Linguaggi di programmazione: Il linguaggio C

Un programma è la rappresentazione di un algoritmo in un particolare *linguaggio di programmazione*.

La programmazione è basata sul concetto di **astrazione**:

- *L'astrazione è il processo con cui si descrive un fenomeno attraverso un sottoinsieme delle sue proprietà.*

☞Ogni programma è un'astrazione del problema da risolvere: si costruisce un modello astratto del problema, concentrandosi soltanto sulle proprietà importanti ai fini della risoluzione, e trascurando eventuali altre caratteristiche, considerate irrilevanti.

Potere espressivo di un linguaggio

È la capacità di fornire costrutti di astrazione il più possibile simili ai concetti utilizzati nella descrizione del metodo risolutivo.

☞Quanto più il potere espressivo di un linguaggio è elevato, tanto più il programmatore è facilitato nella fase di formalizzazione del metodo risolutivo in programma.

In particolare, il potere espressivo di un linguaggio si manifesta:

- nei **tipi di dato** che si possono esprimere e manipolare;
- nelle operazioni esprimibili, cioè nell'insieme di istruzioni previste per esprimere il **controllo** del flusso di esecuzione.

Programma = Dati + Controllo

Il linguaggio C

Progettato nel 1972 da D. M. Ritchie presso i laboratori AT&T Bell per poter riscrivere, in un linguaggio di alto livello, il codice del sistema operativo UNIX.

Definizione formale nel 1978 (B. W. Kernighan e D. M. Ritchie)

Nel 1983 è stato definito uno standard (**ANSI C**) da parte dell'American National Standards Institute.

Caratteristiche principali

- **Elevato potere espressivo:**
  - Tipi di dato primitivi e tipi di dato definibili dall'utente
  - Strutture di controllo (programmazione strutturata, funzioni e procedure)
- Caratteristiche di **basso livello** (gestione della memoria, accesso alla rappresentazione dei dati).
- Stile di programmazione che incoraggia lo sviluppo di programmi per passi di raffinamento successivi (sviluppo **top-down**).
- Sintassi definita formalmente.

Elementi del testo di un programma C

Nel **testo** di un programma C possono comparire:

- parole chiave
- commenti
- caratteri e stringhe
- numeri (interi o reali) = valori costanti
- identificatori

Parole chiave

`auto`	`break`	`case`	`const`
`continue`	`default`	`do`	`double`
`else`	`enum`	`extern`	`float`
`for`	`goto`	`if`	`int`
`long`	`register`	`return`	`short`
`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`unsigned`	`void`
`volatile`	`while`		

Le parole chiave sono **parole riservate**, cioè non possono essere utilizzate come identificatori (ad esempio di variabili).

Commenti

Sono sequenze di caratteri ignorate dal compilatore.

Vanno racchiuse tra **`/*`** … **`*/`**:

```
/* questo è
   un commento
   dell'autore */
```

I commenti vengono generalmente usati per introdurre note esplicative nel codice di un programma.

Costanti numeriche

Numeri interi

Rappresentano numeri relativi (quindi con segno):

	2 byte	**4 byte**
base decimale	12	70000, 12L
base ottale	014	0210560
base esadecimale	0xFF	0x11170

Numeri reali

Varie notazioni

```
24.0   2.4E1   240.0E-1
```

Suffissi: `l, L, u, U` (interi-long, unsigned)
`f, F` (reali - floating)

Prefissi

`0` (ottale)
`0x, 0X` (esadecimale)

Costanti carattere

Dipendono dall'insieme dei caratteri disponibili (è dipendente dalla implementazione). In genere, questo corrisponde al set ASCII esteso (256 caratteri). Si indicano tra singoli apici:

```
'a'   'A'
```

Caratteri speciali

newline (fine linea)	**`\n`**
tab (tabulazione)	**`\t`**
backspace (spazio indietro)	**`\b`**
form feed (fine pagina)	**`\f`**
carriage return (ritorno carrello)	**`\r`**
codifica ottale	`\xxx` (x = cifra ottale 0-7) `\041` è la codifica del carattere !

Il carattere `\` inibisce il significato predefinito di alcuni caratteri "speciali" (es. `'`, `"`, `\`, etc.)

`\'` `\\` `\"` `\0` (carattere nullo)

Costanti stringa

Sono sequenze di caratteri tra doppi apici `" "`.

`"a"` **`"aaa"`** **`""`** (stringa nulla)

Esempio:

`printf` è l'istruzione per la stampa

```
printf("Prima riga\nSeconda riga\n");
printf("\\\"/");
```

Effetto ottenuto:

```
Prima riga
Seconda riga
\"/
```

Identificatori

Un **identificatore** è un nome che denota un oggetto usato nel programma (ad esempio: variabili, costanti, tipi, procedure e funzioni).

- Deve iniziare con una lettera (o con il carattere '_'), alla quale possono seguire lettere e cifre in numero qualunque:

<identificatore> ::= <lettera> {<lettera> | <cifra>}

- Distinzione tra maiuscole e minuscole (linguaggio **case-sensitive**).

Esempio:

Sono identificatori validi:

```
Alfa    beta
Gamma1  Gamma2
```

Non sono identificatori validi:

```
3X      int
```

Regola generale

☞Prima di essere usato, un identificatore deve essere già stato definito in una parte di testo precedente.

Struttura di un programma C

Nel caso più semplice, un **programma C** consiste in:

<programma> ::=
[<parte-dich-globale>]
<main>
[{<altre-funzioni>}]

Struttura del main

La parte **<main>** di un programma è suddivisa in:

<main> ::=
`main() {`
<parte-dichiarazioni>
<parte-istruzioni>
`}`

☞Il **<main>** è costituito da due parti:

- Una **parte di dichiarazioni** (variabili, tipi, costanti, etichette, etc.) in cui vengono descritti e definiti gli oggetti che vengono utilizzati dal main.
- Una **parte istruzioni** che descrive l'algoritmo risolutivo utilizzato, mediante istruzioni del linguaggio.

Esempio:

```
/* programma che, letti due numeri a
terminale, ne stampa la somma */

#include <stdio.h>

main()
{

int X,Y; /* p. dichiarativa */

scanf("%d%d",&X,&Y); *p. istruzioni*/
printf("%d",X+Y);

}
```


Variabili

Una **variabile** rappresenta un dato che può cambiare il proprio valore durante l'esecuzione del programma.

Regola generale

In ogni linguaggio di alto livello una variabile è caratterizzata da un nome (identificatore) e quattro attributi base:

- **campo d'azione** (scope): è l'insieme di istruzioni del programma in cui la variabile è nota e può essere manipolata;
  - C, Pascal, determinabile staticamente
  - LISP, dinamicamente
- **tempo di vita** (o durata o estensione): è l'intervallo di tempo in cui un'area di memoria è legata alla variabile;
  - ForTRAN, allocazione statica
  - C, Pascal, allocazione dinamica
- **valore**: è rappresentato (secondo la codifica adottata) nell'area di memoria legata alla variabile;
- **tipo**: definisce l'insieme dei valori che la variabile può assumere e l'insieme degli operatori applicabili.

Variabili in C

- Ogni variabile, per poter essere utilizzata dalle istruzioni del programma, deve essere preventivamente **definita**.

Definizione di variabili

La **definizione** di variabile associa ad un identificatore (**nome** della variabile) un **tipo**.

<def-variabili> ::=
<identificatore-tipo> <identificatore-variabile>
{, <identificatore-variabile>};

Esempi:

```
int A, B, SUM;     /* Variabili A, B, SUM intere */

float root, Root;  /* Variab. root, Root reali */

char C;            /* Variabile C carattere */
```

Effetto della definizione di variabile

- La definizione di una variabile provoca come effetto l'allocazione in memoria della variabile specificata (allocazione *automatica*).
- Ogni istruzione successiva alla definizione di una variabile A, potrà utilizzare A.

Istruzione di assegnamento

Il concetto di **variabile** nel linguaggio C rappresenta un'astrazione della cella di memoria.

L'istruzione di **assegnamento**, quindi, è l'astrazione dell'operazione di scrittura nella cella che la variabile rappresenta.

Assegnamento

<identificatore-variabile> = <espressione>

Esempi

```
main() {
 int a; /* definizione di a */
 ...
 a=100; /* assegnamento ad a del
           valore 100 */ }
```

```
#include <stdio.h>
main() {
 float X, Y;

 scanf("%f", &X); /* lettura */
/* assegnamento del risultato di una
espr. aritmetica: */
 Y = 2*3.14*X;
 printf("%f", Y); /* stampa */ }
```

Costanti

Una **costante** rappresenta un dato che **non** può cambiare di valore nel corso dell'esecuzione.

La dichiarazione di una costante associa ad un identificatore (**nome** della costante) un **valore** (espresso eventualmente mediante altra costante).

<dich-costante> ::=
const <tipo> <id.-costante> = <costante>
<costante> ::= (+ | -) <id.-costante> |
(+|-) <numero-senza-segno> | <altre-costanti>

```
const float pigreco = 3.14;
```

```
const float pigreco = 3.1415926,
  e = 2.7182,
  menoe = - e;
```

☞ Anche in questo caso, prima di essere usato, un identificatore deve essere già stato definito (ad es., `e` per definire `menoe`).

Vantaggi derivanti dall'uso di costanti

Leggibilità e modificabilità dei programmi.

Esempio:

```
#include <stdio.h>

main()
{
/* programma che, letto un numero da
terminale, stampa il valore della
circonferenza del cerchio con raggio
pari a tale numero */

 const float pigreco = 3.1415926;
 float X, Y;

 scanf("%f", &X); /*legge X */
 Y = 2 * pigreco * X;
 printf("%f", Y); /* stampa Y */
}
```

Tipo di dato

Un **tipo di dato** *T* è definito come:

- Un insieme di valori *D* (**dominio**);
- Un insieme di funzioni (**operazioni**), f_1, …, f_n, definite sul dominio *D*.

In pratica

Un tipo *T* è definito:

- dall'insieme di valori che le variabili di tipo *T* possono assumere;
- dall'insieme di operazioni che possono essere applicate ad operandi del tipo *T*.

Esempio:

Consideriamo i numeri *naturali*

Tipo_naturali = (N, {+, -, *, /, =, >, <, etc })

- N è il dominio;
- {+, -, *, /, =, >, <, etc } è l'insieme di operazioni.

Il concetto di *tipo*

Un linguaggio di programmmazione si dice *tipato* se prevede costrutti specifici per attribuire tipi ai dati utilizzati nei programmi.

Se un linguaggio è tipato:

☞Ogni dato (variabile o costante) del programma deve appartenere ad **uno ed un solo** tipo.

☞Ogni operatore richiede **operandi** di tipo specifico e produce **risultati** di tipo specifico.

Vantaggi:

- **Astrazione**: l'utente esprime e manipola i dati ad un livello di astrazione più alto della loro organizzazione fisica ➠ maggior portabilità.
- **Protezione**: il linguaggio protegge l'utente da combinazioni errate di dati ed operatori (**controllo statico** sull'uso di variabili, etc., in fase di compilazione).
- **Portabilità**: l'indipendenza dall'architettura rende possibile la compilazione dello stesso programma su macchine profondamente diverse.

Tipo di dato in C

Il C è un linguaggio tipato.

Classificazione dei tipi di dato in C

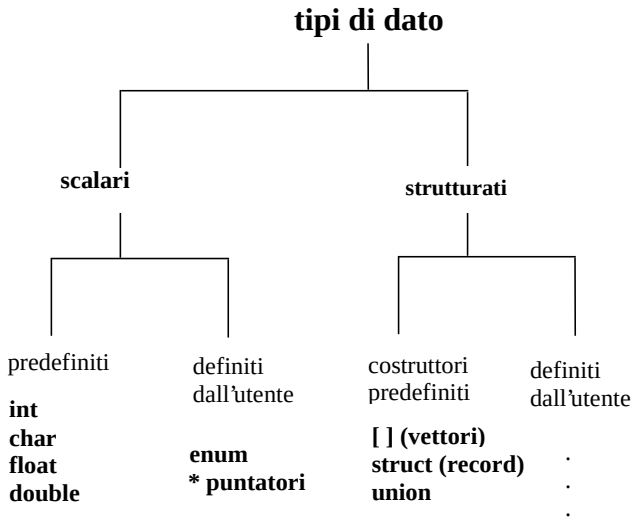
Esiste distinzione tra:

- tipi ***primitivi***: sono tipi di dato previsti dal linguaggio (built-in) e quindi rappresentabili direttamente;
- tipi ***non primitivi***: sono tipi ***definibili dall'utente*** **(**mediante appositi costruttori di tipo, v. *typedef*).

Inoltre, esiste distinzione tra:

- tipi ***scalari***, il cui dominio è costituito da elementi *atomici*, cioè logicamente non scomponibili;
- tipi ***strutturati,*** il cui dominio è costituito da elementi non atomici (e quindi scomponibili in altri componenti).

Classificazione dei tipi di dato in C



Tipi primitivi

Il C prevede quattro tipi primitivi:

- **`char`** (caratteri)
- **`int`** (interi)
- **`float`** (reali)
- **`double`** (reali in doppia precisione)

☞È possibile applicare ai tipi primitivi dei **quantificatori** e dei **qualificatori**:

Quantificatori

- I ***quantificatori*** (*long* e *short*) influiscono sullo spazio in memoria richiesto per l'allocazione del dato.
  - **`short`** (applicabile al tipo **`int`**)
  - **`long`** (applicabile ai tipi **`int`** e **`double`**)

Esempio:

```
int X; /* se X è su 16 bit...*/
long int Y; /*... Y è su 32 bit */
```

Qualificatori

- I ***qualificatori*** condizionano il dominio dei dati:
  - **`signed`** (applicabile ai tipo **`int`** e **`char`**)
  - **`unsigned`** (applicabile ai tipo **`int`** e **`char`**)

```
int A; /* A in [-2e15, 2e15-1] */
unsigned int B; /* B in [0, 2e16-1] */
```


Il tipo `int`

Dominio

Il dominio associato al tipo `int` rappresenta l'insieme dei numeri interi (cioè **Z**, insieme dei numeri relativi): ogni variabile di tipo `int` è quindi l'astrazione di un intero.

Esempio: definizione di una variabile intera

```
int A; /* A è un dato intero */
```

☞Poiché si ha sempre a disposizione un numero **finito** di bit per la rappresentazione dei numeri interi, il dominio rappresentabile è di estensione finita.

Ad esempio:

Se il numero *n* di bit a disposizione per la rappresentazione di un intero è 16, allora il dominio rappresentabile è composto di:

$(2^n - 1) = 2^{16} - 1 = 65.536$ valori

Uso dei quantificatori `short/long`

Aumentano/diminuiscono il numero di bit a disposizione per la rappresentazione di un intero:

spazio(**`short int`**) <= *spazio*(**`int`**) <= *spazio*(**`long int`**)

Uso dei qualificatori:

- **`signed`:** viene usato un bit per rappresentare il segno (in realtà si usa la rappresentazione in complemento a 2). Quindi l'intervallo rappresentabile è:

$$[-2^{n-1}, +2^{n-1}-1]$$

- **`unsigned`:** vengono rappresentati valori a priori positivi. Intervallo rappresentabile:

$$[0, 2^{n-1}-1]$$

Operatori per il tipo `int`

Al tipo **`int`** (ed ai tipi ottenuti da questo mediante qualificazione/quantificazione) sono applicabili i seguenti operatori:

Operatori aritmetici

Forniscono risultato intero:

`+, -, *, /`	somma, sottrazione, prodotto, divisione intera
`%`	operatore *modulo*: resto della divisione intera `10 % 3` ➠ `1`
`++, --`	*incremento* e *decremento*: richiedono un solo operando (una variabile) e possono essere postfissi (**`a++`**) o prefissi (**`++a`**) (v. espressioni)

Operatori relazionali

Si applicano ad operandi interi e producono risultati ***"booleani"*** (cioè il cui valore può assumere soltanto uno dei due valori {*vero*, *falso*}):

`==, !=`	uguaglianza, disuguaglianza `10 == 3` ➠ *falso*, `10 != 3` ➠ *vero*
`<, >, <=, >=`	minore, maggiore, minore o uguale, maggiore o uguale `10 >= 3` ➠ *vero*

Booleani

Sono dati il cui dominio è di due soli valori (valori *logici*):

{*vero, falso*}

☞In C **non esiste** un tipo primitivo per rappresentare dati booleani.

Come vengono rappresentati i risultati di espressioni relazionali?

- Il C prevede che i valori logici restituiti da espressioni relazionali vengano rappresentati attraverso gli interi {0, 1} secondo la convenzione:
  - **0 equivale a *falso***
  - **1 equivale a *vero***

Esempio:

L'espressione `A == B` restituisce:

- ➠ **0**, se la relazione non è vera
- ➠ **1**, se la relazione è vera

Operatori logici

Si applicano ad operandi di tipo `int` e producono risultati *booleani*, cioè interi appartenenti all'insieme {0, 1} (il valore 0 corrisponde a "falso", il valore 1 corrisponde a "vero"). In particolare l'insieme degli operatori logici è:

`&&`	operatore AND logico
`||`	operatore OR logico
`!`	operatore di negazione (NOT)

Tabella di verità degli operatori logici

a	b	a && b	a || b	!a
falso	*falso*	*falso*	*falso*	*vero*
falso	*vero*	*falso*	*vero*	*vero*
vero	*falso*	*falso*	*vero*	*falso*
vero	*vero*	*vero*	*vero*	*falso*

Operatori logici in C

In C, gli operatori logici agiscono su operandi di tipo **`int`**:

- Se il valore di un operando è **diverso da zero**, viene interpretato come ***vero***.
- Se il valore di un operando è **uguale a zero**, viene interpretato come ***falso***.

Definizione degli operatori logici in C

a	b	a && b	a || b	!a
0	0	0	0	1
0	≠ 0	0	1	1
≠ 0	0	0	1	0
≠ 0	≠ 0	1	1	0

Esempi sugli operatori tra interi:

```
37 / 3 ➠ 12

37 % 3 ➠ 1

7 < 3 ➠ 0

7 >=3 ➠ 1

0 || 1 ➠ 1

0 || -123 ➠ 1

12 && 2 ➠ 1

0 && 17 ➠ 0

! 2 ➠ 0
```

I tipi `float` e `double` (reali)

Dominio

Concettualmente, rappresenta l'insieme dei numeri reali $\Re$.

In realtà, è un sottoinsieme di $\Re$ a causa di:

- **precisione** limitata;
- **limitatezza** del dominio.

Lo spazio allocato per ogni numero reale (e quindi l'insieme dei valori rappresentabili) dipende dal metodo di rappresentazione adottato.

Differenza tra `float/double`

`float` *singola* precisione
`double` *doppia* precisione (maggiore numero di bit per la mantissa)

Uso del quantificatore `long`

Si può applicare a **`double`**, per aumentare ulteriormente la precisione:

spazio(**`float`**) <= *spazio*(**`double`**) <= *spazio*(**`long double`**)

Esempio: definizione di variabili reali

```
float x;
double A, B;
```


Operatori per i tipi `float/double`

Operatori aritmetici

`+, -, *, /` si applicano a operandi reali e producono risultati reali.

Operatori relazionali

Hanno lo stesso significato visto nel caso degli interi:

`==, !=` uguale, diverso

`<, >, <=, >=` minore, maggiore, etc.

Overloading

In C (come in Pascal, Fortran e molti altri linguaggi) operazioni primitive associate a tipi diversi possono essere denotate con lo stesso simbolo (ad esempio, le operazioni aritmetiche su reali od interi).

Esempi:

```
5.0 / 2 ➠ 2.5

2.1 / 2 ➠ 1.05

7.1 < 4.55 ➠ 0

17 == 121 ➠ 0
```

☞A causa della rappresentazione finita, ci possono essere errori di conversione. Ad esempio, i test di uguaglianza tra valori reali (in teoria uguali) potrebbero non essere verificati.

```
(x / y) *y == x
```

Meglio utilizzare "un margine accettabile di errore":

```
(x == y) ➠ (x >= y - epsilon) && (x <= y + epsilon)
```

dove, ad esempio:

```
const float epsilon = 0.000001;
```


Il tipo `char`

Carattere

Ogni simbolo grafico rappresentabile all'interno del sistema. Ad esempio:

- le lettere dell'alfabeto (maiuscole, minuscole)
- le cifre decimali ('`0`', …, '`9`')
- i segni di punteggiatura ('`,`', '`.`' etc.)
- altri simboli di vario tipo ('`+`', '`-`', '`&`', '`@`', etc.)
- i caratteri di controllo (*`bell`*, *`lf`*, *`ff`*, etc.)

Dominio del tipo `char`

È l'insieme dei *caratteri* disponibili sul sistema di elaborazione (*set* di caratteri).

Tabella dei codici

Di solito, si fa riferimento ad una tabella dei codici (ad esempio: ASCII). In ogni tabella dei codici, ad ogni carattere viene associato un intero che lo identifica univocamente: il **codice**.

- Il dominio associato al tipo **char** è **ordinato**: l'ordine dipende dal codice associato ai vari caratteri.

Tabella ASCII

Di solito, vengono usati **8 bit** → 256 valori possibili

0	NUL	42	*	84	T	126	~	168	®	210	“	252	¸		
1	SOH	43	+	85	U	127		169	©	211	”	253	˝		
2	STX	44	,	86	V	128	Ä	170	™	212	‘	254	˛		
3	ETX	45	-	87	W	129	Å	171	´	213	‘	255	ˇ		
4	EOT	46	.	88	X	130	Ç	172	¨	214	÷				
5	ENQ	47	/	89	Y	131	É	173	≠	215	◊				
6	ACK	48	0	90	Z	132	Ñ	174	Æ	216	ÿ				
7	BEL	49	1	91	[	133	Ö	175	Ø	217	Ÿ				
8	BS	50	2	92	\	134	Ü	176	∞	218	⁄				
9	HT	51	3	93	]	135	á	177	±	219	¤				
10	LF	52	4	94	^	136	à	178	≤	220	‹				
11	VT	53	5	95	_	137	â	179	≥	221	›				
12	FF	54	6	96	`	138	ä	180	¥	222	fi				
13	CR	55	7	97	a	139	ã	181	µ	223	fl				
14	SO	56	8	98	b	140	å	182	∂	224	‡				
15	SI	57	9	99	c	141	ç	183	∑	225	·				
16	DLE	58	:	100	d	142	é	184	∏	226	,				
17	DC1	59	;	101	e	143	è	185	π	227	„				
18	DC2	60	<	102	f	144	ê	186	∫	228	‰				
19	DC3	61	=	103	g	145	ë	187	ª	229	Â				
20	DC4	62	>	104	h	146	í	188	º	230	Ê				
21	NAk	63	?	105	i	147	ì	189	Ω	231	Á				
22	SYN	64	@	106	j	148	î	190	æ	232	Ë				
23	ETB	65	A	107	k	149	ï	191	ø	233	È				
24	Can	66	B	108	l	150	ñ	192	¿	234	Í				
25	EM	67	C	109	m	151	ó	193	¡	235	Î				
26	SUB	68	D	110	n	152	ò	194		236	Ï				
27	ESC	69	E	111	o	153	ô	195	√	237	Ì				
28	FS	70	F	112	p	154	ö	196	ƒ	238	Ó				
29	GS	71	G	113	q	155	õ	197	≈	239	Ô				
30	RS	72	H	114	r	156	ú	198	Δ	240	□				
31	US	73	I	115	s	157	ù	199	«	241	Ò				
32		74	J	116	t	158	û	200	»	242	Ú				
33	!	75	K	117	u	159	ü	201	…	243	Û				
34	"	76	L	118	v	160	†	202		244	Ù				
35	#	77	M	119	w	161	°	203	À	245	ı				
36	$	78	N	120	x	162	¢	204	Ã	246	ˆ				
37	%	79	O	121	y	163	£	205	Õ	247	˜				
38	&	80	P	122	z	164	§	206	Œ	248	¯				
39	‘	81	Q	123	{	165	•	207	œ	249	˘				
40	(	82	R	124	|	166	¶	208	–	250	˙				
41	)	83	S	125	}	167	ß	209	—	251	˚				

Il tipo `char`

Definizione di variabili di tipo `char`

```
char C1, C2;
```

Costanti di tipo `char`:

Ogni valore di tipo **`char`** viene specificato tra singoli apici.

Ad esempio:

```
'a' 'b' 'A' '0' '2'
```

Rappresentazione dei caratteri in C

Il linguaggio C rappresenta i dati di tipo **`char`** come degli **interi**:

- Ogni carattere viene rappresentato dal suo codice (cioè **dall'intero** che lo indica nella tabella ASCII)

Operatori per il tipo `char`

I `char` sono rappresentati da interi (su 8 bit):

☞Sui dati **`char`** è possibile eseguire tutte le operazioni previste per gli interi. Ogni operazione, infatti, è applicata ai codici associati agli operandi.

Operatori relazionali

`==`, **`!=`**, **`<`**, **`<=`**, **`>=`** ,**`>`** per i quali valgono le stesse regole viste per gli interi

Ad esempio:

```
char x,y;
```

`x < y` se e solo se *codice*(`x`) < *codice*(`y`)
`'a' > 'b'` ***falso*** perché *codice*(`'a'`) < *codice*(`'b'`)

Operatori aritmetici

Sono gli stessi visti per gli interi.

Operatori logici:

Sono gli stessi visti per gli interi.

Esempi:

`'A' < 'C'` ➠ `1` (infatti 65 < 67 è *vero*)

`'"' + '!'` ➠ **`'C'`** (*codice*(**`"`**)+*codice*(**`!`**) = 67)

`! 'A'` ➠ `0` (*codice*(`'A'`) è diverso da zero)

`'A' && 'a'` ➠ `1`

Uso dei qualificatori

È possibile, come per gli interi, applicare i qualificatori `signed`, `unsigned` a variabili di tipo `char`:

```
signed char C;
unsigned char K;
```

Espressioni omogenee ed eterogenee

In C è possibile combinare tra di loro operandi di tipo diverso:

- espressioni **omogenee**: tutti gli operandi sono dello stesso tipo
- espressioni **eterogenee**: gli operandi sono di tipi diversi.

Regola adottata in C:

- Sono eseguibili le espressioni eterogenee in cui tutti i tipi referenziati risultano ***compatibili*** (cioè i tipi che, dopo l'applicazione della regola automatica di conversione implicita di tipo del C, risultano omogenei).
- Non sono eseguibili le espressioni eterogenee se tutti i tipi referenziati risultano non **compatibili** (cioè restano eterogenei anche dopo l'applicazione della regola automatica di conversione implicita di tipo del C).

Compatibilità fra tipi di dato

Definizione

Un tipo di dato T_1 è **compatibile** con un tipo di dato T_2 se il dominio D_1 di T_1 è contenuto in D_2, dominio di T_2.
Ad esempio: gli interi sono compatibili con i reali, perché $Z \subset \Re$

☞La relazione di compatibilità **non è simmetrica:**

- se T_1 è compatibile con T_2, non è detto che T_2 sia compatibile con T_1.

Proprietà

- Se T_1 è compatibile con T_2, un operatore *Op* definito per T_2 può essere anche utilizzato con argomenti T_1.

Quindi:

Se *Op* è definito per T2 come:

$$Op: D2 \times D2 \rightarrow D2$$

Allora può essere utilizzato anche come:

$$Op: D1 \times D2 \rightarrow D2$$
$$Op: D2 \times D1 \rightarrow D2$$

Compatibilità tra tipi primitivi

In C è definita la seguente gerarchia tra i tipi primitivi:

```
char < short < int < unsigned < long <
unsigned long < float < double < long
double
```

dove il simbolo < indica la relazione di **compatibilità**.

☞La gerarchia associa un ***rango*** a ciascun tipo:

Ad esempio:

rango(`int`) < *rango*(`double`)

Regola di conversione implicita

Facendo riferimento alla gerarchia tra tipi C primitivi, ad ogni espressione *x op y* viene applicata automaticamente la seguente **regola**:

1. Ogni variabile di tipo **`char`** o **`short`** (eventualmente con qualifica **`signed`** o **`unsigned`**) viene convertita nel tipo **`int`**.

2. Se, dopo il passo 1, l'espressione e` ancora eterogenea, si converte temporaneamente l'operando di tipo *inferiore* al tipo *superiore* (***promotion***).

3. A questo punto l'espressione è **omogenea** e viene eseguita l'operazione specificata. Il risultato è di tipo uguale a quello prodotto dall'operatore effettivamente eseguito (in caso di overloading, quello di rango più alto).

Compatibilità e conversione implicita di tipo

☞La compatibilità viene, di solito, controllata staticamente, applicando le regole di conversione implicita in fase di **compilazione** senza conoscere i valori attribuiti ai simboli (**tipizzazione forte**).

Esempio 1:

Espressione semplice

```
x / y
```

```
3 / 3.0 ➠ 1.0 (reale)
3.0 / 3 ➠ 1.0 (reale)
3 / 3 ➠ 1 (intero)
```

Esempio 2:

Espressione composta

```
int x;
char y;
double r;

(x + y) / r
```

È necessario conoscere:

- **Priorità** degli operatori (definita dallo standard)
- **Ordine di valutazione** degli operandi (lo standard non lo indica)

- **Ipotesi**: gli operandi vengono valutati da sinistra a destra.
- **passo 1:**

```
(x + y)
```

  - `y` viene convertito nell'intero corrispondente
  - viene applicata la somma tra interi

  ➠ **risultato intero** `tmp`

- **passo 2:**

```
tmp / r
```

  - `tmp` viene convertito nel corrispondente `double`
  - viene applicata la divisione tra reali

  ➠ **risultato reale**

Conversione esplicita

In C si può forzare la conversione di un dato in un tipo specificato, mediante l'operatore di **cast**:

(<nuovo tipo>) <dato>

Il <dato> viene convertito esplicitamente nel <nuovo tipo>:

```
int A, B;
float C;

C = A / (float)B;
```

⇒ viene eseguita la divisione tra reali.

Tipi primitivi nel linguaggio C: *Integral* Types e *Floating* Types

I tipi primitivi (scalari) del C possono essere suddivisi in tipi **enumerabili** (*Integral Types*) e non (*Floating Types*).

Tipi enumerabili (*Integral Types*)

Gli elementi del dominio associato al tipo sono rigidamente ordinati:

- È possibile fare riferimento al primo valore del dominio ed all'ultimo.
- Per ogni elemento è sempre possibile individuare l'elemento *precedente* (se non è il primo) ed il *successivo* (se non è l'ultimo).

☞A ciascun elemento del dominio può essere associato un valore intero positivo, che rappresenta il numero d'ordine dell'elemento nella sequenza ordinata dei valori.

Tipi enumerabili in C

- `char`
- `int`

Tipi non enumerabili (*Floating Types*)

Concettualmente, il dominio $\Re$ è un insieme *denso*: dati due elementi x_1 ed x_2 del dominio distanziati tra loro di un ε piccolo a piacere, esiste sempre un'infinità di valori di R contenuti nell'intervallo $[x_1, x_2]$.

Tipi non enumerabili in C

- `float`
- `double`

{Integral Types, Floating Types}

L'insieme di queste due categorie costituisce i **tipi aritmetici.**

Definizione ed inizializzazione delle variabili di tipo semplice

Definizione di variabili

Tutti gli identificatori di tipo primitivo descritti fin qui possono essere utilizzati per definire variabili.

Ad esempio:

```
char lettera;
int, x, y;
unsigned int P;
float media;
```

Inizializzazione di variabili

È possibile specificare un valore iniziale di una variabile in fase di definizione.

Ad esempio:

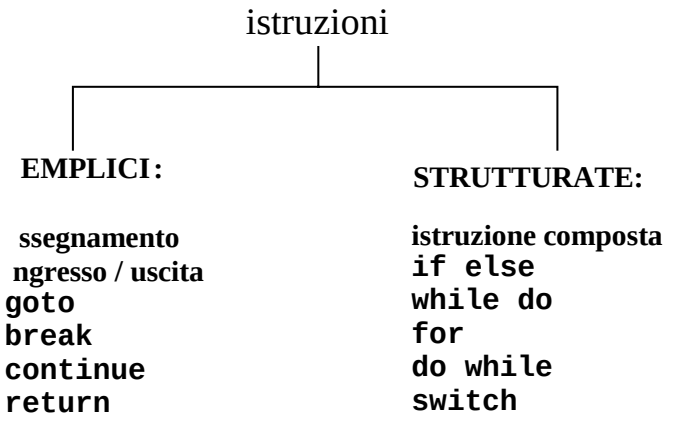
```
int x =10;
char y = 'a';
double r = 3.14*2;
```

☞Differisce dalla definizione di costanti, poiché i valori delle variabili, durante l'esecuzione del programma, potranno essere modificati.

Istruzioni: classificazione

In C le istruzioni possono essere classificate in due categorie:

- istruzioni **semplici**
- istruzioni **strutturate**: si esprimono mediante composizione di altre istruzioni (semplici e/o strutturate)



Regola sintattica generale

In C ogni istruzione è terminata da un punto e virgola (fa eccezione l'istruzione composta).

Assegnamento

È l'istruzione con cui si modifica il valore di una variabile.

Sintassi:

<istruzione-assegnamento> ::=
<identificatore-variabile> = <espressione>;

Esempio:

```
int A, B;

A=20;
B=A*5; /* B=100 */
```

Compatibilità di tipo ed assegnamento

In un assegnamento, l'identificatore di variabile e l'espressione devono essere dello stesso tipo (eventualmente, conversione implicita).

Esempio:

```
int x;
char y = 'a'; /* codice(a) = 97 */
double r;

x = y; /* char->int: x = 97*/
x = y + x; /* x = 194 */
r = y + 1.33; /* char->int->double */
x = r; /* troncamento: x = 98*/
```


Esempio:

```
main()
{
 /* definizioni variabili: */
int X, Y;
unsigned int Z;
float SUM;
 /* segue parte istruzioni */
 X = 27;
 Y = 343;
 Z = X + Y -300;
 X = Z / 10 + 23;
 Y = (X + Z) / 10 * 10;
 /* qui X = 30, Y = 100, Z = 70 */
 X = X + 70;
 Y = Y % 10;
 Z = Z + X -70;
 SUM = Z * 10;
 /* X = 100, Y = 0, Z = 100,
    SUM = 1000.0*/
}
```

Assegnamento come operatore

Formalmente, l'istruzione di assegnamento è un'espressione:

☞Il simbolo = è **un operatore**
- l'istruzione di assegnamento è un'espressione;
- restituisce un valore:
  - il valore restituito è quello assegnato alla variabile a sinistra del simbolo =;
  - il tipo del valore restituito è lo stesso della variabile oggetto dell'assegnamento.

Ad esempio:

```
const int valore = 122;
int K, M;

K = valore + 100;
   /* K = 122; l'espressione
   produce il risultato 222 */
M = (K = K / 2) + 1;
   /* K = 111, M = 112 */
```


Altri tipi di assegnamento

In C sono disponibili operatori che realizzano particolari forme di assegnamento.

Operatori di incremento e decremento

- Determinano l'incremento/decremento del valore della variabile a cui sono applicati.
- Restituiscono come risultato il valore incrementato/decrementato della **variabile** a cui sono applicati.

```
int A = 10;

A++; /* equivale a: A = A + 1; */
A--; /* equivale a: A = A - 1; */
```

Differenza tra notazione prefissa e postfissa

- **Notazione Prefissa**: (ad esempio, **++A**) significa che l'incremento viene fatto prima dell'impiego del valore di A nell'espressione.
- **Notazione Postfissa:** (ad esempio, **A++**) significa che l'incremento viene effettuato dopo l'impiego del valore di A nell'espressione.

Esempi:

```
int A = 10, B;
char C = 'a';

B = ++A; /*A e B valgono 11 */
B = A++; /* A = 12, B = 11 */
C++; /* C vale 'b' */


int i, j, k;
k = 5;
i = ++k; /* i = 6, k = 6 */
j = i + k++; /* j = 12, i = 6, k = 7 */
```

Infatti:

```
j = i + k++;
```

equivale a:

```
j = i + k;
k = k + 1;
```

☞In C l'ordine di valutazione degli operandi non è indicato dallo standard: si possono scrivere espressioni il cui valore è difficile da predire.

Operatore di assegnamento *abbreviato*

È un modo sintetico per modificare il valore di una variabile. Sia **v** una variabile, ***op*** un operatore (ad esempio, `+`, `-`, `/`, etc.), ed **e** una espressione.

```
v op= e
```

è ***quasi*** equivalente a:

```
v = v op (e)
```

Esempio:

```
k += j; /* equivale a k = k + j */
k *= a + b;
   /* equivale a k = k * (a + b) */
```

☞Le due forme sono **quasi equivalenti** perché in:

```
v op= e
```

v viene valutato una sola volta, mentre in:

```
v = v op (e)
```

v viene valutato due volte.

Espressioni sequenziali

Un'espressione sequenziale si ottiene concatenando tra loro più espressioni con l'operatore virgola (`,`).

- Il risultato prodotto da un'espressione sequenziale è il risultato ottenuto dall'ultima espressione della sequenza.
- La valutazione dell'espressione avviene valutando nell'ordine testuale le espressioni componenti, da sinistra verso destra.

Esempio:

```
int A = 1;
char B;
A = (B = 'k', ++A, A*2); /* A = 4 */
```


Precedenza ed associatività degli operatori

In ogni espressione, gli operatori sono valutati secondo una **precedenza** stabilita dallo standard, seguendo opportune regole di **associatività**:

- La **precedenza** indica l'ordine con cui vengono valutati operatori diversi;
- L'**associatività** indica l'ordine in cui operatori di pari priorità (cioè aventi la stessa precedenza) vengono valutati.

☞È possibile forzare le regole di precedenza mediante l'uso delle parentesi.

Regole di precedenza ed associatività degli operatori C

In ordine di priorità decrescente:

Operatore	Associatività
`() [] ->`	da sinistra a destra
`! ~ ++ -- & sizeof`	da destra a sinistra
`* / %`	da sinistra a destra
`+ -`	da sinistra a destra
`<< >>`	da sinistra a destra
`< <= > >=`	da sinistra a destra
`== !=`	da destra a sinistra
`&`	da sinistra a destra
`^`	da sinistra a destra
`|`	da sinistra a destra
`&&`	da sinistra a destra
`||`	da sinistra a destra
`?:`	da destra a sinistra
`=+ =- =* =/`	da destra a sinistra
`,`	da sinistra a destra

Esempi:

```
3 * 5 % 2          ➠ (3 * 5) % 2
X + 7 – A          ➠ (X + 7) – A
3 < 0 && 3 < 10    ➠ (3 < 0) && (3<10)
                   ➠ 0 && 1
3 < (0 && 3) < 10  ➠ (3 < 0) < 10
                   ➠ 0 < 1
0 == 7 == 3        ➠ 0 == (7 == 3)
                   ➠ 0 == 0
```

Valutazione a "corto circuito" *(short-cut)*:

- Nella valutazione di una espressione C, se un risultato intermedio determina a priori il risultato finale della espressione, il resto dell'espressione non viene valutato.

Ad esempio, espressioni logiche:

Ipotesi: Valutazione degli operandi da sinistra a destra

`(3 < 0) && (X < Y)` ➠ solo primo operando
`falso` `&& vero`

☞Bisognerebbe evitare di scrivere espressioni che dipendono dal metodo di valutazione usato ➠ scarsa portabilità (ad es., in presenza di funzioni con effetti collaterali).

Esercizi:

Sia V = 5, A = 17, B = 34. Determinare il valore delle seguenti espressioni logiche:

```
A<=20 || A>=40
!(B=A*2)
A<=B && A<=V
A>=B && A>=V
!(A>=B && A<=V)
!(A>=B) || !(A<=V)
```

Soluzioni:

Ipotesi: valutazione degli operandi da sinistra a destra

`A<=20 || A>=40`	➠ vero {`A<20`, shortcut}
`!(B=A*2)`	➠ falso {`B=34=17*2`}
`A<=B && A<=V`	➠ falso {`A>V`}
`A>=B && A>=V`	➠ falso {`A<B`}
`!(A<=B && A<=V)`	➠ vero
`!(A>=B) || !(A<=V)`	➠ vero

Precedenza ed associatività degli operatori

La **precedenza degli operatori** è stabilita dalla sintassi delle espressioni.

```
a + b * c
```

equivale **sempre** a

```
a + (b * c)
```

L'**associatività** è ancora stabilita dalla sintassi delle espressioni.

```
a + b + c
```

equivale **sempre** a

```
(a + b) + c
```

Istruzioni di ingresso ed uscita (input/output)

L'immissione dei dati di un programma e l'uscita dei suoi risultati avvengono attraverso operazioni di lettura e scrittura.

☞Il C non ha istruzioni predefinite per l'input/output.

Libreria standard di I/O

In ogni versione ANSI C, esiste una *Libreria Standard* di input/output (**stdio**) che mette a disposizione alcune funzioni (dette *funzioni di libreria*) che realizzano l'ingresso e l'uscita da/verso i dispositivi standard di input/output.

Dispositivi standard di input e di output

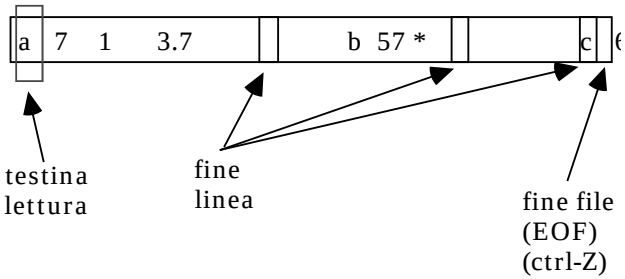
Per ogni macchina, sono periferiche predefinite (di solito, tastiera e video).

Input/output

Il C vede le informazioni lette/scritte da/verso i dispositivi standard di I/O come **file *sequenziali***, cioè come **sequenze di caratteri**.

I file standard di input/output possono contenere dei caratteri di controllo:

- **End Of File (EOF)** indica la fine del file
- **End Of Line** indica la fine di una linea
- …



Funzioni di libreria per

- **Input/Output a caratteri**
- **Input/Output a stringhe di caratteri**
- **Input/Output con formato**

I/O con formato

Nell'Input ed Output con formato occorre specificare il formato dei dati che si vogliono leggere oppure scrivere.

Il formato stabilisce:

- come interpretare la sequenza dei caratteri immessi dal dispositivo di ingresso (nel caso della **lettura**)
- con quale sequenza di caratteri rappresentare in uscita i valori da stampare (nel caso di **scrittura**)

Il formato viene indicato con opportune direttive del tipo:

```
%<direttiva>
```

Formati più comuni

		short	long
signed int	**%d**	**%hd**	**%ld**
unsigned int	**%u** (decimale)	**%hu**	**%lu**
	%o (ottale)	**%ho**	**%lo**
	%x (esadecimale)	**%hx**	**%lx**
float	**%e, %f, %g**		
double	**%le, %lf, %lg**		
carattere singolo	**%c**		
stringa di caratteri	**%s**		
puntatori (indirizzi)	**%p**		

Lettura con formato: `scanf`

La funzione **`scanf`** assegna i valori letti dal file standard di input alle variabili specificate come argomenti.

Sintassi

`scanf(<stringa-formato>, <sequenza-variabili>);`

Esempio:

```
int X;
float Y;
scanf("%d%f", &X, &Y);
```

La funzione **`scanf`**:

- legge una serie di valori in base alle specifiche contenute in <*stringa-formato*>: in questo caso `"%d%f"` indica che i due dati letti dallo standard input devono essere interpretati rispettivamente come un valore intero decimale (`%d`) ed uno reale (`%f`);
- memorizza i valori letti nelle variabili specificate come argomenti nella <*sequenza_variabili*> (`X`, `Y` nell'esempio);
- è una *funzione* che restituisce il numero di valori letti e memorizzati, oppure EOF in caso di *end of file*.

scanf

Separatori

Ogni direttiva di formato prevede dei separatori specifici:

Tipo di dato	Direttive di formato	Separatori
Intero	**%d**, **%x**, **%u**, etc.	Spazio, EOL, EOF.
Carattere	**%c**	Nessuno
Stringa	**%s**	Spazio, EOL, EOF

☞Se la stringa di formato contiene N direttive, è necessario che le variabili specificate nella sequenza siano esattamente N.

☞Gli identificatori delle variabili a cui assegnare i valori sono sempre preceduti dal simbolo & (infatti, le variabili devono essere specificate attraverso il loro indirizzo ➠ operatore & [v. puntatori e funzioni]).

☞La <stringa_formato> può contenere dei caratteri qualsiasi (che vengono scartati, durante la lettura), che rappresentano separatori aggiuntivi rispetto a quelli standard.

Esempio:

```
scanf("%d:%d:%d", &A, &B, &C);
```

=> richiede che i tre dati da leggere vengano immessi separati dal carattere ":".

Scrittura con formato: printf

Nel caso più generale, la funzione `printf` viene utilizzata per fornire in uscita il risultato di una espressione.

Sintassi

`printf(<stringa-formato>,<sequenza-elementi>)`

Esempio:

```
int X;
float Y;
printf("%d%f", X*X, Y);
```

- scrive sul dispositivo di standard output una serie di valori in base alle specifiche contenute in <*stringa-formato*>. In questo caso, un intero (`%d`), ed un reale (`%f`);
- i valori visualizzati sono i risultati delle espressioni che compaiono come argomenti nella <*sequenza_elementi*>;
- restituisce il numero di caratteri scritti.

printf

☞La stringa di formato della funzione `printf` può contenere sequenze costanti di caratteri da visualizzare

☞È possibile inserire nella stringa di formato anche **caratteri di controllo**:

newline	**\n**
tab	**\t**
backspace	**\b**
form feed	**\f**
carriage return	**\r**

☞Per la stampa del carattere '%' si usa: **%%**

Esempio:

```
char Nome = 'M';
char Cognome = 'P';
printf("%s\n%c. %c. \n\n%s\n",
   "Programma scritto da:",
   Nome, Cognome, "Fine");
```

Stampa:

```
Programma scritto da:
M. P.

Fine
```


Input/output

Se, all'interno di un programma C, si vogliono usare le funzioni della libreria **stdio** è **necessario** inserire all'inizio del file sorgente la linea:

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

☞**#include** è una direttiva per il **preprocessore** C:

- Nella fase precedente alla compilazione del programma ogni direttiva "`#...`" viene eseguita, provocando delle modifiche testuali al programma sorgente.
- Nel caso di **`#include <stdio.h>`** la direttiva stessa viene sostituita con il contenuto del file `stdio.h`.

Esempio:

```
#include <stdio.h>

main()
{
 int k;

 scanf("%d", &k);
 printf("Quadrato di %d: %d", k, k*k);
}
```

printf/scanf

Esempio:

```
scanf("%c%c%c%d%f", &c1, &c2, &c3, &i, &x);
```

Se in ingresso vengono dati:

```
ABC 3 7.345
```

Le variabili assumono i seguenti valori:

```
char c1  'A'
char c2  'B'
char c3  'C'
int i     3
float x   7.345
```


Esempio:

Stampa della codifica (decimale, ottale e esadecimale) di un carattere dato da input.

```
#include <stdio.h>

main()
{
 int a;
 printf("Dai un carattere e ottieni il
  valore decimale, ottale e hex");
 scanf("%c", &a);
 printf("\n%c vale %d in decimale,
  %o in ottale e %x in hex.\n",
  a, a, a, a);
}
```

✎ Esercizio

Calcolo dell'orario previsto di arrivo.
Scrivere un programma che legga tre interi positivi da terminale, rappresentanti l'orario di partenza (ore, minuti, secondi) di un vettore aereo, legga un terzo intero positivo rappresentante il tempo di volo in secondi e calcoli quindi l'orario di arrivo.

Approccio top-down

Nel caso di un problema complesso può essere utile adottare una metodologia ***top-down*:**

Si applica il principio di scomposizione, dividendo un problema in sotto-problemi e sfruttando l'ipotesi di conoscere le loro soluzioni per risolvere il problema di partenza:

a) dato un problema *P* ricavarne una soluzione attraverso la sua scomposizione in sottoproblemi più semplici;
b) eseguire a), b) per ogni sottoproblema individuato, che non sia risolvibile attraverso l'esecuzione di un sottoproblema elementare.

La soluzione finale si ottiene dopo una sequenza finita di passi di raffinamento relativi ai sottoproblemi via via ottenuti, fino ad ottenere sottoproblemi elementari.

Prima specifica:

```
main()
{
 /*dichiarazione dati */
 /* leggi i dati di ingresso */
 /*calcola l'orario di arrivo */
 /*stampa l'orario di arrivo */
}
```

Codifica:

Come dati occorrono tre variabili intere per l'orario di partenza ed una variabile intera per i secondi di volo.

```
/*definizione dati */
long unsigned int Ore, Minuti, Secondi,
TempoDiVolo;

/*leggi i dati di ingresso*/
/*leggi l'orario di partenza*/
printf("%s\n","Orario di partenza
(hh,mm,ss)?");
scanf("%ld%ld%ld\n", &Ore, &Minuti,
&Secondi);
/*leggi il tempo di volo*/
printf("%s\n","Tempo di volo (in
sec.)?");
scanf("%ld\n", &TempoDiVolo);

/*calcola l'orario di arrivo*/
Secondi = Secondi + TempoDiVolo;
Minuti = Minuti + Secondi / 60;
Secondi = Secondi % 60;
Ore = Ore + Minuti / 60;
Minuti = Minuti % 60;
Ore = Ore % 24;

/*stampa l'orario di arrivo*/
printf("%s\n","Arrivo previsto alle
(hh,mm,ss):");
printf("%ld%c%ld%c%ld\n", Ore, ':',
Minuti, ':', Secondi);
```

Versione finale:


```
#include <stdio.h>
main()
{
/*dichiarazione dati */
long unsigned int Ore, Minuti, Secondi,
TempoDiVolo;

/*leggi l'orario di partenza*/
printf("%s\n","Orario di partenza
(hh,mm,ss)?");
scanf("%ld%ld%ld", &Ore, &Minuti,
&Secondi);
/*leggi il tempo di volo*/
printf("%s\n","Tempo di volo (in
sec.)?");
scanf("%ld", &TempoDiVolo);

/*calcola l'orario di arrivo*/
Secondi = Secondi + TempoDiVolo;
Minuti = Minuti + Secondi / 60;
Secondi = Secondi % 60;
Ore = Ore + Minuti / 60;
Minuti = Minuti % 60;
Ore = Ore % 24;

/*stampa l'orario di arrivo*/
printf("%s\n","Arrivo previsto alle
(hh,mm,ss):");
printf("%ld%c%ld%c%ld\n", Ore, ':',
Minuti, ':', Secondi);
}
```

Dichiarazioni e definizioni

Nella parti dichiarative di un programma C possiamo incontrare:

- ***definizioni*** (di variabile, o di funzione)
- ***dichiarazioni*** (di tipo o di funzione)

Definizione

Descrive le proprietà dell'oggetto definito e ne determina l'esistenza.

Ad esempio, ***definizione*** di una variabile:

```
int V;
```

la variabile intera V viene creata (allocata in memoria).

Dichiarazione

Descrive soltanto delle proprietà di oggetti, che verranno (eventualmente) creati mediante definizione.

Ad esempio, ***dichiarazione*** di un tipo non primitivo.

Dichiarazione di tipo

- La dichiarazione di tipo serve per introdurre **tipi non primitivi**.
- Associa ad un **tipo di dato** non primitivo un **identificatore** (scelto arbitrariamente dal programmatore).
- Aumenta la leggibilità e la modificabilità del programma.
- In C per dichiarare un nuovo tipo, si utilizza la parola chiave `typedef`.
- In C sono possibili dichiarazioni di tipi scalari non primitivi:
  - tipi ***ridefiniti***
  - tipi ***enumerati***

Tipo ridefinito

Un nuovo identificatore di tipo viene associato ad un tipo già esistente (primitivo o non).

Sintassi

```
typedef <TipoEsistente> <NuovoTipo>;
```

Esempio:

```
typedef int MioIntero;
MioIntero X, Y, Z;
int W;
```


Tipo enumerato

Un **tipo enumerato** viene specificato attraverso l'esplicitazione di un **elenco di valori** che rappresenta il suo **dominio**.

Sintassi

```
typedef enum {a1, a2, a3, …, an} <EnumType>;
```

Una volta dichiarato, un tipo enumerato può essere utilizzato per definire variabili.

Esempio:

```
typedef enum{nord, sud, est, ovest} direzione;
direzione D=nord;
```

Proprietà

Dati di tipo enumerato sono **enumerabili**: il dominio è strettamente ordinato, in base all'ordine testuale con cui si indicano gli elementi del dominio nella dichiarazione.

Ad esempio:

```
typedef enum{nord, sud, est, ovest} direzione;
```

☞`nord` *precede* `sud`, `est` *segue* `nord` e `sud`, etc.

Tipo enumerato in C

☞In C il tipo enumerato equivale a una ridefinizione del tipo `int`.

Ogni elemento del dominio viene rappresentato come un **intero**, che viene utilizzato nella valutazione di espressioni, relazioni ed assegnamenti.

☞**`enum`** in C ha stessa occupazione, stesso range e **stesso utilizzo** di **`int`**.

Convenzione (default)

Il primo elemento del dominio viene rappresentato con il valore 0, il secondo con 1, etc.

Esempio:

```
typedef enum {lu,ma,me,gi,ve,sa,do}
Giorno;
/* lu=0, ma=1, me=2, ..., do=6*/
typedef enum {cuori, picche, quadri,
fiori} Carta;
/* cuori=0, picche=1, ... */

Carta C1, C2, C3, C4, C5;
Giorno G;

G = lu; /* G = 0*/
G = ma; /* G = 1 */
...
C1 = cuori; /* C1 = 0 */
...
```

☞L'utilizzo di tipi ottenuti per enumerazione rende più leggibile il codice.

☞Un identificatore di un valore scalare definito dall'utente (ad es., `lu`) deve comparire nella definizione di **un solo** tipo enumerato.

```
typedef enum {lu,ma,me,gi,ve,sa,do}
Giorno;
typedef enum{lu,ma,me} PrimiGiorni;
 /*scorretto */
```

Operatori sul tipo enumerato

Il tipo **enum** è un tipo **totalmente ordinato** (inoltre è un *Integral Type*).

Essendo i dati di tipo enumerato mappati su interi, su di essi sono disponibili gli stessi operatori visti per gli interi. In particolare:

Operatori relazionali

```
lu < ma          ➠ 0 < 1 ➠ vero
lu <= v          ➠ 0 < 5 ➠ vero
lu >= s          ➠ 0 > 6 ➠ falso
cuori <= quadri  ➠ 0 < 2 ➠ vero ...
```

☞Valori di tipo enumerato non sono stampabili né leggibili (non esiste, cioè, una direttiva di formato specifica per gli enumerati).

```
typedef enum {cuori, picche, quadri,
fiori} seme;
seme S;
 ...
 if(S == cuori)
  printf("%s","cuori");
```


Tipi enumerati in C

È possibile forzare il meccanismo di associazione automatico di interi a valori del dominio di tipi **enum**.

```
typedef enum {Nord=90, Sud, Est, Ovest}
direzione;
/* Nord=90, Sud=91, Est=92, Ovest=93 */
direzione D;
```

Attenzione!

```
d = 0;
```

viene accettato ed eseguito!

☞Non c'è controllo sugli estremi dell'intervallo di interi utilizzato!

Esempi di tipi enumerati

Vogliamo realizzare il tipo **`boolean`**: D = {*falso*, *vero*}

- È un tipo predefinito in altri linguaggi di programmazione (ad esempio, in Pascal).
- **Non è previsto in C**, dove, però:
  - il valore 0 (zero) indica ***FALSO***
  - ogni valore diverso da 0 indica ***VERO***

Prima soluzione:

Per definire le due costanti booleane:

```
#define FALSE 0
#define TRUE 1
```

☞**`#define`** è una direttiva del *preprocessore* C; provoca una sostituzione nel testo:

- dove c'è "FALSE" -> 0
- dove c'è "TRUE" -> 1

(non si alloca spazio in memoria)

Seconda soluzione:

Uso del tipo enumerato:

```
typedef enum {false, true} Boolean;
Boolean flag1,flag2;

 flag1 = true;
 if(flag1) ... /* flag vale 1 */

 flag2 = -37 /* !!! */
 if(flag2) ... /*funziona lo stesso!*/
```

Equivalenza tra tipi di dato

Quando due oggetti hanno lo stesso tipo? Dipende dalla realizzazione del linguaggio. Due possibilità:

- equivalenza **strutturale**
- equivalenza **nominale**

Equivalenza strutturale

Due dati sono considerati di tipo equivalente se hanno la stessa struttura.

Ad esempio:

```
typedef int MioIntero;
typedef int NuovoIntero;
MioIntero A;
NuovoIntero B;
```

A e B hanno lo stesso tipo.

Equivalenza nominale

Due dati sono considerati di tipo equivalente solo l'identificatore di tipo che compare nella loro definizione è lo stesso.

Ad esempio:

```
typedef int MioIntero;
typedef int NuovoIntero;
MioIntero A;
NuovoIntero B;
```

A e B vengono considerati di tipo diverso.

Equivalenza di tipo in C

Lo standard non stabilisce il tipo di equivalenza da adottare.

☞Per garantire la portabilità, è necessario sviluppare programmi che presuppongano l'equivalenza nominale.

Programmazione strutturata (Dijkstra, 1969)

La programmazione strutturata nasce come proposta per regolamentare e standardizzare le metodologie di programmazione.

Obiettivo:

- Rendere più facile la lettura dei programmi (e quindi la loro modifica e manutenzione).

Idea di base

La parte istruzioni (o parte esecutiva) di un programma viene vista come un comando (complesso, o *strutturato*) ottenuto componendo altri comandi elementari (assegnamento, chiamata di procedura) e non , mediante alcune regole di composizione (strutture di controllo).

Strutture di controllo:

- **concatenazione** (o composizione);
- **alternativa** (o istruzione condizionale);
- **ripetizione** (o iterazione).

☞Eliminazione dei salti incondizionati.

Programmazione strutturata

☞**Abolizione di salti incondizionati** (`goto`) nel flusso di controllo.

Vantaggi:

- **Leggibilità**.
- Supporto alla metodologia di progetto top-down: soluzione di problemi complessi attraverso scomposizione in sotto-problemi, a loro volta scomponibili in sotto problemi, etc:
  - la soluzione si ottiene componendo le soluzioni dei sottoproblemi attraverso concatenazione, alternativa e ripetizione.
- Supporto a metodologia **bottom-up**: la soluzione di problemi avviene aggregando componenti già disponibili mediante concatenazione, alternativa e ripetizione (programmazione per componenti).
- Facilità di **verifica** e **manutenzione**.

Teorema di Böhm e Jacopini

Le strutture di **concatenazione**, **iterazione** e **alternativa** costituiscono un insieme completo in grado di esprimere tutte le funzioni calcolabili.

☞L'uso di queste sole strutture di controllo non limita il potere espressivo.

Un linguaggio composto dalle istruzioni

- lettura, scrittura (`scanf`, `printf`)
- assegnamento
- istruzione composta (`{`...`}`)
- istruzione condizionale(`if`... `else`...)
- istruzione di iterazione (`while`...)

è un ***linguaggio completo*** (in grado di esprimere tutte le funzioni calcolabili).

Istruzioni strutturate in C

- istruzione composta: (o blocco) **`{ }`**
- alternativa: `if, switch`
- istruzioni di iterazione: `while, do, for`

Le istruzioni strutturate (o di controllo) sono alla base della **programmazione strutturata**.

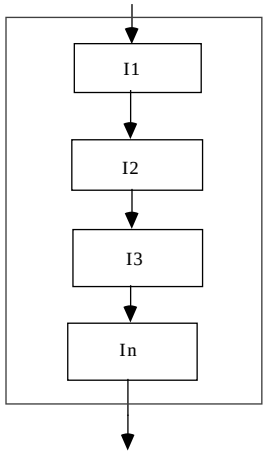
Istruzione composta { }

Determina l'esecuzione nell'ordine testuale delle istruzioni componenti.

Sintassi:

```
{
  <Dichiarazioni e Definizioni>
  <Sequenza di Istruzioni>
}
```

```
<sequenza-istruzioni> ::=
      <istruzione>; { <istruzione;> }
```



Sintatticamente equivalente a una singola istruzione (strutturata).

Istruzione composta

Dichiarazioni e definizioni:

È possibile definire variabili che hanno visibilità e tempo di vita limitato al blocco stesso.

Esempio:

```
...
{
 int A;
 A = 100;
 printf("Valore di A: %d\n", A);
}
...
```

Formalmente, il corpo del `main` è costituito da un'istruzione composta:

```
main()
{
 int A;
 A = 100;
 printf("Valore di A: %d\n", A);
}
```

Esempio di istruzione composta

```
/* programma che, letti due numeri a
terminale, ne stampa la somma */

#include <stdio.h>

main()
{
  int X, Y;

  scanf("%d%d", &X, &Y);
  printf("%d", X + Y);
}
```

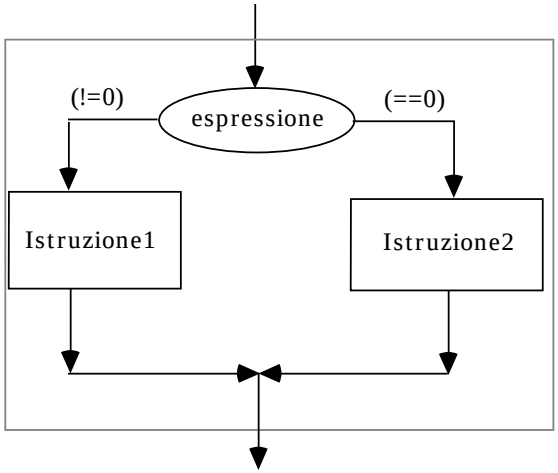

Istruzioni di alternativa

Istruzione `if`:

Seleziona l'esecuzione di una sola tra le istruzioni componenti in base al risultato di un'espressione logica (detta selettore).

```
if (<espressione>)
  <istruzione1>
[else
  <istruzione2>]
```

Se il risultato di <espressione> è *vero* (cioè è diverso da zero) viene eseguita <istruzione1>, altrimenti viene eseguita <istruzione2>.



Istruzione `if`

Esempio:

```
#include <stdio.h>
main()
{
 int A, B;
 scanf("%d%d", &A, &B);
 if(B==0)
  printf ("B vale zero!\n");
 else
  printf("Quoziente: %d\n", A/B);
}
```

La parte **`else`** dell'istruzione **`if`** è opzionale.

```
#include <stdio.h>
main()
{
 int A, B;
 scanf("%d%d", &A, &B);
 if(B==0) return;
  /* termina l'esecuzione del main */
 printf("Quoziente: %d\n", A/B);
}
```

Indentazione

L'"*indent*" delle linee del testo del programma rispetta l'annidamento delle varie istruzioni ➠ si aumenta la leggibilità del programma (modifica più facile).

Istruzione `if`

`if` (<espressione>)
** <istruzione1>**
[`else`
** <istruzione2>]**

☞<istruzione1> ed <istruzione2> possono essere di tipo **qualunque**, semplice o strutturato (ad es. istruzione composta, o `if`).

Esempio:

```
...
int Eta;

if (Eta >= 6)
{
 if (Eta <= 14)
 printf("%s", "scolare");
}
else
{
 printf("%s", "Non scolare");
 printf("%d", Eta);
}
...
```

Oppure:

```
int Eta;

if ((Eta >= 6) && (Eta <=14 ))
  printf("%s", "scolare");
else
{
  printf("%s", "non scolare");
  printf("%d", Eta);
}
```


if: esempi

Esempio 1:

Programma che legge due numeri e determina qual è il maggiore.

```
/* determina il maggiore tra due numeri
file c2.c */

#include <stdio.h>
main()
{
  int primo, secondo;

  scanf("%d%d", &primo, &secondo);
  if(primo > secondo)
    printf("%d", primo);
  else printf("%d", secondo);
}
```

Esempio 2:

```
/* Programma che calcola le radici di
un'equazione di secondo grado */

#include <stdio.h>
#include <math.h> /* lib. matematica */
main()
{
 float a, b, c;
 float d, x1, x2;

 scanf("%f%f%f", &a, &b, &c);
 if ((b*b) < (4*a*c))
  printf("%s", "radici complesse");
 else
 {
  d = sqrt(b*b-4*a*c);
  x1 = (-b+d)/(2*a);
  x2 = (-b-d)/(2*a);
  printf("%f%f", x1, x2);
 }
}
```


Esempio 3:

Risolvere un sistema lineare di due equazioni in due incognite

$$\mathbf{a_1x + b_1y = c_1}$$
$$\mathbf{a_2x + b_2y = c_2}$$

$x = (c_1b_2\text{-}c_2b_1) / (a_1b_2\text{-}a_2b_1) = XN / D$
$y = (a_1c_2\text{-}a_2c_1) / (a_1b_2\text{-}a_2b_1) = YN / D$

Soluzione:

```
#include <stdio.h>
main()
{
 float A1, B1, C1, A2, B2, C2;
 float XN, YN, D, X, Y;

 scanf("%f%f%f%\n", &A1, &B1, &C1);
 scanf("%f%f%f%\n", &A2, &B2, &C2);
 XN = (C1*B2 - C2*B1);
 D = (A1*B2 - A2*B1);
 YN = (A1*C2 - A2*C1);
 if(D == 0) {
  if(XN == 0)
   printf("sist. indeterm.\n");
  else
   printf("Nessuna soluz.\n");
 }
 else {
  X = XN/D;
  Y = YN/D;
  printf("%f%f\n", X, Y);
 }
}
```

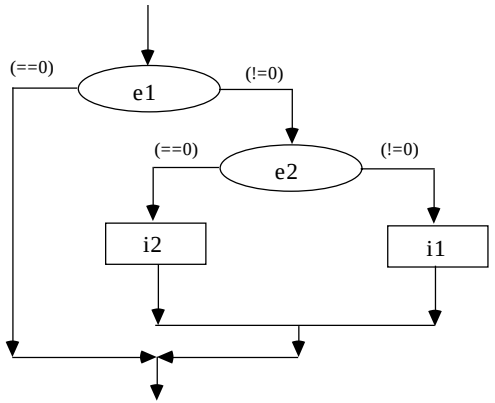
`if` annidati

```
if (<espressione>)
  <istruzione1>
[else
  <istruzione2>]
```

- <istruzione1> ed <istruzione2> possono essere ancora istruzioni `if`.

☞Si possono avere più **istruzioni `if` annidate**:

```
if(<e1>)
  if(<e2>)
    <i1>;
  else
    <i2>;
```



L'**`else`** si riferisce sempre all'**`if`** più "interno", se non specificato altrimenti.

```
if(n>0)
  if(a>b)
    n = a; /*n > 0 && a > b */
  else
    n = b; /* n > 0 && a <= b */
```

☞Se si vuole riferire l'**`else`** all'**`if`** più esterno, occorre utilizzare le parentesi graffe:

```
if(n>0)
  { if (a>b) n = a; } /* n>0 && a>b */
else
  n = b; /* n <= 0 */
```

✎ Esercizio

Dati tre numeri interi positivi che rappresentano i tre lati di un triangolo, si stampi il tipo del triangolo (equilatero, isoscele o scaleno).

Prima specifica:

```
main()
{
 /* dichiarazione dati */
 /* leggi le lunghezze dei lati */
 /* se triangolo, stampane il tipo */
}
```


Codifica:

Come dati occorrono tre variabili intere per rappresentare le lunghezze dei segmenti.

```
 /* dichiarazione dati */
unsigned int primo, secondo, terzo;

 /* leggi le lunghezze dei segmenti */
scanf("%d%d%d", &primo, &secondo,
&terzo);

 /* se triangolo, stampane il tipo */
if(primo+secondo>terzo)
 if(primo==secondo) { /*det. il tipo*/
  if(secondo==terzo)
   printf("equilatero");
  else
   printf("isoscele");
 }
 else {
  if(secondo==terzo)
   printf("isoscele");
  else {
   if(primo==terzo)
    printf("isoscele");
   else
    printf("scaleno");
  }
 }
```

Esempio: `if` annidati

Stampa di una variabile di tipo enumerato.

```
typedef enum {cuori, picche, quadri,
fiori} seme;

main()
{
 seme S;
...
 <assegnamento di un valore a S>
...
 if(S==cuori)
  printf("%s", "cuori");
 else
  if (S==picche)
   printf("%s", "picche");
  else
   if(S==quadri)
    printf("%s", "quadri");
   else
    if(S==fiori)
     printf("%s", "fiori");
}
```


Istruzione `switch`

Consente di selezionare l'esecuzione di uno (o più) tra gli N blocchi di istruzioni componenti, in base al valore di una espressione.

Sintassi:

```
switch (<EspressioneIntegralType>) {
  case <costante1>: <blocco1>; [break;]
  case <costante2>: <blocco2>; [break;]
  ...
  case <costanteN>: <bloccoN>; [break;]
  [default: <bloccoDiDefault>;]
}
```

- L'espressione è detta **selettore**. Deve restituire un valore di tipo IntegralType (enumerabile).
- Ogni costante associata a una "etichetta" **`case`** deve essere dello stesso tipo del selettore.
- Un valore può comparire al più in un'etichetta.

Istruzione `switch`

Significato

Se l'espressione restituisce un valore uguale ad una delle costanti indicate (per esempio <costante1>), si esegue il <blocco1> *e tutti i blocchi dei rami successivi*.

☞I blocchi non sono mutuamente esclusivi: possibilità di eseguire in sequenza più blocchi di istruzioni.

Per ottenere la mutua esclusione tra i blocchi:

☞ Necessità di **forzare l'uscita** mediante l'istruzione **`break`** (che provoca l'uscita forzata dallo `switch`).

Esempio:

```
int X;
switch (X%2)
{
 case 0: printf("X è pari"); break;
 case 1: printf("X è dispari"); break;
}
```

Ramo di *default*:

È possibile specificare un'etichetta **`default`**: essa viene eseguita per qualunque valore restituito dal selettore.
In pratica, consente di eseguire un blocco nel caso in cui il valore dell'espressione non corrisponde ad alcuna etichetta.

Esempio:

Calcolo della durata di un mese

Prima soluzione:

```
#define GENNAIO 1
#define FEBBRAIO 2
#define MARZO 3
...
#define NOVEMBRE 11
#define DICEMBRE 12
...
int mese, anno, giorni;
...
switch (mese)
{
 case GENNAIO: giorni = 31; break;
 case FEBBRAIO:
  if (<anno bisestile>) giorni = 29;
  else giorni = 28;
  break;
 case MARZO: giorni = 31; break;
 case APRILE: giorni = 30; break;
 case MAGGIO: giorni = 31; break;
 case GIUGNO: giorni = 30; break;
 case LUGLIO: giorni = 31; break;
 case AGOSTO: giorni = 31; break;
 case SETTEMBRE: giorni = 30; break;
 case OTTOBRE: giorni = 31; break;
 case NOVEMBRE: giorni = 31; break;
 case DICEMBRE: giorni = 31;
}
```

Seconda soluzione:

```
switch (mese)
{
 case FEBBRAIO:
   if (<bisestile>) giorni = 29;
   else giorni = 28;
   break;
 case APRILE: giorni = 30; break;
 case GIUGNO: giorni = 30; break;
 case SETTEMBRE: giorni = 30; break;
 case NOVEMBRE: giorni = 30; break;
 default: giorni = 31;
}
```

Terza soluzione:

```
switch (mese)
{
 case FEBBRAIO:
   if (<bisestile>) giorni = 29;
   else giorni = 28;
   break;
 case APRILE:
 case GIUGNO:
 case SETTEMBRE:
 case NOVEMBRE: giorni = 30; break;
 default: giorni = 31;
}
```


Esempio: una calcolatrice

Si vuole realizzare un programma che emuli il comportamento di una calcolatrice in grado di effettuare le operazioni aritmetiche su due operandi A e B.
Si supponga che l'utente immetta l'operazione nel formato:

A *op* B =

dove *op* può valere +, -, *, /, %.

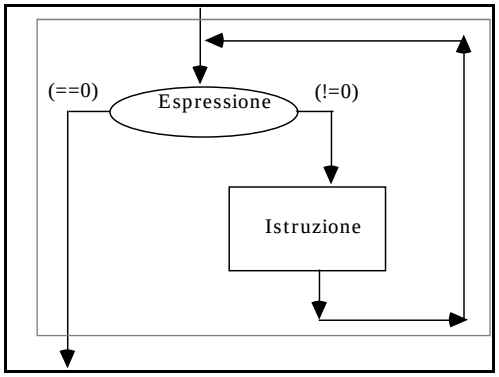
```
#include <stdio.h>

main()
{
  int A, B;
  char op;

  printf("Digita l'operazione che vuoi
effettuare [formato da usare A op B=]:\n");
  scanf("%d%c%d =", &A, &op, &B);
  switch (op)
  {
    case '+': printf("%d\n", A+B); break;
    case '-': printf("%d\n", A-B); break;
    case '*': printf("%d\n", A*B); break;
    case '%': printf("%d\n", A%B); break;
    case '/': printf("%d\n", A/B); break;
    case ':': printf("%f\n", (float)A/B);
break; /* divisione non intera */
    default: printf("\n operazione non
prevista\n");
  }
}
```

Istruzioni di iterazione: `while`

Consente la ripetizione dell'istruzione componente in modo controllato da una espressione.

Sintassi:

```
while (<espressione>)
      <istruzione>;
```



Significato

L'espressione (condizione di ripetizione) viene valutata all'**inizio di ogni ciclo**.

- L'<istruzione> viene eseguita finché il valore di <espressione> rimane *vero* (diverso da zero).
- Si esce dal ciclo quando l'espressione restituisce un valore =0 (*falso* logico).
- Se, inizialmente, <espressione> ha valore zero, il corpo del ciclo non viene **mai** eseguito.

Esempio 1:

Somma dei primi dieci interi

```
#include <stdio.h>

main()
{
  int somma, j;
  somma = 0;
  j = 1;
  while (j <= 10)
  {
    somma = somma + j;
    j++;
  }
  printf("Risultato: %d\n", somma);
}
```

Esempio 2:

Scarto dei blank in lettura

```
char car = ' ';
while (car == ' ')
{
  scanf("%c", &car);
}
```

✎ Esercizio

Scrivere un programma che calcoli la media degli N voti riportati da uno studente.

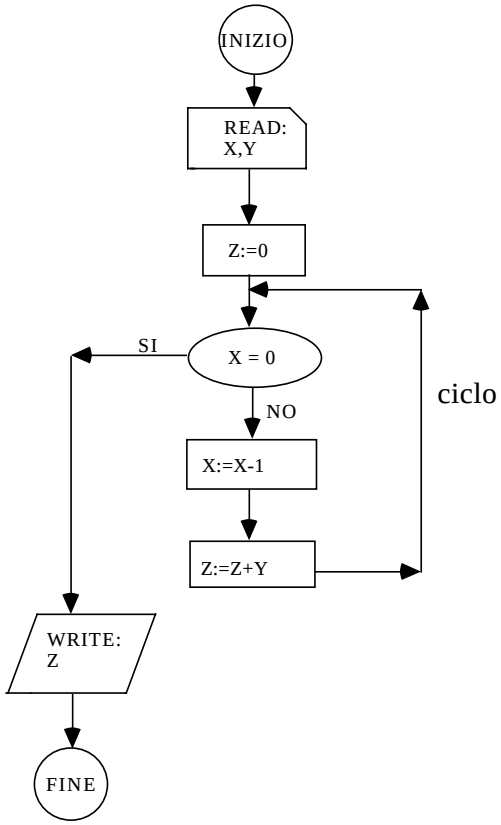
```
/* Media di n voti */
#include <stdio.h>
main()
{
 int voto, N;
 float media, sum;

 printf("Quanti sono i voti ?");
 scanf("%d", &N);
 sum = 0;
/* ripeti ... */
   printf("Dammi un voto:");
   scanf("%d", &voto);
   sum = sum + voto;
/* ... per N volte */
 media = sum / N;
 printf("Risultato: %f", media);
}
```


Codifica:

```
/* Media di n voti */
#include <stdio.h>
main()
{
 int voto, N, i=1;
 float media, sum;

 printf("Quanti sono i voti ?");
 scanf("%d", &N);
 sum = 0;
 while (i <= N)
 { /* corpo ciclo while */
  printf("Dammi il voto n. %d:", i);
  scanf("%d", &voto);
  sum = sum + voto;
  i++;
 }
 media = sum / N;
 printf("Risultato: %f",media);
}
```

Esercizio

Programma che calcola il prodotto X*Y come sequenza di somme (si supponga Y > 0, X >= 0).



Codifica:

```
/* moltiplicazione come sequenza di
somme */
#include <stdio.h>
main()
{
 int X, Y, Z;

 printf("Dammi i fattori:");
 scanf("%d%d", &X, &Y);
 Z = 0;
 while(X!=0)
 { /* corpo ciclo while */
  Z = Z + Y;
  X = X - 1;
 }
 printf("%d", Z);
}
```

Cicli innestati

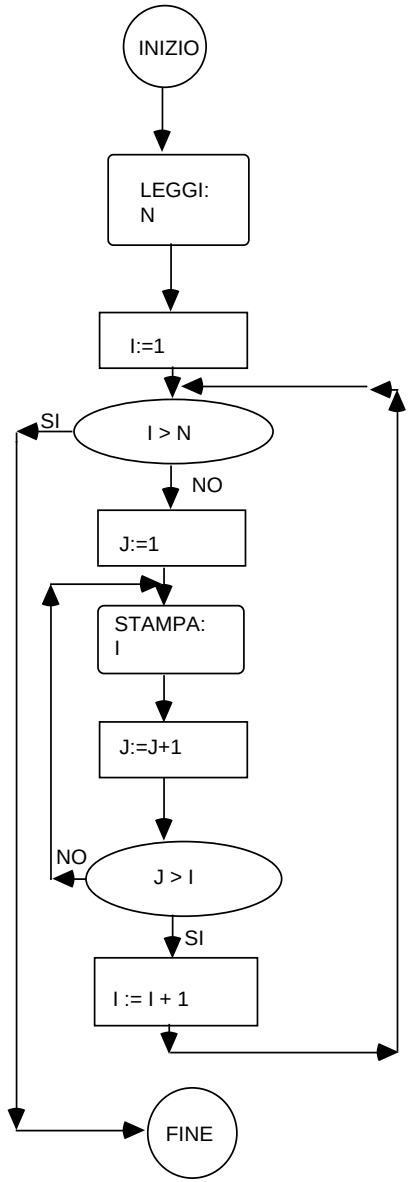
```
while (<espressione>)
      <istruzione>;
```

- <istruzione> può essere una qualunque singola istruzione (semplice o strutturata): eventualmente anche una istruzione `while` ➠ cicli **innestati**.

Esercizio:

Si legga un numero naturale N. Stampare una volta il numero 1, due volte il numero 2, …, N volte il numero N.

- Dominio di ingresso (0, 1, 2, …, N)
- Dominio di uscita ((), (1), (1, 2, 2), (1, 2, 2, 3, 3, 3), …).



Codifica:

```
#include <stdio.h>
main()
{
 int N, I, J;

 printf("dammi N:");
 scanf("%d", &N);
 I = 1;
 while(I<=N)
 { /* corpo ciclo esterno */
  printf("Prossimo valore: ");
  printf("%d", I);
  J = 1;
  while(J<I)
  { /* corpo ciclo interno */
   printf("%d", I);
   J = J + 1;
  }
  I = I + 1;
 }
}
```


✎ Esercizio

Stabilire se un numero naturale N è primo. Un numero naturale N è primo, se non è divisibile per alcun numero intero minore di esso (a parte 1).

☞Non è necessario eseguire tutte le divisioni di N per 2, 3, …, (N-1), ma basta eseguire le divisioni per i naturali minori o uguali alla radice quadrata di N.

```
/* Numero primo */
#include <stdio.h>
#include <math.h>
main()
{
 typedef enum {false,true} boolean;
 int N, I;
 float N1;
 boolean primo;

 scanf("%d", &N);
 N1=sqrt(N);
 I = 2;
 primo = true;
 while((I<=N1) && (primo==true)) {
  if(((N/I)*I)==N) primo = false;
  else I = I + 1;
 }
 if(primo==true)
  printf("numero primo");
 else printf("numero non primo");
}
```

Esercizio

Calcolo del **fattoriale** di un numero intero non negativo N:

- fattoriale(0) = 1
- fattoriale(N) = N * (N-1) * … * 1 = fattoriale(N-1) * N

```
/* Calcolo del fattoriale */
#include <stdio.h>
#include <math.h>
main()
{
 int N, F, I;

 printf("Dammi N:");
 scanf("%d",&N);
 F = 1;
 I = 2;
 while(I <= N)
 {
  F = F * I;
  I = I + 1;
 }
 printf("%s%d","Fattoriale: ", F);
}
```


Esercizio

Divisione tra numeri interi positivi attraverso somma e sottrazione.

```
/* divisione tra interi positivi */
#include <stdio.h>
main()
{
 unsigned int X, Y, Quoz, Resto;

 scanf("%u%u", &X, &Y);
 Resto = X;
 Quoz = 0;
 while(Resto >= Y)
 {
  Quoz = Quoz + 1;
  Resto = Resto - Y;
 }
 printf("%u%s%u%s%u%s%u", X,
  " diviso ", Y, " = ", Quoz,
  " resto ", Resto);
}
```

Istruzioni di iterazione: `do ... while`

Nell'istruzione **`while`**, la condizione di ripetizione viene verificata **all'inizio di ogni ciclo**

Istruzione `do`:

Consente di ripetere l'istruzione eseguendo il controllo a fine iterazione.

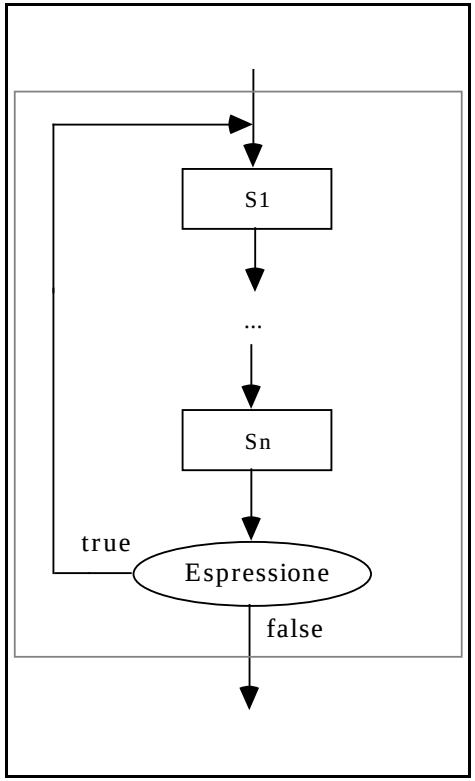
Sintassi:

```
do
    <istruzione>
while (<espressione>);
```

☞La condizione di ripetizione viene verificata **alla fine di ogni ciclo**.

☞La prima ripetizione viene **sempre** eseguita.

Istruzione `do`



Esempio:

```
do
  scanf("%c", &car)
while(car == ' ')
  /* salta spazi bianchi */
```

Prima dell'esecuzione del ciclo, il valore di `car` e` indeterminato.

Con il while:

```
car = ' ';
while (car == ' ')
  scanf("%c", &car);
```


Istruzioni ripetitive: `for`

È una istruzione di ripetizione particolarmente adatta per realizzare *cicli a contatore*.

Sintassi:

```
for(<espressione1>; <espressione2>; <espressione3>)
        <istruzione>;
```

Significato:

- <espressione1> è l'espressione di inizializzazione: viene eseguita una volta sola, prima di entrare nel ciclo;
- <espressione2> rappresenta la condizione di permanenza nel ciclo (valutata all'inizio di ogni iterazione);
- <espressione3> è l'espressione di passaggio al ciclo successivo (valutata alla fine di ogni iterazione).

`for` è equivalente a:

```
<espressione1>; /*inizializzazione*/
while (<espressione2>)
  /*condizione di ripetizione*/
{
  <istruzione>;
  <espressione3>; /*inizio nuovo ciclo*/
}
```

Esempio `while`/`for`

Con `while`:

```
somma = 0;
j = 1;
while (j <= n)
{
  somma = somma + j;
  j++;
}
```

Con `for`:

```
somma = 0;
for(j=1; j<=n; j++)
  somma = somma + j;
```


Cicli `for` particolari

`for`(<espressione1>; <espressione2>; <espressione3>)
<istruzione>;

- Ognuna delle espressioni **può essere omessa** (il punto e virgola deve rimanere).
- Se manca <espressione2>, si ha un ciclo infinito.

Cosa eseguono i seguenti **for** ?

```
for(i=1; i<=n; i++) printf("%d ", i);
for(;;) { ... }
```

Esempio:

Fattoriale con istruzione `for`.

```
/* Calcolo del fattoriale */
#include <stdio.h>
#include <math.h>
main()
{
 int N, F, I;

 printf("Dammi N:");
 scanf("%d", &N);
 F = 1;
 I = 2;
 for(I=2, F=1; I<=N; I++)
  F = F * I;
 printf("%s%d", "Fattoriale: ", F);
}
```


Istruzioni per il trasferimento del controllo

Istruzione `break`

L'istruzione **`break`** provoca l'uscita immediata dal **ciclo** (o dall'istruzione **`switch`**) in cui è racchiusa.

Istruzione `continue`

L'istruzione **`continue`** provoca l'inizio della successiva iterazione del **ciclo** in cui è racchiusa (non si applica a **`switch`**).

```
for(i = val1; i <= val2; i++)
{
 ...;
 if(...) continue;
 else ...
}
```

Esempio:

```
for(i=N; i>0; i--)
{
 /* salta alla prossima ripetizione se
N è multiplo di i */
 if(N%i) continue;
 /* esci dal ciclo se i vale 55 */
 if(i == 55) break;
 ...
}
```

Istruzione `goto`

`goto <label>`

L'istruzione **`goto`** provoca il salto all'istruzione etichettata dall'etichetta <label>.

Una **label** è un identificatore seguito da un due punti e può essere applicata ad una qualunque istruzione della medesima funzione in cui si trova il `goto`.

```
          ...
          goto errore;
          ...
errore:   ...
          ...
```

Formalmente l'istruzione **`goto`** non è necessaria.

☞Usare il **`goto`** solo nei rari casi in cui il suo uso rende più leggibile il codice. Ad esempio:

- per uscire dall'interno di strutture molto nidificate;
- per convergere in caso di errore, in un unico punto da punti diversi del programma.

Problemi se si entra in flusso innestato (procedure o blocchi)

✎ Esercizi

Leggere una sequenza di numeri interi a terminale e calcolarne la somma, nei seguenti casi:

1) La lunghezza della sequenza non è nota, ma l'ultimo valore è seguito da un intero negativo (while);

2) Come il caso 1, ma sicuramente la sequenza ha lunghezza maggiore o uguale ad uno (do);

3) La sequenza di numeri da sommare è preceduta da un numero intero che rappresenta la lunghezza della sequenza (for);

Espressioni (complementi)

Operatori sui bit

<<	shift a sinistra	k<<4 shift a sinistra di 4 bit equivale a k*16
>>	shift a destra	k>>4 shift a destra di 4 bit equivale a k/16
&	and bit a bit	
|	or inclusivo bit a bit	
^	or esclusivo bit a bit	
~	Complemento a 1	

Operatore condizionale

? :=

<condizione> ? <parteTrue> : <parteFalse>

- La <**parteTrue**> viene valutata solo se la condizione è verificata (valore diverso da 0).
- La <**parteFalse**> viene valutata solo se la condizione **non** è verificata (valore uguale a zero)

```
x = (y != 0 ? 1/y : INFINITY);
k=(a<b)? a : b; /* assegna il minore */
```


✎ Esercizi

`i = (j = k + 1)`

`i = (k = j + 12) - ( j = k + 56 - j)`

`(k & 1)`

`i >>= 1` e `j <<= 1`

`1 + ~ i` (~ è il complemento a 1 bit a bit)

`~ ( ~ i | ~ j )` vs. `i & j`

`! ( ! a || ! b )` vs. `a && b`

`a && b` vs. `a ? b : 0`

`a || b` vs. `a ? 1 : b`

`! a` vs. `a ? 0 : a`

`(x & MASK) == 0` vs. `x & MASK == 0`