera_operazione,
         secondo_operando,  risultato);
  return 0;
}
```

Figure 2: Una variante del programma C di Figura 1

- Per quanto riguarda le *istruzioni*, devono essere uno dei seguenti tipi:

  istruzione vuota ovvero **;** (punto e virgola senza niente prima); non fa nulla, ma in alcuni casi è utile

  assegnamento è del tipo `<nome_var> = <espressione>`

  - `<espressione>` dev'essere dello stesso tipo dichiarato per la variabile a sinistra dell'uguale
  - può essere complicata a piacere, contenere sia operazioni che chiamate a funzioni (vedere più avanti), coinvolgere sia costanti che variabili (anche la stessa variabile che si trova a sinistra dell'uguale)
  - nei casi più semplici, può essere solo una costante o solo una variabile (del giusto tipo, ovviamente)
  - per quanto riguarda la *semantica* dell'istruzione di assegnamento, vale quanto segue:
  - **l'effetto è quello di memorizzare, nei byte di RAM dedicati alla variabile a sinistra dell'uguale, il valore risultante dell'espressione nella parte destra, con ciò cancellando il precedente valore della variabile**
  - ad esempio, le righe 24 e 25 di Figura 2 sono degli assegnamenti
  - da notare che, nella riga 24 di Figura 2, la variabile `vera_operazione` è stata dichiarata di tipo `char` (riga 15), e le viene assegnato il valore tornato dalla funzione `piu_o_meno`, che è correttamente un `char` (riga 8)

  chiamata a funzione (come visto, è combinabile con l'assegnamento), è del tipo `<nome_funz>(<input1>, ..., <input`n`>)`

  - notare come la sintassi sia diversa dalla definizione di funzione (niente tipo di ritorno, niente tipo degli argomenti, niente corpo della funzione)
  - gli argomenti `<input1>, ..., <input`n`>` devono essere in generale delle espressioni, e ciascuna di queste espressioni deve avere lo stesso tipo dichiarato per il corrispondente argomento nella definizione della funzione
  - per quanto riguarda la *semantica* dell'istruzione di chiamata a funzione, vale quanto segue:
  - **l'effetto è quello di sospendere l'esecuzione della funzione in cui ci si trova attualmente, eseguire la funzione chiamata (dove gli input hanno il valore specificato dagli argomenti della chiamata) e poi ritornare il controllo alla funzione in cui ci si trovava in precedenza**
    - per "ritornare il controllo" si intende che, alla fine dell'esecuzione della funzione chiamata, si riprende ad eseguire la funzione chiamante *dal punto immediatamente seguente a quello in cui c'era la chiamata*

    - il valore ritornato dalla funzione chiamata (cioè il valore che si trova dopo l'istruzione `return`) viene usato opportunamente nella funzione chiamata, a seconda di come è strutturata la chiamata stessa
    - ad esempio, se è usata a destra di un assegnamento, allora il valore ritornato viene assegnato alla variabile che si trova a sinistra dell'assegnamento
    - se la funzione chiamata ha come tipo di output `void`, allora nessun valore viene ritornato, e la chiamata a funzione deve essere un'istruzione "secca" (non si deve cioè trovare a destra di un assegnamento)
- ad esempio, le righe 24 e 25 di Figura 2 contengono delle chiamate a funzione (in questo caso, quindi, non sono usate come istruzioni a sé stanti)
    - in entrambi i casi, la funzione chiamante è il `main`
    - la funzione chiamata, invece, è `piu_o_meno` alla riga 24 e `esegui_op` alla riga 25, sempre in Figura 2
- più in particolare (nel seguito si fa sempre riferimento alla Figura 2), nella riga 24, alla funzione `piu_o_meno` viene passata la variabile `operazione`, che sulla base della dichiarazione di riga 15 è correttamente un `char`, esattamente come l'argomento `op` della funzione `piu_o_meno` stessa (riga 8)
- l'effetto sarà quello di sospendere l'esecuzione della funzione `main` in cui ci si trova, eseguire la funzione `piu_o_meno` dove l'input `op` prende il valore che ha la variabile `operazione` nel `main`, e poi restituire il risultato (uno dei due `return` alle righe 9 o 10) alla funzione `main` (che lo assegnerà alla variabile `vera_operazione`)
- analogamente, nella riga 25, alla funzione `esegui_op` vengono passate le variabili `primo_operando`, `operazione` e `secondo_operando`, che sulla base della dichiarazioni alle righe 14 e 15 sono correttamente un `int`, un `char` e un `int` rispettivamente, esattamente come gli argomenti `primo`, `op` e `secondo` della funzione `esegui_op` stessa (riga 3)
- quando, nella funzione `esegui_op`, verrà eseguito uno dei due `return` alle righe 4 o 5, il controllo ritornerà alla funzione `main` e il valore ritornato da `esegui_op` (ovvero il risultato della somma o della sottrazione) sarà assegnato alla variabile `risultato`
- sono chiamate a funzione anche quelle effettuate a `printf` e `scanf` (dichiarate dentro `stdio.h` e definite in modo speciale)
- si tratta di funzioni molto particolari, perché la loro definizione prevede un numero indefinito di argomenti:
  ```
  int printf(char *fmt, ...)
  int scanf(char *fmt, ...)
  ```

- vuole dire che hanno per forza un argomento di tipo stringa, e poi ne *possono* avere degli altri
- dalla documentazione fornita con Linux (ottenibile digitando `man 3 printf` e `man 3 scanf` da un terminale, o anche cercando su Internet) si può leggere che tali argomenti devono essere tanti quanti i caratteri % presenti nella stringa passata come primo argomento
- ad esempio, nelle righe 17, 18, 20 e 22, non essendoci alcun carattere % nella stringa (costante) passata come primo argomento, non occorrono altri argomenti
- invece, nelle righe 19, 21 e 23 è presente un %, quindi occorre un secondo argomento
- analogamente, nelle righe 26-27 ci sono 4 caratteri % nella stringa costante passata come primo argomento, e quindi occorrono altri 4 argomenti
- da notare che gli argomenti aggiuntivi delle `scanf` sono variabili precedute da `&`
- per ora, basti dire che ciò deriva dal fatto che la `scanf` ha come effetto quello di *modificare* il valore di tali variabili
- nel caso della `printf`, gli argomenti aggiuntivi sono semplicemente delle variabili senza `&` (ma possono essere delle generiche espressioni) in quanto la `printf` non effettua modifiche a tali variabili
- il carattere (o in alcuni casi i caratteri) che segue il % viene usato per definire il tipo del corrispondente parametro aggiuntivo: `%c` corrisponde al `char`, `%u` corrisponde all'`unsigned`, `%d` corrisponde all'`int`, `%s` corrisponde al `char *`, `%f` corrisponde al `float`, `%lf` corrisponde al `double`, `%lu` corrisponde al `unsigned long`, `%ld` corrisponde all'`int long`

controllo di flusso ricadono in questa categoria le seguenti istruzioni:

- `if`; `if ... else`; `while`; `do ... while`; `for`; `switch ... case`; `continue`; `break`; `goto`; `return`
- il `goto` va evitato in quanto porta spesso a programmi difficili da capire
- queste istruzioni si chiamano così perché alterano la normale sequenza di esecuzione delle istruzioni (non più l'una dopo l'altra, ma ci possono essere dei salti)
- sintassi dell'`if`:
  `if (<condizione>) <blocco_istruzioni>;`
  - se <blocco_istruzioni> è costituito da più di una istruzione, allora va racchiuso tra parentesi graffe; altrimenti, le parentesi graffe sono opzionali
    - ad esempio, le graffe alle righe 15 e 18 di Figura 1

        * <condizione> dev'essere un'espressione valutabile come un valore *booleano* (vero o falso)
        * per il momento, si assumerà che le espressioni booleane siano quelle ottenibili tramite confronti tra variabili e/o costanti dello stesso tipo, tramite quindi gli operatori `==` (uguaglianza, notare la differenza con l'assegnamento), `<=`, `>=`, `<`, `>`, `!=` (diverso)
        * per quanto riguarda la *semantica* dell'istruzione di `if`, vale quanto segue:
        * **l'effetto è quello di eseguire <blocco_istruzioni> se <condizione> è vera, altrimenti non fa nulla**
        * notare che mettere il punto e virgola dopo la condizione non è sintatticamente sbagliato, ma porta a risultati inaspettati (sostanzialmente, il punto e virgola verrebbe interpretato come l'istruzione vuota, e si prosegue in ogni caso con <blocco_istruzioni>, sia che <condizione> sia vera sia che sia falsa)
    - sintassi dell'`if ... else`:
      `if (<condizione>) <blocco_istruzioni1>; else <blocco_istruzioni2>;`
        * vale quanto detto per l'`if`
        * per quanto riguarda la *semantica* dell'istruzione di `if ... else`, vale quanto segue:
        * **l'effetto è quello di eseguire <blocco_istruzioni1> se <condizione> è vera, altrimenti esegue <blocco_istruzioni2>**

  sequenza le istruzioni vanno separate le une dalle altre per mezzo di `;`
    - non sempre valido (almeno all'apparenza) per le istruzioni di controllo di flusso

- **Esercizio**: modificare il programma di Figura 1 in modo tale che faccia la sottrazione se viene immesso `'-'` da tastiera, mentre con un qualsiasi altro carattere viene fatta la somma

- **Esercizio**: modificare il programma di Figura 1 in modo tale che faccia la divisione (intera, ovvero il quoziente) se viene immesso `'/'` da tastiera, mentre con un qualsiasi altro carattere viene fatto il prodotto

- **Esercizio (difficile)**: modificare il programma di Figura 1 in modo tale che la variabile `operazione` sia di tipo intero, e che venga fatta la somma se `operazione` vale 1, la sottrazione se `operazione` vale 2, il quoziente se `operazione` vale 3, e il prodotto con qualsiasi altro valore
    - suggerimento: <blocco_istruzioni> può contenere qualsiasi categoria di istruzioni; quindi anche un altro `if`...

- **Esercizio**: modificare il programma fatto al punto precedente in modo tale che la variabile `operazione` sia nuovamente di tipo carattere, e che venga fatta la somma se `operazione` vale `'1'`, la sottrazione se `operazione` vale `'2'`, il quoziente se `operazione` vale `'3'`, e il prodotto con qualsiasi altro valore

- **Esercizio**: modificare il programma fatto al punto precedente in modo tale che la variabile `operazione` sia nuovamente di tipo carattere, e che venga fatta la somma se `operazione` vale `'+'`, la sottrazione se `operazione` vale `'-'`, il quoziente se `operazione` vale `'/'`, e il prodotto con qualsiasi altro valore

- **Esercizio**: fare gli stessi esercizi di prima, ma modificando il programma in Figura 2.

- **Esercizio**: compilare ed eseguire i programmi di cui sopra, con diversi possibili input