

Tecniche di Programmazione Funzionale e Imperativa

Ivano Salvo

*Programmazione C: Specifiche
Pre, Post-condizioni, invarianti*

Corso di Laurea in Informatica, III anno



SAPIENZA
UNIVERSITÀ DI ROMA

Lezione 17, 3 maggio 2021

Lezione 17a

Tiny C e Asserzioni Logiche

Struttura di un programma C -

Ogni buon corso di C comincia con il programma Hello World!

definizione di una funzione

Include le librerie di input/output standard, che permette di usare printf()

```
#include<stdio.h>

int main(int argc, char* argv[]){
    /* programma Hello World! */
    printf("Hello World!\n");
    return 0;
}
```

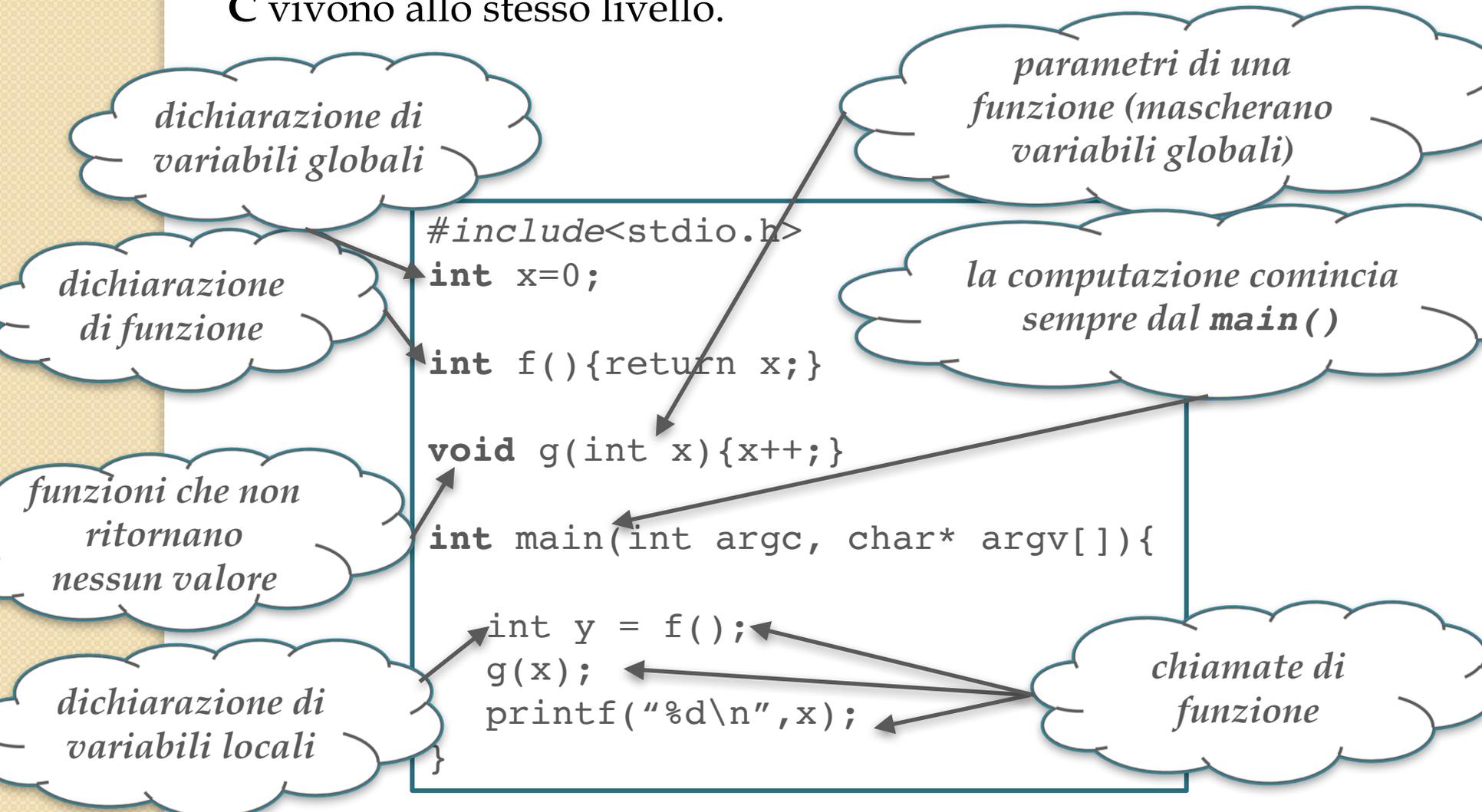
commenti

ritorna un valore. 0 significa "tutto bene", altrimenti si esce con un codice di errore

eventuali parametri letti dall'esterno quando comincia l'esecuzione

Struttura di un programma C

La computazione comincia sempre da `main()`. Tutte le funzioni C vivono allo stesso livello.



Tiny C

Cominciamo con un frammento minimale del C, contenente **if**, **while**, sequenza, assegnazioni e variabili intere.

L'unica espressione ammessa è +1 e la verifica di uguaglianza ==.

Cominciamo con un semplice programma che calcola la somma tra due numeri interi positivi.

*clausola **modifies**:
esprime eventuali
side-effects*

*alla fine di un
ciclo è garantita
l'asserzione INV
& negazione della
guardia*

```
int somma(int m, int n)
{
  REQ: m, n >= 0
  * ENS: ritorna m+n
  * MOD: -
  */
  int i = 0;
  int s = m;
  while (i != n){
    /* INV: s = m + i
     s = s + 1;
     i = i + 1;
    */
  }
  return s;
} /* INV & i=n => s=m+n
```

*clausola **require** o
precondizioni: esprime
assunzione sui parametri*

*clausola **ensure** o
postcondizioni: esprime
condizioni sui risultati*

***Invarianti**: esprimono
condizioni logiche sempre
soddisfatte durante
l'esecuzione di un ciclo*

Vero Programmatore C

Ovviamente, un vero programmatore C baratterebbe un anno di vita per risparmiare una riga di codice o anche pochi spazi.

Adeguiamoci subito al galateo del Vero Programmatore C.

Ricordate che **`i++`** è “equivalente” a **`i = i + 1`** e non a `i+1`!!

C'è anche il pre-incremento `++i`.

```
int somma(int m, int n){  
    int i=0;  
    while (i++ != n) m++;  
    return m;  
}
```

*postincremento: viene
eseguito dopo aver
valutato l'espressione i*

*non è necessario introdurre
una nuova variabile s: **m** è
già una copia*

Osservazione: comandi ed espressioni

A rigore, `i++` **non è un'espressione**, ma piuttosto un **comando**.

In C però c'è una totale promiscuità tra comandi (= che modificano la memoria) ed espressioni (che valutano un valore).

In particolare, tutti i comandi restituiscono un valore. In particolare le **assegnazioni restituiscono il valore assegnato**.

Attenzione! Questo può portare a errori subdoli. Ad esempio:

```
if (x=0) printf("pippo"); else printf("pluto");
```

non viene segnalata come errore dal compilatore. L'espressione valuta sempre a 0 (= false) e x viene assegnata con 0.

Il test di uguaglianza è `==`.

Ovviamente, tutto ciò permette dei programmi molto compatti anche se non sempre leggibili. Un gusto tutto C.

Verifiche di Correttezza

Faremo prove di correttezza dei programmi, basate sulle asserzioni logiche. Tali asserzioni logiche dipendono dai valori delle variabili di programma.

Converremo di scrivere: m_0, n_0 per i **valori iniziali** delle variabili m, n all'interno della funzione.

In un ciclo scriveremo m', n' per i valori assunti dalle variabili m e n **dopo l'esecuzione del ciclo** e m, n i valori assunti **prima**.

Nel nostro esempio, avremo che nel ciclo è sempre vero che $m=m_0+i$. Infatti, **è vero all'ingresso del ciclo** in quanto $i=0$ & $m_0=m$. Si mantiene a ogni esecuzione perché $m'=m+1$ e $i'=i+1$ e quindi: $m=m_0+i \Rightarrow m+1=m_0+i+1 \Rightarrow m'=m_0+i'$.

La **terminazione** è garantita dal fatto che $n' - i' < n - i$ e all'interno del ciclo $n - i > 0$ (questo è garantito dalla guardia del while).

```
int somma(int m, int n){
    int i=0;
    while (i++ != n) m++;
    return m;
}
```

Teorema di iterazione finita

Teorema. Sia φ una proprietà che dipende dalle variabili x_1, \dots, x_n di un programma. Se:

- la proprietà φ è soddisfatta all'ingresso del ciclo **while**(B)C
- l'esecuzione di C conserva la proprietà φ
- il ciclo termina

Allora, l'asserzione $\varphi \ \& \ \neg B$ è soddisfatta all'uscita del ciclo.

Teorema. Sia t una funzione a valori interi che dipende dalle variabili x_1, \dots, x_n di un programma. Se all'interno di un ciclo **while**(B)C con invariante φ :

- $\varphi \ \& \ B$ implica $t(x_1, \dots, x_n) \geq 0$
- $\varphi \ \& \ B$ implica $t(x'_1, \dots, x'_n) < t(x_1, \dots, x_n)$

Allora il ciclo termina dopo un numero finito di passi.

Dim: (Sketch) Non ci sono catene infinite decrescenti in \mathbb{N} . □

Volendo, potremo giocare con un altro frammento del C che io chiamo **TinyReC**, in cui ho solo **if** e funzioni ricorsive.

Ovviamente, anche questo frammento permette di calcolare qualunque cosa (è una “specie” di lambda-calcolo – se non uso neanche le assegnazioni!)

```
int sommaRec(int m, int n){
    return sommaRecAux(m, n, 0);
}

int sommaRecAux(int m, int n, int i){
    if (i==n) return m;
    return sommaRecAux(m+1, n, i+1);
}
```

Ricorsione

Possiamo scrivere una funzione da **Vero Programmatore C** che evita la funzione ausiliaria e il parametro ausiliario *i*:

```
int sommaRecVPC(int m, int n){  
    if (!n) return m;  
    return sommaRecVPC(m+1, n-1);  
}
```

Attenzione invece che il seguente programma **non funziona!**

```
int sommaRecWrong(int m, int n){  
    if (!n) return m;  
    return sommaRecVPC(m++, n--);  
}
```

infatti l'incremento/decremento avviene **dopo la valutazione dell'espressione** per il passaggio dei parametri.

Funzionerebbe con `++m` e `--n`, ma tutto sommato non ha molto senso modificare i valori di *m* ed *n*.

Predecessore

Ci siamo presi già la libertà di sottrarre 1, ma in realtà è possibile definire anche la funzione predecessore, partendo da +1.

Idea: usare due indici, mantenendo la proprietà $i=j+1$ fino a che i non diventa n . $i=j+1$ & $i=n$ implica $n=j+1$ (per sostituzione) e quindi $j=n-1$.

Terminazione: La funzione $n - j$ è sempre decrescente. Infatti ho $n' - j' = n - (j + 1) = n - j - 1$, ma non è necessariamente sempre positiva! Infatti, se n fosse 0, $n - j = -1$. Infatti, in tal caso il programma non termina!

```
int pred(int n){
    int i = 0;
    int j = 1;
    while (j != n){
        /* INV: i = j + 1 */
        i++;
        j++;
    }
    return i;
}
```

Predecessore: assert

In realtà la funzione è corretta rispetto alla specifica:

$$\text{pred } n = n - 1 \quad n > 0$$

Completare questa funzione facendo tornare un valore arbitrario su 0 (per esempio 0 oppure 42) **non è più corretto** rispetto alla funzione non terminante!

La cosa migliore, è far terminare il programma con un'eccezione. Cogliamo l'occasione per mostrare un programma da Vero programmatore C con una variabile in meno.

```
#include<assert.h>
int pred(int a){
  /* REQ: a>0
   * ENS: return a-1
   */
  assert(a>0);
  int i = 0;
  while (i+1 != a) i++;
  /* INV: i < a */
  return i;
}
```

*Valuta la condizione a>0
e blocca il programma se
la condizione fallisce*

Predecessore: Sperimentazioni

Verificate cosa succede in **C** se eseguite il programma predecessore su input 0.

Fate lo stesso esperimento in Python.

Rifate lo stesso esperimento (in Python, **C** e Haskell) dopo che abbiamo scritto la versione ricorsiva.

Cosa ne deducete?

Comparazione

Scriviamo un programma `int compareTo(int m, int n)` che restituisce 1 se $m > n$, 0 se $m = n$ e -1 altrimenti.

```
int compareTo(int m, int n){
  /* REQ: m,n>=0 */
  if (m==n) return 0;
  while (m && n){
    m = pred(m);
    n = pred(n);
  }
  if (n) return -1;
  return 1;
}
```

Esercizio: scrivere invarianti, funzione di terminazione etc.

Note sulla complessità in Tiny C

Anche se Turing-completo, il TinyC (o TinyReC) introduce enormi inefficienze dovute alla rappresentazione unaria dei naturali.

Funzioni semplicissime, come il predecessore sono lineari nel numero su cui fare il predecessore.

Ad esempio, **un programma che conta in avanti è più efficiente di uno che conta all'indietro.**

In generale, il programma che calcola una funzione (che cresce molto rispetto a n) $f(n)$ **ha una complessità almeno di $f(n)$** : infatti nel caso fortunato possiamo costruire il valore $f(n)$.

Nella Teoria degli Algoritmi si fa l'ipotesi che un'operazione aritmetica costi sempre $\mathcal{O}(1)$ (ipotesi del **costo costante**).

Funziona quando i programmi non calcolano numeri particolarmente grandi. Non funziona quando si calcolano funzioni che crescono molto (come fattoriale o fibonacci etc.) nel qual caso è più opportuno fare l'ipotesi del **costo logaritmico**.

Comparazione revisited

Scriviamo un programma `int compareTo(int m, int n)` che restituisce 1 se $m > n$, 0 se $m = n$ e -1 altrimenti.

Questa volta però contiamo in avanti e scegliamo la comodità al posto dell'eleganza:

```
int compareTo(int m, int n){
  /* REQ: m,n>=0 */
  int i=0;
  if (m == n) return 0;
  while (42){
    if (i == m) return -1;
    if (i == n) return 1;
    i++;
  }
}
```

Esercizio: scrivere invarianti, funzione di terminazione etc.

Condizioni Logiche in C

Abbiamo giocato a usare poche istruzioni: condizioni logiche solo nella forma $exp_1 == exp_2$ oppure $exp_1 != exp_2$.

In C **non c'è** il tipo **booleano**: in compenso, a **tutti i valori di ogni tipo di dato base è associato un valore di verità**.

Per gli interi, 0 vale false e diverso da 0 vale true.

Per i puntatori, NULL vale false, e diverso da 0 vale true.

Per cui, espressioni come:

```
if (4) printf("pluto"); else printf("pippo");
```

sono perfettamente legittime (e in questo caso stampa pluto).

Possiamo scrivere un programma super-compresso per la somma:

```
int somma(int m, int n){  
    while (n--) m++;  
    return m;  
}
```

Lezione 17b

Sintesi di Programmi da Asserzioni Logiche

Divisione Intera

Scriviamo un programma che calcola la divisione intera (o il resto) per un esecutore che sa fare solo somme/differenze.

Ricordiamo cosa significa fare la divisione intera: dati due numeri m ed $n > 0$, occorre trovare due numeri tali che:

$$m = qn + r \ \& \ 0 \leq r < n$$

Questa è quindi la post-condizione della nostra funzione. L'idea standard in questi casi è **indebolire** la post-condizione: ci sono infinite coppie q, r tali che $m = qn + r$ (è l'equazione di una retta).

È sufficiente trovare l'unica coppia tale che $0 \leq r < n$.

Quindi: $m = qn + r$ è l'invariante naturale del ciclo: ponendo $q=0$ e $r = m$ otteniamo facilmente una coppia di valori che lo soddisfa.

Incrementando q di 1, l'invariante si mantiene sottraendo n ad r . Infatti: $q'n' + r' = (q+1)n + (r - n) = qn + n + r - n = qn + r$.

$r \geq n$ è la guardia naturale (esco quando soddisfo la seconda condizione, cioè $0 \leq r < n$). Osserviamo che sotto l'ipotesi $n > 0$, $r \geq n$ è una funzione di terminazione.

Divisione intera iterativa

Fatte le precedenti osservazioni, la funzione si scrive da sola.

Cosa succede se $n = 0$?

Osservate che la stessa funzione può essere usata per calcolare il resto (usualmente chiamato mod).

```
int div(int m, int n){
  /* REQ: n > 0 */
  int q = 0;
  int r = m;
  while (r >= n){
    /* INV: m = q*n + r */
    r -= n;
    q++;
  }
  return q;
  /* return r calcola il resto */
}
```

r -= n (o +=, *= etc.) è
equivalente a **r = r - n**

Moltiplicazione Egiziana: def

Problema 1: *Scrivere una funzione che calcoli la moltiplicazione sfruttando le seguenti uguaglianze:*

$$\begin{aligned}m \times 2n &= (m + m) \times n \quad (n > 1) \\m \times (2n + 1) &= m \times 2n + m \\m \times 0 &= 0\end{aligned}$$

Idea: somme e moltiplicazioni $\times 2$ sono operazioni più facili di fare prodotti generali (**provare per credere**).

Si tratta di una definizione induttiva alternativa del prodotto, che definisce il prodotto per **induzione** sul secondo parametro.

Specifica quanto vale il prodotto in 3 casi (gra loro mutuamente esclusivi): 0 (**caso base**) e numero pari o dispari, sempre in termini di prodotti con un secondo fattore **minore**.

Multiplic. Egiziana ricorsiva

Data una definizione induttiva, modulo un po' di sintassi, è immediato scrivere un programma ricorsivo anche in C:

```
int multEgypt(int m, int n){
    /* caso base */
    if (n==0) /* if (!n)
        return 0;
    /* caso dispari */
    if (n % 2 == 1) /* if (n % 2)
        return m + multEgypt(m, n-1);
    /* caso pari */
    return multEgypt(m+m, n/2)
}
```

Moltiplic. Egiziana iterativa

Già per questo programma molto semplice, la soluzione iterativa è meno ovvia.

Come spesso accade l'idea è quella di usare una **variabile accumulatore**, diciamo p (per prodotto).

Operando direttamente su due variabili di ingresso m ed n , l'idea è di mantenere **invariante** la quantità $p+mn = m_0n_0$.

È facile entrare nel ciclo soddisfacendo l'invariante, semplicemente inizializzando p a 0.

- Se n è **dispari**, sottraggo 1 a n e sommo m a p . Infatti, in tal caso: $p'+m'n'=p+m+m(n-1)=p+m+mn-m=p+mn = m_0n_0$
- Se n è **pari**, dimezzo n e raddoppio m . Infatti, in tal caso: $p'+m'n'=p+2m(n/2)=p+mn = m_0n_0$

Ovviamente, quando n diventa 0 ho che $p=m_0n_0$. E quindi n è la funzione di **terminazione** (decrece a ogni iterazione) e $n \neq 0$ la guardia.

Moltiplic. Egiziana iterativa

Fatti i ragionamenti alla slide precedente, il programma si scrive da solo dalle asserzioni logiche:

```
int multEgypt(int m, int n){
    int p = 0;
    while (n != 0){
        /* INV:  $r + m * n = m_0 * n_0$ 
         *  $T = n$ 
         */
        if (n%2) { p += m;
                  n--;
                } else { m = m * 2;
                          n = n / 2;
                } /* endelse */
    } /* endwhile */
    return p;
}
```

Lezione 17

That's all Folks!

Grazie per l'attenzione...

...Domande?