Steve Blackbird (TardoFreak)

PROGRAMMAZIONE (2): IL PRIMO PROGRAMMA.

26 September 2016

In questo articolo proviamo a scrivere il primo programma. Segue un po' di teoria, il "minimo sindacale" che serve per poter poi continuare ed affrontare altri argomenti. Con il primo programma si può anche verificare il buon funzionamento del sistema di sviluppo in modo da avere uno strumento valido per provare gli esempi e fare le proprie sperimentazioni.

Il primo programma scritto in C

Dopo la teoria passiamo ai fatti e proviamo a vedere come scrivere un programma semplice, per toccare con mano la programmazione. Incominciamo con il C:

- Lanciamo NetBeans
- Dal menù *File* scegliamo *New Project*. Si apre una finestra che ci invita a scegliere il tipo di progetto che vogliamo creare.
- Nella finestra di sinistra clicchiamo sulla cartella *C/C++*.
- Nella finestra di sinistra clicchiamo su *C/C++ Application*.
- Premiamo il pulsante *Next* ed apparirà un'altra finestra.
- Nel campo *Project Name* scriviamo **ProgC** che vorrebbe significare programma in C.
- Accertiamoci che la casella **Create Main File** si spuntata, lasciamo il nome main ed accertiamoci che il menu a tendina a fianco indichi C.
- Premiamo il tasto *Finish*
- Sulla sinistra, nello spazio di navigazione del progetto, apriamo il gruppo *Source Files*. Troveremo un file chiamato **main.c**
- Doppio click sul main.c e comparirà lo scheletro del nostro primo programma in C

```
/*
 * File:   main.c
 * Author: TardoFreak
 *
 * Created a long time ago
 */

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

/*
 *
```

```
 */
int main(int argc, char** argv) {

  return (EXIT_SUCCESS);
}
```

- Nella linea vuota prima dell'istruzione return (EXIT_SUCCESS); scriviamo

```
printf("Il mio primo programma scritto in C");
```

- Il sorgente del nostro programma sarà diventato

```
/*
 * File:   main.c
 * Author: Stefano
 *
 * Created on 19 settembre 2016, 16.18
 */

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

/*
 *
 */
int main(int argc, char** argv) {
  printf("Il mio primo programma scritto in C");
  return (EXIT_SUCCESS);
}
```

- Dal menù *Run* scegliamo *Run Project (ProgC)* per far partire la compilazione e, una volta terminata, il programma. Questa operazione può anche essere fatta semplicemente premendo il triangolo/freccia verde nella barra degli strumenti.

Nella parte bassa dell'area di lavoro si aprirà una scheda a tabulazioni intitolata *ProgC (Build,Run)* e poco tempo dopo un'altra scheda intitolata *ProgC(Run)* dove leggeremo

```
Il mio primo programma scritto in C
RUN SUCCESSFUL (total time 213ms)
```

Congratulazioni! Abbiamo scritto il nostro primo programma in C e questo funziona pure! Non fa chissà che cosa, scrive semplicemente una linea di testo, ma è un programma, il primo programma. Vediamolo nel dettaglio:
Le linee dalla 1 alla 6 sono commenti. I commenti in C sono racchiusi fra /* e */

```
/* questo è un commento.
Posso anche andare a capo e continuare a scrivere perché
il commento finisce quando si incontra questo-> */
```

sono solo dei commenti, non fanno niente di niente e vengono completamente ignorati, quindi potremmo eliminare tutte le sei linee. Anche le linee 11,12 e 13 sono commenti (vuoti, tra l'altro) e quindi potremmo eliminare anche queste. Negli esempi che vedremo da qui in avanti i commenti li toglierò lasciando solo quello che veramente interessa.

Le linee 8 e 9 sono importanti perché servono per "inglobare" nel programma delle funzioni di libreria la funzione **printf** è una una funzione di libreria, e non un'istruzione del linguaggio. Per ora sono un mistero ma il loro significato sarà più chiaro in seguito. Un piccolo atto di fede e prendiamo la cosa per buona. Una volta eliminati i commenti il programma si riduce a questo:

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

int main(int argc, char** argv)
{
  printf("Il mio primo programma scritto in C");
  return (EXIT_SUCCESS);
}
```

Ed ora vediamo come funziona. I programmi in C sono un insieme di **funzioni** ma, all' avvio del programma, ne viene eseguita una in particolare: la funzione **main**. Cosa vuol dire "viene eseguita"? Che la lista di istruzioni racchiuse fra le due parentesi graffe vengono eseguite in sequenza una dopo l'altra.
La prima istruzione che viene eseguita è quindi la *printf*. Come accennato prima questa non è un'istruzione del linguaggio ma è una funzione di libreria che stampa un testo e/o dei numeri sul dispositivo di uscita, nel nostro caso il terminale di NetBeans. Parleremo diffusamente delle funzioni più avanti, per ora fermiamoci su questa. Fra le due parentesi è racchiuso il cosidetto **parametro attuale** della funzione che andiamo ad impiegare. In buona sostanza abbiamo scritto "Esegui (chiama) la funzione prinf e fagli stampare la scritta *Il mio primo programma scritto in C*". Il calcolatore fa proprio questo e noi ci ritroviamo nel terminale quella scritta.
La seconda linea è necessaria e bisogna metterla alla fine del programma. Il perché non ha importanza ora, piccolo atto di fede e prendiamo la cosa così com'è. Tutto sarà chiaro quando si parlerà delle funzioni.
Nota ai più attenti non sarà sfuggito un particolare: ho spostato la prima parentesi graffa. La posizione della parentesi non ha nessun effetto sul funzionamento del programma, è solo una questione di stile. Quando scrivo in C uso questo stile, questo modo di scrivere le parentesi un po' per tradizone (una volta i programmi si scrivevano così) ma anche per sapere a colpo d'occhi se quello che scrivo è C o Java. Come ho detto prima le sintassi si assomigliano tantissimo e mi è capitato di fare errori proprio perché, usando ambedue i linguaggi, mi evita confusione.

Il primo programma scritto in Java

Bene, ora chiudiamo il progetto in C che abbiamo creato prima scegliendo dal menù *File* la voce *Close Project (ProgC)*, e creiamo il progetto in Java

- Dal menù File scegliamo New Project. Si apre una finestra che ci invita a scegliere il tipo di progetto che vogliamo creare.* Nella finestra di sinistra clicchiamo sulla cartella *Java*.* Nella finestra di sinistra clicchiamo su *Java Application'*.
- Premiamo il pulsante *Next* ed apparirà un'altra finestra.
- Nel campo *Project Name* scriviamo **ProgJava**
- Accertiamoci che la casella Create Main Class sia spuntata e premiamo il tasto *Finish*. Si aprirà immediatamente il sorgente del programma

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package progjava;

/**
 *
 * @author Stefano
 */
public class ProgJava {

  /**
   * @param args the command line arguments
   */
  public static void main(String[] args) {
    // TODO code application logic here
  }

}
```

Anche in Java i commenti sono racchiusi fra /* e */ quindi li elimino per non creare inutile confusione.

```
package progjava;

public class ProgJava {

  public static void main(String[] args) {
    // TODO code application logic here
```

```
  }
}
```

C'è un altro modo per scrivere commenti che è identico sia in Java che in C, e serve per scrivere commenti che stanno in una singola linea, sono chiamati **line comment**, iniziano con // e finiscono al termine della linea. Sono più facili e veloci da scrive e, personalmente, li preferisco a quelli scritti con /* e */.
Bene, ora rimpiazziamo la linea del commento

```
// TODO code application logic here
```

con

```
System.out.println("Il mio primo programma scritto in Java");
```

ottenendo così questo sorgente

```
package progjava;

public class ProgJava {

  public static void main(String[] args) {
    System.out.println("Il mio primo programma scritto in Java");
  }
}
```

- Dal menù *Run* scegliamo *Run Project (ProgJava) e facciamo partire il programma.*

Nella parte bassa dell'area di lavoro si aprirà una scheda a tabulazioni intitolata *ProgJava(Run)* dove leggeremo

```
run:
Il mio primo programma scritto in Java
BUILD SUCCESSFUL (total time: 1 second)
```

Ri-congratulazioni! Abbiamo scritto il nostro primo programma in Java e questo funziona. Fa esattamente quello che fa quello scritto in C e, se notate, si assomigliano molto. Analizziamolo nel dettaglio.
La linea *package progjava;* è una dichiarazione. Dice che questo programma, o meglio questa **classe** farà parte di un pacchetto di classi. Una sorta di libreria di classi, più o meno come nel C le istruzioni *include*. Questi pacchetti sono molto utili nel caso di riutilizzo delle classi da parte di altri programmi che si scriveranno in futuro. A noi, considerato che ci concentriamo sulla programmazione strutturata, questo non interessa e lo lasciamo lì dov'è.
Una cosa che salta all'occhio è che anche in Java c'è la funzione main, ma in Java le funzioni si chiamano **metodi**. Tuttavia il calcolatore si comporta allo stesso modo. Essendoci una sola classe

principale (che racchiude tutto il programma) all'avvio il calcolatore esegue il metodo **main**, cioè inizia ad eseguire in sequenza le istruzioni racchiuse fra le due parentesi graffe, una dopo l'altra.
Come nel programma in C la prima istruzione non è un'istruzione ma una chiamata al metodo System.out.println (come la prinf). Questo metodo serve per stampare testo e/o numeri nel dispositivo di uscita (cioè la finestra a video dell'ambiente NetBeans). . Fra le parentesi tonde abbiamo scritto il **parametro attuale** per questo metodo. Quindi, anche in questo caso, è come se noi dicessimo al calcolatore "Chiama il metodo System.out.println e fagli stampare la scritta *Il mio primo programma scritto in Java*", e anche in questo caso noi ci troveremo la scritta sul terminale di NetBeans.
Nota: le parole *public static* rimarranno un mistero, almeno per quanto riguarda la programmazione strutturata. Anche qui serve un atto di fede: bisogna scriverle e le scriveremo.

Istruzioni e blocchi

Un programma è costituito da una lista di **linee di programma**, ogni linea di programma contiene istruzioni (ad esempio la dichiarazione di una variabile che vedremo poco più avanti) che terminano con un punto e virgola detto **terminatore di linea**. Questo significa che volendo si potrebbe scrivere una **linea logica** su più **linee fisiche** (le linee dell'editor di testi). I caratteri di ritorno a capo <CR> (che ci sono e non si vedono ma hanno la funzione di permettere la scrittura nella riga successiva), tranne pochi casi molto particolari (ad esempio i line comment che terminano con il termine della linea fisica), sono semplicemente non presi in considerazione dal compilatore. Quindi si possono anche scrivere più linee logiche in una sola linea fisica. A volte questo può essere un pregio ma di solito non lo è.
Sovente diverse linee logiche sono raggruppate in un'entità chiamata **blocco**. Negli esempi dei primi programmi in C e Java il **corpo** della funzione o metodo "main" è un blocco. La porzione di codice, di linee di programma, che forma un blocco è racchiuso fra **parentesi graffe**.
Un aspetto interessante del blocco è che può essere considerato come un unica istruzione, una sorta di "macro istruzione" composta da più istruzioni, direttive e altri blocchi. Esempio di blocco che contiene un line comment e tre linee di programma.

```
if (a > b) {
  // blocco
  int temp = a;
  a = b;
  b = temp;
}
```

Nota: il blocco è solo quello racchiuso fra parentesi graffe, l'istruzione prima del blocco (if (a > b) è un costrutto che vedremo più avanti.

Per scrivere il sorgente in modo ordinato e comprensibile si usa scrivere il contenuto di un blocco spostato verso destra di un certo numero di spazi. Di solito sono quattro ma io ne uso due, sono più che sufficienti per avere un sorgente facilmente leggibile. Es:

```
{
  // Blocco 1
  System.out.println("Blocco 1);
  {
    // Blocco 2
    System.out.println("Blocco 2);
  }

  {
    // Blocco 3
    System.out.println("Blocco 3);
    {
      // Blocco 4
      System.out.println("Blocco 4);
    }
  }
}
```

Nell'esempio il blocco 1 contiene tutto, potrebbe essere ad esempio il metodo/funzione main. I blocchi 2 e 3 sono blocchi interni al blocco 1 e quindi le istruzioni si scrivono spostate verso destra di due spazi. Il blocco 4 è all'interno del blocco 3 quindi le istruzioni in esso contenute vengono scritte con due ulteriori spazi verso destra.
Questo tipo di allineamento è comunemente chiamato **indentazione**, termine che deriva dall'inglese *indent* permette anche di valutare visivamente l'annidamento dei vari blocchi.

Algoritmi

Un algoritmo è un insieme di direttive che servono per risolvere un problema. Queste direttive, per essere considerate un algoritmo, devono essere espresse in modo estremamente preciso affinché chiunque possa capirle senza dovere introdurre ulteriori dettagli. Essendo un insieme di direttive, un algoritmo può essere scritto in pseudocodice (una combinazione di termini in lingua italiana e di costrutti di programmazione) o mediante un linguaggio di programmazione specifico.
Un esempio può essere utile per capire meglio. Supponiamo di dover calcolare il costo dei componenti di una scheda elettronica. In una scheda elettronica ci sono diversi componenti in diverse quantità. Ad esempio possiamo avere 3 transistor BC337, 4 resistenze da 10K, un circuito stampato e via dicendo. L'algoritmo per calcolare con una calcolatrice il costo totale dei materiali della scheda può essere definito in questo modo:

- Cancellare il contenuto della memoria in modo che valga 0 premendo il tasto MC
- Ripetere le seguenti operazioni per ogni componente

- Moltiplicare il costo del componente per il numero di componenti di quel tipo

- Aggiungere il risultato al contenuto della memoria premendo M+

- Visualizzare il contenuto della memoria premendo MR

Un algoritmo può avere dei risultati intermedi e la necessità di memorizzarli e quindi conservarli da qualche parte. Nell'esempio appena visto il display della calcolatrice contiene il costo dei componenti di un certo tipo, la memoria della calcolatrice funziona invece come totalizzatore. Il calcolatore memorizza questi valori nella sua memoria sotto forma di **variabili**.

Variabili e Tipi

Le variabili sono da considerarsi come dei contenitori che contengono un valore che può essere letto e modificato. I linguaggi di alto livello permettono di dare un nome a piacere (con alcune limitazioni) alle variabili. Tale nome identificherà il contenitore, infatti i nomi delle variabili, e anche quelle delle classi e dei metodi o funzioni, sono detti **identificatori**.
Che cosa possono contenere le variabili? Diversi tipi di dato. La scelta del tipo di dato dipende da cosa e come deve essere usata la variabile. Possiamo scegliere fra variabili che possono contenere numeri interi positivi e negativi, o numeri reali con cifre decimali, singoli caratteri, sequenze di caratteri formate da un numero di caratteri a piacere e valori logici (vero o falso). E' chiaro che una variabile che può contenere un valore logico occupa meno byte in memoria di una variabile che può contenere un numero reale a doppia precisione.
Ma come fa il compilatore a sapere se un identificatore è una variabile di un certo tipo? Mediante una **dichiarazione**. Per esempio scrivere la linea

```
int totaleProdotti;
```

sia in C che in Java significa dire in qualche modo al compilatore "sappi che da qui in poi userò una variabile che si chiama *totaleProdotti* e che questa è di tipo **int**, cioè può contenere numeri interi (senza la virgola ovviamente) positivi e negativi".
Note:

- l'identificatore *totaleProdotti* è diverso da *TotaleProdotti* o da *totaleprodotti* perché sia il Java che il C fanno differenza fra caratteri maiuscoli e minuscoli. Questo è di solito una fonte di errori di sintassi.
- per convenzione in Java (ma io la uso anche in C) gli identificatori di variabili iniziano con una lettera minuscola.
- gli identificatori di qualsiasi tipo, compresi quelli dei metodi/classi/funzioni, non possono essere parole chiave. Per esempio *int* è una parola chiave quindi non la si può usare come identificatore. Si può usare invece *iNt* ma non è una pratica accettabile. Una regola non scritta impone di evitare come la peste l'uso di questo tipo di trucchetti per dare agli identificatori i nomi delle parole chiave.
- La dichiarazione di una variabile è una istruzione e una istruzione deve sempre terminare con il punto e virgola.
- Si possono dichiarare più variabili dello stesso tipo elencandole dopo il tipo e separandole con una virgola

```
int pippo, pluto, paperino;
```

che equivale a scrivere

```
int pippo;
int pluto;
int paperino;
```

In Java i tipi di base detti **primitivi** (i tipi di base) sono i seguenti

Nome del tipo	Tipo di valore	Memoria usata	Intervallo valori
byte	Intero	1 byte	da -128 a 127
short	Intero	2 byte	da -32.768 a 32.767
int	Intero	4 byte	da -2.147.483.648 a 2.147.483.647
long	Intero	8 byte	da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	Numero a virgola mobile	4 byte	da ±3,40282347·10^+38 a ±1,40239846·10^-45
double	Numero a virgola mobile	8 byte	da ±1,79769313486231570·10^+308 a ±4,94065645841246544·10^-324
char	Carattere singolo Unicode	2 byte	tutti i caratteri ASCII e Unicode
boolean		1 bit	true o false

Per il C il discorso è un po' diverso perché sia l'intervallo dei valori (chiamato **range**) che la memoria usata variano a seconda del compilatore. Questa è la tabella (molto semplificata) dei tipi indicati dal manuale di riferimento del compilatore Keil per i micro ARM, potenti micro a 32bit.

Nome del tipo	Tipo di valore	Memoria usata	Intervallo valori
char	Intero o carattere ASCII esteso	1 byte	da -128 a 127
short	Intero	2 byte	da -32.768 a 32.767
int	Intero	4 byte	da -2.147.483.648 a 2.147.483.647
long	Intero	8 byte	da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
long long	Intero	8 byte	da -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807
float	Numero a virgola mobile	4 byte	da ±3,40282347·10^+38 a ±1,40239846·10^-45

double	Numero a virgola mobile	8 byte	da ±1,79769313486231570·10^+308 a ±4,94065645841246544·10^-324

Note:

- Alcuni tipi, come "long" e "long double" come pure "long" e "long long" sono identici. Questo perché il compilatore li tratta allo stesso modo. Compilatori per altri processori più potenti potrebbero trattare i "long long" ed i "long double" in modo diverso, ad esempio offrendo un range maggiore per i "long long" (a scapito di una occupazione di memoria di 16 byte).
- E' anche normale per i compilatori per micro, sopratutto per quelli di fascia bassa, trattare gli "int" come degli "short" perché in quei dispositivi la memoria temporanea è limitata e l'occupazione di memoria da parte di una variabile, potrebbe essere un problema. Quindi prima di scrivere un programma in C è **obbligatorio consultare il manuale di riferimento del compilatore per trovare l'elenco dei tipi con le loro caratteristiche**.
- In verità "short" e "long" non sono propriamente dei tipi ma sono dei **modificatori** del tipo "int" ("long" lo è anche per il tipo "double"). Scrivere "short" equivale a scrivere "short int" ma essendo "int" sottinteso, lo si può omettere.
- Esiste un modificatore per gli interi che cambia il range, è il modificatore "unsigned" che, come dice il nome stesso, indica che il numero è un intero senza segno, cioè un intero positivo. Quindi se il tipo "char" ha un range che va da -128 a +127 un "unsigned char" ha un range che va da 0 a 255. Lo stesso per il "short", un "unsigned short" ha un range che va da 0 a 65535 e via discorrendo. Il modificatore "unsigned" non si può applicare ai tipi in virgola mobile "float" e "double".

Visibilità

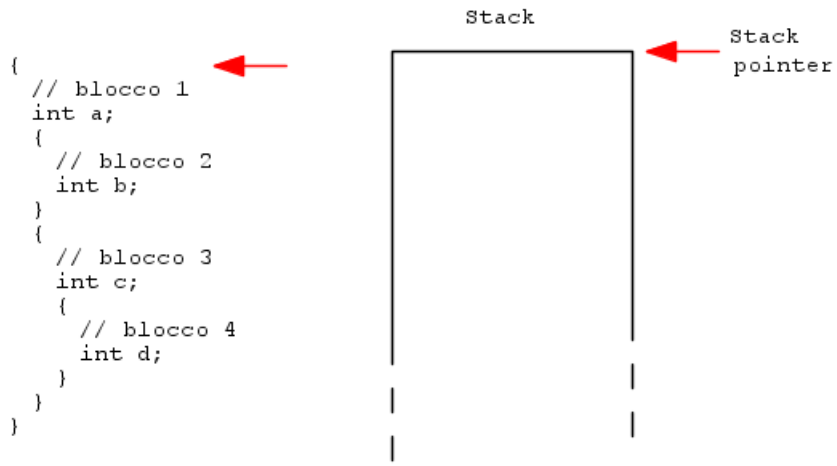
Le dichiarazioni di variabili si scrivono all'interno dei blocchi e/o dei corpi dei metodi o funzioni. In Java si possono dichiarare variabili in qualsiasi punto mentre in C devono tassativamente essere dichiarate all'inizio di ogni blocco o corpo di funzione. Queste sono chiamate **variabili locali** e possono essere lette e modificate solo all'interno del blocco in cui sono state dichiarate. Un esempio aiuterà a capire meglio, prendiamo in considerazione questa porzione di codice (potrebbe essere il corpo di una funzione o metodo) che non fa assolutamente niente se non dichiarare delle variabili in diversi punti

```
{
  // blocco 1
  int a;
  {
    // blocco 2
    int b;
  }
```

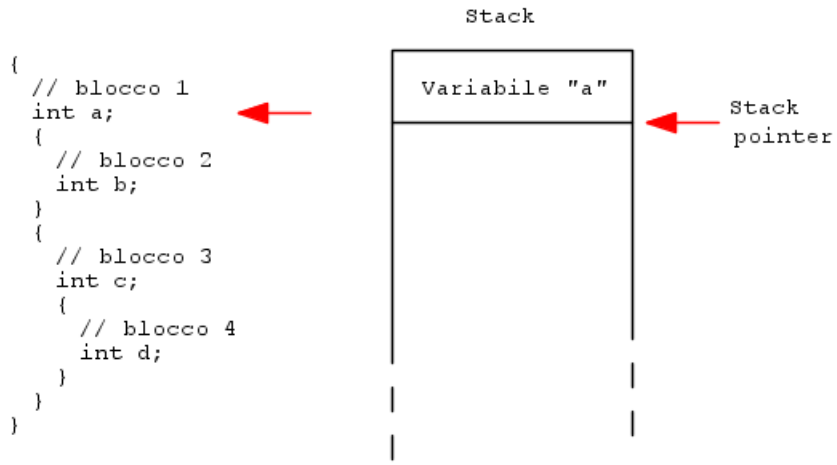
```
  {
    // blocco 3
    int c;
    {
      // blocco 4
      int d;
    }
  }
}
```

- Nel blocco 1 c'è la dichiarazione della variabile "a". Da questo punto in poi "a" sarà **visibile** cioè, si potrà fare riferimento ad essa, in altre parole se ne potrà leggere o modificare il suo valore in tutto il blocco 1.
- Nel blocco 2 c'è la dichiarazione della variabile "b". All'interno di questo blocco oltre ad essere visibile "b" anche "a" è visibile perché dichiarata in un blocco che racchiude anche questo.
- Al termine del blocco 2 la variabile b non esiste più, quindi non è più visibile.
- Nel blocco 3 c'è la dichiarazione della variabile "c". Sono visibili "a" e "c"
- Nel blocco 4 c'è la dichiarazione della variabile "d". Sono visibili "d" perché dichiarata all'interno di questo blocco, "c" perché dichiarata nel blocco 3 che racchiude il blocco 4, e "a" perché dichiarata nel blocco 1 che li contiene tutti. Al termine del blocco 4 "d" non esisterà più.

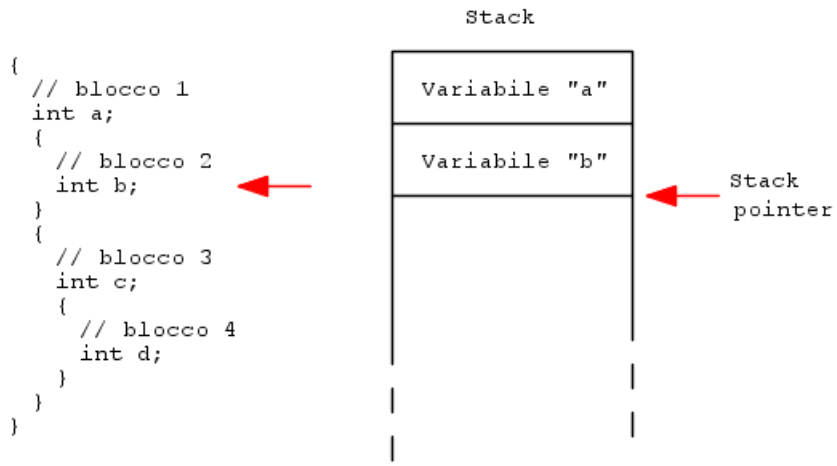
Il perché succede questo lo si può capire meglio analizzando cosa fa effettivamente il calcolatore quando viene dichiarata una variabile, utilizzando un modello che non è esattamente corrispondente alla realtà, ma utile per la comprensione. Abbiamo accennato in precedenza su come il calcolatore memorizza i dati (che ora possiamo chiamare variabili), ed ora vediamo più precisamente il modo in cui lo fa. Per la memorizzazione delle variabili locali il calcolatore si serve di una struttura chiamata **stack** (che è comunque una zona di memoria) ed un registro speciale chiamato **stack pointer** cioè puntatore allo stack. Esso fa riferimento (contiene quindi l'indirizzo) alla prima posizione libera dello stack. Questa è la situazione all'inizio del blocco



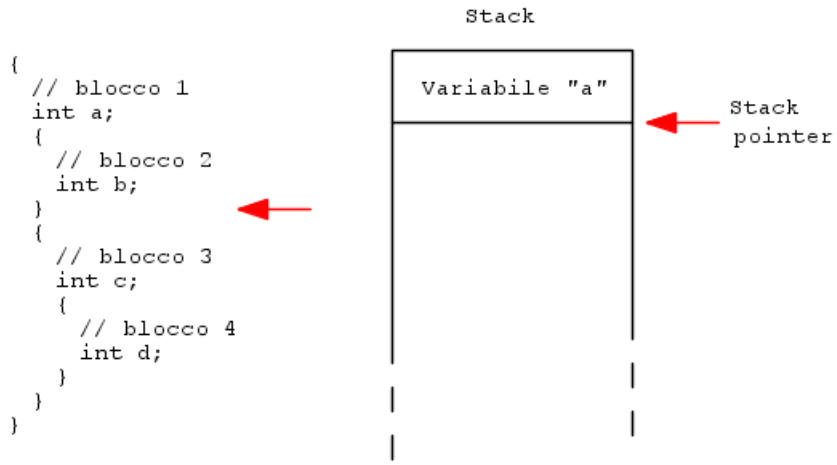
Lo stack è vuoto e lo stack pointer punta all'inizio dello stack. Dopo la dichiarazione della variabile "a" la situazione diventa questa



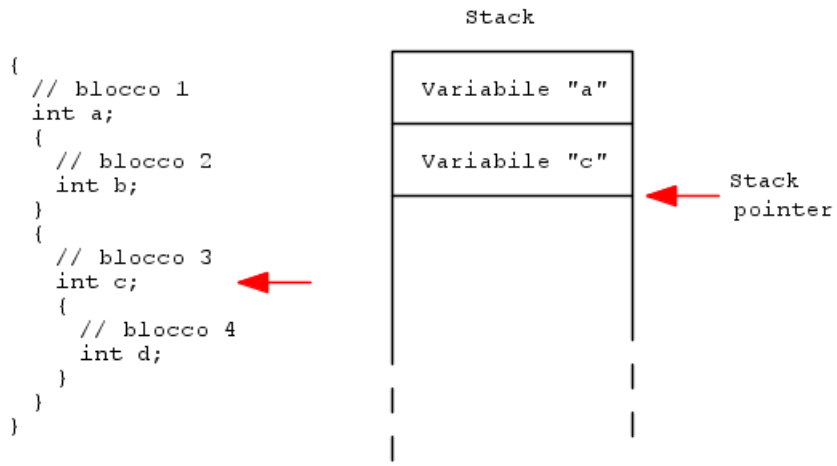
La variabile "a" ha un suo posto nello stack ed è visibile perché se, ad esempio, una espressione la coinvolgesse il calcolatore la cercherebbe nello stack a partire dallo stack pointer andando verso l'alto e, visto che in questo "cammino" la incontra, la può usare senza problemi. Vediamo ora la situazione dopo la dichiarazione di "b"



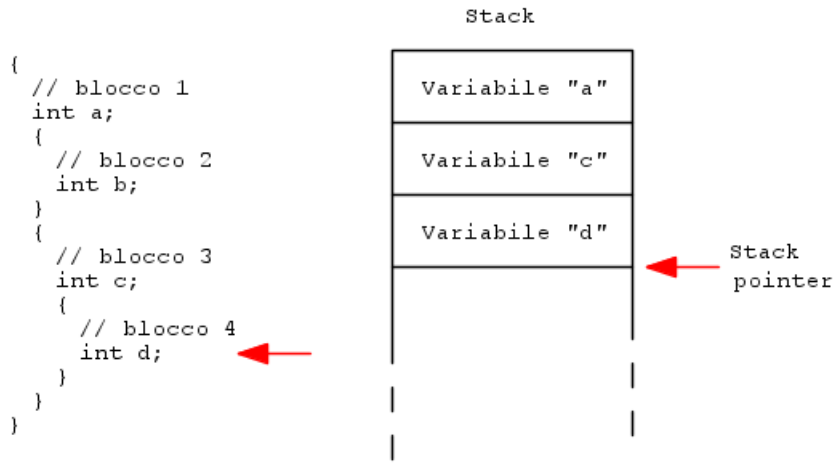
In questa situazione un'eventuale espressione che coinvolgesse "a" e "b" sarebbe perfettamente legale perché il calcolatore troverebbe entrambe le variabili. Dopo che abbiamo incontrato la fine del blocco 2 la situazione ritornerà ad essere questa.



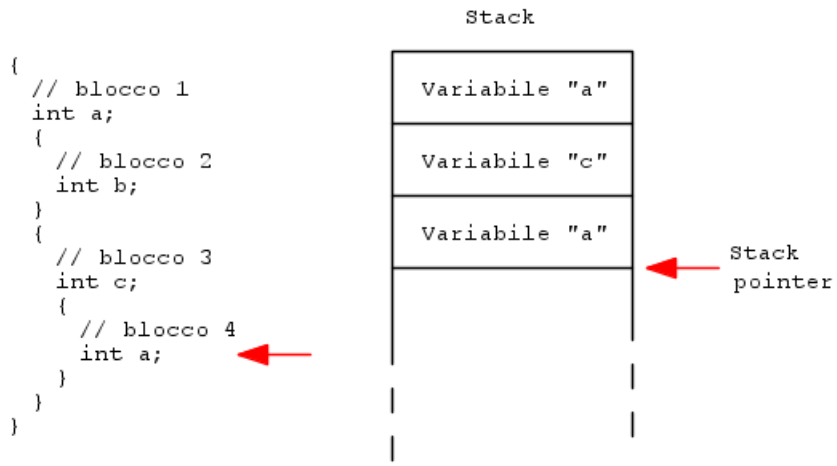
La variabile "b" è sparita dallo stack perché non esiste più. Esiste solo all'interno del blocco 2, ma il blocco 2 è terminato ed il calcolatore ha rimosso la variabile "b". E' facile immaginare la situazione dopo la dichiarazione della variabile "c"



E dopo la dichiarazione della variabile "d"



Il meccanismo dovrebbe essere chiaro.
Ma cosa sarebbe successo se invece di dichiarare la variabile "d" il compilatore avesse incontrato nuovamente la dichiarazione "int a;"? Ci troveremmo in questa situazione



Sarebbe perfettamente legale perché se l'istruzione successiva facesse riferimento ad "a" il calcolatore la cercherebbe dallo stack pointer in su, e sarebbe la prima che incontrerebbe. Ma attenzione, perché farebbe riferimento a quella dichiarata dentro il blocco 4 e non a quella dichiarata nel blocco 1!
In altre parole è possibile dichiarare a **livello di blocco** una variabile con lo stesso nome di un'altra dichiarata in un blocco che lo contiene, a patto di ricordarsi che il calcolatore farà riferimento all'ultima dichiarata. Questo può essere causa di errori logici ma tante volte è utile e semplifica il codice.
E' comunque caldamente consigliato (ed io lo faccio) di **dichiarare una variabile solo nell'ambito in cui serve**. Quindi se mi serve solo all'interno di un blocco non è bene dichiararla al di fuori di questo, ma è meglio dichiararla all'interno, a livello di blocco.
Nota IMPORTANTE: Nel modello che ho utilizzato parlo del "calcolatore" che opera sullo stack. In verità non è così perché tutto il lavoro sullo stack lo fa il compilatore. Questo perché il compilatore, in qualche modo, si crea un'immagine precisa di come verrà utilizzato lo stack. Quindi il compilatore, mentre compila il programma linea dopo linea, sa con esattezza quanto è grande lo stack e dove sono posizionate le variabili prima di compilare l'istruzione successiva.

Variabili globali

Il C permette di dichiarare anche le cosiddette **variabili globali**. Sono variabili dichiarate al di fuori dei corpi delle funzioni e che sono visibili a tutte le funzioni del programma. Questo vuol dire che se alcune funzioni fanno riferimento ad una o più di queste variabili, nel momento in cui il valore di una o più di queste variabili cambia, l'effetto si ripercuote su tutte queste funzioni. Di per se è un qualcosa di molto potente ma fonte di errori logici.
In Java non esistono variabili globali.

Considerazioni finali

Da qui in poi farò solo riferimento a Java. Ho scritto l'esempio del primo programma per fornire una traccia a chi ha voglia di scrivere gli esempi futuri anche in C. Se un programma in C è un insieme di funzioni e dati, un programma in Java è un insieme di classi ed ogni classe è un insieme di metodi e dati. Scrivere un programma in Java con una classe sola e con metodi **public static** è (quasi) come scrivere un programma in C quindi, anche per evitare confusione (un linguaggio basta ed avanza) userò solo Java per quanto possibile. Suggerisco caldamente a chi vuole imparare ad usare i micro o il C in generale, di provare gli esempi anche in C, è una pratica divertente.
Ovviamente porterò degli esempi e delle osservazioni sul C quando questi saranno particolarmente importanti.

Nota Immagine di copertina tratta dal sito www.superedo.it