

# Lezione 12: Variabili e Operatori

Elementi di Architettura e Sistemi Operativi

Docente: Tiziano Villa

Corso di Laurea in Bioinformatica

Aprile 2021

# Argomenti della lezione

- Variabili e operatori in C.
- Traduzione in LC-3.

Fonte:

**Patt & Patel:** “*Introduction to Computing Systems: From Bits and Gates to C and Beyond*”. Ch. 12 section 12.5.

# Elementi di base del C

- Variabili

- ▶ Un dato su cui il programmatore puo' eseguire un'operazione
- ▶ Uno spazio con nome e tipo in memoria
- ▶ Esempio: z, counter

- Operatori

- ▶ Azioni predefinite eseguite sui dati
- ▶ Esempio: \*, ||, |, &&, &, +

- Espressioni

- ▶ Operatori combinati con le variabili/letterali formano espressioni
- ▶ Esempio:  $x * y$

- Enunciati

- ▶ Espressioni raggruppate insieme formano enunciati
- ▶ Unita' di lavoro per linguaggi ad alto livello
- ▶ Terminano con punto e virgola. Esempio:  $z = x * y;$

# Proprieta' delle variabili

- **Identificatore:** Nome della variabile
- **Tipo:** come s'interpreta il dato e quindi quanto spazio richiede in memoria
- **Ambito:** regione del programma in cui la variabile e' viva e accessibile
- **Memoria:** come il compilatore C alloca la variabile in memoria e la sua durata (la variabile perde il suo valore quando termina l'esecuzione del blocco che la contiene ?)

# Proprieta' delle variabili

Ogni combinazione di lettere, numeri e trattino basso ("underscore": \_)

- Sensibile a maiuscola/minuscola
  - ▶ "somma" e' diverso da "Somma"
- Non puo' iniziare con un numero
  - ▶ Per convenzione le variabili che iniziano con un trattino basso si usano solo in procedure speciali di libreria
- I nomi delle variabili possono avere lunghezza variabile, ma il compilatore C considera solo i primi 31 caratteri (dopo non differenzia i nomi delle variabili)
- Non si possono usare come nomi di variabile le parole chiave riservate (es. *int*).

# Esempi d'identificatori

- **Legali**

i

wordsPerSecond

words\_per\_second

\_green

aReally\_longName\_moreThan31chars

aReally\_longName\_moreThan31characters

*(gli ultimi due identificatori sono indistinguibili)*

- **Illegali**

10sdigit

ten'sdigit

done?

double *(parola chiave riservata)*

# Tipi delle variabili

Ci sono tre tipi di dati fondamentali: *int* interi, *char* caratteri, *double* numeri in virgola mobile lunghi.

Il significato di una stringa binaria dipende dal tipo con cui s'interpreta, ad es., 0110 0110 rappresenta *f* se interpretato come un carattere ASCII oppure il numero 102 come un intero in complemento a due.

Il compilatore alloca memoria ed esegue le operazioni in base al tipo dichiarato. Ad esempio, l'addizione di due interi e' eseguita con un'operazione ADD, mentre l'addizione di due numeri in virgola mobile richiede una sequenza d'istruzioni.

# Tipi delle variabili

C ammette molti tipi di variabile:

**int** intero (almeno 16 cifre binarie, di solito 32 cifre binarie)

**long** intero lungo (almeno 32 cifre binarie)

**float** virgola mobile (almeno 32 cifre binarie)

**double** virgola mobile lungo (di solito 64 cifre binarie)

**char** carattere (almeno 8 cifre binarie)

- La dimensione può variare in base al processore
  - ▶ *int* ha la dimensione “naturale” di un intero: in LC-3, sono 16 cifre binarie, ma in quasi tutti i processori moderni sono 32 cifre binarie
  - ▶ Come faccio a sapere la dimensione ?
    - ★ Si chiama la funzione *sizeof*, e.g., *sizeof(int)* restituisce la risposta in bytes.
- Interi con o senza segno:
  - ▶ Per norma, gl'interi s'intendono con segno rappresentati in complemento a due
  - ▶ Si usa la parola chiave *unsigned* for interi senza segno.



# Tipi delle variabili

- **int** in LC-3 il tipo int rappresenta un intero con 16 cifre in complemento a due, cioè un intero con segno tra -32 768 e +32 767.  
*int numeroDiSecondi* dichiara una variabile di tipo intero chiamata *numeroDiSecondi*, LC-3 alloca una parola di memoria.
- **char** bastano 8 cifre binarie per un carattere, ma per semplicità in LC-3 assegniamo una parola (16 cifre binarie) per un carattere.  
*char lucchetto* e *char chiave = 'Q'* dichiarano due variabili di tipo carattere, di cui la seconda è inizializzata a Q.
- **double** rappresentano numeri in virgola mobile in precisione doppia e richiedono 64 cifre binarie (**float** richiede 32 cifre binarie).  
Esempi di dichiarazione: *double costoPerLitro*, *double elettroniPerSecondo*, *double mediaTemperatura*.

# Tipologia di costanti

- Le costanti sono variabili il cui valore non cambia durante l'esecuzione del programma.
- Ci sono tre tipi di valori costanti nel linguaggio C:
  - ▶ **Costanti letterali:** costanti senza nome che compaiono letteralmente nel codice sorgente.
  - ▶ **Costanti con “const”:** dichiarate come costanti premettendo “const” al tipo. Es.: *const double pi = 3.14159.*
  - ▶ **Simboli:** definite con la direttiva di preprocessore *#define*. Es.: *#define RADIUS 15.0.*
    - ★ Denotano valori costanti in una singola esecuzione, ma variabili in esecuzioni diverse o da utente a utente.

# Esempi d'uso dei letterali

## Interi

123 /\* decimal \*/

-123

0x123 /\* hexadecimal \*/

## Numeri in virgola mobile

6.023

6.023e23

5E12

## Caratteri

'c'

'\n' /\* newline \*/

'\xA' /\* ASCII 10 (0xA) \*/

## Numeri in virgola mobile in doppia precisione

```
double twoPointone = 2.1;      /* This is 2.1 */
double twoHundredTen = 2.1E2; /* This is 210.0 */
double twoHundred = 2E2; /* This is 200.0 */
double twoTenths = 2E-1; /* This is 0.2 */
double minusTwoTenths = -2E-1; /* This is -0.2 */
```

# Esempi d'uso delle costanti

```
#define RADIUS 15.0

int main(){
    const double pi = 3.14159;
    double area;
    double circum;

    area = pi * RADIUS * RADIUS;
    circum = 2 * pi * RADIUS;
}
```

- Ogni combinazione di variabili, costanti, operatori e chiamate a funzione
  - ▶ Ogni espressione ha un tipo, ereditato dai tipi dei componenti secondo le regole del C per la composizione dei tipi
- Esempi:  
counter >= STOP  
x + sqrt(y)  
x & z + 3 || 9 - w - % 6

- Esprime un'unità di lavoro completa
  - ▶ eseguita in sequenza
- Un enunciato semplice termina con punto e virgola  
 $z = x * y$ ; /\* assegna il prodotto a z \*/  
 $y = y + 1$ ; /\* incrementa y dopo il prodotto \*/  
; /\* enunciato nullo \*/
- Un enunciato composto raggruppa enunciati semplici mediante parentesi graffe
  - ▶ equivalente sintatticamente a un enunciato singolo  
 $\{ z = x * y; y = y + 1; \}$

- I programmatori modificano le variabili con gli *operatori* messi a disposizione dal linguaggio ad alto livello.
- Le variabili e gli operatori si combinano a formare *espressioni* ed *enunciati* che rappresentano il compito che il programma deve eseguire.
- Ogni operatore può corrispondere a molte istruzioni macchina.
  - ▶ Esempio: l'operatore di moltiplicazione (\*) richiede tipicamente più istruzioni ADD del linguaggio LC-3.



# Operatore di assegnamento

- Tutte le espressioni prendono un valore, anche quelle con l'operatore d'assegnamento.
- Per l'assegnamento, il risultato e' il valore assegnato.
  - ▶ di solito (ma non sempre) e' il valore del lato destro
    - ★ la conversione dei tipi puo' modificare il valore assegnato rispetto al valore calcolato
- L'assegnamento associa da destra a sinistra
  - ▶ Es.,  $y = x = 3$ ;  $y$  prende il valore 3 poiche' a  $x = 3$  si assegna il valore 3

# Ambito delle variabili: globale e locale

- Risponde alla domanda: dove e' accessibile una variabile ?
- **Globale**: accessibile in tutto il programma
- **Locale**: accessibile solo in una certa regione del programma
- Il compilatore inferisce l'ambito da dove la variabile e' dichiarata
  - ▶ non c'e' bisogno che il programmatore dichiari esplicitamente l'ambito
- Una variabile e' locale al blocco in cui e' dichiarata
  - ▶ il blocco e' definito da graffe aperte e chiuse { }
  - ▶ si puo' accedere a ogni variabile dichiarata in qualsiasi blocco "contenente"
- Una variabile globale e' dichiarata all'esterno di ogni blocco

# Esempio di validita' dell'ambito

```
#include <stdio.h>
int itsGlobal = 0;
main()
{
    int itsLocal = 1; /* locale al main */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
    {
        int itsLocal = 2; /* locale a questo blocco */
        itsGlobal = 4; /* cambia la variabile globale */
        printf("Global %d Local %d\n", itsGlobal, itsLocal);
    }
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

## Uscita del programma:

Global 0 Local 1

Global 4 Local 2

Global 4 Local 1

# Classe di memoria di una variabile

Ci sono due classi di memoria in C: *statica* e *automatica*.

- Le variabili statiche mantengono il loro valore tra una chiamata e l'altra.
  - ▶ In C le variabili globali sono statiche, mantenendo il loro valore fino a quando il programma termina.
- Le variabili automatiche perdono il loro valore quando il loro blocco termina.
  - ▶ In C le variabili locali per regola sono automatiche.
  - ▶ In C le variabili locali possono essere dichiarate come statiche, premettendo la parola chiave *static* nella dichiarazione.
    - ★ Es. la dichiarazione *static int localVar* implica che *localVar* mantiene il suo valore durante l'intera esecuzione del programma. Il compilatore alloca spazio nell'area delle variabili globali, ma la mantiene visibile solo per il suo blocco.

# Classe di memoria di una variabile

## Esempio di uso della parola chiave *static*

```
int Count (int x)
{
    static int y;

    y++;
    printf("Questa funzione e' stata chiamata %d volte", y);
}
```

- In questo caso il compilatore alloca una variabile statica locale *y* nella sezione globale e l'aggiorna ad ogni chiamata senza perderne il valore tra una chiamata e l'altra.
- Le variabili globali e tutte le variabili della classe statica sono inizializzate a zero dal compilatore quando inizia l'esecuzione.
- Tutte le variabili automatiche e quindi le variabili locali (tranne quelle definite come statiche) devono essere inizializzate dal programmatore altrimenti contengono un valore imprevedibile memorizzato al momento dell'ultima scrittura in quella cella di memoria.

# Classe di memoria di una variabile

- La parola chiave *register* indica che un oggetto di tipo automatico e' usato spesso nel codice e converrebbe memorizzarlo in un registro.
- La parola chiave *extern* indica che la memoria per una funzione o variabile e' allocata in un altro modulo oggetto che sara' assemblato con il modulo corrente quando si costruisce l'eseguibile.

# Il punto di vista del compilatore

- Il compilatore tiene traccia delle variabili mediante la *tavola dei simboli* e assegna la memoria alle variabili secondo la loro tipologia (ad es. globali o locali, etc.).
- Quando il compilatore legge una dichiarazione di variabile, crea un nuovo elemento corrispondente nella tavola dei simboli. Tale elemento contiene:
  - ▶ nome, cioè il suo identificatore
  - ▶ tipo, es. *int* o *char*
  - ▶ posizione relativa della variabile nella regione di memoria assegnata ad essa
  - ▶ ambito, cioè identificatore del blocco in cui la variabile è dichiarata

# Allocazione delle variabili in memoria

Nel linguaggio C, ci sono due regioni della memoria in cui si alloca spazio alle variabili.

- la *sezione dei dati globale* (*global data section*) dove si memorizzano le variabili globali, e in generale le variabili della classe di memoria statica (*static storage class*)
- la *pila al tempo d'esecuzione* (*run-time stack*) dove si memorizzano le variabili locali della classe di memoria automatica (*automatic storage class*)
  - ▶ Tutte le variabili locali di una funzione sono allocate in una regione di memoria chiamata *finestra di attivazione* o *struttura di pila* (*activation record* o *stack frame*) che contiene tutte le variabili locali di una data funzione
  - ▶ Ogni invocazione di una funzione ha una finestra di attivazione



# Allocazione delle variabili in memoria

- La posizione relativa o spostamento (“offset”) di una variabile indica la sua distanza (numero di locazioni) dalla base della sezione di memoria
  - ▶ Es. se la variabile globale *Temp* ha spostamento 4 rispetto alla base della sezione globale che inizia a 0x5000 *Temp* si trova all'indirizzo 0x5004
- Per convenzione memorizziamo nel registro *R4* l'indirizzo dell'inizio della sezione dati globale
  - ▶ Es. LDR R3, R4, #4 carica nel registro R3 il valore della variabile globale precedente *Temp*

# Allocazione delle variabili in memoria

- Per convenzione memorizziamo nel registro *R5* (chiamato *puntatore alla struttura*, “*frame pointer*”) l’indirizzo dell’inizio della finestra di attivazione.
  - ▶ Le variabili sono indicate nella finestra in ordine inverso rispetto a quello delle loro dichiarazioni poiché la pila dell’esecuzione cresce per indirizzi decrescenti della memoria. Es., se la variabile locale *Secondi* è dichiarata per sesta, l’istruzione `LDR R0, R5, #-5` ne carica il valore nel registro *R0*.
- Anticipo: quando si chiama una funzione in C s’infilta la sua finestra d’attivazione in cima alla pila d’esecuzione e *R5* è fatto puntare all’inizio di tale finestra, per cui gli accessi alle sue variabili locali sono corretti. Alla fine della chiamata, la finestra è sfilata dalla pila e *R5* è fatto puntare alla finestra del chiamante.
  - ▶ Nel frattempo *R6* (*puntatore alla pila*, “*stack pointer*”) punta sempre alla cima della pila dell’esecuzione.



# Allocazione delle variabili in memoria

- Pila dell'esecuzione
  - ▶ Le variabili locali vanno qui
  - ▶ R6 punta alla cima della pila
  - ▶ R5 punta all'inizio della finestra di attivazione nella pila
  - ▶ Nuova finestra di attivazione per ogni blocco (eliminata quando si esce dal blocco)
- Spostamento (Offset) = distanza dall'inizio dell'area di memoria
  - ▶ Globale: LDR R1, R4, #4
  - ▶ Locale: LDR R2, R5, #-3

# Ripartizione della memoria

- Il testo del programma occupa una regione della memoria, così come la sezione dati globale e la pila dell'esecuzione. La dimensione della pila dell'esecuzione è variabile ed è decisa durante l'esecuzione.
- Si riserva un'altra regione di memoria per i dati allocati dinamicamente: *"heap"* (*mucchio*) di dimensione variabile decisa durante l'esecuzione.
- Se una funzione ne chiama un'altra, la pila dell'esecuzione cresce perché s'infiltra una nuova finestra di attivazione sulla pila, con gli indirizzi di memoria che decrescono verso 0x0000 (con le variabili locali puntate nell'ordine da R5, R5-1, etc.). Invece il mucchio cresce verso 0xFFFF.

# Esempio complessivo - Codice C

```
#include <stdio.h>

int inGlobal;

main()
{
    int inLocal ; /* local to main */
    int outLocalA ; /* local to main */
    int outLocalB ; /* local to main */

    inLocal = 5;
    inGlobal = 3;

    outLocalA = inLocal & ~inGlobal;
    outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

    printf("outLocalA %d, outLocalB %d\n",
           outLocalA, outLocalB);
}
```

# Esempio complessivo - Analisi del codice C

- Il codice contiene una variabile globale *inGlobal* e tre variabili locali *inLocal*, *outLocalA*, *outLocalB*.
- Il compilatore LC-3 assegna alla variabile globale *inGlobal* il primo posto disponibile nella sezione dati globale, dalla posizione 0.
- Analizzando la funzione *main* il compilatore assegna alle variabili locali *inLocal*, *outLocalA*, *outLocalB* rispettivamente le posizioni 0, -1, -2 nella finestra d'attivazione di *main* ("activation record" di *main*).

| Identificatore   | Tipo | Locazione | Ambito  | Altro |
|------------------|------|-----------|---------|-------|
| <i>inGlobal</i>  | int  | 0         | globale | ..    |
| <i>inLocal</i>   | int  | 0         | main    | ..    |
| <i>outLocalA</i> | int  | -1        | main    | ..    |
| <i>outLocalB</i> | int  | -2        | main    | ..    |

# Esempio complessivo - Codice LC3

main:

...

<codice d'inizializzazione>

...

AND R0, R0, #0

ADD R0, R0, #5 ; inLocal parte da 0 in main

STR R0, R5, #0 ; inLocal = 5

AND R0, R0, #0

ADD R0, R0, #3 ; inGlobal parte da 0 in Globals

STR R0, R4, #0 ; inGlobal = 3

LDR R0, R5, #0 ; acquisisci valore di inLocal

LDR R1, R4, #0 ; acquisisci valore di inGlobal

NOT R1, R1 ; ~inGlobal

AND R2, R0, R1 ; inLocal + ~inGlobal

STR R2, R5, #-1 ; outLocalA = inLocal + ~inGlobal;  
outLocalA parte da -1 in main



# Esempio complessivo - Codice LC3

```
LDR R0, R5, #0 ; acquisisci valore di inLocal
LDR R1, R4, #0 ; acquisisci valore di inGlobal
ADD R0, R0, R1 ; inLocal + inGlobal
```

```
LDR R2, R5, #0 ; acquisisci valore di inLocal
LDR R3, R4, #0 ; acquisisci valore di inGlobal
NOT R3
ADD R3, R3, #1 ; ~inGlobal
```

```
ADD R2, R2, R3 ; inLocal - inGlobal
NOT R2
ADD R2, R2, #1 ; ~(inLocal - inGlobal)
```

```
ADD R0, R0, R2 ; (inLocal + inGlobal) - (inLocal - inGlobal)
STR R0, R5, #-2 ; outLocalB; outLocalB parte da -2 in mai
...
<codice per chiamare la funzione printf>
...
```

# Esempio complessivo - Analisi del codice C

- Il registro R4 punta a *inGlobal* nella posizione 0 della sezione dati globale.
- Il registro R5 punta a *inLocal* nella posizione 0 della pila in tempo reale, mentre *outLocalA* e *outLocalB* sono nelle posizioni -1 e -2 della pila in tempo reale.
- La memoria cresce dalla posizione x0000 alla posizione xFFFF.
- La sezione dati globale cresce verso posizioni crescenti della memoria, per cui se ci fosse una seconda variabile globale essa si troverebbe alla posizione R4+1.
- La pila in tempo reale cresce verso posizioni decrescenti della memoria, per cui se la prima variabile locale *inLocal* si trova all'indirizzo R5=YZ, la seconda variabile locale *outLocalA* si trova all'indirizzo R5-1=YZ-1, e la terza variabile locale *outLocalB* si trova all'indirizzo R5-2=YZ-2.