

Fondamenti di Informatica

Allievi Automatici

A.A. 2015-16

Introduzione al C

Breve storia del linguaggio C

- Il linguaggio C:
 - Evoluzione (D. Ritchie) di linguaggi precedenti (tra cui il B)
 - Usato per sviluppare il sistema operativo UNIX
 - Motivo iniziale del suo successo
 - Gran parte dei sistemi operativi oggi sono scritti in C (C++)
 - Portabile tra calcolatori differenti
 - Consolidato già intorno agli anni '70
- Standardizzazione (importante in informatica!):
 - Anni '80: molte varianti leggermente diverse e incompatibili
 - 1989: comitato per la standardizzazione (aggiornato nel '99)

Le librerie standard del C

- I programmi C consistono di “pezzi” (meglio: moduli) chiamati ***funzioni***:
 - Il programmatore può:
 - creare egli stesso le sue funzioni
 - usare le funzioni già offerte dal compilatore (di libreria)
 - organizzare le sue funzioni in librerie
- Le funzioni (programmate o preesistenti) si usano come “mattoncini” (blocchi base/building blocks)
- Le funzioni di libreria sono scritte con accuratezza, sono efficienti e sono ***portabili***
 - **Quindi**: se esiste già una funzione, è **inutile** riprogrammarla!

Il nostro primo programma C

```
/* The first C program */

#include <stdio.h>

int main()
{
    printf("Hello World!\n");

    return 0;
}

> Hello World!
>
```

- commenti

- testo racchiuso da `/* ... */`
- è ignorato dal compilatore
- per descrivere i programmi

- `#include`

- direttiva al *preprocessore* [#]
- carica il contenuto di un file
 - in questo caso `stdio.h`
[**s**tandard **i**nput/**o**utput]
- Inclusione delle **librerie**

Osservazioni (I)

- `int main() { ... corpo ... }`
 - I programmi C contengono la definizione di una o più funzioni, una delle quali deve essere il `main()`
 - In questo caso c'è la definizione della sola funzione `main()`
 - Le parentesi tonde () indicano che è una funzione
 - Le funzioni tipicamente “restituiscono” un valore
 - Al termine della loro esecuzione, per comunicarne “l'esito”
 - `int` significa che il `main` restituisce un valore intero
 - Il **corpo** di ogni funzione è incluso in un **blocco**, racchiuso in parentesi graffe { }
 - Il corpo costituisce la definizione della funzione. Raccoglie le istruzioni che specificano il “comportamento” della funzione

Osservazioni (II)

- `printf("Hello World!\n");`
- È una **istruzione** di stampa
 - L'intera riga si chiama istruzione (o *statement*)
 - Visualizza una sequenza di caratteri indicata tra le doppie virgolette "..."
- L'istruzione richiede l'esecuzione di una funzione (la funzione printf)
 - Si dice che l'istruzione costituisce una chiamata alla funzione
 - La funzione è definita altrove (in una libreria standard)
 - Nel nostro programma c'è solo la “chiamata”
 - Le parentesi tonde raccolgono i parametri passati alla funzione
 - `f(x) ... printf("...stringa...")` Qui il parametro è una stringa
- Il carattere '`\`' (*backslash*) si premette ai cosiddetti “caratteri di escape”:
 - il **carattere** '`\n`' indica che `printf()` deve fare qualcosa di speciale
 - è il carattere di escape che indica “new line” (vai a capo)

Osservazioni (III)

- **return 0;**
 - È un modo di terminare una funzione
 - Lo 0 è “restituito” come effetto dell’esecuzione
 - Restituito a chi? A chi la ha invocata!!
 - È compatibile con la dichiarazione **int main()**
 - **return 0**, in questo caso significa che il programma è terminato senza anomalie
 - è una semplice convenzione: 0 indica la fine di una esecuzione corretta, ad altri valori si associano interpretazioni legate al tipo di errore che si è verificato
 - è una convenzione tipica del main. Per le altre funzioni normalmente si usano convenzioni e interpretazioni legate al significato della funzione

Collegatore (Linker)

- Quando si chiama una funzione (`printf` nell' esempio), il collegatore la cerca nelle librerie
 - nella libreria standard, ed eventualmente in quelle indicate da una apposita direttiva al preprocessore
 - `#include <nomelibreria.h>`
- La funzione è copiata dalla libreria e inserita nel programma oggetto
- Se c'è un errore nel nome della funzione il collegatore lo rileva (non riesce a trovarla)
 - Il linker è normalmente integrato con il compilatore nell'ambiente di sviluppo


```
/* Sum of two integers */  
#include <stdio.h>
```

Un altro programma C

```
int main() {  
    int integer1, integer2, sum;           /* declaration */  
  
    printf("Enter first integer\n");       /* prompt */  
    scanf("%d", &integer1 );              /* read an integer */  
    printf("Enter second integer\n");      /* prompt */  
    scanf("%d", &integer2);               /* read an integer */  
    sum = integer1 + integer2;             /* assignment */  
    printf("Sum is %d\n\n", sum );         /* print sum */  
  
    return 0;                             /* successful end */  
}
```

```
> Enter first integer  
> 45  
> Enter second integer  
> 72  
> Sum is 117  
>  
>
```

Osservazioni (I)

- `int integer1, integer2, sum;`
 - Dichiarazione delle **variabili**
 - *locazioni di memoria* dove sono memorizzati i dati manipolati dal programma
 - `int` : le variabili conterranno numeri interi
 - `integer1, integer2, sum` - nomi di variabili
 - Le dichiarazioni devono apparire *prima delle istruzioni eseguibili*
 - le variabili **prima** si dichiarano, **poi** si usano

Osservazioni (II)

- **scanf ("%d", &integer1) ;**
 - Acquisisce un dato in ingresso dall'utente:
 - Qui abbiamo due argomenti:
 - **%d** : indica che il dato atteso è un intero decimale
 - e sarà *interpretato* come intero decimale
 - **&integer1** - indica l'indirizzo della locazione di memoria corrispondente alla variabile **integer1**
 - &: operatore che estrae l'indirizzo delle variabili
 - **scanf** **interrompe il flusso di esecuzione** (bloccante)
 - L'utente risponde alla **scanf** digitando un numero e premendo *enter* (*invio*) per **far ripartire l'esecuzione**

Osservazioni (III)

- = (operatore di assegnamento)
 - Assegna un valore a una variabile
 - È un operatore *binario* (cioè con *due* operandi):
`sum assume il valore variable1 + variable2`
- `printf("Sum is %d\n\n", sum);`
 - Come in `scanf`, `%d` indica un valore decimale
 - Cioè che il contenuto della variabile sarà interpretato come tale
 - `sum` indica quale variabile sarà visualizzata a terminale
 - La `printf` può avere un numero variabile di parametri
 - `printf("Sum of %d and %d is %d\n\n", integer1, integer2, sum);`
 - Intere espressioni possono essere argomenti della `printf`.
Ad esempio, si poteva anche fare direttamente il calcolo:
`printf("Sum is %d\n\n", integer1+integer2);`

Il “livello” del linguaggio C

- Il C ha un livello più alto rispetto al linguaggio assembler
 - L'uso di **funzioni** è il segno della maggiore astrazione ammessa dal C
 - Permette di non dettagliare **ogni volta** tutte le operazioni
 - Permette di scrivere programmi (→algoritmi) **sintetici**
 - **Un algoritmo sintetico è tipicamente più comprensibile per l'occhio umano**
 - Compromesso tra la pedante esattezza della specifica in linguaggio macchina e l'estrema sintesi dell'intuizione umana
 - $(x+y)-(z+w)$
 - Quindici istruzioni che “fanno perdere di vista l'obiettivo”

Il programma $(x+y)-(z+w)$ in C e in assembler

```
int main() {  
    int x, y, z, w;  
    scanf("%d%d%d%d", &x, &y, &z, &w);  
    printf("\nRisultato:%d", (x+y)-(z+w));  
    return 0;  
}
```

In questa versione del programma C la “variabile d’appoggio” RIS non è esplicitata: il compilatore si fa carico della gestione dei risultati intermedi (linguaggio di livello più alto)

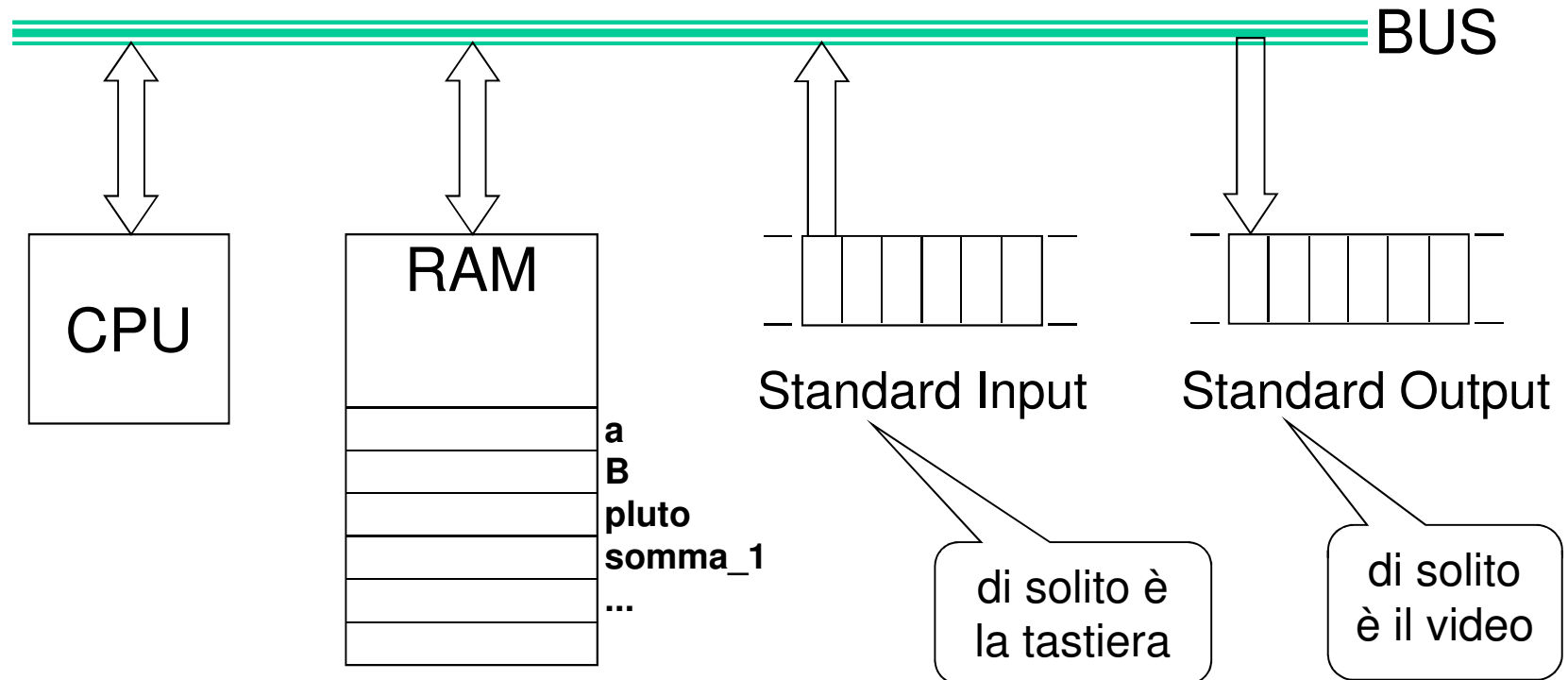
*Disponendo dei soli registri A e B, è **necessario** allocare una variabile in memoria. Avendo a disposizione un processore con più di due registri, non lo sarebbe*

0	READ	X
1	READ	Y
2	READ	Z
3	READ	W
4	LOADA	Z
5	LOADB	W
6	ADD	
7	STOREA	RIS
8	LOADA	X
9	LOADB	Y
10	ADD	
11	LOADB	RIS
12	DIF	
13	STOREA	RIS
14	WRITE	RIS
15	HALT	
16	...int....(X).....	
17	...int....(Y).....	
18	...int....(Z).....	
19	...int....(W).....	
20	...int...(RIS)...	

Un po' di ordine...

La macchina astratta C

Algoritmi e programmi sono definiti in funzione del loro esecutore
L'esecutore dei programmi C è una macchina astratta



Standard I/O

- Un programma C ha due periferiche “standard” di ingresso e uscita
 - **stdin** : standard input (tastiera)
 - **stdout** : standard output (terminale video)
- che possono essere viste come “**sequenze**” (flussi) di byte, di norma interpretati come caratteri

Memoria

- Divisa in celle elementari
- Ogni cella può contenere un dato
- I dati possono essere
 - Numeri
 - Caratteri
 - Stringhe (sequenze di caratteri in celle adiacenti)
 - ...
- Semplificazioni / idealizzazioni / approssimazioni / astrazioni
 - Nessun limite al numero delle celle
 - Nessun limite ai valori numerici contenuti

Le variabili in memoria centrale

- Variabili: corrispondono a *locazioni* di memoria
 - Ogni variabile ha
 - un **nome** (*identificatore*)
 - un **tipo** (insieme dei *valori* e delle *operazioni* ammissibili)
 - una **dimensione** (*normalmente misurata in Byte*)
 - un **indirizzo** (*individua la cella [o le celle] di memoria*)
 - un **valore**
 - La lettura *non modifica* i valori delle variabili
 - Inserendo un nuovo valore (per assegnamento, o con una **scanf**), si *sostituisce* (e quindi si **distrugge**) il vecchio valore



Le variabili

- Rappresentano i *dati su cui lavora il programma*
- Sono denotate mediante identificatori:

`a, B, Pluto, somma_1, ...`

- Variabili diverse devono avere identificatori diversi
- A una variabile ci si riferisce sempre con lo stesso identificatore
- Durante l'esecuzione del programma le variabili hanno sempre un valore ben definito
 - Può essere significativo o non significativo, ma c'è
 - Perciò le variabili devono essere sempre inizializzate in modo opportuno [per evitare errori e sorprese]

Identificatori e parole chiave

- I nomi di variabili devono essere **identificatori**:
 - Sequenze di *lettere*, *cifre*, *underscore* ‘_’,
 - *Case sensitive*
 - cioè “sensibili al maiuscolo”: A è identificatore diverso da a
 - Il primo carattere dev'essere una lettera
 - Esempi:
 - h12 i a_modo_mio identificatoreValidoPerUnaVariabileC DopoDomani
nonce2senza4_v01 v09 v10 v17 V17 h7_25 lasciapassareA38 b8ne
- Ci sono “parole-chiave” (o *keyword*) riservate
 - ad esempio `int`, `return`, ...
 - Non si possono usare come nomi di variabili
 - Occorre ricordarle a memoria
 - non sono tantissime, è facile (*meglio: non è difficile*)

Struttura di un programma C



Struttura di un programma C

- **Parte dichiarativa:** contiene le dichiarazioni degli elementi del programma
 - Dati, ed eventualmente funzioni (ma solo nella parte globale)
- **Parte esecutiva:** contiene le istruzioni da eseguire, che ricadono nelle categorie:
 - Istruzioni di assegnamento (=)
 - Strutture di controllo:
 - Condizionali (if-then-else e switch)
 - Iterative, o **cicli** (while, do e for)
 - Istruzioni di Input/Output (printf, scanf, ...)

Variabili globali e locali

- Variabili **globali**: sono dichiarate fuori dalle funzioni
 - per convenzione: all'inizio, dopo le `#include`
- Variabili **locali**: sono dichiarate internamente alle funzioni, prima della parte esecutiva
- In caso di programmi monomodulo (cioè contenenti una sola funzione, il `main()`), variabili globali e locali sono equivalenti
 - È pertanto indifferente il luogo in cui sono dichiarate
- La differenza si ha nei programmi multimodulo

Commenti

- Porzioni di testo racchiuse tra `/*` e `*/`
 - Se un commento si estende da un certo punto fino alla fine di una sola riga si può anche usare `//`

```
/* Programma che non fa nulla ma mostra
   i due modi per inserire commenti in C */
int main {
    int valore; // Dich. inutile: variabile mai usata
    return 0;
}
```
- I commenti sono ignorati dal compilatore
- Aumentare leggibilità e facilità di modifica di un programma
 - A distanza di tempo, per persone diverse dall'autore, ...

Istruzioni semplici e composte

(simple and compound statements)

- Sequenze di *istruzioni semplici*
 - Ogni istruzione semplice termina con ;
 - ; è detto il “terminatore” dell’istruzione
- Si possono raggruppare più istruzioni in sequenza tra { e } a costituire un **blocco**
 - Il blocco costituisce una “super-istruzione”
- Non è necessario il ; dopo }, in quanto il blocco è già una istruzione
 - e non necessita del terminatore per diventarla

Istruzioni di assegnamento

<variabile> = <espressione> ;

dove *<espressione>* può essere

- un valore ***costante***
- una ***variabile***
- una combinazione di espressioni costruita mediante ***operatori*** (e.g. aritmetici +, −, *, /, %) e ***parentesi***

Esempi di assegnamento

```
x = 23;  
w = 'a';  
y = z;  
alfa = x + y;  
r3 = ( alfa * 43 - xgg ) * ( delta - 32 * j );  
x = x + 1;
```

Abbreviazioni (*operatori* di assegnamento):

```
a = a + 7;    a = a * 5;    a = a + 1;    a = a - 1;  
a += 7;       a *= 5;       ++a;         --a;
```

Istruzioni della forma *variabile = variabile operatore espressione*;
si possono scrivere come: *variabile operatore = espressione*;

Esecuzione degli assegnamenti

1. **valutazione dell'espressione** che compare a *destra* del simbolo =
(il valore delle variabili che vi compaiono si trova memorizzato nelle celle corrispondenti, e da lì è letto)
2. **memorizzazione del risultato** dell'espressione nella variabile a *sinistra* del simbolo =

Aritmetica (I)

- Operatori aritmetici in C:
 - $*$ per la moltiplicazione e $/$ per la divisione
 - La divisione tra interi *elimina il resto* (**quoziente**):
 $13 / 5$ è uguale a 2
 - L'operatore *modulo* calcola il **resto** della divisione:
 $13 \% 5$ è uguale a 3
- Precedenza degli operatori:
 - Come in aritmetica, $*$ e $/$ hanno priorità su $+$ e $-$
 - si usano le parentesi quando c'è ambiguità
 - Per esempio: la media aritmetica di **a**, **b**, **c**:
 $a + b + c / 3$ NO !!!!
 $(a + b + c) / 3$ SI

Aritmetica (II)

Operazione	Operatore C	Espr. aritmetica	Espr. C
Addizione	+	$f+7$	f + 7
Sottrazione	-	$p-c$	p - c
Moltiplicazione	*	bm	b * m
Divisione	/	x/y	x / y
Modulo	%	$r \text{ mod } s$	r % s

Operatori C	Operazioni	Precedenza
()	Parentesi	Valutate per prime. Se ci sono degli annidamenti, si valuta prima la coppia più interna. Se ci sono più coppie allo stesso livello, si valuta da sinistra a destra.
* , / , %	Moltiplicazione, Divisione, Modulo	Valutate per seconde. Se ce ne sono diverse, si valutano da sinistra a destra.
+ , -	Addizione, Sottrazione	Valutate per ultime. Se ce ne sono diverse, si valutano da sinistra a destra.

Variabili e tipi di dato

- Tutte le variabili devono essere dichiarate, specificandone il ***tipo***
- La dichiarazione deve ***precedere*** l'uso
- Il ***tipo*** è un concetto astratto che esprime:
 - Come deve essere *interpretato* il dato
 - L'allocazione di spazio per la variabile
 - Le operazioni permesse sulla variabile
- Perché dichiarare?
 - per poter controllare il programma in compilazione

```
int pippo;  
pippo = pippo+1;
```

```
int beppe;  
beppe = 'a';
```

in realtà il C è tollerante
e permissivo

Tipi predefiniti in C

- **char, int, float e double** (*NO boolean*)

int i = 0; /* dichiarazione con inizializzazione */

char a;

const float pi = 3.14; /* pi non è più modificabile */

double zeta = 1.33;

- Consiglio: inizializzare sempre esplicitamente le variabili
 - Si migliora la leggibilità
 - Non conviene fidarsi delle inizializzazioni implicite che l'ambiente potrebbe effettuare
 - Inizializzazioni che in C **non** sono garantite

Variabili, costanti, inizializzazioni

- **int**

```
int a;
```

```
int b, c = 0;    /* Attenzione: a e b non inizializzate */
```

```
const int d = 5;
```

```
b = -11;
```

```
c = d;    /* OK: possiamo leggere le costanti */
```

```
d = c;    /* KO: non possiamo modificarle */
```

Variabili, costanti, inizializzazioni

- **float**

```
float a;
```

```
float b, c = 0;
```

```
const float d = 5.0;
```

```
b = -11;
```

```
a = d;          /* OK */
```

```
d = a;          /* KO */
```

```
a = 4 / 5;      /* Che cosa succede? Perché? */
```

```
a = 4.0 / 5.0; /* Che cosa succede? Perché? */
```

```
b = 4 / 5.0;   /* Che cosa succede? Perché? */
```

Variabili, costanti, inizializzazioni

- **char**

char a;

char b, c = 'Q'; /* Le costanti di tipo carattere si indicano con ' */

const char d = 'q'; /* OK: d non sarà più modificato */

a = "q"; /* KO: "q" è una *stringa*, anche se di un solo carattere */

a = '\n'; /* OK: i caratteri di escape sono caratteri a tutti gli effetti */

b = "ps"; /* KO: non si possono assegnare stringhe ai char */

c = 'ps'; /* KO: 'ps' non è una costante valida, non ha senso */

a = 75; /* Che cosa succede? */

Teorema di Böhm e Jacopini

- **Tutti** i programmi possono essere scritti in termini di tre *strutture di controllo*:
 - **Sequenza**: istruzioni eseguite in ordine
 - **Selezione**: istruzioni che permettono di prendere strade diverse *in base a una condizione* (costrutto di tipo **se-allora**)
 - **Iterazione**: istruzioni che permettono di *eseguire ripetutamente* un certo insieme di altre istruzioni (costrutti di tipo **fintantoché**)


Sequenza

```
int main()
{
    int integer1, integer2, sum;
    printf ("Enter first integer\n");
    scanf ("%d", &integer1 );
    printf ("Enter second integer\n");
    scanf ("%d", &integer2);

    sum = integer1 + integer2;

    printf ("\nSum is %d\n\n", sum );

    return 0;
}
```



Selezione

```
int main()
{
    int n;
    printf ("Inserisci un numero\n");
    scanf ("%d", &n );
    if ( n > 0 )
        printf ("Un numero positivo ! \n");
    else
        printf ("Un numero negativo o nullo\n");
    printf ("Fine del programma\n");
    return 0;
}
```

condizione



Iterazione

```
int main()
{
    int n = 9;
    printf ( " PRONTI...\n " );
    while ( n > 0 ) {
        printf ( " ...meno %d ...\n", n);
        n = n-1;
    }
    printf ( " ...VIA!!! \n " );
    return 0;
}
```

condizione



Istruzioni **condizionali**

- L'esecuzione dipende da **condizioni** *sul valore di espressioni booleane*, costruite mediante operatori:
 - **Relazionali** (predicano sulla relazione tra due valori)
==, !=, <, >, <=, >=
 - **Logici** (predicano sul valore di verità di espressioni logiche)
 - ! (NOT)
 - || (OR)
 - && (AND)

Condizioni: esempi

$X == 0$

$X > 0 \ \&\& \ A \neq 3$

$!((x+5)*10 \geq ALFA3 / (Beta_Due+1))$

N.B. : esistono regole di precedenza

$! a \ || \ b \ \&\& \ c$

prima !, poi &&, poi ||

$((!a) \ || \ (b \ \&\& \ c))$

in caso di dubbio, usare le parentesi (tonde)

Dettagli

- && e || si valutano da sinistra a destra
- La valutazione di una espressione logica procede finché necessario per dedurre la verità o falsità, e si arresta appena è definita:

`(x != 0) && ((100 / x) == 0)`

Se x vale zero l'espressione risulta falsa e non si verifica alcun errore (di divisione per zero), perché è **inutile** procedere a valutare oltre && (*ovviamente è inutile solo se x vale zero*)

Vero/falso in C

- Una **condizione** (*espressione relazionale o logica*) assume il valore
 - 0 se risulta FALSA
 - 1 se risulta VERA
- Ogni valore “non zero” è considerato vero
 - if (3) ...
 - if (-1) ...
 - while(1) ... /* continua **PER SEMPRE** !! */
 - if (a – a) ... /* sarà sempre FALSO,
indipendentemente dal valore di a */

Assegnamento (=) e uguaglianza (==)

- L'istruzione di assegnamento

```
int a = 0, b = 4;  
a = b;  
printf( "%d", a );
```



- Il predicato di confronto

```
int a = 0, b = 4;  
if( a == b )  
    printf( "uguali" );  
else  
    printf( "diversi" );
```



ATTENZIONE !

```
int a = 0, b = 4;  
if ( a = b )  
    printf( "uguali" );  
else  
    printf( "diversi" );
```

```
int a = 0, b = 4;  
if ( b = a )  
    printf( "uguali" );  
else  
    printf( "diversi" );
```

Istruzioni condizionali (if-then-else)

- Costrutto a **selezione singola**:

```
if( espressione )  
    istruzione
```

ramo
then

- oppure a **selezione doppia**:

```
if( espressione )  
    istruzione1  
else  
    istruzione2
```

ramo
then

ramo
else

- I rami possono contenere istruzioni composte:

```
if( espressione )  
    { seq.1 di istruzioni }  
else  
    { seq.2 di istruzioni }
```

sequenze di più
istruzioni in un
blocco

- Gli `if` possono essere annidati

Istruzione condizionale semplice

```
if (x < 0) x = -x; else x = x + 10;
```

Per maggior leggibilità, è bene usare regole di incolonnamento (**indentazione**)

```
if (x < 0)
```

```
    x = -x;
```

```
else
```

```
    x = x + 10;
```



è una questione di ***stile***
ma è ***molto*** importante

Esempi

```
if ( x < 0 )  
    x = - x;    /* trasforma x nel suo valore assoluto */
```

```
if ( a > b ) {    /* indica il massimo tra due valori */  
    max = a;  
    printf("massimo: %d", max);  
}  
else {  
    max = b;  
    printf("massimo: %d", max);  
}
```

```
if ( a > b )  
    max = a;  
else  
    max = b;  
printf("massimo: %d", max);
```


Istruzioni condizionali

(selezione singola)

```
#include <stdio.h>          /* Calcolo del valore assoluto */
int main() {                /* programma principale */
    int numero, valass;     /* dichiarazione delle variabili */
    printf("Calcolo Valore Assoluto.\n\n");
    printf("Inserisci Numero Intero:");
    scanf("%d", &numero);   /* acquisizione valore */
    if( numero < 0 )        condizione
        valass = 0 - numero;
    if( numero >= 0 )       ramo
        valass = numero;    then
    printf("Numero: %d\n", numero);
    printf("Valore assoluto: %d\n", valass);
    return 0;
}
```

Istruzioni condizionali

(variante con selezione doppia)

```
#include <stdio.h>          /* Calcolo del valore assoluto */
int main() {                 /* programma principale */
    int numero, valass;      /* dichiarazione delle variabili */
    printf("Calcolo Valore Assoluto.\n\n");
    printf("Inserisci Numero Intero:");
    scanf("%d", &numero);    /* acquisizione valore */
    if( numero < 0 )         condizione
        valass = 0 - numero;  ramo then
    else                      ramo else
        valass = numero;
    printf("Numero: %d\n", numero);    /* output */
    printf("Valore assoluto: %d\n", valass); /* output */
    return 0;
}
```

condizione

ramo then

ramo else

```

/* Uso di strutture condizionali, operatori relazionali e uguaglianza */
#include <stdio.h>

int main() {
    int num1, num2;
    printf("Enter two integers to check their relationships: ");
    scanf("%d%d", &num1, &num2);    /* lettura di due numeri interi */
    if( num1 == num2 )
        printf("%d is equal to %d\n", num1, num2);
    if( num1 != num2 )
        printf("%d is not equal to %d\n", num1, num2);
    if( num1 < num2 )
        printf("%d is less than %d\n", num1, num2);
    if( num1 > num2 )
        printf("%d is greater than %d\n", num1, num2);
    if( num1 <= num2 )
        printf("%d is less than or equal to %d\n", num1, num2);
    if( num1 >= num2 )
        printf("%d is greater than or equal to %d\n", num1, num2);
    return 0;    /* il programma è terminato con successo */
}

```

```
> Enter two integers, and I will tell you
> the relationships they satisfy: 3 7
> 3 is not equal to 7
> 3 is less than 7
> 3 is less than or equal to 7
```

```
> Enter two integers, and I will tell you
> the relationships they satisfy: 22 12
> 22 is not equal to 12
> 22 is greater than 12
> 22 is greater than or equal to 12
```

if-then-else annidati

```
if( i < 100 )
{
    if( i > 0 )
        printf ("Minore di 100 e maggiore di zero\n");
    else
    {
        if( i == 0 )
            printf("Uguale a zero\n");
        else
            printf("Minore di zero\n");
    }
}
else
{
    if( i == 100 )
        printf( "Uguale a 100\n");
    else
        printf ("Maggiore di 100\n");
}
```

Annidamento, blocchi, indentazione

Potenziale ambiguità:

if (n > 0) **if** (a > b) z = a; **else** z = b;

ogni else si associa
all'if più vicino

*l'indentazione lo rende evidente
se incerti, usare le parentesi*

```
if (n > 0) {  
    if (a > b)  
        z = a;  
    else  
        z = b;  
}
```

Sequenze di if

- Spesso accade di voler scrivere molti if annidati (alternative multiple):

```
if (...)
    fai qualcosa1;
else
    if (...)
        fai qualcosa2;
    else
        if (...)
```

Esempio

```
if (n % 2 == 0)
    printf("%d è pari", n);
else
    if (n % 3 == 0)
        printf("%d è multiplo di 3", n);
    else
        if (n % 5 == 0)
            printf("%d è multiplo di 5", n);
        else
            if (n % 7 == 0)
                printf("%d è multiplo di 7", n);
            else
                if (n % 11 == 0)
                    printf("%d è multiplo di 11", n);
                else
                    if (n % 13 == 0)
                        printf("%d è multiplo di 13", n);
                    else
                        printf ("il numero %d non ha divisori primi < 15", n);
```


Una rappresentazione più leggibile

```
if (n % 2 == 0)
    printf("%d è pari", n);
else if (n % 3 == 0)
    printf("%d è multiplo di 3", n);
else if (n % 5 == 0)
    printf("%d è multiplo di 5", n);
else if (n % 7 == 0)
    printf("%d è multiplo di 7", n);
else if (n % 11 == 0)
    printf("%d è multiplo di 11", n);
else if (n % 13 == 0)
    printf("%d è multiplo di 13", n);
else printf ("il numero %d non ha divisori primi < 15", n);
```

/* Se un numero n ha divisori primi < 15 ,
stampa il minimo di tali divisori,
altrimenti stampa un messaggio
che lo segnala */

Istruzioni di I/O

- `scanf (...);` → ingresso
- `printf (...);` → uscita

– Esempio

`printf("%d", (a-z)/10);`

equivale a

`temp = (a-z)/10;`

`printf("%d", temp);`

(dove `temp` è una variabile non usata altrove)

printf

- `printf (stringa_controllo, elementi...);`
 - `%d` intero decimale
 - `%f` floating point
 - `%c` carattere
 - `%s` stringa (...???)
 - `\n` (new line) manda a capo
 - `\t` (tabulazione) stampa spazi fino al successivo punto di
“allineamento” [si usa per incolonnare]
 - `\a` (bell o alarm) emette un “beep” [è un carattere speciale]
 - `\` (carattere di **escape**: annulla il significato del
successivo – si usa per stampare caratteri “riservati”
come ad esempio `\' , \” , \% , \\`)

Un “tipo” in più: le stringhe

- Una sequenza di caratteri si dice *stringa*
- Le stringhe si indicano racchiuse tra doppi apici
 - Esempio: **"scrittura bustrofedica"**
 - Stringa costante di 22 caratteri
 - N.B. anche lo spazio è un carattere!
- "In turco, \"luna\" si dice \"ay\""**
 - stringa costante di 29 caratteri
 - le sequenze di escape sono necessarie per includere le virgolette nella stringa

Esempio di printf

```
float tC, tF; /* temperatura in gradi Celsius e Fahrenheit */  
....  
tC = 5.0 / 9.0 * ( tF - 32 ); // legge di conversione °F→°C  
printf ("%s\n%s%f\n%s%f\n", "temperatura",  
        "in celsius=", tC, "in fahrenheit=", tF);
```

*Ogni %s è sostituito dalla corrispondente stringa costante.
L'istruzione equivale a:*

```
printf("temperatura\nin celsius=%f\nin fahrenheit=%f\n",  
        tC, tF);
```

scanf

- Una stringa di controllo specifica come interpretare i dati letti da stdin
- I nomi delle variabili di cui leggere il valore devono essere preceduti da & (ampersand)
 - La funzione è definita in modo da richiedere come parametri gli *indirizzi* delle variabili in cui memorizzare i valori acquisiti

Esempio di scanf

```
int a, b;
```

```
char c;
```

```
float d;
```

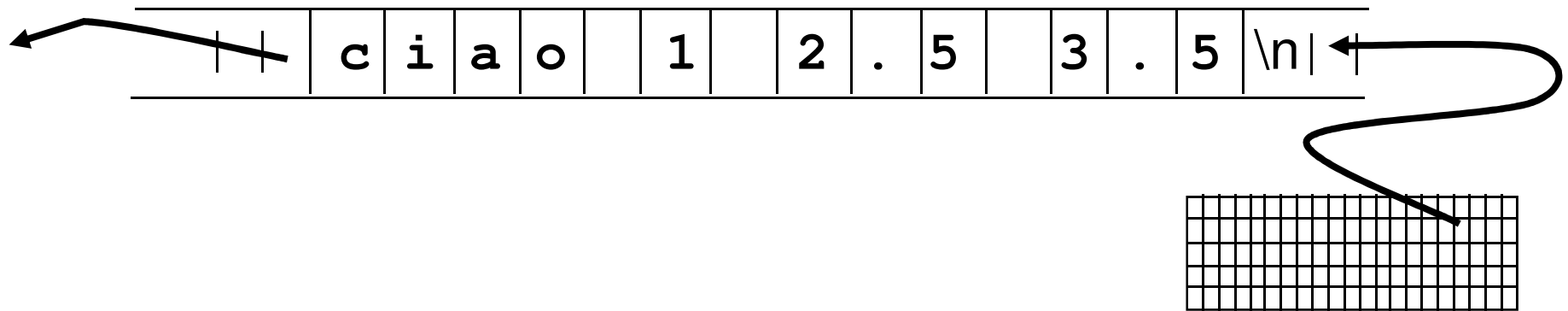
```
scanf("%d%d%c%f", &a, &b, &c, &d);
```

Esempio di programma

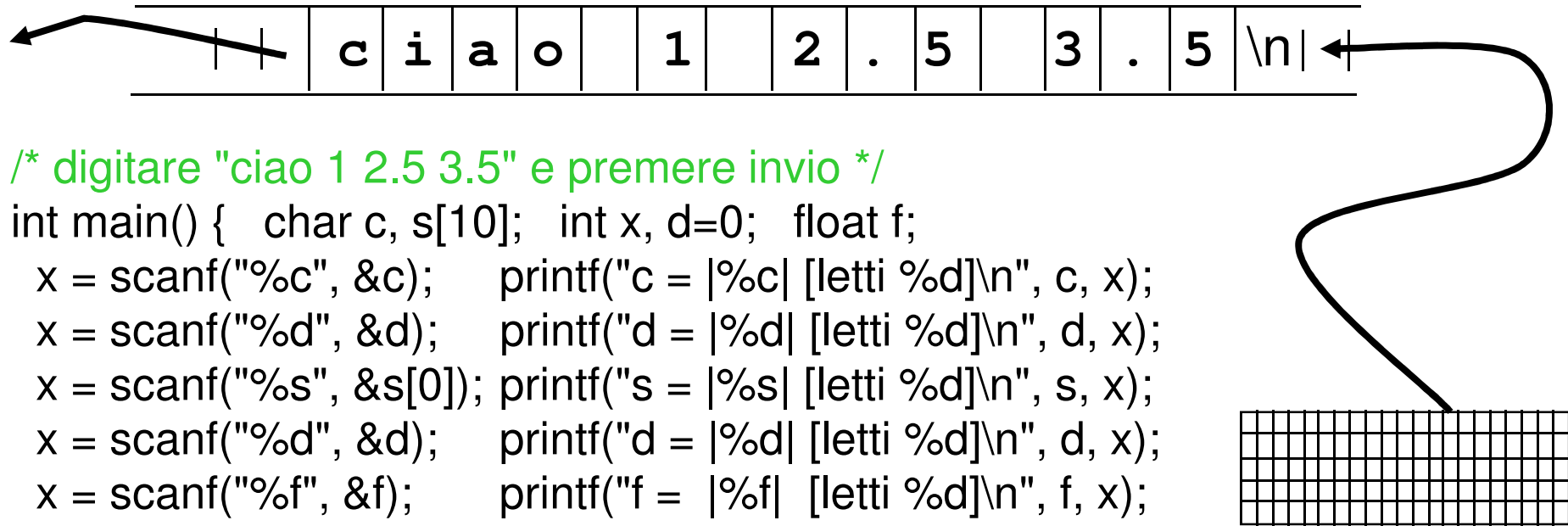
```
#include <stdio.h>    /* calcolo del massimo tra 2 interi */
int main() {
    int a, b, max;
    printf("dammi due numeri interi : ");
    scanf("%d%d", &a, &b);
    if( a > b )
        max = a;
    else
        max = b;
    printf("massimo = %d\n", max);
    return 0;
}
```


Approfondiamo: come funziona la scanf()

- I caratteri (digitati, o comunque letti da stdin) si accodano in un'area di memoria detta "buffer di input"
- Se la scanf riesce a decifrare l'input in base alla specifica di formato (%d, %c, ...) lo "consuma", altrimenti non lo tocca (si verifica un "matching failure")
- Termina restituendo il numero di elementi letti con successo



Approfondiamo: come funziona la scanf()



/ digitare "ciao 1 2.5 3.5" e premere invio */*

```
int main() { char c, s[10]; int x, d=0; float f;
  x = scanf("%c", &c);    printf("c = |%c| [letti %d]\n", c, x);
  x = scanf("%d", &d);    printf("d = |%d| [letti %d]\n", d, x);
  x = scanf("%s", &s[0]); printf("s = |%s| [letti %d]\n", s, x);
  x = scanf("%d", &d);    printf("d = |%d| [letti %d]\n", d, x);
  x = scanf("%f", &f);    printf("f = |%f| [letti %d]\n", f, x);
  x = scanf("%d", &d);    printf("d = |%d| [letti %d]\n", d, x);
  return 0;
}
```

Un passo ulteriore

- Verifica dei dati di ingresso:
 - Consiste nell'intercettare valori *inaccettabili* e nel segnalare il *motivo* per cui sono tali
 - Fa parte della gestione dell'*interazione* con l'*utente*
 - A volte risulta più complicata o onerosa dell'algoritmo che risolve il problema vero e proprio

Verifica dei dati d'ingresso

```
/* Calcolo dell'area di un triangolo */
#include <stdio.h>
int main( ) {
    int base, altezza, area;
    printf("Area del triangolo.\n\n");
    printf("Inserisci Base: ");
    scanf("%d", &base);
    printf("Inserisci Altezza: ");
    scanf("%d", &altezza);
    if( base >= 0 && altezza >= 0 ) {          /* dati          */
        area = (base * altezza) / 2;          /* accettabili*/
        printf("Base: %d\n", base);
        printf("Altezza: %d\n", altezza);
        printf("Area: %d\n", area);
    }
    else
        printf("Valori inaccettabili (non positivi).\n");
    return 0;
}
```

Riflessione

- Occorre progettare il dialogo di I/O
 - far precedere sempre un input da un output di richiesta ("prompt")
 - È buona norma non procedere nell'esecuzione finché non si sia esplicitamente verificato che tutti i dati letti soddisfano le ipotesi per il funzionamento corretto del programma
 - L'utente umano è il più debole degli anelli, è difficilmente controllabile, ed è prono agli errori

Un problema più generale

- La verifica dei dati d'ingresso è un aspetto della tecnica di *raffinamento dell'algoritmo per passi successivi*:
 - Dapprima si progetta l'algoritmo trascurando volontariamente alcuni dettagli
 - Così ci si può concentrare sul problema fondamentale
 - In seguito si introducono miglioramenti, che a poco a poco affrontano i dettagli trascurati
 - La verifica della validità dei dati in input è un tipico “perfezionamento” che si applica nei raffinamenti successivi

Un'altra riflessione sul programma

- Il programma per l'area del triangolo ha un limite
 - **Base** e **altezza** sono rappresentate da valori interi
 - È un'ipotesi di per sé molto limitante, ma non introduce errori nel calcolo dell'area
 - Anche l'**area** è rappresentata da un valore intero
 - Le approssimazioni introdotte sono pesanti. Sono accettabili?
 - Un triangolo con base = altezza = 1 ha area...
- Il programma **NON** è un buon **MODELLO**
 - L'approssimazione dipende (fortemente) dai valori
- Se l'area fosse **float** l'errore risulterebbe attenuato
 - A patto di effettuare correttamente la conversione int-float...

Il ciclo (loop) while

- Itera l'esecuzione di **una istruzione** **fintantoché** una certa **condizione** è vera

```
int a, b;  
scanf("%d%d", &a, &b);  
while ( b > 0 ) {  
    a = a + a;  
    --b;  
}  
printf ("Il valore di a ora è %d", a);
```

condizione di
PERMANENZA
nel ciclo

notare lo stile di
incolonnamento

Il ciclo (loop) while

- Itera l'esecuzione di una istruzione fintantoché una certa **condizione** è vera

```
int a, b;  
scanf("%d%d", &a, &b);
```

Che cosa calcola?

```
while ( b > 0 ) {
```

```
    a = a + a;
```

```
    --b;
```

```
}
```

```
printf ("Il valore di a ora è %d", a);
```

la funzione $f(a,b)$ = {

M.C.D. di due interi positivi

1. Acquisisci i valori di N ed M
2. Calcola MIN, il minimo tra N ed M
3. Parti con $X=1$ ed assumi che MCD sia 1
4. Fintantoché $X < \text{MIN}$
 1. incrementa X di 1
 2. se X divide sia N che M, assumi che MCD sia X
5. Mostra come risultato MCD

Esempio: MCD (0)

```
x = 1;
mcd = 1;
if ( N < M )
    min = N;
else
    min = M;
while ( x < min ) {
    ++x;
    if ( (N % x) == 0 && (M % x) == 0 )
        mcd = x;
}
printf("massimo comune divisore: %d", mcd);
```

Esempio: MCD (1)

```
printf("dammi due valori interi positivi : ");
```

```
scanf("%d%d", &n, &m);
```

```
x = 1;
```

```
if ( n < m )  
    min = n;
```

```
else  
    min = m;
```

```
while ( x <= min ) {  
    if ( (n % x)==0 && (m % x)==0 )  
        mcd = x;  
    ++x;
```

```
}
```

/* Si scandiscono tutti i naturali da 1
al min tra n e m. **L'ultimo** divisore
comune trovato è il MCD */

Esempio: MCD (2)

```
printf("dammi due valori interi positivi : ");  
scanf("%d%d", &n, &m);  
if ( n < m )  
    x = n;  
else  
    x = m;  
while ( (n % x) != 0 || (m % x) != 0 )  
    --x;  
mcd = x;  
...
```

/* Si scandiscono i naturali
diminuendo a partire dal
minimo tra n e m. **Il primo**
divisore comune trovato è
il MCD */

Osservazione:
si tratta della *negazione* della
condizione precedente

Rispetta la fondamentale
legge di DeMorgan
 $!(A \ \&\& \ B) \Leftrightarrow !A \ || \ !B$

Esempio: MCD (3)

/ Codifica dell'algoritmo di Euclide */*

```
printf("dammi due valori interi positivi : ");
```

```
scanf("%d%d", &n, &m);
```

```
while ( m != n )
```

```
    if ( n > m )
```

```
        n = n - m;
```


```
    else
```

```
        m = m - n;
```

```
mcd = n;
```

```
....
```

Scegliere m o n come MCD è indifferente:
al momento dell'uscita dal ciclo while,
infatti, m e n sono **certamente** uguali.



Analisi critica

Siamo sicuri che il ciclo termini sempre?

- È cruciale l'ipotesi che n e m siano positivi
 - Un programmatore scrupoloso effettuerebbe un opportuno controllo sui dati in ingresso
- Sotto questa ipotesi, ad ogni passo o n o m decresce, ma resta positivo
- Non esiste una sequenza di coppie (n, m) che rispetti queste proprietà e che non sia finita

Riconosciamo nel corpo del ciclo una sezione che garantisce un progressivo avvicinamento alla condizione di uscita (terminazione)

Esercizio (terminazione dei cicli)

Stampa dei numeri
pari minori di N

```
int main() {  
    int N, pari = 0;  
    printf("dammi un intero positivo : ");  
    scanf("%d", &N);  
    if ( N <= 0 )  
        printf("Non era positivo\n");  
    else  
        while ( pari != N ) {  
            printf("%d\n", pari);  
            pari += 2;  
        }  
    return 0;  
}
```

errore!!!
dove?

Calcolo del fattoriale

Definito solo su interi non negativi

$$n! = \begin{cases} n * (n-1) * \dots * 2 & \text{se } n > 1 \\ 1 & \text{se } n = 0, 1 \\ \text{indefinito} & \text{altrimenti} \end{cases}$$

Calcolo del fattoriale

```
#include <stdio.h>
int main( ) {
    int n, m, fatt = 1;
    printf("Inserisci n: ");
    scanf("%d", &n);
    if( n < 0 )                /* verifica dati d'ingresso */
        printf("Numero negativo, fattoriale indefinito");
    else {
        m = n;
        while( m > 1 ) {
            fatt = fatt * m;
            --m;
        }
        printf("Fattoriale di %d : %d\n", n, fatt);
    }
    return 0;
}
```

Gli operatori ++ e --

- Attenzione all'uso degli operatori di incremento e decremento (++ e --)
- ++i è un'espressione che prima incrementa i e poi ne fornisce il valore (**pre**-incremento)
- i++ è un'espressione che prima fornisce il valore di i e poi la incrementa (**post**-incremento)
 - Esempio: sia la dichiarazione `int c = 5;`
`printf("%d", ++c);` stampa 6
`printf("%d", c++);` stampa 5
in **entrambi** i casi al termine `c` ha valore 6
- Quindi `if (i++ > 0)...` è **diverso** da `if (++i > 0)...`

Gli operatori ++ e --

Attenzione !!!

Quando la variabile non è parte di una espressione, pre-incremento e post-incremento hanno lo stesso effetto

All'interno di una espressione, però, è determinante la regola del pre- e del post-

```
int a = 32;
```

```
if ( a++ % 13 < 7 ) ...    vero o falso? E con ++a?
```

ULTERIORE ATTENZIONE: i post-(inc|dec)rementi sono applicati al termine della valutazione **dell'INTERA espressione** (*in questo C e Java differiscono... i=i++;...*)

Studio dei cicli: il contatore

- Quando il numero di iterazioni è predeterminato (ad esempio N)

```
int contatore = 1;  
while ( contatore <= N ) {  
    ...;  
    contatore++;  
}
```

```
int contatore = 1;  
while (contatore <= N) {  
    ...;  
    contatore++;  
}
```

se nel ciclo
non si usa mai
il contatore

```
int contatore = 1;  
while (contatore++ <= N) {  
    ...;  
}
```

Formulazione di un algoritmo: ciclo controllato da un contatore

- Il corpo del ciclo è ripetuto fino a quando la variabile “contatore” raggiunge un valore definito
- Il valore è **definito**: il numero di ripetizioni del ciclo è **noto** a priori
- Esempio: ***Una classe di 10 studenti ha ottenuto i voti di un compito. Trovare la media aritmetica di questi voti***

Raffinamenti successivi

- Raffinamento ***top-down***:
 - Si comincia con una rappresentazione in pseudocodice del “top”: *Determinare la media della classe*
 - Si divide il “top” in sottoproblemi da risolvere (in ordine):
 1. *Inizializzazione delle variabili*
 2. *Lettura, addizione e conteggio dei voti*
 3. *Calcolo e scrittura della media*

Raffinamenti successivi

- Pseudocodice:
 - 1.1 Inizializza il totale a zero*
 - 1.2 Inizializza il contatore a uno*
 - 2. WHILE contatore minore o uguale a dieci*
 - 2.1 Leggi da terminale il prossimo voto*
 - 2.2 Aggiungi il voto al totale*
 - 2.3 Aggiungi uno al contatore*
 - 3.1 Calcola la media (uguale a totale diviso per 10)*
 - 3.2 Stampa a terminale la media*

```
/* Programma per il calcolo della media di una classe
   tramite ciclo controllato da un contatore */
```

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

```
int main() {
```

```
    int contatore=0, totale=0, voto;
```

```
    float media;
```

```
    while( contatore < 10 ) {
```

```
        printf("Prossimo voto: ");
```

```
        scanf("%d", &voto);
```

```
        totale += voto;
```

```
        contatore++;
```

```
    }
```

```
    media = totale / 10.0;
```

```
    printf("Media della classe: %f\n", media);
```

```
    return 0;
```

```
}
```

```
> Prossimo voto: 8
```

```
> Prossimo voto: 6
```

```
> Prossimo voto: 5
```

```
> Prossimo voto: 7
```

```
> Prossimo voto: 3
```

```
> Prossimo voto: 9
```

```
> Prossimo voto: 7
```

```
> Prossimo voto: 9
```

```
> Prossimo voto: 3
```

```
> Prossimo voto: 4
```

```
> Media della classe: 6.1
```

```
>
```

Studio dei cicli: la “sentinella”

- Elaborazione di sequenze di valori
- Iterazione controllata da "sentinella"
- L'iterazione avviene un numero di volte non noto a priori
- Procede fintantoché resta vera una condizione
 - L'inserimento, da parte dell'utente, della sentinella al posto del valore da elaborare
- *NB: si sceglie come sentinella un valore che non appartiene all'insieme dei valori da elaborare!*

```
printf("fornisci valore da elaborare, o,  
      se hai finito, fornisci %...", sentinella);  
scanf("%...", valore);  
while (valore != sentinella) {  
    elabora valore  
    printf("fornisci valore da elaborare.  
          Se hai finito, fornisci %...", sentinella);  
    scanf("%...", valore);  
}
```

Esempio di ciclo con sentinella

- Calcolare la media di una sequenza di numeri interi positivi inseriti dall'utente
- Il valore -1 indica la fine della sequenza
 - *Non sappiamo quanto durerà il "flusso" di dati, ma sappiamo che -1 ne indica la fine*

```

int main() {
    int valore, sum=0, num=0, sentinella=-1;
    printf("\nInserisci un valore da elaborare, o,
           se hai finito, fornisci %d", sentinella);
    scanf("%d", &valore);
    while( valore != sentinella ) {
        sum += valore;
        num++;
        printf("\nFornisci valore da elaborare;
               se hai finito, fornisci %d", sentinella);
        scanf("%d", &valore);
    }
    printf("Media dei %d valori: %d", num, sum/num);
    return 0;
}

```

Ancora raffinamenti successivi

- Variante del problema della media dei voti della classe:
*Scriviamo un programma che accetti un **numero arbitrario** di voti*
- *1. Inizializzazione delle variabili (raffinato in):*
 - Inizializza il totale a zero*
 - Inizializza il contatore a zero*

Ancora raffinamenti successivi

- 2. *Lettura, addizione e conteggio dei voti (raffinato in):*
Leggi da terminale il prossimo voto (potrebbe essere la sentinella)
WHILE l'utente non ha ancora immesso la sentinella
Aggiungi il voto al totale
Aggiungi uno al contatore
Leggi il prossimo voto (potrebbe essere la sentinella)
- 3. *Calcolo e scrittura della media (raffinato in):*
IF il contatore è diverso da zero
Calcola la media (uguale a totale / contatore)
Stampa a terminale la media
ELSE
Stampa a terminale "Non è stato immesso alcun voto"


```
/* Programma per il calcolo della media di una classe
   ciclo controllato da sentinella */
#include <stdio.h>

int main() {
    float media;
    int contatore=0, totale=0, voto;

    printf("Prossimo voto, -1 per terminare: ");
    scanf("%d", &voto);

    while( voto != -1 ) {
        totale = totale + voto;
        contatore = contatore + 1;
        printf ("Prossimo voto, -1 per terminare: ");
        scanf ("%d", &voto);
    }
}
```

```

if( contatore != 0 ) {
    media = (float) totale / contatore;
    printf("Media della classe: %.2f", media);
}
else
    printf("Non è stato immesso alcun voto.\n");

return 0;
}

```

Prossimo voto, -1 per terminare: 7
 Prossimo voto, -1 per terminare: 9
 Prossimo voto, -1 per terminare: 9
 Prossimo voto, -1 per terminare: 8
 Prossimo voto, -1 per terminare: 7
 Prossimo voto, -1 per terminare: 6
 Prossimo voto, -1 per terminare: 8
 Prossimo voto, -1 per terminare: 8
 Prossimo voto, -1 per terminare: -1
 Media della classe: 7.75

Una “classe di problemi”

- L'uso combinato di if e while in un opportuno ciclo “a sentinella” permette di analizzare sequenze di dati di lunghezza arbitraria, ignota a priori
 - Estrazione di parametri della sequenza mediante uso di opportuni accumulatori
 - “Filtri a memoria finita”
 - Durante la scansione della sequenza si aggiornano progressivamente gli accumulatori, che riassumono lo “stato” della sequenza analizzata fino a quel momento

Filtri a memoria finita: un esercizio

- L'utente immette una sequenza (di lunghezza libera) di caratteri alfabetici terminata dal carattere '.'
- Il programma deve ignorare ogni carattere non alfabetico diverso da '.'
 - Va bene accettare solo caratteri alfabetici minuscoli
- Al termine dell'elaborazione il programma segnala
 - la vocale inserita il maggior numero di volte [indicando anche quante volte]
 - In caso di “equinumerosità” tra più vocali, va bene una qualsiasi
 - il numero totale di consonanti

Filtri a memoria finita: un esempio (esercizio)

- Varianti (per renderlo più “difficile”):
 - Considerare i caratteri case-insensitive (cioè **d** è uguale a **D**)
 - Indicare tutte le vocali, in caso di “equinumerosità”
 - Segnalare anche la lunghezza della massima sottosequenza di caratteri consecutivi uguali, considerando che comunque i caratteri non alfabetici interrompono il conteggio di tali sottosequenze
[esempio: a_**bB**@**bb**ì⁺pc**D**dg+**pPP**ba°p**aAbP***Pab. ⇒ **3**]
 - Considerare significative anche le cifre [0, 1, ... , 9]
 - Fare operazioni di filtraggio anche sulle cifre
 - ...

Studiamo meglio i tipi di dato

Il tipo `int`

- Approssima il dominio degli interi
- Normalmente utilizza 1 parola
- Esistono anche `short int` e `long int`
 - tipico: long 32 bit, short 16 bit e int 16 o 32
 - lo standard ANSI richiede che
 - spazio per short <= spazio per int <= spazio per long
 - si può usare char per interi tra -127 e 127
 - si può anche specificare signed / unsigned
 - signed è ridondante (implicito) per i numeri
 - possiamo avere unsigned int, signed char, ...

Interi lunghi e corti

- **Calcolatori tipo PC (con Windows)**

- `long int` (32 bit, in C_2)
- `int` (16 bit, in C_2)
- `signed char` (8 bit, in C_2)
- `unsigned long int` (32 bit, in bin. nat.)
- `unsigned int` (16 bit, in bin. nat.)
- `unsigned char` (8 bit, in bin. nat.)

In C i caratteri si possono usare come numeri interi!

- **Calcolatori "tipo WorkStation" (con Unix/Linux)**

- `long int` (64 bit, in C_2)
- `int` (32 bit, in C_2)
- `short int` (16 bit, in C_2)
- `signed char` (8 bit, in C_2)

Tutto il resto va di conseguenza...

I tipi float e double

- Approssimano i numeri razionali
- Le costanti si possono scrivere
 - 315.779
 - 3.73E-5
- double (di solito) occupa più memoria di float
 - tipicamente 4 byte per float e 8 byte per double
 - (...esistono anche i long double, che devono occupare almeno tanto spazio quanto i double)

Cautele

- Attenzione ai confronti tra float

float a, b;

...

if (a == b) ...


può non aver senso, a causa delle
approssimazioni nella memorizzazione!!

Il tipo char

- Di solito è rappresentato con un byte (8 bit)
 - Consente quindi di rappresentare i caratteri ASCII
- I caratteri sono rappresentati da numeri interi e quindi tra essi è definito un ordinamento
 - Esempio: il carattere '1' è rappresentato dall'intero 49, il carattere 'A' dall'intero 65, ...
 - quindi, nell'ordinamento, risulta '1' < 'A'
 - Per le lettere dell'alfabeto è stata scelta una codifica tale per cui l'ordinamento dei codici ASCII coincide con l'usuale ordinamento alfabetico: 'A' < 'B' ecc.
 - Lettere alfabeticamente adiacenti hanno codifiche adiacenti
 - Le maiuscole sono codificate “prima” delle minuscole
 - Quindi ad esempio 'A' < 'a' ma anche 'Z' < 'a'

Somiglianza tra char e int

```
int main() { /* Legge un carattere e ne stampa il codice ASCII */
    char c; /* se è una lettera minuscola; altrimenti termina */
    int i;
    printf("scrivi un car. minuscolo (maiuscolo per finire)\n");
    scanf("%c", &c);
    while( c >= 'a' && c <= 'z' ) {
        i = c;
        printf("valore ASCII per %c risulta %d\n", c, i);
        printf("scrivi un car. minuscolo (ogni altro per finire)\n");
        scanf("%c", &c);
    }
}
```



Attenzione (una nota pratica)

- Nell'esercizio precedente, premendo "invio" ('\n') subito dopo il carattere digitato, il ciclo termina
- Lo fa terminare il carattere '\n'
 - rimane nel “buffer di input”, ed è letto dalla `scanf` in fondo al `while`, sicché la condizione risulta falsa
- Se si vuole evitare questo problema:
 - si può duplicare `scanf ("%c", &c);`
 - si possono usare altre funzioni (`getc()`, ...)
 - si può introdurre una `fflush(stdin);`
 - ...

Tipi integral

char	signed char	unsigned char
signed short int	signed int	signed long int
unsigned short int	unsigned int	unsigned long int

- "int" si può omettere nella specifica del nome dei tipi
- "signed" è implicito se non specificato

Tipo predefinito	Denominazioni alternative
signed short int	signed short, short
signed int	signed, int
signed long int	long int, signed long, long
unsigned short int	unsigned short
unsigned int	unsigned
unsigned long int	unsigned long

Dichiarazione di variabili enumerative

```
enum { QUI, QUO, QUA } papero;
```

```
enum { gennaio, febbraio, marzo,  
      aprile, maggio, giugno, luglio,  
      agosto, settembre, ottobre,  
      novembre, dicembre } mese;
```

- Si possono pensare le etichette di una variabile enumerativa come “nomi alternativi” degli interi

```
papero = QUI;      ⇔      papero = 0;
```

```
mese = aprile;     ⇔      mese = 3;
```

Dichiarazione di variabili enumerative

```
printf("%d", giugno);
```

➤ 5

```
scanf("%d", &mese);
```

➤ 11 \Leftrightarrow mese assume il valore "dicembre"

- Le etichette di un tipo enumerativo sono poste in corrispondenza coi numeri interi, partendo da 0
- Si può controllare la corrispondenza tra valori ed etichette di modo che non inizi da 0:

```
enum {sufficiente=6, buono, distinto, ottimo, sublime};
```


Vantaggi del concetto di tipo

- Si sa quanta memoria riservare alle variabile, in base al loro tipo
 - int: di solito una parola; float: di solito due parole
- Il compilatore può rilevare errori di uso delle variabili
 - un linguaggio è **fortemente tipizzato** se garantisce:
 - correttezza delle istruzioni rispetto al tipo degli operandi, verificabile dal compilatore
 - che non sorgano errori di tipo in esecuzione
- In linea di massima il C persegue la tipizzazione forte
 - però è molto permissivo, infatti...

Conversioni di tipo

- Se in C un operando non è del tipo atteso subisce di solito una **conversione automatica** di tipo
 - È detta anche *cast implicito*
 - Si opera la conversione anziché segnalare l'errore
 - A volte è accompagnata da un “avvertimento” (warning)
- Esempio
 - `int i; float f;`
 - la valutazione dell'espressione `i + f` effettua prima la conversione di `i` in `float` e poi la somma

Conversioni di tipo: esempi

```
int n, m;    float x;
```

x = n + x; (n è convertito in “float” e poi sommato a x)
n = x; (x è troncato, ne sopravvive la parte intera)
x = n; (n da “int” è convertito in (promosso a) “float”)
n = n / m; (il risultato della divisione è un intero)
n = n / x; (n è convertito in “float”, poi si esegue la divisione,
e infine il risultato è troncato a “int”)
x = n / x; (come sopra ma il risultato resta “float”)
x = n / m; (**attenzione:** la divisione tra int tronca (quoziente))

ESEMPIO DI CONVERSIONE ESPLICITA (cast esplicito):

```
x = (float) n / m;
```

ATTENZIONE

- Il cast (implicito o esplicito che sia) non modifica il tipo della variabile o delle variabili coinvolte, ma solo **il tipo associato al valore dell'espressione**
- Le variabili in memoria continuano a essere del tipo dichiarato staticamente nella parte dichiarativa del programma

Espressioni: tipo e valore

- Ogni espressione, infatti:
 - ha un tipo, che dipende dagli operatori, dai tipi delle variabili, dai tipi delle costanti, dall'effetto dei cast impliciti ed espliciti...
 - una volta valutata, rappresenta un valore



- **OGNI ESPRESSIONE HA UN TIPO**
- **OGNI ESPRESSIONE HA UN VALORE**

Ogni espressione ha un tipo

È il tipo associato al risultato della valutazione

Le **espressioni** si compongono di:

- **variabili**
 - Il cui tipo è noto in base alla dichiarazione, e **NON cambia**
- **costanti**
 - Il cui tipo è deducibile da “come sono scritte”
 - 3 : int
 - 3.0 : float
 - '3' : char
 - "3" : *stringa*
 - impareremo che il tipo "*stringa*" è “puntatore costante a char”

Ogni espressione ha un tipo

[*continua...*]

[*Le **espressioni** si compongono di :*]

- **invocazioni di funzioni**

- Il cui **tipo** è quello del risultato (valore restituito), così come è dichiarato nella definizione della funzione

- **operatori** (i cui operandi sono... **espressioni**)

- Effettuano le opportune conversioni automatiche (congetturali e conservative)
- Il **tipo** del risultato dipende dagli operandi e dalla definizione dell'operatore

Ogni espressione ha anche un valore

Tutte le espressioni sono “valutate”, e hanno un valore

Anche le condizioni e gli assegnamenti hanno un tipo e un valore!!

Abbiamo già visto che gli operatori booleani “valgono” 0 o 1

Condizioni:

$(x \neq y)$	può valere 0 o 1 (dipende dai valori di x e y)
$(3 == 3)$	vale certamente 1
$(3 == 3) == 1$	vale certamente 1
$(x \neq x) == 0$	vale certamente 1
$(3 == 4) == 1$	vale certamente 0

Ogni espressione ha anche un valore

Gli assegnamenti sono espressioni, e hanno un tipo e un valore

In particolare, assumono il valore e il tipo *della variabile assegnata*

Quindi... si possono anche “accodare”

`x = (y = (z = (w = (q = 3))));` o anche `x = y = z = w = q = 3;`

– L’assegnamento è un operatore associativo a destra

– Tutte le variabili risultano assegnate a 3

```
int x, y = ( 3 == 4 );           /* che cosa stampa? */
if( ( x = y++ ) == ( x = y-1 ) )
    printf("%d", x+y);
```

ATTENZIONE

Possiamo usare gli assegnamenti nelle condizioni, e viceversa

Di solito però... succede quando si sta commettendo un errore...!

```
x = y == 3;    /* Assegna a x il valore 0 o 1 e non modifica y */  
if ( 3 = x )   /* Errore di sintassi (3 non è una variabile) */
```

Ma soprattutto...

```
if ( x = 3 ) ... È SEMPRE VERO!!! (per ogni valore precedente di x)  
if ( x = 0 ) ... È SEMPRE FALSO!!! (per ogni valore precedente di x)  
if ( x = y ) ... equivale a scrivere  x = y; if ( y != 0 ) ...
```

E, ancora peggio...

```
while ( ! (x = 0) ) ... NON TERMINA MAI !!!!
```

Ancora conversioni

- In C i tipi sono ordinati in base alla *precisione*:
 - $\text{char} < \text{short} < \text{int} < \text{long} < \text{float} < \text{double} < \text{long double}$
- Si consideri un **operatore binario**
 - Se un operando è long double, converti l'altro a long double, altrimenti
 - Se uno è double, converti l'altro in un double, altrimenti
 - Se uno è float, converti l'altro a float, altrimenti
 - Converti char e short a int
 - Quindi, se un operando è long, converti l'altro a long
- Si converte l'operando “meno ricco” nel tipo di quello “più ricco” (o “più largo”, o “più preciso”)
 - il tipo più preciso domina nella conversione
 - il risultato è un valore del tipo più preciso

Ancora conversioni

- Si consideri un generico **assegnamento**
 - Il valore dell'espressione della parte destra è convertito nel tipo che sta a sinistra dell' =
 - i **char** sono convertiti in **int**
 - interi lunghi (p.es. **int**) sono convertiti in **int** più corti (p. es. **char**) troncando bit
 - **float x; int i;**
 - sia **x = i;** che **i = x;** causano conversioni
 - quando un **double** è convertito in un **float** si può verificare troncamento o arrotondamento (dipende dall'implementazione, IEEE 754 *docet*)

Ancora sulla dichiarazione dei dati: le variabili `const`

```
const int modelloauto = 159;
```

```
const float pigreco = 3.14159265;
```

- Si inizializza in fase di dichiarazione
- Non si può più modificare
- Causa allocazione di memoria
 - È una “variabile blindata”
- Esiste anche un altro modo di definire un valore una volta per tutte
 - la direttiva `#define`

Definizione di macro: #define

#define PIGRECO 3.141592

- La **#define** è una **direttiva al preprocessore**
- Non è terminata dal punto e virgola
 - una direttiva al precompilatore *non è un'istruzione C*
- Non causa allocazione di memoria
 - **PIGRECO** è una "costante simbolica"
- Il **simbolo PIGRECO** viene sostituito nel codice con il valore 3.14 ... **prima che il programma sia compilato**
- Si dice che **PIGRECO** è una macro-definizione (o semplicemente una MACRO)
- Per **convenzione**, le costanti definite tramite macro sono interamente maiuscole (es: TIPICA_COSTANTE)

Definizione di macro: #define

```
#define VERO 1
```

```
#define FALSO 0
```

```
printf("%d %d %d", FALSO, VERO, VERO+1) ;
```

```
> 0 1 2
```

- Si possono costruire MACRO a partire da altre MACRO, e una macro può anche essere dotata di uno o più parametri:

```
#define AREACERCHIO(x) (PIGRECO*(x)*(x))
```

```
area = AREACERCHIO(4) ;
```

Diventa \Rightarrow `area = (3.141592*(4)*(4)) ;`

Altre direttive al precompilatore

- Abbiamo già visto:

#include <stdio.h>

- Serve a richiamare librerie
- comanda al preprocessore di leggere *anche da un altro file sorgente*

- Altra direttiva usata frequentemente:

#undef VERO

- Serve ad annullare la definizione di una MACRO
- Da quel punto in poi la costante **VERO** *non è più definita*

Varianti sintattiche

Ogni cosa, in C, si può fare in molti modi

Teoria e pratica

- In **teoria**

- sequenze, if-else e while sono **complete**
- bastano a codificare *qualsiasi* algoritmo eseguibile da un computer
 - Teorema di Boehm e Jacopini

- In **pratica**

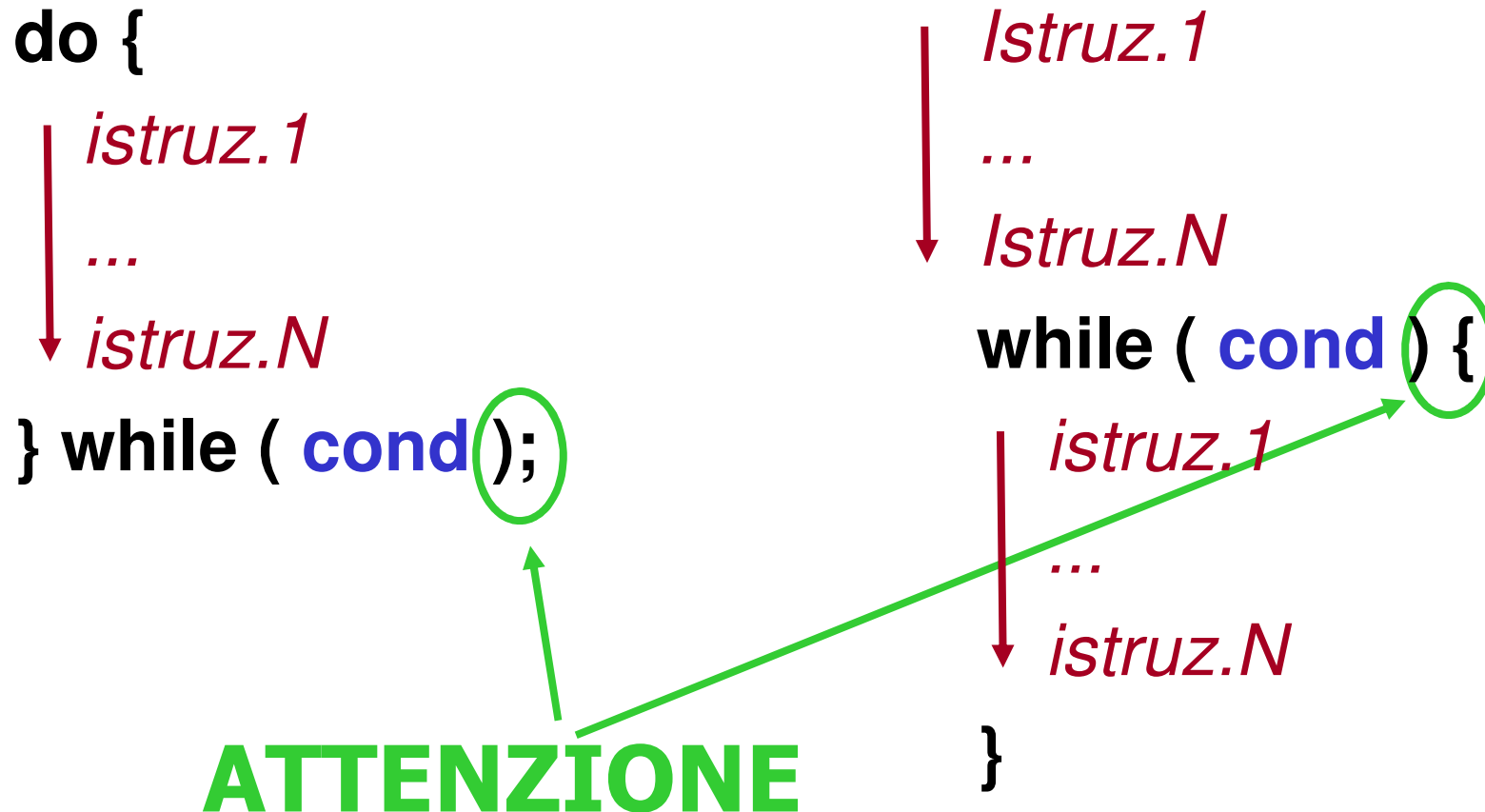
- ~~per aumentare il potere espressivo~~

per **agevolare la scrittura** dei programmi, i linguaggi introducono altre istruzioni di controllo e altre funzionalità

No!!



Il ciclo do-while



La "trasformazione inversa"

```
while ( cond ) {
```

istruz.1
...
istruz.N




```
}
```

*Prima si ripeteva il
codice, qui si ripete
la condizione*

```
do {
```

```
  if ( cond ) {
```

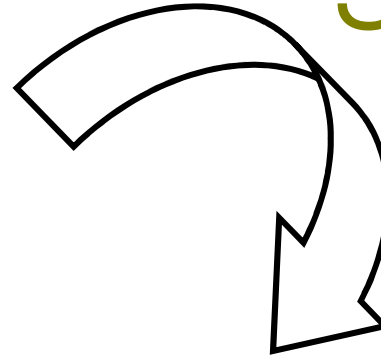
istruz.1
...
istruz.N



```
  }
```

```
} while ( cond );
```

Sentinella



```
printf("prompt");  
scanf("%...", valore);  
while (valore != sentinella) {  
    elabora valore;  
    printf("prompt");  
    scanf("%...", valore);  
}
```

```
do {  
    printf("prompt");  
    scanf("%...", valore);  
    elabora valore;  
} while (valore != sentinella);
```

Il ciclo for

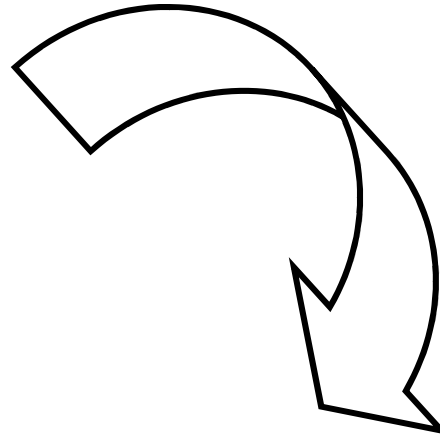
```
for ( exp.A; cond; exp.I ) {  
    ist.1;  
    ...  
    ist.N;  
}
```

ATTENZIONE

```
exp.A;  
while ( cond ) {  
    ist.1;  
    ...  
    ist.N;  
    exp.I;  
}
```

Ciclo a contatore

```
cont = 0;  
while ( cont < N ) {  
    ...;  
    cont++;  
}
```



```
for ( cont = 0; cont < N; cont++ ) {  
    ...;  
}
```

Espressioni condizionali

operando1 ? operando2 : operando3

- Se *operando1* è vero, l'espressione vale *operando2*
- altrimenti l'espressione vale *operando3*

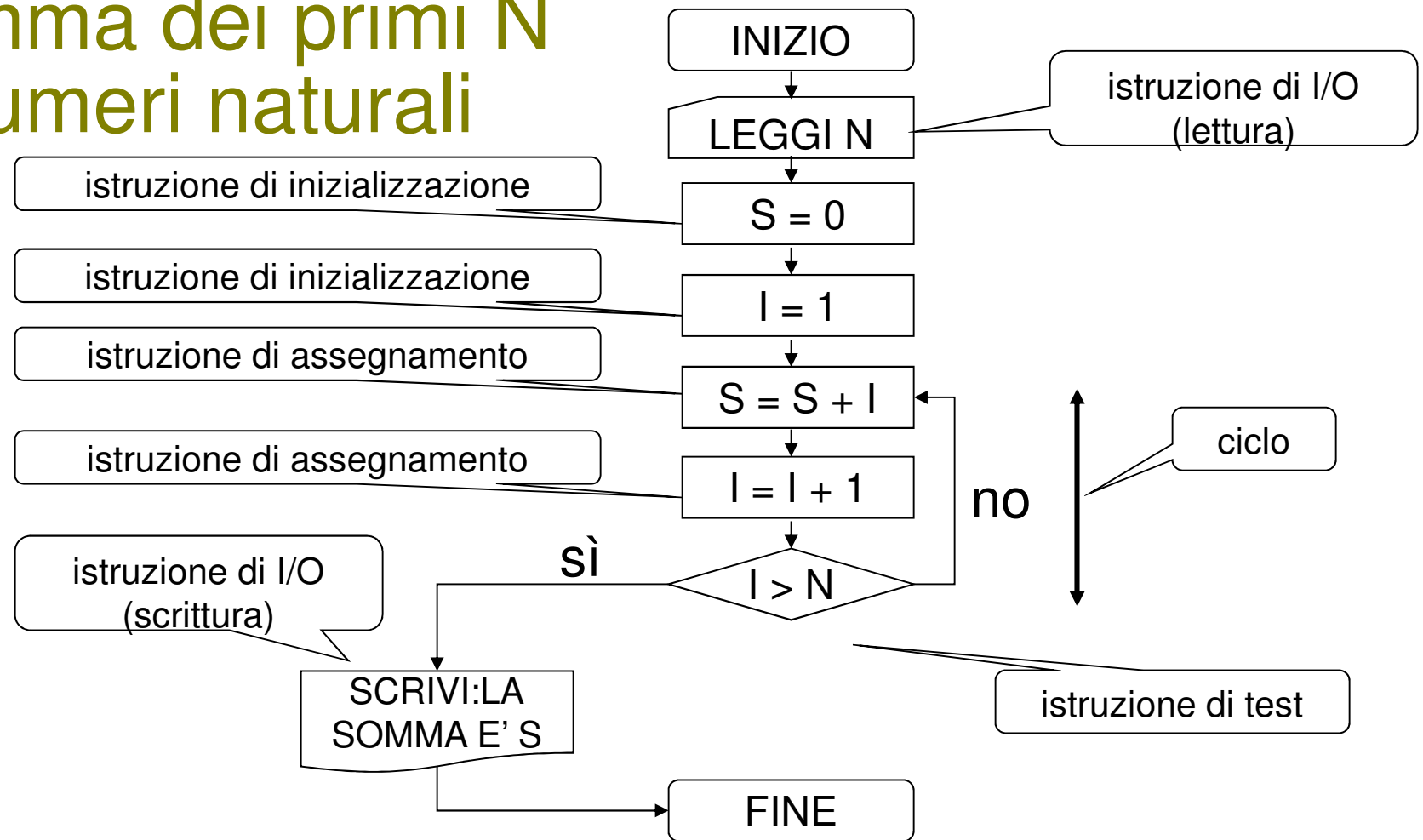
- Esempio: massimo tra due numeri

$x = (y > z) ? y : z;$

- Equivale a

if $(y > z)$ { $x = y;$ } else { $x = z;$ }

Somma dei primi N numeri naturali

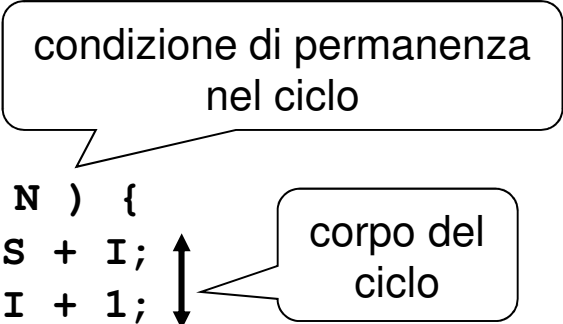


Istruzioni iterative: ciclo a condizione iniziale - **while**

```
1.  read (N)
2.  if N >= 0 then {
3.      S = 0
4.      I = 1
5.      while I <= N do {
5.          S = S + I
6.          I = I + 1
7.      }
8.  }
9.  write ("Sum is" S)
10. end
```

```
/* Somma dei primi N naturali */
```

```
int main( ) {
    int N, S, I;
    printf("Inserisci N: ");
    scanf("%d", &N);
    if( N >= 0 ) {
        S = 0;
        I = 1;
        while( I <= N ) {
            S = S + I;
            I = I + 1;
        }
        printf ("\nSum is: %d\n", S);
    }
    return 0;
}
```



Istruzioni iterative: ciclo a condizione finale - do

```
#include <stdio.h>
/* Somma dei primi N numeri naturali */
int main( ) {
    int N, S, I;                                /* dichiarazione variabili */
    printf("Inserisci N: ");
    scanf("%d", &N);                             /* input */
    if( N > 0 ) {                                 /* accetto solo N positivo */
        S = 0;
        I = 1;
        do {                                     /* ciclo a condizione finale */
            S = S + I;
            I = I + 1;
        } while( I <= N );
        /* il ciclo è eseguito almeno una volta ! */
        printf("\nSum is: %d\n", S);             /* output */
    }
    return 0;
}
```

The diagram consists of two callout boxes. The first box, labeled "corpo del ciclo", has a pointer to the body of the do-while loop, which is enclosed in a double-headed vertical arrow. The second box, labeled "condizione di permanenza nel ciclo", has a pointer to the while condition "I <= N".

Versione con il for

```
#include <stdio.h>
/* Somma dei primi N numeri naturali */
int main( ) {
    int N, S, I; /* dichiarazione variabili */
    printf("Inserisci N: ");
    scanf("%d", &N); /* input */
    if( N > 0 ) { /* se N è positivo */
        S = 0;
        for( I = 1; I <= N; I++ )
            /* Si noti che all'inizio I = 1 */
            S = S + I;
        printf("Sum is %d\n", S); /* output */
    }
    return 0;
}
```

The diagram consists of two callout boxes. The first box, labeled "Intestazione del ciclo", has a line pointing to the header of the for loop: `for(I = 1; I <= N; I++)`. The second box, labeled "corpo del ciclo", has a line pointing to the first line of the loop body: `S = S + I;`.

Alternativa per $N \geq 0$

```
#include <stdio.h>
/* Somma dei primi N numeri naturali */
int main( ) {
    int N, S, I;           /* dichiarazione variabili */
    printf("Inserisci N: ");
    scanf("%d", &N);       /* input */
    if( N >= 0 ) {         /* Se N non è negativo */
        S = 0;
        for( I = 0; I <= N; I++ )
            /* Si noti che all'inizio I = 0 */
            S = S + I;
        printf ("Sum is %d\n", S);    /* output */
    }
    return 0;
}
```

corpo
del ciclo

intestazione
del ciclo

Formulazione del fattoriale con il **for**

```
#include <stdio.h>
/* Calcolo del fattoriale */
int main( ) {
    int n, fatt, m;
    printf ("Inserisci n: ");
    scanf ("%d", &n);
    if( n > 0 ) {
        fatt = n;
        for( m = n; m > 2; m-- )
            /* si noti che m decresce */
            fatt = fatt * (m - 1);
        printf ("Il fattoriale di %d vale: %d\n", n, fatt);
    }
    return 0;
}
```

corpo
del ciclo

intestazione
del ciclo

Switch

```
if ( var == v1 )      ist.1;  
else if ( var == ... ) ist.2;  
else if ( var == vi || var == vj )  
                                ist.ij;  
else if ( var == ... ) ist.3;  
else if ( var == vN ) ist.N;  
else    ist.U;
```

```
switch ( var ) {  
    case v1:    ist.1; break;  
    case ...:   ist.2; break;  
    case vi:  
    case vj:    ist.ij; break;  
    case ...:   ist.3; break;  
    case vN:    ist.N; break;  
    default:    ist.U; break;  
}
```



```
char c;  
int n_cifre, n_separatori, n_altri;  
...
```

```
c = ....; /* acquisizione */  
switch ( c ) {
```

```
case '0' :
```

```
case '1' :
```

```
.....
```

```
case '9' : n_cifre++;
```

```
break;
```

```
case ' ' :
```

```
case '\n' :
```

```
case ';' :
```

```
case ':' :
```

```
...
```

```
case ',' : n_separatori++;
```

```
break;
```

```
default :
```

```
n_altri++;  
}
```

deve essere integral

Che cos'è?

Un “filtro a memoria finita” che “filtra” stdin contando le cifre (caratteri numerici), i separatori, e i caratteri che non appartengono a queste due categorie

evita di passare al caso successivo

necessario per trattare i casi non esplicitamente elencati


```

int main(){
    char c, throw_away;
    int n_cifre=0, n_separatori=0, n_altri=0;
    do {
        printf("dammi un carattere; ! per terminare "); scanf("%c", &c);
        switch (c) {
            case '0': case '1': case '2': case '3': case '4':
            case '5': case '6': case '7': case '8': case '9':    n_cifre++;
                break;
            case '.': case ',' : case '\n' : case ' ' : case ':': case ',': n_separatori++;
                break;
            default:
                n_altri++;
        }
        printf("\n");
    } while( c!='!' );
    printf("cifre  %d\nseparatori  %d\naltri  %d\n", n_cifre, n_separatori, --n_altri);
    return 0;
}

```

/* Programma che legge una sequenza di caratteri (# termina) e la trasforma in melodia associando ogni carattere a una nota */

```
int main() {
    char C; int resto;
    printf("Inserisci il primo carattere del tuo nome\n"); scanf("%c", &C);
    while (C != '#') {
        resto = C % 7;
        switch (resto) {
            case 0: printf("Il car. %c corrisponde a 'do'\n", C); break;
            case 1: printf("Il car. %c corrisponde a 're'\n", C); break;
            ... 2 3 4 5 ... ... mi fa sol la ...
            case 6: printf("Il car. %c corrisponde a 'si'\n", C); break;
        }
        scanf("%c", &C); /* ignora il '\n' digitato dopo la scanf() precedente */
        printf("Inserisci il prox car. del nome; # termina il programma\n");
        scanf("%c", &C);
    }
    return 0;
}
```

Istruzioni *break* e *continue*

- *break*; fa uscire dal corpo di un ciclo o da uno switch
 - Di fatto, dal *blocco* in cui si trova
- *continue*; interrompe l'iterazione corrente di un ciclo (do, while o for) e dà inizio all'iterazione successiva
 - Non si esegue (per la sola iterazione in corso) la parte "restante" del corpo del ciclo

Esempio

/* ciclo con elaborazioni su una serie di (al più N) valori, assunti successivamente dalla variabile intera **x** saltando i valori negativi e interrompendo l'elaborazione al primo valore nullo incontrato */

```
for( i = 0 ; i < N ; i++ ) {  
    printf("immettere un intero > "); scanf("%d", &x);  
    if ( x < 0 )  
        continue;  
    if ( x == 0 )  
        break;  
    .... /*elaborazione elementi positivi */  
}
```

Istruzione goto

- Trasferimento esplicito e incondizionato del flusso di esecuzione
- Quasi sempre da evitare! (*spaghetti code*)

```
scanf("%d%d", &x, &y);  
if ( y == 0 )  
    goto error;  
printf("%f\n", x/y);  
...  
error: printf("y non può essere uguale a 0\n");
```

Conoscenza del linguaggio e stile di programmazione

- È **necessario** scrivere i programmi pensando alla loro leggibilità
- Conviene evitare di usare caratteristiche del linguaggio che sono legali, ma deprecabili
 - Esempio: condizioni e espressioni la cui interpretazione dipenda fortemente dall'ordine di valutazione delle sotto-espressioni

Consigli (I)

- Fare in modo che il flusso di controllo sia poco intricato (evitare goto)
- Usare cicli for quando il numero di iterazioni è noto a priori
- Usare il ciclo for in maniera disciplinata
 - rispettando il ruolo del contatore
 - usando inizializzazione, condizione e passo di incremento che siano semplici e chiari

Consigli (II)

- Attenzione a non dimenticare i break negli switch
- Mettere il caso di default come ultimo
 - In questo caso non necessita di break
- Usare **MOLTO parsimoniosamente** break e continue nei cicli

Last, but **not** least

- Esaminare **SEMPRE** con attenzione le condizioni di inizializzazione e **terminazione** dei cicli
 - (rivedere, ad esempio, il ragionamento fatto per l'algoritmo di Euclide per il MCD)
 - Capita (più spesso di quanto si immagini) che erroneamente si scrivano cicli infiniti!

Un altro esempio di costruzione incrementale dei programmi (approccio top down)

- Si leggono sequenze di gruppi di numeri naturali; i gruppi sono separati dal valore 0
- L'ultimo gruppo è terminato dal valore -1
- Si stampi in output una sequenza di naturali corrispondenti alle somme dei valori contenuti nei singoli gruppi

int i; passo 2
scanf("%d", &i);
while (i != -1) {
 calc. somm. gruppo;
 stampa sommatoria;
 passa al prox gruppo;
}

cerca gruppo; passo 1
while (esiste un gruppo) {
 calc. somm. gruppo;
 stampa sommatoria;
 passa al prox gruppo;
}

int i; int sum;
scanf("%d", &i); passo 3
while (i != -1) {
 sum=0;
 accumula sommatoria;
 printf ("%d \n", sum);
 passa al prox gruppo;
}

```
passo 4  int i; int sum;
scanf("%d", &i);
while ( i != -1 ) {
    sum=0;
    while ( gruppo non finito ) {
        sum += i;
        scanf ("%d", &i);
    }
    /* i carattere che segue gruppo */
    printf ("%d \n", sum);
    if ( i! = -1 )
        scanf("%d", &i);
}
```

gruppo non finito → i != 0 && i != -1

Che cosa abbiamo fatto

- Il programma è ottenuto per passi successivi di raffinamento
- È scritto ad ogni passo in un misto di C e di linguaggio naturale (pseudo-codice)
- Alla fine del processo di raffinamento risulta scritto in C ed è eseguibile
- I passi scritti in linguaggio naturale e raffinati al passo successivo possono diventare commenti

Come scrivere un programma

- Prima:
 - Avere capito bene il problema \Rightarrow i problemi si risolvono eseguendo ***una serie di azioni in un certo ordine***
 - Avere impostato l'***algoritmo*** \Rightarrow un algoritmo specifica una procedura in termini di ***azioni da eseguire*** e di ***ordine di esecuzione***
- Durante:
 - Capire quali “blocchetti” usare
 - Usare i principi di buona programmazione (*programmazione strutturata*)
- Dopo:
 - Saper fare test efficaci di correttezza (è **difficile!!**)

Come specificare l'algoritmo

- Diagrammi di flusso / Pseudocodice
- Codice
- Punti critici:
 - Blocchi ad esecuzione sequenziale
 - In cui le istruzioni vengono eseguite nell'ordine in cui sono scritte
 - Trasferimento del controllo:
 - In cui si comanda l'esecuzione di un'istruzione che non è la prossima scritta nel programma
 - Cioè: cicli, selezioni, sottoprogrammi, ...

Strutture di controllo annidate (I)

- Problema:
 - Un collegio ha una lista di risultati di test (1 = ammesso, 2 = respinto) per 10 studenti
 - Scrivere un programma che analizzi i risultati, e, se si ammettono più di 8 studenti, stampi “alzare la retta”
- Note:
 - Il programma deve analizzare 10 risultati: conviene usare un ciclo controllato da un contatore del numero di studenti
 - Si possono usare altri due contatori: uno per contare gli studenti ammessi, l'altro per i respinti
 - Ogni risultato è 1 oppure 2: se non è 1, si suppone sia 2

Strutture di controllo annidate (II)

- Livello top:
Analizza i risultati del test e decidi se bisogna alzare la retta
- Primo Raffinamento:
Inizializza le variabili
Leggi da terminale i dieci risultati e conta gli studenti ammessi e respinti
Stampa a terminale il riassunto dei risultati e decidi se alzare la retta

Strutture di controllo annidate (III)

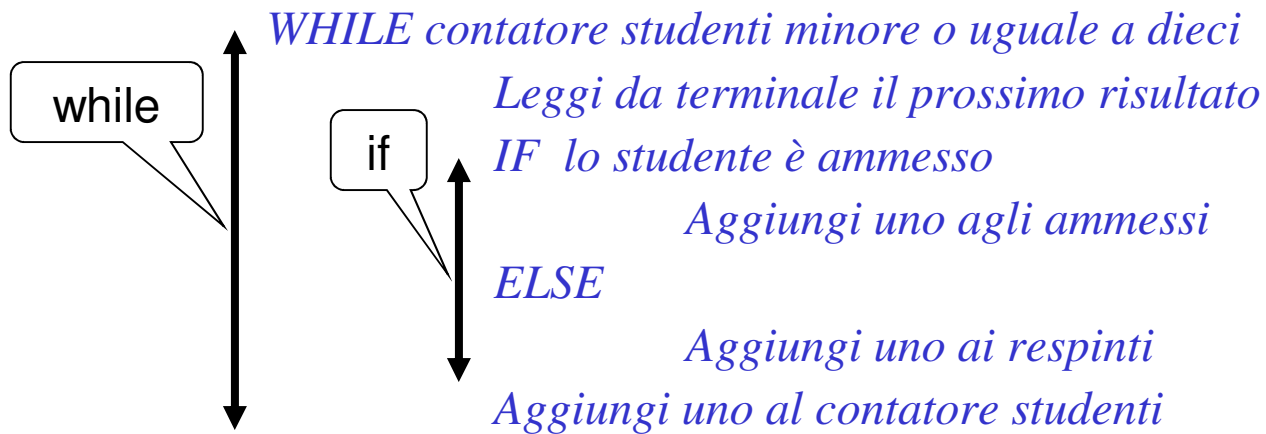
- *Inizializza le variabili:*

Inizializza ammessi a zero

Inizializza respinti a zero

Inizializza contatore studenti a uno

- *Leggi i dieci risultati del test e conta ammessi e respinti:*



- *Stampa riassunto dei risultati e decidi se alzare la retta:*

Stampa a terminale il numero di ammessi

Stampa a terminale il numero di respinti

IF più di 8 studenti sono ammessi stampa "alzare la retta"

```

/* Analisi dei risultati d'esame */
#include <stdio.h>

int main() {
    /* inizializzazione nella dichiarazione */
    int ammessi = 0, respinti = 0, studenti = 1, risultato;

    /* ciclo controllato dal contatore */
    while( studenti <= 10 ) {
        printf("Introdurre un risultato (1=amm.,2=resp.): ");
        scanf("%d", &risultato );

        if( risultato == 1 )
            ammessi = ammessi + 1;
        else
            respinti = respinti + 1;

        studenti = studenti + 1;
    }

    printf("Amemssi: %d\n", ammessi);
    printf("Respinti: %d\n", respinti);

    if( ammessi > 8 )
        printf("Alzare la retta\n");

    return 0;    /* terminato con successo */
}

```

```

Introdurre un risultato (1=amm.,2=resp.): 1
Introdurre un risultato (1=amm.,2=resp.): 2
Introdurre un risultato (1=amm.,2=resp.): 2
Introdurre un risultato (1=amm.,2=resp.): 1
Introdurre un risultato (1=amm.,2=resp.): 1
Introdurre un risultato (1=amm.,2=resp.): 1
Introdurre un risultato (1=amm.,2=resp.): 2
Introdurre un risultato (1=amm.,2=resp.): 1
Introdurre un risultato (1=amm.,2=resp.): 1
Introdurre un risultato (1=amm.,2=resp.): 2
Amemssi: 6
Respinti: 4

```