

Linguaggio C

Fondamenti del linguaggio

Prof. M. Camurri

Linguaggio C

- Il linguaggio C è il linguaggio di programmazione progettato da **Dennis Ritchie** nel **1972** durante la scrittura del sistema operativo UNIX



- Standardizzato da ANSI nel 1989 (C89 o ANSI-C) e da ISO nel 1990

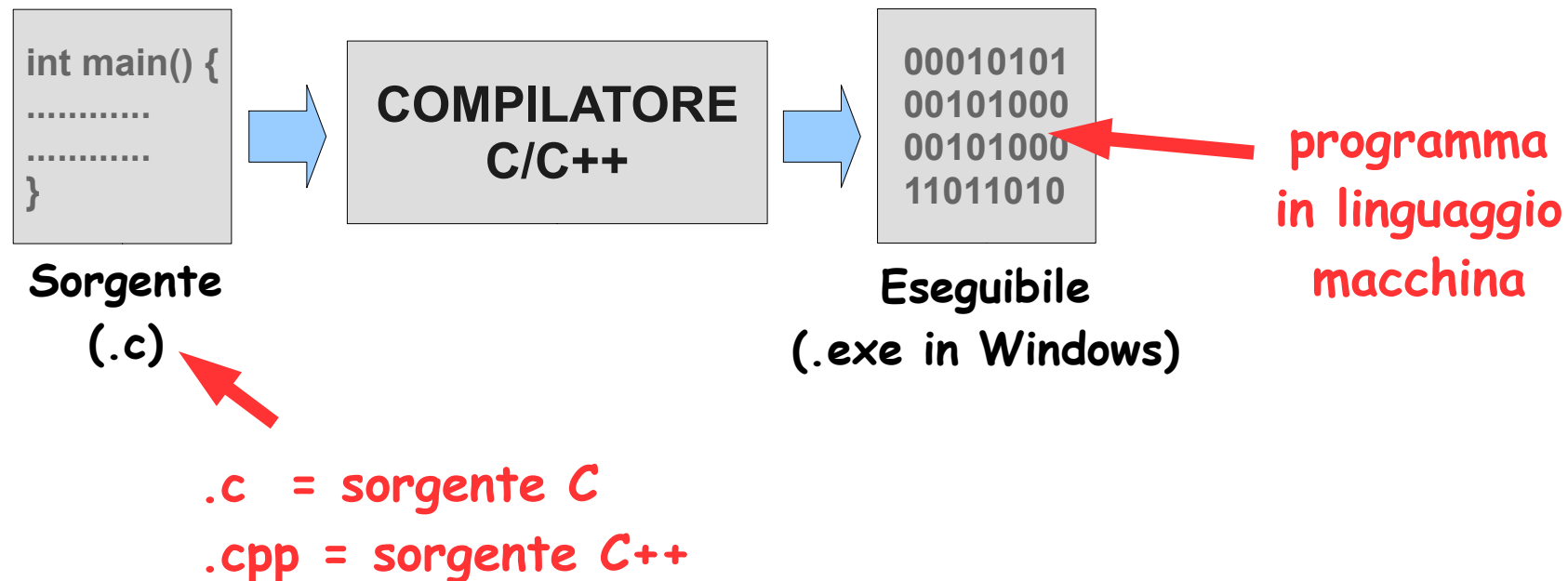
Linguaggio C

E' importante perché:

- è alla base della maggior parte dei linguaggi di programmazione moderni come **C++**, **C#**, **Java**, **JavaScript**, **PHP**, ...
- esiste moltissimo software ancora in uso scritto in C, tra cui buona parte del codice di sistemi operativi come Linux, Android e Windows!
- è un linguaggio fondamentale per la **programmazione di sistema** (scrittura di parti del S.O, driver di dispositivi, ecc.)
- è spesso utilizzato per la programmazione di sistemi particolari in ambito industriale, come **micro-controllori**, "schede" , "centraline" di controllo, ecc.

Caratteristiche del linguaggio C

- E' un linguaggio di alto livello, ma permette operazioni di basso livello (esempio: indirizzare la memoria in modo assoluto)
- Non è un linguaggio a oggetti (al contrario di C++, C#, Java ...)
- E' un linguaggio compilato



Struttura base di un programma in C

- Codice sorgente di un programma C che stampa il messaggio "Hello World!" sullo schermo e termina:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      printf("Hello World!");
6  }
```

- Salviamo il programma in un normale file di testo di nome **HelloWorld.c**
- Nota: è importante salvare il file con estensione .c e non .cpp , poichè la seconda estensione indica un file C++
- C e C++ sono due linguaggi diversi. C++ è un'estensione del C, quindi non tutto il codice C++ è valido in C.

Generazione del file eseguibile

- Per poter eseguire il programma precedente occorre prima compilarlo. La **fase di compilazione** verifica se il programma è sintatticamente corretto e, solo in caso affermativo, genera un file eseguibile (estensione .exe su Windows).
- La **sintassi** di un linguaggio di programmazione è l'insieme delle regole che definiscono la struttura delle istruzioni valide. Ad esempio, una regola sintattica del C afferma che tutte le istruzioni terminano con il carattere "punto e virgola" (;).
- Il compilatore generalmente pone il file eseguibile nella stessa cartella che contiene il file sorgente, ma è possibile modificare la cartella di output del compilatore tramite apposite opzioni. I dettagli dipendono dall'ambiente di sviluppo utilizzato.

Generazione del file eseguibile

- Un ambiente di sviluppo o **IDE** (Integrated Development Environment) è un programma che consente di scrivere il sorgente (**edit**), compilare (**compile**) e infine eseguire (**run**) un programma in uno o più linguaggi di programmazione.
- Dev-cpp e Codeblocks sono esempi di IDE per il linguaggio C/C++.
- Una volta generato il file eseguibile, questo può essere eseguito su computer privi dell'ambiente di sviluppo, poichè il sorgente non è più necessario...
- Sulla maggior parte dei sistemi il file eseguibile può essere mandato in esecuzione da linea di comando, cioè digitando il nome del programma dalla shell del sistema operativo (cmd su Windows), oppure cliccando sul file eseguibile.

Attendere la pressione di un tasto

- A seconda del metodo scelto per eseguire il programma, il programma HelloWorld può presentare questo problema: la finestra si chiude immediatamente senza dar tempo all'utente di leggere il messaggio "Hello World!".
- Per risolvere il problema, inseriamo l'istruzione **getchar()** che blocca il programma finché non viene digitato il tasto INVIO:

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World! \n\n");
5      printf("Premi INVIO per chiudere la finestra");
6      getchar();
7  }
```


Metodo alternativo (solo Windows!)

- Per attendere la pressione di un tasto esiste un metodo alternativo, valido però solo su Windows:

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World! \n\n");
5      system("PAUSE");
6  }
```

- La funzione **system** richiama un comando del sistema operativo sottostante, in questo caso "Pause". Poichè "Pause" è un comando di Windows, l'eseguibile prodotto funzionerà solo su tale sistema operativo, per cui la prima soluzione è preferibile!
- D'ora in poi, in tutti gli esempi di questo corso, le istruzioni di attesa finale NON saranno inserite

Spiegazione di HelloWorld.c (1/2)

- Il **main** è il blocco principale di codice del programma. In linguaggio C, un blocco di codice come questo, a cui è assegnato un nome, è detto **funzione**. Quindi il main è la funzione principale del programma e deve sempre essere presente con questo nome! Vedremo in seguito che è possibile creare nuove funzioni oppure richiamare altre funzioni già esistenti. L'esecuzione di un programma inizia sempre dalla prima istruzione del main.
- La riga **#include <stdio.h>** indica che vogliamo utilizzare le funzioni della **libreria** stdio.h (una *libreria* è un insieme di funzioni).
- Poiché praticamente ogni programma usa istruzioni di I/O, inseriremo sempre questa riga. Il nome **stdio** è l'abbreviazione di **standard input output**.

Spiegazione di HelloWorld.c (2/2)

- L'istruzione **printf("Hello World!");** è un'istruzione di output che visualizza una stringa sul dispositivo di output predefinito (normalmente lo schermo). La "f" finale sta per "formatted".
- una **stringa** è una sequenza di caratteri, e in C deve essere racchiusa tra virgolette doppie
- printf è una funzione contenuta nella libreria stdio.h, quindi per utilizzarla occorre aver incluso la libreria.
- E' possibile inserire nella stringa caratteri speciali usando una **sequenza di escape**. Ad esempio, la sequenza **\n** indica il carattere di andata a capo (**new line**), quindi per andare a capo dopo "Hello" scriveremo:

```
printf("Hello \n World!");
```

Return

- Quando un programma termina, è possibile comunicare al sistema operativo se il programma è terminato "normalmente" o se invece si è verificato qualche errore.
- A tale scopo si utilizza la parola chiave `return`, come in questo esempio:

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello World! \n\n");
5      return 0;
6  }
```

Return

- L'istruzione **return** restituisce al Sistema Operativo un valore numerico che indica l'esito complessivo del programma. Questi valori sono anche detti **codici di ritorno** (return code). Per convenzione, il valore 0 indica che NON CI SONO STATI ERRORI.
- Se si omette l'istruzione return 0; il programma termina comunque correttamente appena si raggiunge la graffa finale del programma, quindi d'ora in poi la ometteremo per brevità.
- Attenzione: l'utilizzo di return in un qualsiasi punto del main diverso dall'ultima riga, provoca la terminazione anticipata e immediata del programma.

Variabili e tipi di dato

- Una variabile è un'area di memoria (ram) identificata da un nome, e in grado di contenere un valore
- In un programma C è possibile creare variabili di vario tipo: variabili di tipo int, char, float, double ...
- int, char, float, double sono detti **tipi di dato** (*data type*)
- Il tipo di dato di una variabile determina l'insieme dei possibili valori che possono essere inseriti nella variabile. Ad esempio: una variabile di tipo int può contenere solo numeri interi, le variabili di tipo float e double possono contenere numeri con la virgola, le variabili di tipo char possono contenere singoli caratteri come lettere, cifre o simboli di punteggiatura.

Variabili e tipi di dato

- Per utilizzare una variabile in C occorre prima dichiararla, cioè indicare il tipo di dato e il nome della variabile. La dichiarazione ha l'effetto di creare la variabile.

```
int main() {  
    int x;    // Dichirazione della variabile x, di tipo intero.  
    x = 5;    // assegno alla variabile x il valore 5  
}
```

- E' possibile inizializzare una variabile mentre la si dichiara, quindi le due righe precedenti si possono riassumere in:

```
int main() {  
    int x = 5; // Dichiarazione e inizializzazione di x  
}
```

Variabili e tipi di dato

- Più variabili dello stesso tipo possono essere dichiarate su un'unica riga. Esempio:

```
int main() {  
    float a, b; // create due variabili di tipo float  
    a = 2.5;  
    b = a/2;  
}
```


Tipi di dato

Il linguaggio C prevede i seguenti tipi di dato:

Nome	Dimensione tipica*	Intervallo di valori rappresentabili (range)
char	1 byte	Numeri interi da -128 a 127 o singoli caratteri ASCII
short	2 byte	Numeri interi da -32.768 a 32.767
int	4 byte	Numeri interi da -2.147.483.648 a 2.147.483.647
float	4 byte	Numeri frazionari positivi e negativi, con valore assoluto compreso tra circa 10^{-38} e circa 10^{38} e precisione di 6 cifre decimali
double	8 byte	Numeri frazionari positivi e negativi, con valore assoluto compreso tra circa 10^{-308} e circa 10^{308} e precisione di 15 cifre decimali

* In C la dimensione effettiva dei tipi di dato non è fissata dal linguaggio, ma può variare a seconda dell'architettura del sistema. Il range mostrato in tabella è riferito alla dimensione tipica.

Tipi di dato

char, short e int possono essere preceduti dalla parola chiave **unsigned**. La dimensione delle variabili rimane immutata, ma gli intervalli di valori rappresentabili cambiano come mostrato in tabella:

Nome	Dimensione tipica *	Intervallo di valori rappresentabili (range)
unsigned char	1 byte	numeri interi da 0 a 255 o caratteri ASCII
unsigned short	2 byte	numeri interi da 0 a 65.535
unsigned int	4 byte	numeri interi da 0 a 4.294.967.295

Output in C: la funzione *printf*

- La funzione printf consente di comporre messaggi che contengono porzioni di testo costanti e porzioni variabili.

Esempio:

```
int main() {  
    int x = 1, y=2;  
    printf("il valore di x e' %d e quello di y e' %d", x, y );  
    // stampa:  il valore di x e' 1 e quello di y e' 2  
}
```

- Il messaggio contiene dei segnaposto (%d). La funzione printf sostituisce i segnaposto con il valore delle variabili elencate dopo il messaggio. Quindi il primo %d è sostituito con il valore di x, e il secondo con il valore di y.
- %d indica che in questa posizione comparirà un valore di tipo intero espresso in base dieci (la d infatti sta per decimal)

Output in C: la funzione *printf*

- Per stampare il contenuto di variabili di tipo float, occorre utilizzare %f , mentre per le variabili double si usa %lf (long float) e per le variabili di tipo char si usa %c

```
int main() {  
    float x = 1.333;  
    char y = 'A';  
    printf("il valore di x e' %f \n", x );  
    printf("il valore di y e' %c \n", y );  
}
```

Output in C: la funzione *printf*

- **%d** , **%f**, **%lf** e **%c** sono detti **specificatori di formato** e la stringa che li contiene è detta **stringa di formato**
- Dopo la stringa di formato, si possono utilizzare anche espressioni al posto di variabili. Esempio:

```
float x = 1.333;  
printf("il quadrato di x e' %f e il cubo e' %f", x*x, x*x*x );
```

- Nel caso di %f e %lf, è possibile specificare il numero di cifre dopo la virgola che si vogliono visualizzare in questo modo:

```
float x = 5.38;  
printf("x = %.1f", x); // stampa x=5.4
```

Input in C: la funzione *scanf*

- Per acquisire un valore dall'esterno e porlo in una variabile, si utilizza la funzione `scanf`
- Esempio: per acquisire un numero intero e inserirlo nella variabile `x`, l'istruzione di input è:

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

- La funzione `scanf` acquisisce un valore dal dispositivo di input predefinito, che nel caso sarà sempre la tastiera
- Occorre sempre indicare uno specificatore di formato, che indica il tipo valore da acquisire, e il nome della variabile di destinazione preceduto dal simbolo `&`

Input in C: la funzione *scanf*

- Il simbolo & davanti al nome della variabile consente di estrarre l'indirizzo di memoria della variabile, cioè la posizione della variabile all'interno della memoria principale
- Il motivo per cui è necessario indicare alla funzione scanf l'indirizzo della variabile sarà chiaro più avanti, quando tratteremo i puntatori. Per ora è sufficiente ricordare che, quando si indica un nome di variabile nella funzione scanf, questo deve sempre essere preceduto dal carattere &.

Esempio: area del cerchio

- Vediamo un esempio completo di I/O, che introduce anche la direttiva `#define`:

```
1  #define PIGRECO 3.14
2  #include <stdio.h>
3
4  int main() {
5      float r, area;
6
7      printf("Inserisci il raggio: ");
8
9      scanf("%f",&r);
10
11     area = r*r*PIGRECO; // formula dell'area
12
13     printf("L'area del cerchio di raggio %f e' %f ", r, area);
14
15     /* questo è un commento
16     su più righe */
17 }
```

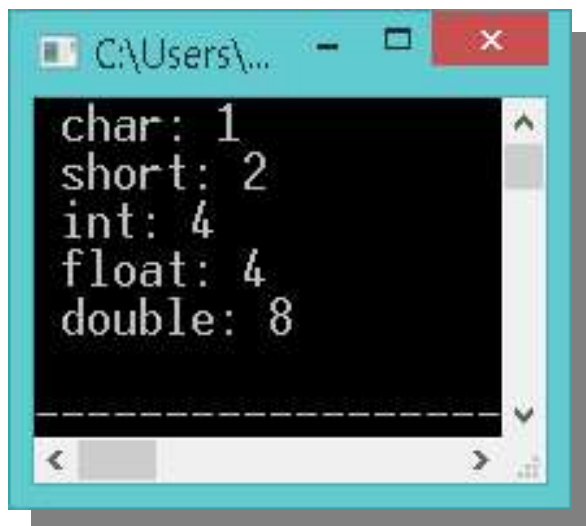

Analisi dell'esempio

- Le righe che iniziano con cancelletto (#) sono **direttive per il pre-compilatore**. Il pre-compilatore è un programma in grado di apportare alcune modifiche al sorgente prima della compilazione vera e propria.
- La direttiva **#define** serve a definire nuove costanti. Il pre-compilatore sostituisce ogni occorrenza della costante con il valore specificato dal programmatore. Attenzione: non mettere un punto e virgola alla fine di una definizione.
- I **commenti su singola** riga si aprono con **//** e terminano con l'andata a capo
- I **commenti su più righe** si aprono con **/*** e si chiudono con ***/**

L'operatore sizeof

Per verificare la dimensione effettiva dei tipi di dato sul proprio sistema, è possibile utilizzare l'operatore sizeof:

```
1  #include <stdio.h>
2  int main() {
3      printf(" char: %d \n", sizeof(char)    );
4      printf(" short: %d \n", sizeof(short)  );
5      printf(" int: %d \n", sizeof(int)      );
6      printf(" float: %d \n", sizeof(float)  );
7      printf(" double: %d \n", sizeof(double);
8  }
```

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Users\...' and standard window controls. The command prompt displays the output of the program: 'char: 1', 'short: 2', 'int: 4', 'float: 4', and 'double: 8'. Each line is on a new line, separated by a newline character. The window has a black background and white text.

L'operatore sizeof

- E' possibile utilizzare l'operatore sizeof anche passando come argomento una variabile, anzichè un tipo di dato. Quindi si può scrivere:

```
int x, y;  
y = sizeof(x); // se x occupa 4 byte, allora dopo  
               // questa istruzione y = 4
```

Operatori

Aritmetici

+	addizione
-	sottrazione
*	moltiplicazione
/	divisione
%	modulo

Relazionali

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

Logici

!	NOT
&&	AND
	OR

Assegnamento

=	assegnamento
+=	somma e assegnamento
-=	sottrazione e assegnamento
/=	divisione e assegnamento
*=	moltiplicazione e assegnamento

Incremento / Decremento

++	incremento
--	decremento

Operatori aritmetici

- In un programma C è possibile assegnare ad una variabile il risultato di un'espressione contenente le usuali operazioni matematiche.

```
int x=1, y=2, z;  
z = 1+x*y;
```

- Per modificare l'ordine delle operazioni si usano le parentesi tonde:

```
int x=1, y=2, z;  
z = (1+x)*y;
```

L'operatore modulo

- L'operatore **%** è detto **modulo** e calcola il resto della divisione tra il primo operando e il secondo. Esempio:

```
int x;  
x = 10%3;    // leggi "ad x assegna 10 modulo 3" .  
             // Ora x vale 1
```

Divisione intera e non intera

- L'operatore `/` in C è particolare, in quanto si comporta in modo diverso a seconda che gli operandi siano entrambi di tipo `int` oppure no.
- Se gli operandi sono **entrambi** interi, l'operatore `/` calcola il **quoziente** della divisione, cioè la parte intera del risultato della divisione. Esempio:

```
int x=9, y;  
y = x / 2; // poichè sia x che 2 sono di tipo intero,  
           // l'operatore / produce il valore 4
```

- Anche se dichiarassimo la variabile `y` di tipo `float`, il risultato non cambierebbe: la variabile `y` conterrebbe ancora il valore 4. Infatti, in un'assegnazione viene sempre prima calcolato il lato destro, che in questo caso è il valore 4, e poi il risultato viene posto nella variabile di sinistra.

Divisione intera e non intera

- Se almeno uno dei due operandi è di tipo float o double, l'operatore / effettua la divisione reale, producendo un valore che può avere cifre dopo la virgola. Esempio:

```
int x=9;  
float y;  
y = x / 2.0; // poichè 2.0 non è considerato un intero  
             // l'operatore / produce il valore 4.5
```


L'operazione di *cast*

- A volte si ha la necessita di modificare "temporaneamente" il tipo di dato di una variabile all'interno di un'espressione, in modo che, solo per quel calcolo, essa venga trattata come se il fosse di tipo diverso.
- Esempio: supponiamo di avere due variabili intere e voler calcolare il risultato della loro divisione reale.
- Il seguente codice non funziona:

```
int a=9,b=2;  
float c;  
c = a/b;  // non funziona! c contiene il valore 4 anzichè 4.5
```

L'operazione di *cast*

- Per ottenere il risultato voluto, occorre trasformare temporaneamente almeno uno dei due operandi in un valore float. L'esempio seguente mostra come trasformare temporaneamente b in un float, ottenendo il risultato cercato:

```
int a=9,b=2;  
float c;  
c = a / (float) b; // in questa espressione b è considerata  
                  // come se fosse di tipo float.  
                  // Ora c contiene il valore 4.5
```

L'operazione di *cast*

- Come mostrato, il cast di una variabile si realizza anteponendo al nome della variabile una coppia di parentesi tonde che racchiude il tipo di dato di "destinazione".
- L'operatore di cast ha una precedenza più alta rispetto alla divisione, quindi l'istruzione:

```
c = (float) a / b;
```

è da interpretare come:

```
c = ( (float) a ) / b;
```

Operatori di incremento e decremento

- In C è possibile incrementare o decrementare una variabile di una unità in modo molto compatto e veloce, utilizzando gli operatori ++ e --
- L'istruzione **x++;** è equivalente a **x = x + 1;**
- Questi operatori possono essere usati in due diverse modalità, che in generale **NON** sono **equivalenti**:
 - in modalità **post-incremento** o **post-decremento**, l'operatore è posizionato dopo la variabile. Esempio: x++;
 - in modalità **pre-incremento** o **pre-decremento**, l'operatore è posizionato prima della variabile. Esempio: ++x;

Differenza fra $x++$ e $++x$

- Pre-incremento e post-incremento producono effetti diversi quando l'operatore è utilizzato in una espressione
- Ad esempio, le due istruzioni seguenti NON sono equivalenti:

$y = ++x;$ $y = x++;$

- In entrambi i casi il valore di x è incrementato di una unità al termine dell'istruzione. Ciò che cambia è il valore di y . Infatti nel primo caso la variabile x è incrementata prima che il suo valore sia utilizzato nell'espressione, quindi y riceve il valore di x già incrementato. Nel secondo caso, la variabile x è incrementata solo dopo aver usato il suo valore nell'espressione, quindi y riceve il valore che x aveva prima dell'incremento.

Differenza fra `x++` e `++x`

```
y = x++;
```

=

```
y = x;  
x = x + 1;
```

la variabile è incrementata
DOPO aver usato il suo
valore nell'espressione.

```
y = ++x;
```

=

```
x = x + 1;  
y = x;
```

la variabile è incrementata
PRIMA di usare il suo
valore nell'espressione.

Operatori logici

&&

	Vero	Falso
Vero	Vero	Falso
Falso	Falso	Falso

||

	Vero	Falso
Vero	Vero	Vero
Falso	Vero	Falso

!

Vero	Falso
Falso	Vero

Il C utilizza gli interi per rappresentare i valori logici:

- 0 indica FALSO
- 1 indica VERO (ma ogni altro valore diverso da 0 è considerato VERO)

Operatori logici

- Esempio:

```
int main() {  
    int p = 0, q = 1;  
    printf("%d %d %d", p&&q, p||q, !p );  
}
```


Leggi di De Morgan

$$\neg(P \wedge Q) = \neg P \vee \neg Q$$

P	Q	$\neg(P \wedge Q)$	$\neg P \vee \neg Q$
0	0	1	1
0	1	1	1
1	0	1	1
1	1	0	0

$$\neg(P \vee Q) = \neg P \wedge \neg Q$$

P	Q	$\neg(P \vee Q)$	$\neg P \wedge \neg Q$
0	0	1	1
0	1	0	0
1	0	0	0
1	1	0	0

La selezione

- L'istruzione di selezione (o scelta) consente di dividere il flusso del programma in due rami: un blocco di istruzioni è eseguito quando una certa condizione è vera, e l'altro quando è falsa. Si realizza in C con le parole chiave **if** e **else**. La sintassi è:

```
if ( condizione ) {  
    istruzione  
    istruzione  
    ...  
}  
else {  
    istruzione  
    istruzione  
}
```

il ramo else non è obbligatorio e può essere omesso.

La selezione

- Esempio:

```
int x;  
printf("Quanti anni hai?");  
scanf("%d",&x);  
if (x>=18) {  
    printf("Maggiorenne");  
}  
else {  
    printf("Minorenne");  
}
```

Switch

- L'istruzione **switch** consente di scrivere in modo più compatto una sequenza di if..else.
- Ad esempio, consideriamo il codice seguente:

```
int x;  
scanf("%d",&x);  
if ( x==1 ) {  
    printf("Caso A");  
}  
else if ( x==2 ) {  
    printf("Caso B");  
}  
else if ( x==3 ) {  
    printf("Caso C");  
}  
else {  
    printf("Caso non valido");  
}
```

Switch

- Usando l'istruzione switch, il codice precedente diventa:

```
int x;  
scanf("%d",&x);  
switch( x ) {  
    case 1:  
        printf("caso A");  
        break;  
    case 2:  
        printf("caso B");  
        break;  
    case 3:  
        printf("caso C");  
        break;  
    default:  
        printf("caso non valido");  
}
```

Switch

- La sintassi dello switch è:

```
switch( espressione ) {  
    case valore1:  
        istruzione  
        istruzione  
        ...  
        break;  
    case valore2:  
        istruzione  
        istruzione  
        ...  
        break;  
    ...  
    default:  
        istruzione  
        istruzione  
        ...  
}
```

Switch

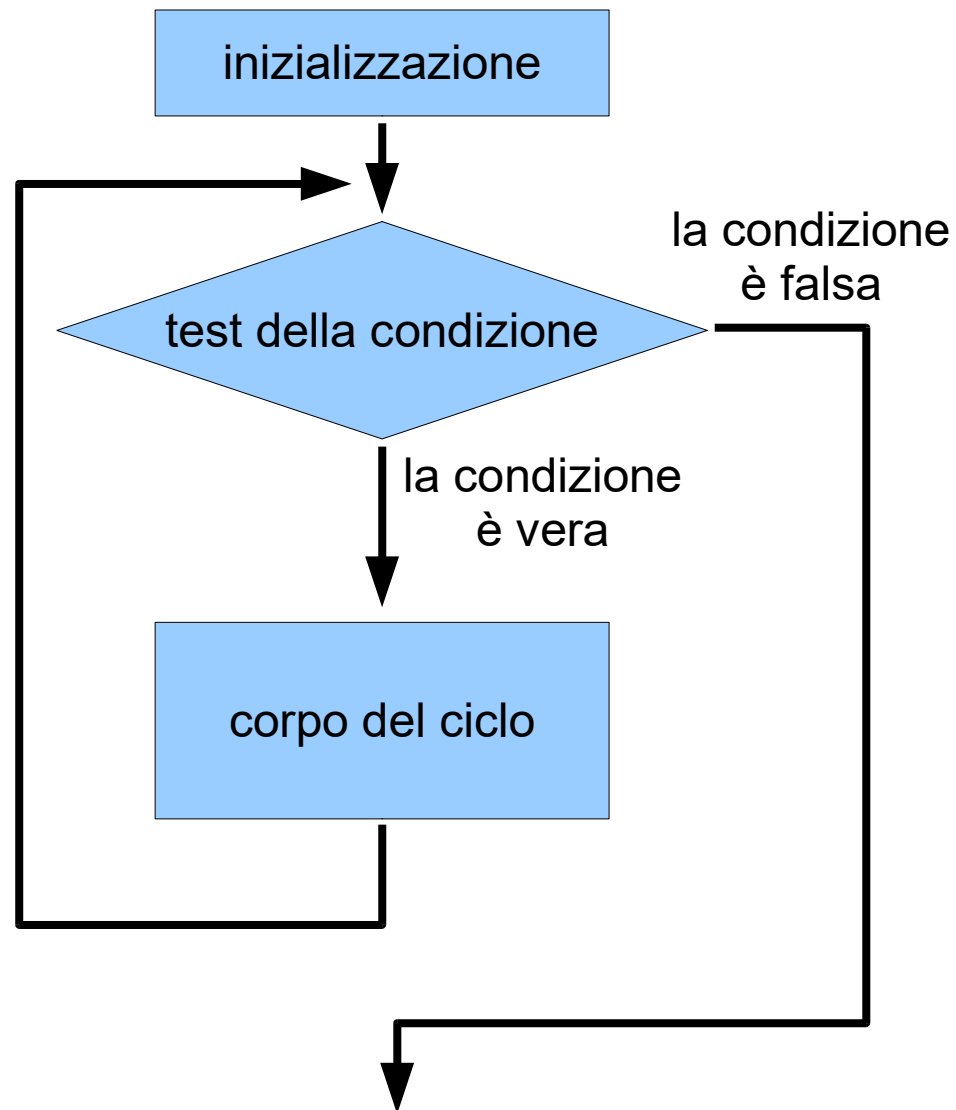
- Funzionamento dello switch: l'espressione contenuta tra parentesi tonde viene confrontata con i valori dei vari "casi" (che devono essere tutti diversi tra loro).
- Se il valore dell'espressione è uguale a uno dei casi, allora vengono eseguite le istruzioni di quel caso, altrimenti vengono eseguite le istruzioni del caso di default.
- L'istruzione **break** serve ad uscire dallo switch, cioè a saltare alla prima istruzione successiva alla parentesi graffa di chiusura dello switch. Se non si inserisce il break, dopo aver aver eseguito le istruzioni di un caso verrebbero eseguite anche le istruzioni dei casi successivi (fino alla fine dello switch o fino al primo break successivo).

Ciclo while

- Il ciclo while consente di ripetere un blocco di istruzioni, dette **corpo** del ciclo, finché rimane vera una determinata condizione. In altre parole: il ciclo termina quando la condizione diventa falsa. La struttura di un ciclo while è:

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


Diagramma di flusso del ciclo *while*



Ciclo while

- Esempio:

```
i=0;
while(i<4) {
    printf("ciao \n");
    i++;
};
```

- Funzionamento del ciclo while: per prima cosa viene verificata la condizione. Se questa è vera, si eseguono le istruzioni tra parentesi graffe (corpo del ciclo) e si verifica di nuovo la condizione. Se è ancora vera, si esegue di nuovo il corpo. Se la condizione è falsa si esce dal ciclo, cioè si passa ad eseguire la prima istruzione dopo il ciclo.

Ciclo while

- Poichè nel ciclo while la condizione è verificata prima di ogni ripetizione, può capitare che il corpo del ciclo sia eseguito zero volte!
- Esempio di ciclo while ripetuto zero volte (errore da evitare!)

```
i=0;
while(i == 10) {
    printf("ciao \n"); // istruzioni mai eseguite
    i++;
};
```

Ciclo do-while

- Il ciclo do-while è un ciclo in cui la condizione è verificata alla fine di ogni ripetizione. Come nel ciclo while, il corpo è ripetuto se la condizione è vera, e quindi il ciclo termina quando la condizione diventa falsa. La sintassi è:

```
inizializzazione  
do {  
    istruzione  
    istruzione  
    ...  
}  
while ( condizione );
```

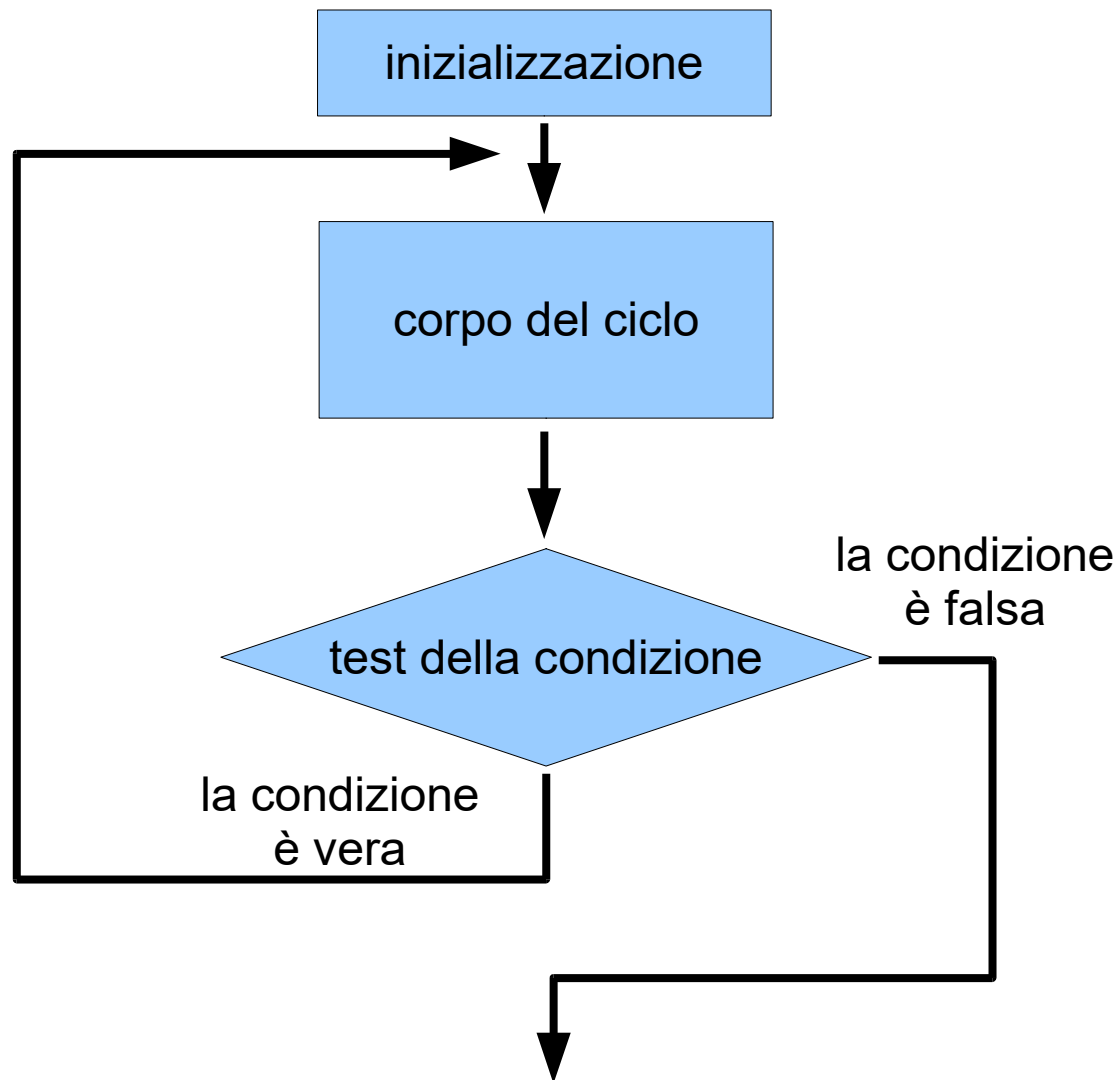
- A differenza del ciclo while, il corpo del ciclo do-while è eseguito sempre almeno una volta, poichè la condizione è verificata per la prima volta solo dopo la prima ripetizione!

Ciclo do-while

- Esempio:

```
i=0;  
do {  
    printf("ciao \n");  
    i++;  
} while( i<4 );
```

Diagramma di flusso del ciclo *do-while*



Controlli dell'input

- Il ciclo do-while è particolarmente indicato per verificare la validità degli input forniti dall'utente, poichè in questi casi non ha senso verificare la condizione prima di aver ricevuto in input il valore da controllare.
- Esempi:

```
int eta;  
do {  
    printf("Quanti anni hai? ");  
    scanf("%d",&eta);  
} while( eta < 0 ) ;
```

```
int anno;  
do {  
    printf("Quale anno stai frequentando (da 1 a 5) ? ");  
    scanf("%d",&anno);  
} while( anno < 1 || anno > 5 ) ;
```

Ciclo for

- In linguaggio C il ciclo **for** è un modo alternativo e più compatto per scrivere un ciclo while. La sintassi è:

```
for( inizializzazione ; condizione ; incremento ) {  
    istruzione  
    istruzione  
    ...  
};
```

- Esempio:

```
// ciclo che stampa: 0 1 2 3  
for(i=0 ; i<4 ; i++) {  
    printf("%d",i);  
};
```


Ciclo for

- Qualsiasi ciclo for è equivalente ad un ciclo while avente la stessa condizione, e in cui l'inizializzazione è posta immediatamente prima del while e l'incremento è posto come ultima istruzione del corpo. Quindi ogni ciclo for può essere tradotto in while, e viceversa.
- Ad esempio, i due cicli seguenti sono equivalenti:

```
for(i=0 ; i<4 ; i++) {  
    printf("%d ",i);  
};
```

```
i=0;  
while( i<4 ) {  
    printf("%d ",i);  
    i++;  
}
```

- La versione con for è più compatta e più chiara... ed è quasi impossibile dimenticarsi dell'inizializzazione e dell'incremento, in quanto entrambi vanno specificati già sulla prima riga!