Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2020/2021

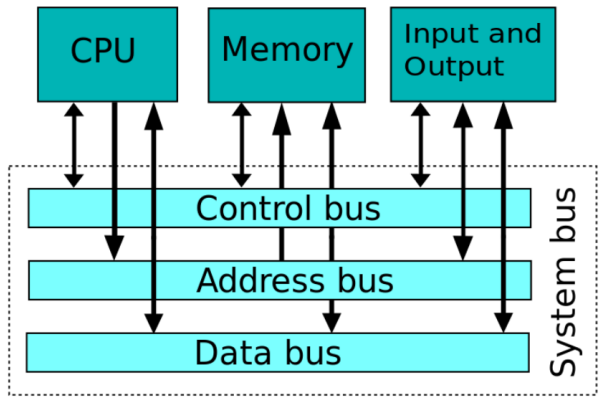
2021

Indice

1. Da Python a C

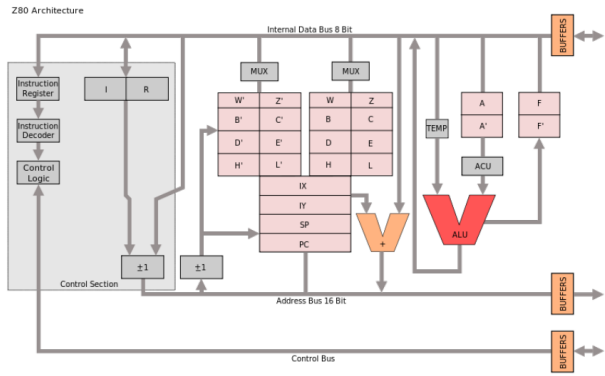
1.1. Architettura del calcolatore

L'architettura di un calcolatore (architettura di Von Neumann) è costituita da CPU, memoria e dispositivi di input/output connessi attraverso dei bus.



1.1.1. Architettura della CPU

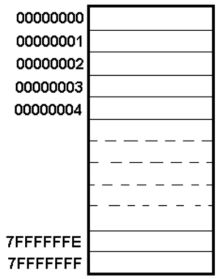
L'architettura di una CPU è costituita da una serie di registri di memoria, da un'unità di calcolo che svolge calcoli su dati contenuti nei registri, e un'unità di controllo che si interfaccia con i bus per leggere/scrivere dati in memoria.



Il trasferimento dei dati memoria-CPU è molto più lento dei calcoli all'interno della CPU.

1.1.2. Architettura della memoria

Per gli scopi di questo corso, la memoria (virtuale) può essere immaginata come una sequenza di celle indicizzate da un indirizzo.



1.1.3. Programmazione del calcolatore

La programmazione del calcolatore consiste nel sintetizzare procedure automatiche (algoritmi) per la soluzione di problemi mediante un linguaggio di programmazione.

Il programma risiede in memoria e viene eseguito tramite la CPU elaborando dati in memoria e interagendo con l'utente tramite i dispositivi di input/output.

La memoria deve essere organizzata e gestita in maniera opportuna per consentire l'esecuzione dei programmi nella CPU.

Scopo principale di questo corso: *comprendere le tecniche di programmazione per la gestione della memoria di un calcolatore durante l'esecuzione di programmi complessi*.

Esempio: somma di due numeri

Python:

```
a = b + c
print(a)
```

C:

```
int a = b + c;
printf("%d\n",a);
```

Questi due frammenti di programma svolgono la stessa funzione e sono molto simili sintatticamente, ma l'uso della memoria del calcolatore durante l'esecuzione dei due programmi è completamente diversa.

1.2. Linguaggi di programmazione

Esistono moltissimi linguaggi di programmazione: una vera e propria *torre di Babele*.

Infatti, per definire un linguaggio X basta realizzare un programma I che consente di utilizzare il linguaggio X, possibilmente su diversi tipi di calcolatore. Utilizzare un linguaggio significa poter eseguire programmi scritti nel linguaggio stesso.

Il programma I deve tradurre i programmi scritti nel linguaggio X in istruzioni direttamente eseguibili dal calcolatore. In alcuni casi, I esegue direttamente le istruzioni del linguaggio.

1.2.1. Linguaggio macchina e linguaggi alto livello

- Linguaggio macchina
  `21 40 16 100 163 240`

- Linguaggio assemblativo (Assembly)
  ```
  iload intRate
  bipush 100
  if_icmpgt intError
  ```

- Linguaggi ad alto livello (ad es. Python e C)
  `if (intRate > 100) ...`

Il linguaggio del modello di elaboratore di Von Neumann è un esempio molto semplificato di linguaggio macchina. C è il linguaggio di alto livello più “vicino" alla macchina. Python è un linguaggio di alto livello, molto “vicino" al programmatore.

1.2.2. Interpreti e compilatori

Il programma *I* può essere realizzato in due diverse modalità:

- *Interprete*
- *Compilatore*

L’interprete (es. Python) considera le istruzioni del linguaggio di programmazione una alla volta traducendole direttamente in istruzioni in linguaggio macchina ed eseguendole direttamente.

Il compilatore di un linguaggio effettua la traduzione in linguaggio macchina dell’intero programma producendo un programma in linguaggio macchina. L’esecuzione del programma avviene separatamente dal processo di traduzione. C è un linguaggio compilato.

Principali vantaggi dei linguaggi compilati (ad es. C):

- Velocità di esecuzione
- Verifica presenza di errori prima dell’esecuzione

Principali vantaggi dei linguaggi interpretati (ad es. Python):

- Semplicità d’uso

1.2.3. Esempio programma Python

calcololungo.py

```
import time

s = 0
for i in range(0,100):
    s += 2**i
    time.sleep(0.1)

print "Il risultato e': "
print S
```

Individuazione di errori solo a tempo di esecuzione.

Nell'esempio precedente, solo dopo aver svolto tutti i calcoli l'interprete Python rileva e segnala l'errore causato dalla richiesta di stampare il valore di una variabile indefinita (S invece di s), senza poter stampare il risultato dell'elaborazione.

1.2.4. Paradigmi di programmazione

Un ulteriore modo di classificare i linguaggi di programmazione considera i *paradigmi* di programmazione. Questi si distinguono per l'enfasi che pongono sui due aspetti fondamentali: oggetti e operazioni e per il modo con cui operano sui dati.

I paradigmi di programmazione principali sono:

- **imperativo**: enfasi sulle operazioni intese come azioni, comandi, istruzioni che cambiano lo stato dell'elaborazione; gli oggetti sono funzionali alla elaborazione

- **funzionale**: enfasi sulle operazioni intese come funzioni che calcolano risultati; gli oggetti sono funzionali alla elaborazione

- **orientato agli oggetti**: enfasi sugli oggetti che complessivamente rappresentano il dominio di interesse; le operazioni sono funzionali alla rappresentazione

In genere in un programma sono utilizzati più paradigmi di programmazione. Quindi i linguaggi di programmazione forniscono supporto (in misura diversa) per i vari paradigmi.

1.3. Il linguaggio C

Queste dispense trattano il linguaggio di programmazione **C**. C è un linguaggio di programmazione di *alto livello*, compilato, che supporta i paradigmi di programmazione *imperativo* e *funzionale*. La sua estensione C++ supporta anche il paradigma di programmazione *orientata agli oggetti*.

Caratteristiche generali di C:

- estremamente efficiente;

- consente un accesso diretto alla memoria;

- molto usato in ambito scientifico e per la realizzazione di sistemi complessi;
- dispone di librerie di programma ricche e ben sviluppate.

1.3.1. Tecniche di programmazione in C

Scopo del corso è quello di approfondire i principali aspetti della programmazione imperativa e funzionale. Il linguaggio C è lo strumento utilizzato nella trattazione.

In particolare, attraverso il C vengono presentati i concetti di base dei linguaggi di programmazione, con particolare riferimento al modello di esecuzione e all'utilizzo della memoria.

La presentazione del C viene fatta in relazione alle conoscenze già acquisite in Python.

1.3.1.1. Esempio di programma in C

primo.c

```c
/* definizione di funzioni di I/O (printf)*/
#include <stdio.h>

int main() {
  printf("Il mio primo programma C\n");
  printf("... e non sara' l'ultimo.\n");
  return 0;
}
```

Il programma è costituito da una serie di istruzioni o direttive per il compilatore.

Le istruzioni hanno il seguente significato:

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

direttiva per includere la definizione di funzioni e variabili predefinite nel linguaggio relative all'input/output.

```c
int main() {
   ...
}
```

definizione della *funzione* main che racchiude il programma principale.

```
    printf("Il mio primo programma C ");
    printf("... e non sara' l'ultimo.\n");
```

istruzioni di stampa su video della frase:

```
Il mio primo programma C
... e non sara' l'ultimo.
```

printf è la funzione di stampa.
\n è il carattere di ritorno a capo nello schermo.

La sequenza di due istruzioni comporta l'esecuzione delle due istruzioni nell'ordine in cui sono scritte.

```
return 0;
```

è l'istruzione che termina l'esecuzione della funzione main e restituisce il risultato 0.

1.3.1.2. Formato del programma

- Le parole in un programma C sono separate da spazi. Es. `int main`
- C (come Python) è **case-sensitive**, cioè distingue tra caratteri minuscoli e caratteri maiuscoli.
- Si possono lasciare un numero di spazi (e/o di linee vuote) a piacere. Andare a capo equivale a separare due elementi del programma (come uno spazio).
- In C, a differenza di Python, l'**indentazione** non ha alcun effetto sull'esecuzione del programma, essa è comunque molto importante perché rende i programmi più leggibili.
- In C tutte le istruzioni terminano con il carattere ;
- Le sequenze di escape comuni sono le stesse in C e Python (\b: backspace; \n: newline; \t: tab orizzontale; \\: backslash; \': apice singolo; \": apice doppio).

- Una differenza fondamentale tra i due linguaggi consiste nel fatto che il C non include le stringhe tra i tipi primitivi.

Il programma `primo2.c` è equivalente al programma `primo.c`.

primo2.c

```
#include <stdio.h>
int main() { printf("Il mio primo programma C\n");
      printf
("... e non sara' l'ultimo.\n"
      );
    return
      0;
                }
```

1.3.1.3. Commenti

È possibile annotare il testo del programma con dei **commenti**. C dispone di due tipi di commento:

- `/* ... */` delimita un commento che può occupare più righe.
- `//` denota l'inizio di un commento che si estende solo fino alla fine della riga (non sempre supportato)

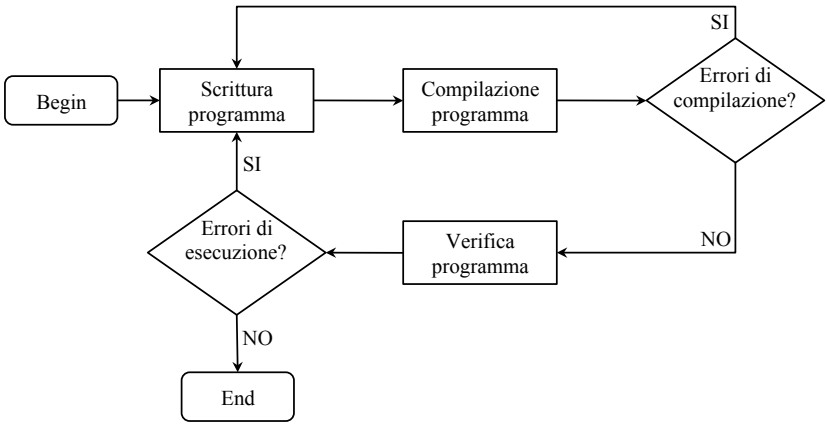
I commenti non hanno alcun effetto sull'esecuzione del programma, sono usati per rendere il programma più leggibile.

1.3.2. Scrivere, compilare ed eseguire un programma C

Le fasi principali dello sviluppo di un programma sono:

1. Preparazione del testo del programma
2. Compilazione del programma
3. Esecuzione del programma compilato

1.3.2.1. Il processo di compilazione



Per un programma C il nome del file deve avere la seguente struttura:

```
nome.c
```

dove:

- *nome* è il nome del file contenente il testo del programma
- `c` è l'estensione che indica che il file contiene un programma C.

Esempio: `primo.c`

1. Scrittura del programma La scrittura di un programma può essere effettuata con qualsiasi programma che consenta la scrittura di un testo (*editor*). Ad esempio: notepad, emacs, gedit, ...

2. Compilazione del programma La compilazione del programma serve a tradurre il programma in una sequenza di comandi direttamente eseguibili dal calcolatore. Il compilatore C più usato è `gcc`, disponibile per diversi sistemi operativi. Per compilare programmi C si può usare anche un compilatore C++, ad esempio `g++`.

In questo corso useremo il compilatore `g++`.

Per compilare un programma C occorre eseguire il seguente comando:

```
g++ -o NomeEseguibile NomeFile.c
```

Esempio: Per compilare il programma `primo.c`, si può usare il comando:

```
g++ -o primo primo.c
```

La compilazione produce come risultato un file chiamato *`NomeEseguibile`*`.exe` in Windows, oppure *`NomeEseguibile`* in Unix/Linux/Mac, che contiene i comandi direttamente eseguibili dal calcolatore.

Se il processo di compilazione è corretto, il compilatore non stampa alcuna informazione, altrimenti stampa dei messaggi di errore.

Talvolta vengono forniti dei *warning*, cioè messaggi che indicano situazioni che potrebbero potenzialmente comportare errori ma che non impediscono al programma di essere compilato.

3. Esecuzione del programma compilato L'esecuzione di un programma si può effettuare solo dopo la compilazione, cioè quando il file `NomeEseguibile.exe` (oppure `NomeEseguibile`) è stato generato.

In C (al contrario di Python) non è possibile eseguire programmi sintatticamente errati.

L'esecuzione del programma avviene inserendo il nome del file eseguibile nel prompt di sistema. Nei sistemi operativi basati su Unix (Linux, Mac, ecc.) è necessario indicare anche il path del file eseguibile, ad esempio usando la sequenza `./` prima del nome del file eseguibile.

Esecuzione in ambiente Windows:

```
NomeEseguibile
```

Esecuzione in ambiente Unix:

```
./NomeEseguibile
```

Esempio: per eseguire il programma `primo.c` si può usare il comando

```
./primo
```

Il risultato dell'esecuzione del programma è quindi la seguente stampa su schermo:

```
Il mio primo programma C
... e non sara' l'ultimo.
```

1.3.2.2. Errori in un programma

Gli errori nei programmi si possono classificare in tre categorie:

Sintassi: riguardano il mancato rispetto delle regole di scrittura del programma e sono individuati dal compilatore.

Semantica: sono dovuti all'impossibilità di assegnare un significato ad un'istruzione (es. variabile non dichiarata). Questi errori a volte sono individuati dal compilatore (errori di semantica statica), altre volte sono individuati a tempo di esecuzione (errori di semantica dinamica).

Logica: sono relativi alle funzionalità realizzate dal programma (differenti da quelle desiderate). Questi errori possono essere individuati solo analizzando il programma o eseguendo test di verifica del programma.

1.3.2.3. Esempio: Soluzione equazioni di secondo grado

equasecondo_err.c

```c
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {

  printF("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  root = sqrt(b*b - 4.0 * a * c)
  root1 = 1/2 * (root + b) / a;
  root2 = -1/2 * (root + b) / a;
  printf("radici di %f x^2 + %f x + %f\n", a, b, c);
  printf("%f e %f\n", root1, root2);
  return 0;
}
```

Il programma precedente contiene diversi tipi di errore.

- printF è scritto in maniera errata (F maiuscola)
- l'istruzione root = sqrt (b*b - 4.0 * a * c) non termina con ;
- tutte le variabili non sono state dichiarate
- 1/2 calcola la divisione tra interi e quindi restituisce 0
- il calcolo di root1 è errato (non calcola correttamente la soluzione)

equasecondo.c

```
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {
  double a,b,c,root,root1,root2;
  printf("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  root = sqrt(b*b - 4.0 * a * c);
  root1 = 0.5 * (root - b) / a;
  root2 = -0.5 * (root + b) / a;
  printf("radici di %f x^2 + %f x + %f\n", a, b, c);
  printf("%f e %f\n", root1, root2);
  return 0;
}
```

1.4. Variabili

Al contrario di Python, in C tutti gli identificatori, in particolare le variabili, devono essere dichiarati prima dell'uso!

La dichiarazione di una variabile richiede di specificare il *tipo* della variabile (ad esempio, `int, double, char,...`) e il nome. Il tipo determina la quantità di memoria allocata alla variabile.

In Python la dichiarazione di variabili non è necessaria in quanto tutte le variabili sono di tipo riferimento ad oggetti.

1.4.1. Dichiarazione di variabili

Le variabili vengono introdotte nel programma attraverso **dichiarazioni di variabile**.

Dichiarazione di variabile

Sintassi:

```
tipo nomeVariabile;
```

- *tipo* è il *tipo* della variabile
- *nomeVariabile* è il nome della variabile da dichiarare

Semantica:
La dichiarazione di una variabile riserva spazio in memoria per la variabile e rende la variabile disponibile nella parte del programma (blocco) dove appare la dichiarazione. È indispensabile dichiarare una variabile prima di usarla.

Esempio:

```
int x;
```

- `int` è il tipo della variabile
- `x` è il nome della variabile

int è il tipo usato per i numeri interi. I tipi di dato primitivi sono illustrati nell'Unità 2.

Dopo questa dichiarazione la variabile x è disponibile per l'utilizzo nel blocco del programma dove appare la dichiarazione (ad esempio, all'interno della funzione main, se la dichiarazione appare lì).

Dichiarazione multipla Si possono anche dichiarare più variabili dello stesso tipo con una sola dichiarazione.

```
tipo nomeVar_1, ..., nomeVar_n;
```

Questa dichiarazione è equivalente a:

```
tipo nomeVar_1;
...
tipo nomeVar_n;
```

1.4.2. Notazione grafica per la rappresentazione di variabili

Una variabile è un riferimento ad un'area di memoria in cui è memorizzato un valore. La dimensione dell'area di memoria dipende dal tipo della variabile ed è assegnata al momento della compilazione.

Per rappresentare le variabili e i loro valori noi useremo la seguente notazione grafica

Il diagramma rappresenta il nome, il tipo, l'indirizzo di memoria e il valore di una variabile. Spesso ometteremo il tipo della variabile e l'indirizzo di memoria. Nel diagramma non viene rappresentata la dimensione dell'area di memoria allocata.

In Python, tutte le variabili sono associate ad aree di memoria della stessa dimensione e contengono riferimenti ad oggetti (che invece sono allocati in aree di memoria di dimensione variabile a seconda del tipo dell'oggetto).

1.4.3. Assegnazione

L'istruzione di assegnazione serve per memorizzare un valore in una variabile.

Assegnazione

Sintassi:

```
nomeVariabile = espressione;
```

- *nomeVariabile* è il nome di una variabile
- *espressione* è una espressione che, valutata, deve restituire un valore del tipo della variabile

Semantica:
Alla variabile *nomeVariabile* viene assegnato il valore dell'*espressione* che si trova a destra del simbolo =. Tale valore deve essere *compatibile* con il tipo della variabile (vedi dopo). Dopo l'assegnazione il valore di una variabile rimane invariato fino alla successiva assegnazione.

In C il risultato dell'espressione a destra dell'operatore = viene memorizzato nell'area di memoria associata alla variabile.

In Python l'assegnazione opera sui riferimenti agli oggetti: il riferimento dell'oggetto risultato dell'espressione a destra dell'operatore = viene associato alla variabile.

Esempio:

```
int x;
x = 3 + 2;
```

- x è una variabile di tipo int
- 3 + 2 è un'espressione intera

Il risultato dell'esecuzione della assegnazione è che dopo di essa x contiene il valore 5.

Esempio:

```
int x;
x = 3.0 + 2.0;
```

Il frammento di codice produce un errore a tempo di compilazione, in quanto il risultato dell'espressione è di tipo reale, mentre la viariabile è di tipo intero. Ulteriori dettagli sulla compatibilità dei tipi sono illustrati nell'Unità 2.

1.4.4. Inizializzazione delle variabili

Inizializzare *una variabile significa specificare un valore da assegnare inizialmente (prima di qualsiasi utilizzo) ad essa.*

Si noti che una variabile non inizializzata non contiene ancora un valore definito e quindi utilizzarla prima di un'assegnazione può comportare errori nel programma. Se la variabile è stata dichiarata ma non ancora inizializzata, allora si può usare l'assegnazione per farlo.

In Python l'uso di una variabile non inizializzata provoca un errore a tempo di esecuzione. In C l'uso di una variabile dichiarata, ma non inizializzata, non genera errori di compilazione o di esecuzione, ma può comportare errori logici.

Esempio: Il seguente programma contiene un errore logico: x e y non sono state inizializzate prima dell'espressione. Il risultato risulta non

definito (dipende effettivamente dallo stato della memoria al momento in cui viene eseguito il programma).

```
int main() {
    int x, y, z;
    // ERRORE le variabili x e y sono indefinite
    z = x * y;
    ...
}
```

Le variabili possono essere inizializzate anche al momento della loro dichiarazione.

```
int main() {
    int x=1, y, z;
    y = 2;
    // OK le variabili x e y sono inizializzate
    z = x * y;
    ...
}
```

1.5. Funzioni

Le *funzioni* sono moduli di programma che svolgono un particolare compito.

Come in Python le funzioni possono essere predefinite o scritte dall'utente.

Vediamo innanzitutto come si invoca (chiama) una funzione.

Invocazione di funzione

Sintassi:

```
nomeFunzione(parametri)
```

- *nomeFunzione*(...) è la funzione invocata
- *parametri* sono i parametri passati alla funzione

Semantica:

Invoca una funzione fornendole eventuali parametri addizionali. L'invocazione di una funzione comporta l'esecuzione

dell'operazione associata ed, in genere, la restituzione di un valore.

Esempio:

```
sqrt(169)
```

- `sqrt` è la funzione che calcola la radice quadrata di un numero
- `169` è il parametro (o argomento) passato alla funzione
- la funzione restituisce un risultato (`13`) che viene usato dall'istruzione che ha chiamato la funzione o dall'espressione in cui la funzione è inserita.

1.5.1. Intestazione di una funzione

L'**intestazione** (o prototipo) di una funzione consiste nella segnatura (nome e parametri) e nel *tipo* del risultato.

Esempi:

```
double sqrt(double x)
```

```
double pow(double b, double e)
```

Nota: Al contrario di Python, in C bisogna definire esplicitamente il tipo del valore restituito da una funzione.

1.5.2. Parametri di una funzione

I parametri di una funzione sono i valori passati dal modulo chiamante alla funzione per poter svolgere i calcoli.

Esempio: la funzione `sqrt(double x)` deve essere invocata passandole un parametro che rappresenta il valore di cui vogliamo calcolare la radice quadrata.

In generale, i parametri passati come argomenti possono essere espressioni complesse formate a loro volta da invocazioni di altre funzioni.

Esempio:

```
sqrt(pow(5,2)+pow(12,2))
```

calcola la radice quadrata di $5^2 + 12^2$.

1.5.3. Risultato di una funzione

Il risultato calcolato da una funzione viene restituito al blocco di codice che ha chiamato la funzione stessa.

Esempio:

l'istruzione

```
printf ("%f", sqrt(pow(5,2)+pow(12,2)));
```

stampa il valore 13 (cioè il risultato di $\sqrt{5^2 + 12^2}$).

Infatti, la funzione `pow(5,2)` restituisce il valore 25, la funzione `pow(12,2)` restituisce 144, l'operatore `+` calcola la somma dei due valori e restituisce 169, la funzione `sqrt` restituisce il valore 13, e infine il valore 13 viene usato dalla funzione `printf` per stampare tale valore sulla console di output.

1.5.4. Funzioni definite in math.h

Le principali funzioni e costanti matematiche sono definite nel file di sistema `math.h` , che bisogna quindi includere mediante la direttiva `#include <...>`.

Per conoscere le funzioni definite in `math.h` (ed in generale le funzioni matematiche messe a disposizione dal C) si veda:

`http://www.gnu.org/software/libc/manual/html_mono/libc.html#Mathematics`

Nome funzione	Significato
`cos(x)`	coseno
`sin(x)`	seno
`tan(x)`	tangente
`acos(x)`	arcocoseno
`asin(x)`	arcoseno
`atan(x)`	arcotangente
`atan2(x,y)`	arcotangente con due parametri ($\tan^{-1}(y/x)$)

Nome funzione	Significato
`pow(x,y)`	potenza (x^y)
`sqrt(x)`	radice quadrata
`exp(x)`	esponenziale (e^x)
`log(x)`	logaritmo naturale
`log10(x)`	logaritmo in base 10

Nome funzione	Significato
`ceil(x)`	arrotondamento in eccesso
`fabs(x)`	valore assoluto
`floor(x)`	arrotondamento in difetto
`round(x)`	arrotondamento all'intero più vicino

1.5.5. La funzione exit

La funzione `exit(int status)` fa terminare il programma, restituendo il valore passato come argomento al sistema operativo. Il valore 0, restituito con `exit(0)`, indica per convenzione che il programma è terminato regolarmente. Valori non nulli (nell'intervallo $[1-255]$) vengono associati a condizioni d'uscita anomale. Tipicamente, il valore 1 viene usato per indicare fallimento generico. Il risultato dell'esecuzione permette al sistema di rispondere in maniera specifica ai diversi tipi di errore.

1.5.6. Funzioni di Input/Output

In C, la principale funzione di output è `printf`, cui vanno specificati il formato caratterizzato dal simbolo `%` ed i parametri, cioè le stringhe o le espressioni da stampare.

Esempio: L'istruzione

```
printf("Primo = %d, secondo = %f", 5, 5.0);
```

esegue la stampa a video della stringa

```
Primo = 5, secondo = 5.000000
```

sostituendo alla stringa `%d`, il valore del secondo argomento (5) della chiamata ed alla stringa `%f` il valore del terzo argomento (5.0). La stringa

%d indica che il valore (5) da stampare è un intero. La stringa %f indica che il valore (5.0) da stampare è un reale. Non c'è limite al numero di argomenti che si possono passare alla funzione.

La funzione scanf permette di leggere input da tastiera, specificando il formato dei dati letti, caratterizzato dal simbolo %, ed i parametri, cioè le variabili da leggere, con la notazione &.

Esempio: Il seguente frammento di codice legge da tastiera due valori reali e li memorizza nelle variabili di tipo double a e b (secondo l'ordine di inserimento).

```
double a,b; // Dichiara due variabili reali
scanf("%lf%lf", &a, &b);
```

La stringa %lf indica che il carattere da leggere è di tipo double.

Non ci sono limiti al numero di valori che si possono leggere in input, e quindi al numero di parametri che si possono passare alla funzione.

Ulteriori dettagli sulle funzioni printf e scanf saranno visti in seguito.

1.6. Blocco di istruzioni

Un **blocco di istruzioni** raggruppa più istruzioni in un'unica istruzione composta.

Blocco di istruzioni

Sintassi:

```
{
  istruzione
  ...
  istruzione
}
```

- *istruzione* è una qualsiasi istruzione C

Semantica:
Le istruzioni del blocco vengono eseguite in sequenza. Le variabili dichiarate internamente al blocco non sono visibili all'esterno del blocco.

Esempio:

```
int a, b;
...
{
  printf("dato1 = %d\n",a);
  printf("dato2 = %d\n",b);
}
```

1.6.1. Campo d'azione delle variabili

Un blocco di istruzioni può contenere al suo interno dichiarazioni di variabili. Una variabile dichiarata dentro un blocco ha come **campo di azione** il blocco stesso, inclusi eventuali blocchi interni. Questo significa che *la variabile è visibile nel blocco e in tutti i suoi blocchi interni, ma non è visibile all'esterno del blocco*.

In Python, una variabile può essere usata dopo la sua inizializzazione indipendentemente dal blocco di codice in cui viene inizializzata. Quindi, una variabile inizializzata in un blocco di codice interno può essere usata in blocchi di codice esterni successivi.

Esempio:

campoDazione.c

```
// campo d'azione
#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {
  double a = 2.3;      // a double
  int i = 1;           // i int
  printf("liv. 0  a = %f\n", a); // 2.3
  printf("liv. 0  i = %d\n", i); // 1
  {
    printf("liv. 1  a = %f\n", a); // 2.3
    printf("liv. 1  i = %d\n", i); // 1

    double i = 0.1;    //  i double
    {
      double r = 5.5; // r double
      i = i + 1;       // i double
      printf("liv. 2.1 r = %f\n", r); // 5.5
      printf("liv. 2.1 i = %f\n", i); // 1.1
    }
    // printf("liv. 1  r = %f\n",r);  // ERRORE
    printf("liv. 1  i = %f\n",i);      // 1.1
```

```
    {
      int r = 4;        // r int
      printf("liv. 2.2 r = %d\n", r); // 4
      printf("liv. 2.2 a = %f\n", a); // 2.3
    }
  }
  i = i + 1;            // i int
  printf("liv. 0  i = %d\n", i); // 2
  return 0;
}
```

1.7. Struttura di un programma

Non ci sono regole stringenti per organizzare un programma, ma la necessità di *definire prima di usare* suggerisce di usare il seguente schema generale:

```
direttive del compilatore
dichiarazione variabili globali (esterne)
dichiarazione intestazione di funzioni
dichiarazione main
dichiarazione funzioni
```

1.7.1. Variabili globali

In C si possono introdurre delle dichiarazioni di variabili prima della dichiarazione della funzione main.

Queste dichiarazioni vengono considerate *globali* (talvolta denominate *esterne*, ma noi utilizzeremo questo termine con un significato specifico che verrà illustrato più avanti).

Una dichiarazione globale ha come campo d'azione il file nel quale è definita.

Esempio:

equasecondo_varglobale.c

```
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>   /* funzioni di I/O */
#include <math.h>    /* funzioni matematiche */

float unmezzo = 0.5;
```

```
int main () {
  double a,b,c,root,root1,root2;
  printf("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  root = sqrt(b*b - 4.0 * a * c);
  root1 = unmezzo * (root - b) / a;
  root2 = - unmezzo * (root + b) / a;
  printf("radici di %f x^2 + %f x + %f\n", a, b, c);
  printf("%f e %f\n", root1, root2 );
  return 0;
}
```

L'uso di variabili globali diventa significativo quando si introducono nel programma diverse definizioni di funzione che possono quindi tutte fare riferimento alle variabili globali del file in cui sono definite.

Nel seguito vedremo come organizzare le definizioni del programma su più file. Questa funzionalità richiederà una rivisitazione del concetto di variabile globale.

1.7.2. Decomposizione in moduli

In C (come in Python) il meccanismo per modularizzare il programma è costituito dalla suddivisione in funzioni (non ci sono script, dato che il linguaggio è compilato).

La definizione di funzioni sarà trattata più avanti.

1.8. Istruzioni condizionali

In C le istruzioni condizionali if-else ed if sono simili a quelle di Python, ma ci sono alcune differenze da sottolineare:

- la sintassi è leggermente diversa (parentesi per la condizione e assenza del carattere :);
- quando ci sono più istruzioni in un ramo occorre racchiuderle in un blocco di istruzioni delimitate da { ... } (non serve l'indentazione! Ma è desiderabile);
- le condizioni in C sono espressioni il cui valore 0 corrisponde a false e qualunque altro valore corrisponde a true.

1.8.1. Soluzione equazioni secondo grado: rivista

equasecondo-if.c

```c
/* Calcolo radici di un'equazione di secondo grado */

#include <stdio.h>  /* funzioni di I/O */
#include <math.h>   /* funzioni matematiche */

int main () {
  double a, b, c;
  printf("Inserisci coefficienti a b c: \n");
  scanf("%lf%lf%lf", &a, &b, &c);
  double delta = b*b - 4.0 * a * c;
  if (delta > 0.0) {
    double root = sqrt(delta);
    double root1 = 0.5 * (root - b) / a;
    double root2 = - 0.5 * (root + b) / a;
    printf("radici reali: %lf e %lf\n", root1, root2);
  } else if (delta < 0.0) {
    double root = sqrt(-delta);
    double real_part = - 0.5 * b / a;
    double imag_part = 0.5 * root / a;
    printf("radici complesse: %lf + i * %lf e %lf - i
    * %lf\n",
     real_part, imag_part, real_part, imag_part) ;
  } else {
    double root1 = -0.5*b/a;
    printf("radici coincidenti: %lf\n", root1);
  }
  return 0;
}
```

1.8.2. L'istruzione if-else

L'istruzione if-else consente di effettuare una *selezione* a 2 vie.

Istruzione if-else

Sintassi:

```
if (condizione)
  istruzione-then
else
  istruzione-else
```

- *condizione* è un'espressione booleana, valutata true oppure false

- *istruzione-then* è una singola istruzione (detta anche il *ramo-then* dell'istruzione if-else)
- *istruzione-else* è una singola istruzione (detta anche il *ramo-else* dell'istruzione if-else)

Semantica:
Viene valutata prima la *condizione*. Se la valutazione fornisce un valore diverso da 0 (true), viene eseguita *istruzione-then*, altrimenti viene eseguita *istruzione-else*. In entrambi i casi l'esecuzione prosegue con l'istruzione che segue l'istruzione if-else.

Esempio:

```
int a, b;
...
if (a > b)
  printf("maggiore = %d\n", a);
else
  printf("maggiore = %d\n", b);
```

L'esecuzione di questa istruzione if-else produce la stampa a video della stringa "maggiore = ", seguita dal valore del maggiore tra a e b.

1.8.3. La variante if

La parte else di un'istruzione if-else è opzionale.

Se manca si parla di istruzione if, la quale consente di eseguire una parte di codice solo se è verificata una condizione.

Istruzione if

Sintassi:

```
if (condizione)
  istruzione-then
```

- *condizione* è un'espressione booleana
- *istruzione-then* è una singola istruzione (detta anche il *ramo-then* dell'istruzione if)

Semantica:
Viene valutata prima la *condizione*. Se la valutazione fornisce un valore diverso da 0 (true), viene eseguita *istruzione-then* e si procede con l'istruzione che segue l'istruzione if. Altrimenti si procede direttamente con l'istruzione che segue l'istruzione if.

Esempio:

```
double a = 1.0;
if (a>0) printf("a positivo.\n");
```

L'esecuzione di questa istruzione if produce la stampa sul canale di output della stringa "a positivo" quando la condizione (a>0) è vera, altrimenti non stampa nulla.

1.8.4. Condizione nell'istruzione if-else

La condizione in un'istruzione if-else può essere costruita usando:
1) operatore di confronto (==, !=, >, <, >=, <=) applicato a variabili (o espressioni) di un tipo primitivo;
Esempio:

```
int a, b, c;
...
if (a > b + c)
   ...
```

2) chiamata ad una *funzione;*
Esempio:

```
int a,b;
...
if (verifica(a,b))
   ...
```

verifica(a,b) è una funzione che verifica una condizione sui valori a e b.

3) espressione booleana *complessa*, ottenuta applicando gli operatori booleani !, && e || a espressioni più semplici;

Esempio:

```
int a, b, c, d;
int trovato;
...
if ((a>(b+c)) || (a == d) && ! trovato)
   ...
```

1.8.5. Attenzione alle condizioni

Ci sono delle formulazioni delle condizioni che possono diventare facilmente sorgenti di errori:

```
int a;
double b;

if (a=0)     // assegnazione
   ...
if (b==0)    // approssimazione
   ...
if (a==b)    // conversione di tipo
   ...
```

Nel primo caso if (a=0) l'errore riguarda l'uso dell'operatore di assegnazione invece dell'operatore di confronto. In questo caso il compilatore non segnala alcuna anomalia, dato che il risultato dell'espressione (a=0) viene convertito nel valore booleano corrispondente cioè false, e sarà sempre eseguito il ramo else dell'istruzione condizionale. Una possibile contromisura potrebbe essere quella di scrivere sempre prima il valore costante (0=a). In questo modo, il compilatore segnala un errore dovuto ad un'assegnazione scorretta (non può esserci un valore nella parte destra dell'operatore).

Nel secondo caso if (b==0), occorre accertarsi che errori di approssimazione non portino ad avere sistematicamente il valore false.

Nel terzo caso bisogna notare che il simbolo == in C denota l'uguaglianza di valori contenuti nella memoria, mentre in Python indica l'uguaglianza *strutturale*.

1.8.6. Valutazione di una condizione complessa

La condizione di un'istruzione if-else può essere un'espressione booleana complessa, nella quale compaiono gli operatori logici &&, ||, e !.

Si deve tenere presente che le sottoespressioni relative a tali operatori vengono valutate da sinistra a destra al seguente modo:

- nel valutare $(e_1$ && $e_2)$, se la valutazione di e_1 restituisce false, allora e_2 **non viene valutata**.

- nel valutare $(e_1 \,||\, e_2)$, se la valutazione di e_1 restituisce true, allora e_2 **non viene valutata**.

Infatti, se e_1 è falso, $(e_1$ && $e_2)$ risulta falso, indipendentemente dal valore di e_2. Analogamente se e_1 è vero, $(e_1$ || $e_2)$ risulta vero, indipendentemente dal valore di e_2.

In generale, bisogna sempre tenere conto del fatto che in questi casi e_2 *non viene valutata.*

Esempio:

```
int i;
...
if (i > 0 && fact(i) > 100) {
  ...
}
```

Si noti che la funzione fact(i) non viene invocata nel caso in cui i abbia valore minore o uguale a 0.

Istruzioni if-else che fanno uso di espressioni booleane complesse potrebbero essere riscritte attraverso l'uso di if-else annidati. In generale questo comporta però la necessità di duplicare codice.

1.8.6.1. Uso dell'operatore di congiunzione &&

```
if ((x < y) && (y < z))
  printf("y compreso tra x e z");
else
  printf("y non compreso tra x e z");
```

corrisponde a

```
if (x < y)
  if (y < z)
     printf("y compreso tra x e z");
  else
    printf("y non compreso tra x e z");
else
  printf("y non compreso tra x e z");
```

Si noti come in questo caso, eliminando la condizione composta, il codice del ramo-else debba essere duplicato.

1.8.6.2. Uso dell'operatore di disgiunzione ||

```
if ((x == 1) || (x == 2))
  printf("x uguale a 1 o a 2");
else
  printf("x diverso da 1 e da 2");
```

corrisponde a

```
if (x == 1)
  printf("x uguale a 1 o a 2");
else if (x == 2)
  printf("x uguale a 1 o a 2");
else
  printf("x diverso da 1 e da 2");
```

Si noti come in questo caso, eliminando la condizione composta il codice del ramo-then debba essere duplicato.

1.8.7. Uso del blocco di istruzioni nell'if-else

Il ramo-then e il ramo-else di un'istruzione if-else possono essere una qualsiasi istruzione C, in particolare un blocco di istruzioni.

```
if (a>b) {
  printf("maggiore = %d\n",a);
  printf("minore = %d\n",b);
}
```

Esempio:

Dati mese ed anno, calcolare mese ed anno del mese successivo.

```
int mese, anno, mesesucc, annosucc;
...
if (mese == 12) {
  mesesucc = 1;
  annosucc = anno + 1;
}
else {
  mesesucc = mese + 1;
  annosucc = anno;
}
```

1.8.8. If annidati

Si hanno quando l'istruzione del ramo-then o del ramo-else è un'istruzione if-else o if.

Esempio:

```
int giorno,mese,anno, giornosucc,mesesucc,annosucc;
...
if (mese == 12) {
  if (giorno == 31) {
    giornosucc = 1;
    mesesucc = 1;
    annosucc = anno + 1;
  }
  else {
    giornosucc = giorno + 1;
    mesesucc = mese;
    annosucc = anno;
  }
}
else {
  ...
}
```

1.8.8.1. If annidati, condizioni mutuamente esclusive

Un caso comune di utilizzo degli if annidati è quello in cui le condizioni degli if annidati si escludono mutuamente.

Esempio:

In base al valore della temperatura (intero) stampare un messaggio secondo la seguente tabella:

temperatura t	messaggio
$30 < t$	molto caldo
$20 < t \leq 30$	caldo
$10 < t \leq 20$	gradevole
$t \leq 10$	freddo

```
int temp;
...
if (30 < temp)
  printf("molto caldo");
else if (20 < temp)
  printf("caldo");
else if (10 < temp)
  printf("gradevole");
else
  printf("freddo");
```

Osservazioni:

- al livello più esterno abbiamo un'unica istruzione if-else
- l'ordine in cui vengono specificate le condizioni è importante
- non serve che la seconda condizione sia composta, ad es. (20 < temp) && (temp <= 30)
- ogni else si riferisce all'if immediatamente precedente

1.8.9. Ambiguità if-else

Consideriamo il seguente frammento di codice:

```
if (a > 0) if (b > 0) printf("b positivo"); else
    printf("???");
```

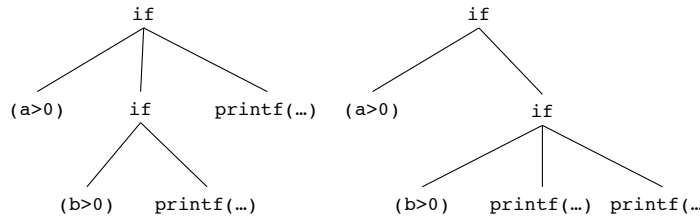
printf("???"); potrebbe essere la parte else

- del primo if: quindi printf("a negativo");
- del secondo if: quindi printf("b negativo");

1.8.9.1. Grammatica delle istruzioni condizionali

```
istruzione = istruzione-semplice | istruzione-composta
istruzione-composta = istruzione-condizionale | ...
istruzione-condizionale = istruzione-if | istruzione-switch
istruzione-if = "if" ( condizione ) istruzione "else" istruzione
              | "if" ( condizione ) istruzione
```

Richiamo: Una grammatica si dice ambigua quando si possono associare 2 alberi sintattici diversi ad una stessa frase riconosciuta dal linguaggio



L'ambiguità si risolve considerando che **un else fa sempre riferimento all'if più vicino**:

```
if (a > 0)
  if (b > 0)
    printf("b positivo");
  else
    printf("b negativo");
```

È sempre possibile usare il blocco di istruzioni { ... } per disambiguare istruzioni if-else annidate. In particolare, perché un else si riferisca ad un if che non è quello immediatamente precedente, quest'ultimo deve essere racchiuso in un blocco:

```
if (a > 0) {
  if (b > 0)
    printf("b positivo");
}
else
    printf("a negativo");
```

1.8.10. Espressione condizionale

C mette a disposizione un operatore di selezione che permette di costruire un'**espressione condizionale**. L'uso di un'espressione condizionale può in alcuni casi semplificare il codice rispetto all'uso di un'istruzione if-else.

Espressione condizionale

Sintassi:

```
condizione?espressione-1:espressione-2
```

- *condizione* è un'espressione booleana
- *espressione-1* e *espressione-2* sono due espressioni qualsiasi, che devono essere dello stesso tipo

Semantica:
Valuta *condizione*. Se il risultato è true, allora valuta *espressione-1* e ne restituisce il valore, altrimenti valuta *espressione-2* e ne restituisce il valore.

Esempio:

```
printf("maggiore = %\n", (a > b)? a : b);
```

L'istruzione nell'esempio, che fa uso di un'espressione condizionale, è equivalente a:

```
if (a > b)
  printf("maggiore = %d", a);
else
  printf("maggiore = %d", b);
```

Si noti che l'operatore di selezione è simile all'istruzione if-else, ma agisce ad un livello sintattico diverso:

- l'operatore di selezione combina *espressioni* e restituisce un'altra espressione; quindi, può essere usato ovunque può essere usata un'espressione;

- l'istruzione if-else raggruppa *istruzioni*, ottenendo un'istruzione composta.

1.8.11. L'istruzione switch

Se dobbiamo realizzare una **selezione a più vie**, possiamo usare diversi if-else annidati. C mette però a disposizione un'istruzione specifica che può essere usata in alcuni casi per realizzare in modo più semplice una selezione a più vie.

Istruzione switch

Sintassi:

```
switch (espressione) {
  case etichetta-1: istruzioni-1
     break;
  ...
  case etichetta-n: istruzioni-n
     break;
  default: istruzioni-default
}
```

- *espressione* è un'espressione intera o di tipo char
- *etichetta-1*, ..., *etichetta-n* sono espressioni intere (o carattere) costanti; ovvero, possono contenere solo letterali interi (o carattere) o costanti inizializzate con espressioni costanti; una espressione non può essere ripetuta come etichetta in più case
- *istruzioni-1*, ..., *istruzioni-n* e *istruzioni-default* sono sequenze di istruzioni qualsiasi
- la parte default è opzionale

Semantica:

1. viene prima valutata *espressione*
2. viene cercato il primo *i* per cui il valore di *espressione* è pari a *etichetta-i*

3. se si è trovato tale *i*, allora vengono eseguite *istruzioni-i* altrimenti vengono eseguite *istruzioni-default*
4. l'esecuzione procede con l'istruzione successiva all'istruzione switch

Esempio:

```
int i;
...
switch (i) {
  case 0: printf("zero"); break;
  case 1: printf("uno"); break;
  case 2: printf("due"); break;
  default: printf("minore di zero o maggiore di due");
}
```

Quando i è pari a 0 (rispettivamente 1, 2) viene stampato zero (rispettivamente uno, due), mentre quando i è minore di 0 oppure maggiore di due, viene stampato minore di zero o maggiore di due.

Se abbiamo più valori per cui eseguire le stesse istruzioni, si possono raggruppare i diversi case:

```
case v1:
case v2:
...
case vn:
  <istruzioni>
  break;
```

Esempio: Calcolo dei giorni di un mese.

```
int mese, giorniDelMese;
...
switch (mese) {
case 4: case 6: case 9: case 11:
  giorniDelMese = 30; break;
case 1: case 3: case 5: case 7: case 8: case 10: case
    12:
  giorniDelMese = 31; break;
case 2:
  giorniDelMese = 28; break;
default:
  giorniDelMese = 0;
  printf("Mese non valido");
}
printf("Giorni del mese %d: %d \n", mese,
    giorniDelMese);
```

1.8.11.1. Osservazioni sull'istruzione switch

L'*espressione* usata per la selezione può essere una qualsiasi espressione C che restituisce un valore *intero* o *carattere* (ma non un valore reale).

I valori specificati nei vari *case* devono invece essere *espressioni costanti*, ovvero il loro valore deve essere noto a tempo di compilazione. In particolare, non possono essere espressioni che fanno riferimento a variabili. Il seguente frammento di codice è sbagliato:

```
int a;
...
switch (a) {
  case a<0: printf("negativo");
            // ERRORE: a<0 non e' una costante
  case 0:   printf("nullo");
  case a>0: printf("positivo");
            // ERRORE: a>0 non e' una costante
}
```

Ne segue che l'*utilità dell'istruzione switch è limitata*.

1.8.11.2. Omissione del break

In realtà, non si richiede che nei case di un'istruzione switch l'ultima istruzione sia break.

In assenza di break si prosegue con l'esecuzione delle istruzioni successive a quella corrispondente all'etichetta valutata, finché non si arriva ad una istruzione break o al termine dell'istruzione switch.

```
int lati;  //  massimo numero di lati del poligono (al
     piu' 6)
...
printf("Poligoni con al piu' %d lati: ",lati);
switch (lati) {
  case 6: printf("esagono, ");
  case 5: printf("pentagono, ");
  case 4: printf("rettangolo, ");
  case 3: printf("triangolo");
          break;
  case 2: case 1: printf("nessuno");
                  break;
  default: printf("\n");
           printf("Immetti un valore <= 6.\n");
```

Se il valore di lati è pari a 5, allora il precedente codice stampa:

```
pentagono, rettangolo, triangolo
```

Nota: quando si omettono i break, *diventa rilevante l'ordine* in cui vengono scritti i vari case. Questo può essere spesso causa di errori. Quindi, *è buona norma mettere break come ultima istruzione di ogni case*.

1.9. Istruzioni di ciclo

In C il ciclo while è simile a quello di Python, ma ci sono alcune differenze da sottolineare:

- la sintassi è leggermente diversa (parentesi per la condizione e assenza del carattere :);
- quando ci sono più istruzioni nel corpo del ciclo occorre racchiuderle in un blocco { ... } (non serve l'indentazione!);
- per le condizioni vale quanto detto a proposito dei condizionali.

1.9.1. Ciclo while

L'istruzione while consente la ripetizione di una istruzione.

Istruzione while

Sintassi:

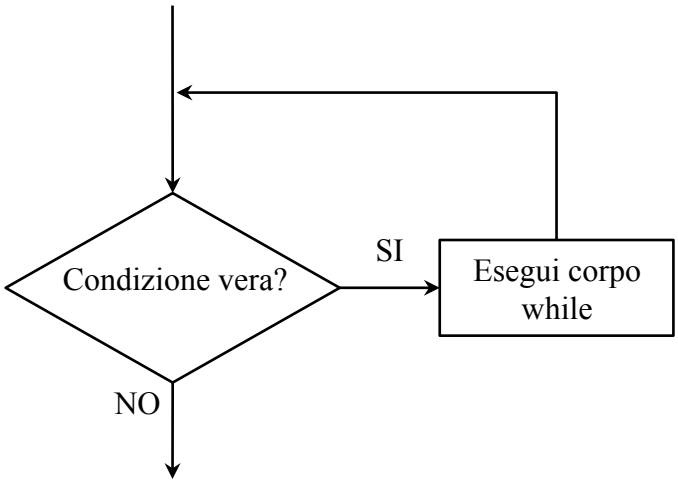
```
while (condizione)
  istruzione
```

- *condizione* è un'espressione booleana
- *istruzione* è una singola istruzione (detta anche il *corpo* del ciclo)

Nota: dato che è possibile, facendo uso di un blocco, raggruppare più istruzioni in una singola istruzione composta, è di fatto possibile avere più istruzioni nel corpo del ciclo.
Semantica:

- viene valutata prima la *condizione*
- se è true (valore diverso da 0), viene eseguita *istruzione* e si torna a valutare la *condizione*, procedendo così fino a quando *condizione* diventa falsa (valore uguale a 0)
- a questo punto si passa ad eseguire l'istruzione che segue il ciclo while

1.9.2. Semantica dell'istruzione di ciclo while



Quindi, il corpo del ciclo viene eseguito finché la *condizione* si mantiene vera. Non appena questa diventa falsa si esce dal ciclo e si continua l'esecuzione con l'istruzione successiva al while.

Esempio: Stampa di 100 asterischi.

```
int i = 0;
while (i < 100) {
    printf("*");
    i++;
}
```

1.9.3. Cicli definiti ed indefiniti

Concettualmente, si distinguono due tipi di ciclo, che si differenziano in base a come viene determinato il numero di *iterazioni* (ripetizioni del corpo del ciclo):

- Nei **cicli definiti** il numero di iterazioni è noto prima di iniziare l'esecuzione del ciclo.

  Esempio: per 10 volte ripeti la stampa di un *.

- Nei **cicli indefiniti** il numero di iterazioni non è noto prima di iniziare l'esecuzione del ciclo, ma è legato al verificarsi di una

condizione (questo dipende a sua volta dalle operazioni eseguite nel corpo del ciclo).

Esempio: finchè l'utente non sceglie di smettere, stampa un * e chiedi all'utente se smettere.

In C, come in altri linguaggi, entrambi i tipi di ciclo possono essere realizzati attraverso l'istruzione while.

1.9.4. Esempio di ciclo while: potenza

```
int base, esponente, potenza;
base = ...;
esponente = ...;
potenza = 1;
while (esponente > 0) {
    potenza = potenza * base;
    esponente--;
}
```

Si noti che si tratta di fatto di un *ciclo definito*, poiché il numero di iterazioni dipende solo dai valori delle variabili che sono stati fissati prima di iniziare l'esecuzione del ciclo.

1.9.5. Elementi caratteristici nella progettazione di un ciclo

```
inizializzazione
while (condizione) {
   operazione
   passo successivo
}
```

- *inizializzazione*: definizione del valore delle variabili utilizzate nel ciclo prima dell'inizio dell'esecuzione del ciclo (prima dell'istruzione di ciclo)
  Es. `potenza = 1;`
- *condizione*: espressione valutata all'inizio di ogni iterazione, il cui valore di verità determina l'esecuzione del corpo del ciclo o la fine del ciclo
  Es. `(esponente > 0)`

- *operazione del ciclo*: calcolo del risultato parziale ad ogni iterazione del ciclo (nel corpo del ciclo)
  Es. potenza = potenza * base;

- *passo successivo*: operazione di incremento/decremento della variabile che controlla le ripetizioni del ciclo (nel corpo del ciclo)
  Es. esponente–;

Una volta progettato il ciclo occorre verificarne la **terminazione**. In particolare, occorre verificare che l'esecuzione delle istruzioni del ciclo possa modificare il valore della condizione in modo da renderla falsa.

Esempio: l'istruzione esponente–; consente di rendere la condizione (esponente > 0) falsa, se esponente è un numero intero positivo.

1.9.6. Errori comuni nella scrittura di cicli while

Dimenticarsi di inizializzare una variabile che viene utilizzata nella condizione del ciclo. Si ricordi che la prima volta che la condizione viene verificata è prima di iniziare ad eseguire il corpo del ciclo.

Esempio:

```
int i;
while (i != 0) {
    printf("%d", i*i);
    printf("prossimo intero");
}
```

La variabile i non è inizializzata, quindi il risultato della condizione del ciclo while risulta indefinita.

Dimenticarsi di aggiornare le variabili che compaiono nella condizione del ciclo. Il ciclo non terminerà mai.

Esempio:

```
int i;
scanf("%d",&i);
while (i != 0) {
    printf("%d", i*i);
    printf("prossimo intero");
}
```

Manca l'istruzione di lettura del prossimo elemento da elaborare.

Sbagliare di 1 il numero di iterazioni Tipicamente è dovuto ad un errore nella condizione del ciclo o nell'inizializzazione di variabili usate nella condizione.

Esempio: Specifica: stampa di 10 asterischi.

```
int i = 0;
while (i <= 10) {  // corretto: (i < 10)
    printf("*");
    i++;
}
```

Ma il programma stampa 11 asterischi.

Per evitare questo tipo di errori, verificare con dati semplici. Nell'esempio, se si verifica con la stampa di 1 asterisco (sostituendo 10 con 1 nella condizione del ciclo), ci si accorge immediatamente che in realtà verrebbero stampati 2 asterischi.

1.9.7. Schemi di ciclo

Esistono alcune operazioni di base molto comuni che richiedono l'utilizzo di cicli:

- *contatore*: conta il numero di valori in un insieme
- *accumulatore*: accumula i valori di un insieme
- *valori caratteristici di un insieme*: determina un valore caratteristico tra i valori in un insieme (ad esempio, il massimo, quando sui valori dell'insieme è definito un ordinamento)

Ciascun tipo di operazione ha alla base uno schema comune dell'istruzione di ciclo.

1.9.7.1. Ciclo controllato da contatore

Una situazione comune di utilizzo dei cicli è quella in cui il ciclo fa uso di una variabile (detta *di controllo*) che ad ogni iterazione varia di un valore costante, ed il cui valore determina la fine del ciclo.

Esempio: stampa i quadrati degli interi da 1 a 10.

```c
int i = 1;
while (i <= 10) {
    printf("%d\n", i*i);
    i++;
}
```

I seguenti elementi sono comuni ai cicli controllati da contatore:

- si fa uso di una *variabile di controllo* del ciclo (detta anche *contatore* o *indice* del ciclo)
  Esempio i
- *inizializzazione* della variabile di controllo
  Esempio i = 1;
- *incremento* (o decremento) della variabile di controllo ad ogni iterazione
  Esempio i++;
- verifica se si è raggiunto il *valore finale della variabile di controllo*
  Es. (i <= 10)

1.9.7.2. Ciclo di accumulazione

Il ciclo di accumulazione è caratterizzato dall'uso di una variabile (detta *accumulatore*) che ad ogni iterazione accumula il risultato di una operazione ed il cui valore alla fine del ciclo è il risultato complessivo dell'operazione.

Esempio: somma i quadrati degli interi da 1 a 10.

```c
int i = 1; // contatore
int s = 0; // accumulatore
while (i <= 10) {
    s += i*i;
    i++;
}
printf("%d\n", s);
```

I seguenti elementi sono comuni ai cicli di accumulazione:

- si fa uso di una *variabile accumulatore*
  Esempio s

- *inizializzazione* della variabile accumulatore all'elemento neutro dell'operazione da svolgere (0 per somma, 1 per prodotto, true per AND logico, false per OR logico, ...)
  Esempio s = 0;
- *accumulazione* dei valori ad ogni iterazione
  Esempio s += *valore*;

1.9.8. Altre istruzioni di ciclo

In C ci sono tre forme di istruzioni di ciclo:

- ciclo while
- ciclo for
- ciclo do

Il ciclo while sarebbe sufficiente per esprimere qualsiasi ciclo esprimibile con for e do. In alcune situazioni è però più conveniente codificare un algoritmo utilizzando gli altri tipi di ciclo.

1.9.9. Ciclo for

Istruzione for

Sintassi:

```
for (inizializzazione; condizione; incremento)
  istruzione
```

- *inizializzazione* è un'espressione con side-effect di inizializzazione di una variabile di controllo (tipicamente un'assegnazione), che può anche essere una dichiarazione con inizializzazione
- *condizione* è un'espressione booleana
- *incremento* è un'espressione con side-effect che tipicamente consiste nell'incremento della variabile di controllo
- *istruzione* è una singola istruzione (detta anche il *corpo* del ciclo for)

Semantica: è equivalente a

```
{ inizializzazione; while (condizione) {
istruzione; incremento; } }
```

Esempio: Stampa di 100 asterischi

```
for (int i = 0; i < 100; i++)
    printf("*");
```

è equivalente a

```
int i = 0;
while (i < 100) {
    printf("*");
    i++;
}
```

1.9.9.1. Osservazioni sul ciclo for

- Se la variabile di controllo è dichiarata nella parte di *inizializzazione*, allora il suo campo di azione è limitato all'istruzione for. Questo si evince dalla traduzione nel ciclo while, nella quale l'intero codice corrispondente al ciclo for è racchiuso in un blocco.

Esempio:

```
for (int i = 0; i < 10; i++) {
    printf("%d\n", i*i);
}

printf("valore di i = %d\n",i);
// ERRORE! i non e' visibile
```

Ciascuna delle tre parti del for (*inizializzazione*, *condizione* e *incremento*) può anche mancare. In questo caso i ";" vanno messi lo stesso. Se manca *condizione*, viene assunta pari a true.

Esempio:

```
for (int i = 0; ; ) { ... }
```

La sintassi del for permette che le tre parti siano delle espressioni qualsiasi, purché *inizializzazione;* e *incremento;* siano delle istruzioni (in particolare, facciano side-effect).

Nell'uso del ciclo for è però buona norma:

- usare le tre parti del for in base al significato sopra descritto, con riferimento ad una *variabile di controllo*;
- non modificare la variabile di controllo nel corpo del ciclo.

In generale, *inizializzazione* e/o *incremento* possono essere una sequenza di espressioni con side-effect separate da ",". Questo permette di inizializzare e/o incrementare più variabili contemporaneamente.

Esempio: calcola e stampa le prime 10 potenze di 2.

```
int i, potDi2;
for (i=0, potDi2=1; i < 10; i++, potDi2 *= 2)
    printf("2 alla %d  = %d\n", i, potDi2);
```

1.9.9.2. Ciclo for: esempi

Il ciclo for è usato principalmente per realizzare *cicli definiti*.

```
for (int i = 1; i <= 10; i++)
// valori: 1, 2, 3, ..., 10
```

```
for (int i = 10; i >= 1; i--)
// valori: 10, 9, 8, ..., 2, 1
```

```
for (int i = -4; i <= 4; i = i+2)
// valori: -4, -2, 0, 2, 4
```

```
for (int i = 0; i >= -10; i = i-3)
// valori: 0, -3, -6, -9
```

1.9.10. Ciclo do

Nel ciclo while la condizione di fine ciclo viene controllata all'inizio di ogni iterazione. Il ciclo do è simile al ciclo while, con la sola differenza che *la condizione di fine ciclo viene controllata alla fine di ogni iterazione*.

Istruzione do

Sintassi:

```
do
  istruzione
while (condizione);
```

- *condizione* è un'espressione booleana
- *istruzione* è una singola istruzione (detta anche il *corpo* del ciclo)

Semantica: è equivalente a

```
istruzione;
while (condizione)
  istruzione
```

Quindi:

- viene prima eseguita *istruzione*
- poi viene valutata prima la *condizione*, e se è vera, si torna ad eseguire istruzione *istruzione*, procedendo così fino a quando *condizione* diventa falsa
- a questo punto si passa ad eseguire l'istruzione che segue il ciclo do

Esempio: stampa di 100 asterischi

```
int i = 0;
do {
    printf("*");
    i++;
} while (i < 100);
```

1.9.10.1. Osservazioni sul ciclo do

Dal momento che la condizione di fine ciclo viene valutata solo dopo aver eseguito il corpo del ciclo, ne segue che:

- *Il corpo del ciclo viene eseguito almeno una volta.* Quindi, il ciclo do consente la ripetizione di istruzioni, quando si vuole che queste istruzioni vengano in ogni caso eseguite almeno una volta.
- Non è in generale necessario inizializzare le variabili che compaiono nella condizione prima di iniziare l'esecuzione del ciclo. È sufficiente che tali variabili vengano inizializzate nel corpo del ciclo stesso.

Esempio: somma gli interi letti da input fino a quando non viene immesso 0.

```
int i;
int somma = 0;
do {
    printf("inserisci intero (0 per terminare): ");
    scanf("%d", &i);
    somma = somma + i;
} while (i != 0);
printf("\n Totale = %d\n", somma);
```

Si noti che la sintassi del ciclo do richiede che ci sia un ";" dopo while *(condizione)*. Per aumentare la leggibilità del programma, in particolare per evitare di confondere la parte while *(condizione)*; di un ciclo do, con un'istruzione while con corpo vuoto, conviene in ogni caso racchiudere il corpo del ciclo do in un blocco, ed indentare il codice come mostrato nell'esempio di sopra.

1.9.10.2. Esempio di ciclo do: validazione dell'input

Spesso è necessario effettuare una validazione di un dato di input immesso dall'utente, e *nel caso l'utente abbia immesso un dato non valido,*

ripetere la richiesta del dato stesso. Questo può essere fatto utilizzando un ciclo do.

Esempio: scrivere un frammento di programma che legge da input un intero, ripetendo la lettura fino a quando non viene immesso un intero positivo, e restituisce l'intero positivo letto.

```
...
int i;
do {
    printf("inserisci un intero positivo: ");
    scanf("%d", &i);
} while (i <= 0);
// all'uscita dal ciclo i e' un intero positivo
...
```

Si noti che il precedente codice non è in grado di gestire correttamente tutte le situazioni di input errato, ad esempio il caso in cui l'utente immette un carattere alfabetico invece di un intero. Una trattazione completa delle tecniche di validazione dell'input esula dagli scopi del presente testo.

1.9.10.3. Equivalenza tra ciclo while e ciclo do

Come segue dalla semantica, ogni ciclo do può essere sostituito da un ciclo while *equivalente*. Questo comporta però la necessità di duplicare il corpo del ciclo do.

Esempio:

```
do {
    printf("inserisci un intero positivo: ");
    scanf("%d", &i);
} while (i <= 0);
```

equivale a

```
printf("inserisci un intero positivo: ");
scanf("%d", &i);
while (i <= 0) {
    printf("inserisci un intero positivo: ");
    scanf("%d", &i);
}
```

1.9.11. Insieme completo di istruzioni di controllo

Due programmi si dicono **equivalenti** se, sottoposti agli stessi dati di ingresso,

- entrambi non terminano, oppure
- entrambi terminano producendo gli stessi risultati in uscita.

Un insieme di istruzioni di controllo del flusso si dice **completo** se per ogni programma nel linguaggio ne esiste uno equivalente scritto solo con le strutture di controllo contenute nell'insieme.

Teorema di Böhm e Jacopini

> La *sequenza*, l'*istruzione if-else* e l'*istruzione while* formano un insieme di istruzioni completo.

1.9.12. Esempio: calcolo del massimo comun divisore (MCD)

Specifica:

Vogliamo realizzare un programma che, dati due interi positivi x ed y, calcoli e restituisca il massimo comun divisore $\text{mcd}(x, y)$.

Il massimo comun divisore di due interi x ed y è il più grande intero che divide sia x che y senza resto.

Es.: $\text{mcd}(12, 8) = 4$
$\text{mcd}(12, 6) = 6$
$\text{mcd}(12, 7) = 1$

1.9.12.1. MCD: sfruttando direttamente la definizione

- cerchiamo il massimo tra i divisori comuni di x ed y
- osservazione: $1 \leq \text{mcd}(x, y) \leq \min(x, y)$
  Quindi, si provano i numeri compresi tra 1 e $\min(x, y)$.
- Conviene iniziare da $\min(\text{x}, \text{y})$ e scendere verso 1. Appena si è trovato un divisore comune di x ed y, lo si restituisce.

Primo raffinamento dell'algoritmo:

```
massimoComunDivisore di int x ed int y {
  int mcd;
  inizializza mcd al minimo tra x ed y
  while ((mcd > 1) && (divisore comune non trovato))
```

```
    if (mcd divide sia x che y)
      si è trovato un divisore comune
    else
      mcd--;
}
```

Osservazioni:

- Il ciclo termina sempre perchè ad ogni iterazione
  - o si è trovato un divisore,
  - o si decrementa mcd di 1 (al più si arriva ad 1).
- Per verificare se si è trovato il mcd si utilizza una variabile booleana (usata nella condizione del ciclo).
- Per verificare se x (o y) divide mcd si usa l'operatore %.

1.9.12.2. MCD: osservazioni

Quante volte viene eseguito il ciclo nel precedente algoritmo?

- *caso migliore:* 1 volta, quando x divide y o viceversa
  Es. mcd(500, 1000)
- *caso peggiore:* min(x,y) volte, quando mcd(x,y) = 1
  Es. mcd(500, 1001)

Quindi, il precedente algoritmo si comporta male se x e y sono grandi e $\mathrm{mcd}(x, y)$ è piccolo.

1.9.12.3. MCD: usando il metodo di Euclide

Il **metodo di Euclide** permette di ridursi più velocemente a numeri più piccoli, sfruttando la seguente proprietà:

$$\mathrm{mcd}(x, y) = \begin{cases} x \,(\text{oppure } y) & \text{se } x = y \\ \mathrm{mcd}(x - y, y) & \text{se } x > y \\ \mathrm{mcd}(x, y - x) & \text{se } x < y \end{cases}$$

Questa proprietà si dimostra facilmente, mostrando che i divisori comuni di x e y sono anche divisori di $x - y$ (nel caso in cui $x > y$).

Es. $\text{mcd}(12,8) = \text{mcd}(12-8,8) = \text{mcd}(4,8-4) = 4$ Per ottenere un algoritmo si applica ripetutamente il procedimento fino a che non si ottiene che $x = y$. Ad esempio:

x	y	maggiore − minore
210	63	147
147	63	84
84	63	21
21	63	42
21	42	21
21	21	$\Longrightarrow$ mcd(21,21) = mcd(21,42) = $\cdots$ = mcd(210,63)

L'algoritmo può essere realizzato in C al seguente modo:

```
int main ()
{
  int x = 210;
  int y = 63;
  while (x != y) {
    if (x > y)
      x = x - y;
    else                    // significa che y > x
      y = y - x;
  }
  printf("mcd = %d\n", x);
  return 0;
}
```

1.9.12.4. MCD: ammissibilità dei valori degli argomenti

Cosa succede nel precedente algoritmo nei seguenti casi:

- Se $x = y = 0$?
  Il risultato è 0.
- Se $x = 0$ e $y > 0$ (o viceversa)?
  Il risultato dovrebbe essere y, ma l'algoritmo entra in *un ciclo infinito*.
- Se $x < 0$ e y è qualunque (o viceversa)?
  L'algoritmo entra in *un ciclo infinito*.

Quindi, se si vuole tenere conto del fatto che il programma possa essere eseguito con valori qualsiasi dei parametri, è necessario inserire una opportuna verifica.

```c
int main ()
{
  int x = 210;
  int y = 63;
  if ((x > 0) && (y > 0)) {
    while (x != y)
      if (x > y)
        x = x - y;
      else                // significa che y > x
        y = y - x;
     printf("mcd = %d\n", x);
  }
  else printf("dati errati");
 return 0;
}
```

1.9.12.5. MCD: osservazioni sul metodo di Euclide

Cosa succede nel precedente algoritmo se x è molto maggiore di y (o viceversa)?

Esempio:

mcd(1000, 2)		mcd(1001, 500)	
1000	2	1001	500
998	2	501	500
996	2	1	500
⋮	⋮	⋮	⋮
2	2	1	1

Per comprimere questa lunga sequenza di sottrazioni è sufficiente osservare che quello che in fondo si calcola è il resto della divisione intera.

1.9.12.6. MCD: metodo di Euclide con i resti

Metodo di Euclide: sia $x = y \cdot k + r$ (con $0 \leq r < y$)

$$\mathrm{mcd}(x,y) = \begin{cases} y, & \text{se } r = 0 \ (x \text{ multiplo di } y) \\ \mathrm{mcd}(r,y), & \text{se } r \neq 0 \end{cases}$$

L'algoritmo può essere realizzato in C nel seguente modo:

```c
int main ()
{
  int x = ...;
  int y = ...;
  while ((x != 0) && (y != 0)) {
    if (x > y)
      x = x % y;
    else
      y = y % x;
  }
  if (x != 0) printf("mcd = %d\n", x);
  else printf("mcd = %d\n", y);
  return 0;
}
```

1.9.13. Cicli annidati (o doppi cicli)

Il corpo di un ciclo può contenere a sua volta un ciclo, chiamato **ciclo annidato**. È possibile annidare un qualunque numero di cicli.

Esempio: stampa della tavola Pitagorica

```c
int main () {
    int N = 10; // dimensione della tavola
    int riga, colonna;
    for (riga = 1; riga <= N; riga++) {
        for (colonna = 1; colonna <= N; colonna++)
            printf(" %3d  ", riga * colonna);
        printf("\n");
    }
    return 0;
}
```

Output prodotto:

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  4   8  12  16  20  24  28  32  36  40
  5  10  15  20  25  30  35  40  45  50
  6  12  18  24  30  36  42  48  54  60
  7  14  21  28  35  42  49  56  63  70
  8  16  24  32  40  48  56  64  72  80
  9  18  27  36  45  54  63  72  81  90
 10  20  30  40  50  60  70  80  90 100
```

1.9.13.1. Esempio di doppio ciclo: stampa di una piramide di asterischi

```c
int altezza;
printf("Altezza = ");
scanf("%d", &altezza);
for (int riga = 1; riga <= altezza; riga++) {
    // 1 iterazione per ogni riga della piramide
    for (int i = 1; i <= altezza - riga; i++)
        printf(" ");  // stampa spazi bianchi iniziali
    for (int i = 1; i <= riga * 2 - 1; i++)
        printf("*"); // stampa sequenza di asterischi
    printf("\n");    // va a capo: fine riga
}
```

Nell'esempio della stampa della tavola pitagorica, il numero di iterazioni del ciclo più interno era fisso. In generale, il numero di iterazioni di un ciclo più interno può dipendere dall'iterazione del ciclo più esterno.

Esempio: stampa di una piramide di asterischi

```
Piramide di altezza 4
   *
  ***
 *****
*******
```

riga	blank	*
1	3	1
2	2	3
3	1	5
4	0	7

Per la stampa della generica riga r: stampa $(\text{altezza} - r)$ blank e $(2 \cdot r - 1)$ asterischi.

1.9.13.2. Esempio: potenza con un doppio ciclo

```
int base = ...;
int esponente= ...;
int risultato = 1;
while (esponente > 0) {
   esponente--;
   // risultato = risultato * base
   int moltiplicando = risultato;
   int moltiplicatore = base;
   int prodotto = 0;
   while (moltiplicatore > 0) {
        moltiplicatore--;
        prodotto = prodotto + moltiplicando;
    }
    risultato = prodotto;
}
printf("%d", risultato);
```

1.9.13.3. Esempio: numeri di Fibonacci

Al nome del matematico Fibonacci è legata una famosa serie numerica, così definita:

- Fib(0)=1;
- Fib(1)=1;
- Fib(n)=Fib(n-1)+Fib(n-2)

Scrivere un programma che legge un intero positivo (altrimenti reitera la richiesta), e calcola il relativo numero di Fibonacci. li

1.9.13.4. Numeri di Fibonacci: soluzione

fibo.c

```
#include <stdio.h>  /* funzioni di I/O */

int main ()
{
  int Fib,N,n1,n2;
  printf("Calcolo il numero di Fibonacci\n");
  do {
    printf("Inserisci un numero (>=0): ");
    scanf("%d",&N);
  } while (N<0);
```

```
    if ((0 == N) || (1 == N))
      Fib=1;
    else {
      n1=1; n2=1;
      for (int i = 2; i <= N; i++) {
        Fib= n2 + n1;
        n2 = n1;
        n1 = Fib;
      }
    }
    printf("\nNumero di Fibonacci di %d = %d\n", N, Fib)
      ;
    return 0;
}
```

1.10. Istruzioni di controllo del flusso

Le istruzioni di controllo del flusso determinano la successiva istruzione da eseguire. In questo senso, le istruzioni if-else, if, switch, while, for, do sono istruzioni di controllo del flusso. Esse non permettono però di stabilire in modo arbitrario la prossima istruzione da eseguire, ma forniscono una *strutturazione del programma* che determina il flusso di esecuzione.

C, come altri linguaggi, permette però anche di usare (seppur limitatatamente) *istruzioni di salto*, che sono istruzioni di controllo del flusso che causano l'interruzione del flusso di esecuzione ed il salto ad una istruzione diversa dalla successiva nella sequenza di istruzioni del programma.

Istruzioni di salto:

- break (salto all'istruzione successiva al ciclo o allo switch corrente)
- continue (salto alla condizione del ciclo)
- goto (salto all'istruzione etichettata)

Note:

- L'uso di istruzioni di salto va in generale evitato. Esse vanno usate solo in casi particolari. In questo corso ne faremo solo un accenno.
- Anche l'istruzione return può essere usata per modificare il flusso di esecuzione, nelle funzioni definite (vedi Unità 5).

1.10.1. Istruzione break per uscire da un ciclo

Abbiamo già visto nell'Unità 3 che l'istruzione break permette di uscire da un'istruzione switch. In generale, *break permette di uscire prematuramente da un'istruzione switch, while, for o do.*

Esempio: ciclo che calcola la radice quadrata di 10 reali letti da input. Si vuole interrompere il ciclo non appena viene immesso un reale negativo.

```
double a;
for (int i = 0; i < 10; i++) {
  printf("Immetti un reale nonnegativo ");
  scanf("%d", &a);
  if (a >= 0)
    printf("%f",sqrt(a));
  else {
    printf("Errore");
    break;
  }
}
```

Nota: nel caso di cicli annidati o di switch annidati dentro un ciclo, l'esecuzione di un break fa uscire di *un solo livello*.

1.10.1.1. Eliminazione del break

L'esecuzione di un'istruzione break *altera il flusso di controllo*. Quindi, quando viene usata nei cicli:

- si perde la strutturazione del programma
- si guadagna in efficienza rispetto ad implementare lo stesso comportamento senza fare uso del break.

In generale, è sempre possibile **eliminare un'istruzione break**. Ad esempio, l'istruzione

```
while (condizione) {
  istruzioni-1
  if (condizione-break) break;
  istruzioni-2
}
```

equivale a

```
finito = 0;
while (condizione && !finito) {
  istruzioni-1
  if (condizione-break) finito = 1;
  else { istruzioni-2 }
}
```

La scelta se eliminare o meno un'istruzione break deve essere fatta valutando da una parte il guadagno in efficienza del programma con break rispetto a quello senza, e d'altra parte la perdita in leggibilità dovuta alla presenza del break.

1.10.1.2. Esempio di eliminazione del break

Con riferimento all'esempio precedente, il codice senza break è il seguente:

```
double a;
int errore = 0;

for (int i = 0; (i < 10) && !errore; i++) {
  printf("Immetti un reale nonnegativo ");
  scanf("%lf", &a);
  if (a > 0)
    printf("%f",sqrt(a));
  else {
    printf("Errore");
    errore = 1;
  }
}
```

1.10.2. Istruzione continue

L'istruzione continue si applica solo ai cicli e *comporta il passaggio alla successiva iterazione del ciclo*, saltando le eventuali istruzioni che seguono nel corpo del ciclo.

Esempio:

```
for (int i = 0; i < n; i++) {
  if (i % 2 == 0)
    continue;
  printf("%d",i);
}
```

stampa i valori dispari tra 0 e n.

Si osservi che l'istruzione continue all'interno di un ciclo for fa comunque eseguire le istruzioni-incremento del ciclo.

Nota: Un possibile uso di continue è all'interno di un ciclo di lettura, nel quale vogliamo effettuare un'elaborazione sui dati letti solo se è verificata una determinata condizione. Bisogna però assicurarsi che ad ogni iterazione del ciclo venga in ogni caso letto il prossimo valore. Altrimenti il ciclo non termina.

```
leggi il prossimo dato;
while (condizione) {
  if (condizione-sul-dato-corrente)
    continue; // ERRORE! viene saltata la lettura del
dato
  elabora il dato;
  leggi il prossimo dato;
}
```

1.10.3. Istruzioni di salto goto

L'istruzione di salto goto deriva dal linguaggio macchina dove ha un ruolo fondamentale per consentire la realizzazione dei cicli. Il citato teorema di Böhm e Jacopini ha mostrato, tra l'altro, che essa non è necessaria ai fini della completezza del linguaggio.

L'istruzione di salto comporta una interruzione del flusso dell'esecuzione del programma, che prosegue con l'istruzione specificata nel goto.

Per consentire il salto, le istruzioni possono avere delle **etichette**:

```
etichetta: istruzione-ciclo;
```

Le etichette devono essere degli identificatori.

L'istruzione goto ha quindi la seguente sintassi:

```
goto etichetta;
```

etichetta è un identificatore che deve individuare una istruzione del programma.

L'esecuzione dell'istruzione goto imposta il programma a proseguire con l'istruzione specificata dall'etichetta.

Esempio: interruzione di un ciclo

```
int i = 0; float x;
while (i < 100) {
  printf("inserisci numero positivo: ");
  scanf("%f", &x);
  if (x<=0) goto errore;
  i++;
}
errore: printf("errore: programma interrotto\n");
```

L'istruzione goto errore interrompe il ciclo.

Nota: l'uso di etichette e di istruzioni goto è considerata una cattiva pratica di programmazione e va riservato a casi particolari. In questo corso *non ne faremo uso*!