Istruzioni semplici e composte

- Una *istruzione semplice* è un'espressione seguita da `;`
- Una *istruzione composta* è una sequenza di istruzioni (non necessariamente semplici) racchiuse tra `{` e `}`
  - Le istruzioni componenti la sequenza vengono eseguite nell'ordine di comparizione nella sequenza;
  - Ogni istruzione viene eseguita solo dopo il termine della esecuzione della precedente
  - La sequenza vuota {} è ammessa
  - In generale, in C il codice può essere scritto in formato libero. L'incolonnamento aiuta la lettura, e, nella pratica, è sempre utilizzato

$$\{\ \langle \text{istr} \rangle_1 ;\ \langle \text{istr} \rangle_2 ;\ \langle \text{istr} \rangle_3 ;\ \}$$

$$\begin{array}{l} \{ \\ \quad \langle \text{istr} \rangle_1 ; \\ \quad \langle \text{istr} \rangle_2 ; \\ \quad \langle \text{istr} \rangle_3 ; \\ \} \end{array}$$

Istruzioni semplici e composte: scambio del valore di due variabili

- **Esempio.** Date due variabili `x`, `y` dello stesso tipo, scrivere una istruzione che ne scambi il valore.
- Due assegnamenti simmetrici non risolvono il problema, indipendentemente dal loro ordine!

Istruzione	`x`	`y`	Istruzione	`x`	`y`
	4	7		4	7
`x=y;`			`y=x;`		
	7	7		4	4
`y=x;`			`x=y;`		
	7	7		4	4

Istruzioni semplici e composte: scambio del valore di due variabili

- Si ricorre ad una variabile di utilizzo temporaneo per memorizzare il valore della prima variabile che viene assegnata, che altrimenti andrebbe perso

Istruzione	temp	x	y
	?	4	7
`temp=x;`			
	4	4	7
`x=y;`			
	4	7	7
`y=temp;`			
	4	7	4

Blocco

- Generalizza l'istruzione composta permettendo l'inserimento di dichiarazioni prima delle istruzioni componenti

$$\langle \text{blocco} \rangle \longrightarrow \texttt{\{}\, \langle \text{dichiarazioni} \rangle\, \langle \text{istruzioni} \rangle\, \texttt{\}}$$

```
{ int temp;
  temp=x;
  x=y;
  y=x;
}
```

- Un blocco è una istruzione; pertanto può essere contenuto (*nidificato*) in altri blocchi

Istruzioni ripetitive

- Costituite da
  - Una *istruzione da ripetere*, o *corpo del ciclo*
  - Una *logica di controllo* della ripetizione
- L'istruzione da ripetere può in particolare essere una istruzione composta
- La logica di controllo consiste nella valutazione di una espressione
  - La prima valutazione dell'espressione può avvenire inizialmente o dopo la prima ripetizione
  - Se il valore dell'espressione risulta diverso da zero, la ripetizione prosegue, diversamente il controllo passa alla prima istruzione seguente l'istruzione ripetitiva

Inizializzazione di una variabile

- L'esecuzione di una istruzione ripetitiva comporta spesso l'utilizzo di una variabile sia come parte del $\langle \text{rvalue} \rangle$ che come $\langle \text{lvalue} \rangle$ di un assegnamento

Esempio. Somma di n numeri

Algoritmo.

```
- finché sono presenti elementi in input
  - leggi un elemento
  - aggiungi elemento alla variabile di accumulo
```

- Se `s` è la variabile di accumulo e `x` l'elemento letto, l'istruzione da ripetere sarà quindi `s = s + x;`

Inizializzazione di una variabile

- Alla prima ripetizione `s` non ha un valore definito
- È necessario che essa abbia ricevuto un valore prima dell'inizio della ripetizione
- L'assegnamento del valore iniziale si dice inizializzazione
- Quale valore deve essere assegnato? Dipende dal problema, ma una regola empirica è

  che valore deve assumere la variabile se la ripetizione avviene zero volte?

- **Esempio.** somma di n numeri: se $n = 0$ il risultato deve essere 0,quindi la corretta inizializzazione è

$$s = 0$$

Operatori relazionali

- Gli operatori relazionali permettono il confronto tra valori
- Il loro risultato è vero o falso

Operatore	Significato
==	uguale
!=	diverso
>	maggiore
>=	maggiore o uguale
<	minore
<=	minore o uguale

Valori e operatori logici

- Algebra di Boole
  - Valori vero, falso
  - operatori logici and, or, not
- Funzionamento (tabella di verità)

P	Q	P and Q	P or Q	not P
falso	falso	falso	falso	vero
falso	vero	falso	vero	vero
vero	falso	falso	vero	falso
vero	vero	vero	vero	falso

Valori e operatori logici

- In C non esiste un tipo predefinito per i valori logici
- Si rappresentano con valori interi con la codifica:
  - il valore 0 denota falso
  - il valore 1 denota vero (ogni altro valore diverso da 0 denota vero)
- Operatori logici in C

simbolo	operatore logico
!	not
&&	and
||	or

P	Q	P && Q	P || Q	!P
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Espressioni

- Rivisitiamo la sintassi delle espressioni, aggiungendo espressioni relazionali e logiche

$$
\begin{aligned}
\langle \text{espr} \rangle &\longrightarrow \langle \text{costante} \rangle \mid \langle \text{nome-var} \rangle \mid \texttt{(}\ \langle \text{espr} \rangle\ \texttt{)} \mid \langle \text{espr-relazionale} \rangle \mid \\
&\quad \langle \text{espr-assegn} \rangle \mid \langle \text{espr-logica} \rangle \mid \langle \text{espr-incr-decr} \rangle \\
\langle \text{espr-relazionale} \rangle &\longrightarrow \langle \text{espr} \rangle\ \langle \text{oper-relazionale} \rangle\ \langle \text{espr} \rangle \\
\langle \text{oper-relazionale} \rangle &\longrightarrow \texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \\
\langle \text{espr-assegn} \rangle &\longrightarrow \langle \text{nome-var} \rangle\ \langle \text{oper-assegn} \rangle\ \langle \text{espr} \rangle \\
\langle \text{oper-assegn} \rangle &\longrightarrow \texttt{=} \mid \texttt{+=} \mid \texttt{-=} \mid \texttt{*=} \mid \texttt{/=} \mid \texttt{\%=} \\
\langle \text{espr-logica} \rangle &\longrightarrow \langle \text{espr} \rangle\ \langle \text{oper-logico} \rangle\ \langle \text{espr} \rangle \\
\langle \text{oper-logico} \rangle &\longrightarrow \quad \texttt{!} \quad \mid \quad \texttt{\&\&} \quad \mid \quad \texttt{||} \\
\langle \text{espr-incr-decr} \rangle &\longrightarrow \texttt{++}\ \langle \text{nome-var} \rangle \mid \texttt{--}\ \langle \text{nome-var} \rangle \mid \\
&\quad \langle \text{nome-var} \rangle\ \texttt{++} \mid \langle \text{nome-var} \rangle\ \texttt{--}
\end{aligned}
$$

Istruzione `while`

- Esegue ripetutamente quanto segue: Calcola il valore della espressione, se è diverso da zero, esegue il blocco; altrimenti passa il controllo all'istruzione successiva
- Sintassi: $\langle \text{istr-while} \rangle \longrightarrow$ `while (` $\langle \text{espr} \rangle$ `)` $\langle \text{istr} \rangle$
- In totale si potranno avere $n+1$ valutazioni (test) e n ripetizioni, con $n \geq 0$
- caso particolare: il test fallisce alla prima valutazione $\rightarrow$ non si ha alcuna ripetizione
- In presenza di istruzioni di ripetizione, un programma può non terminare
- perché una istruzione ripetitiva possa terminare è necessario che l'istruzione ripetuta modifichi almeno una delle variabili coinvolte nella espressione di controllo.

Istruzione `while`

Esempio. Calcolare la divisione fra due numeri interi non negativi usando solo somme algebriche e sottrazioni

- Si sottrae di volta in volta il divisore dal dividendo, si incrementa un contatore del numero di sottrazioni effettuate
- Le sottrazioni termineranno quando il dividendo residuo sarà più piccolo del divisore e il dividendo residuo sarà il resto della divisione
- Istruzione da ripetere

```
Resto = Resto - Divisore;
Quoziente = Quoziente + 1;
```

- Condizione per eseguire la ripetizione

```
Resto >= Divisore
```

Istruzione `while`

- Entrambi `Resto` e `Quoziente` compaiono a destra negli assegnamenti $\rightarrow$ vanno inizializzati
- La ripetizione avverrà zero volte se il divisore è maggiore del dividendo in questo caso deve risultare il resto uguale al dividendo e il quoziente zero
- Inizializzazione

```
Resto = Dividendo;
Quoziente = 0;
```

- Vediamo il programma completo...

Istruzione `while`

- Si consiglia di progettare sempre prima un algoritmo in linguaggio naturale e poi tradurlo nel linguaggio di programmazione scelto

Algoritmo.

```
- acquisisci gli operandi
- inizializza resto e quoziente
- finchè il resto è non inferiore al divisore
  - incrementa il quoziente
  - sottrai divisore da resto
- visualizza quoziente e resto
```

Istruzione `while`

- *acquisisci gli operandi*
- *inizial. resto e quoziente*
- *finchè resto >= divisore*
  - *incrementa il quoziente*
  - *sottrai divisore da resto*
- *visualizza quoziente e resto*

```
#include <stdio.h>
main() {
  int Dividendo, Divisore,
      Quoziente, Resto;
  printf("Dividendo ? ");
  scanf("%d", &Dividendo) ;
  printf("\nDivisore ?" );
  scanf("%d", &Divisore) ;
  Resto = Dividendo;
  Quoziente = 0;
  while (Resto >= Divisore) {
    Quoziente = Quoziente + 1;
    Resto = Resto - Divisore;
  };
  printf ("\n%d, %d",Quoziente,Resto) ;
}
```

Istruzione `do...while`

- Ripete quanto segue: esegue il blocco; calcola il valore della espressione, e, se è zero, passa il controllo all'istruzione successiva
- Sintassi: $\langle$istr-do-while$\rangle \longrightarrow$ `do` $\langle$istr$\rangle$ `while (` $\langle$espr$\rangle$ `)`
- Differisce da `while` in quanto la valutazione dell'espressione viene fatta dopo la esecuzione del blocco; pertanto il blocco viene eseguito almeno una volta
- Si potranno quindi avere n test e n ripetizioni, con $n \geq 1$.
- Ogni `do...while` si può riscrivere con `while`; tuttavia se l'istruzione deve essere eseguita almeno una volta perché l'espressione sia definita, è piú leggibile `do...while`

Istruzione `do...while`

Esempio. Leggere da input caratteri finchè non compare un punto

- si richiede che vengano letti da input n caratteri ed eseguiti n test, di cui $n-1$ falliscono (i caratteri diversi da punto) e uno ha successo (il punto)

```
do scanf("%c",&Ch) while (Ch != '.') ;
```

Utilizzando `while` il test compare prima della prima lettura

è necessario un assegnamento fittizio a `Ch` per garantire l'entrata nel ciclo almeno una volta

```
Ch = '#';
while (Ch != '.') scanf ("%c" ,&Ch) ;
```

Istruzione `do...while`

Esempio. Conta il numero di parole della frase digitata in input. La frase inizia con un carattere diverso dallo spazio ed è terminata da un punto.

- *esegui*
  - *esegui*
    - *acquisisci C*
  - *finché C non è ’ ’ o ’.’*
  - *incrementa il numero di parole*
  - *finché C è ’ ’*
    - *acquisisci C*
- *finché il carattere non è ’.’*
- *visualizza il risultato*

```
#include <stdio.h>
main() {
  char C;
  int n=0;
  do {
    do
      C = getchar();
    while (C != ' ' && C != '.');
    n++;
    while (C == ' ')
      C =getchar();
  } while (C != '.');
  printf("%d",n);
}
```

Esempi comparati `while`, `do...while` ▶

Problema. Sommare una sequenza di numeri interi positivi letti da input

- È necessaria una variabile `Somma` per memorizzare il risultato, inizializzata a zero
- continuare a memorizzare un numero acquisito da input quando è stato già sommato è inutile $\rightarrow$ una sola variabile `I` per contenere i numeri acquisiti da input è sufficiente
- Su `I` si riscriverà di volta in volta il nuovo numero dopo aver effettuato la somma del precedente

Esempi comparati `while`, `do...while`

- È conveniente costruire un programma interattivo $\Longleftrightarrow$ un programma che interagisce con l'utente,
  - sollecitandolo all'introduzione di dati al momento opportuno
  - visualizzando risultati
- Occorre visualizzare messaggi di output che sollecitano l'utente a digitare i numeri
- Corpo del ciclo

```
- visualizzazione di un messaggio per sollecitare
  l'introduzione di un numero
- acquisizione di un numero in I
- somma di I a Somma
```

Esempi comparati `while`, `do...while`

Soluzione. Supponiamo non nota la lunghezza della sequenza

- Come conveniamo di indicare la terminazione? Per esempio, con un numero non positivo (è ammessa una sequenza di zero numeri positivi)
- ad ogni iterazione
  - un test sul numero letto, se è positivo va sommato, altrimenti le iterazioni vengono arrestate
  - Se la sequenza è di n numeri positivi più un numero negativo per la terminazione, si avranno $n+1$ messaggi, $n+1$ acquisizioni, $n+1$ test, l'ultimo test fallisce:

    messaggio, lettura, test,
    somma, messaggio, lettura, test,
    ..., somma, messaggio, lettura, test

Esempi comparati `while, do...while`

- adottiamo come corpo del ciclo

  somma,messaggio,lettura

- Si ricava l'algoritmo seguente

  Algoritmo.
  - *inizializza Somma a zero*
  - *visualizza messaggio*
  - *acquisisci un numero I*
  - *fino a quando I è maggiore di zero*
    - *aggiungi I a Somma*
    - *visualizza messaggio*
    - *acquisisci un numero I*
  - *visualizza il risultato*

  Esercizio. Implementare l'algoritmo in C

Esempi comparati while, do...while

Programma.

```
                    #include <stdio.h>
                    main() {
- iniz. Somma a 0     int I, Somma=0;
- visual.messaggio    printf("Intero in input\n");
                      printf("(negativo o zero per terminare) \n");
- acq. numero I       scanf("%d", &I);
- fino a quando I>0   while (I > 0) {
  - agg. I a Somma      Somma += I;
  - vis. messaggio      printf("Intero in input \n");
                        printf("(negativo o zero per terminare) \n");
  - acq. numero I       scanf("%d", &I);
                      }
- vis. risultato      printf("\n Somma dei numeri positivi: %d",
                             Somma);
                    }
```

Esempi comparati `while`, `do...while`

- Supponiamo ora che
  - si vogliano sommare numeri qualunque
  - sia garantito almeno un numero valido
- Dopo ogni acquisizione si chiede all'utente se vuole continuare
  - $\Longrightarrow$ occorre una variabile di tipo carattere `ch` che assume valori ricevuti in input `'S'` e `'N'`
  - È garantito almeno un numero valido $\Longrightarrow$ la sequenza di operazioni è

    `messaggio, lettura, somma, domanda, test,`
    `...messaggio, lettura, somma, domanda, test`
  - n messaggi, n letture, n domande, n test
  - corpo del ciclo

    `messaggio, lettura, somma, domanda`