Linguaggi di programmazione: Il linguaggio C

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

La programmazione e` basata sul concetto di **astrazione**:

> *L'astrazione e` il processo con cui si descrive un fenomeno attraverso un sottoinsieme delle sue proprieta`.*

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

Potere espressivo di un linguaggio:

E` la capacita` di fornire costrutti di astrazione il piu` possibile simili ai concetti utilizzati nella descrizione del metodo risolutivo.

☞ quanto piu` il potere espressivo di un linguaggio e` elevato, tanto piu` il programmatore e` 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, cioe` l'insieme di istruzioni previste per esprimere il **controllo** del flusso di esecuzione.

Programma = Dati + Controllo

ESEMPIO:

Calcolo radici equazioni di secondo grado

$Ax^2+Bx+C=0$

X è il DATO di output da trovare (x1 e x2)

A, B, C sono i dati di input

SE $b^2-4ac>0$ ALLORA
$X1=(-b+sqrt(b^2-4ac)/2a$
$X2=(-b+sqrt(b^2-4ac)/2a$
ALTRIMENTI
Soluzione Impossibile

E' l'algoritmo di calcolo che bisogna eseguire

Il potere espressivo di un linguaggio si misura da quanto è semplice, naturale, esprimere i dati e gli algoritmi

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. Kernigham e D. M. Ritchie)

Nel 1983 e` 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 delle memoria, accesso alla rappresentazione)

- 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**, cioe` non possono essere utilizzate come identificatori (ad esempio di variabili)

Commenti:

Sono sequenze di caratteri ignorate dal compilatore.
Vanno racchiuse tra **/*** ... ***/:**

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

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

I commenti aumentano la leggibilita' di un programma

Costanti

Caratteri:

Insieme dei caratteri disponibili (e` dipendente dalla implementazione). In genere, ASCII esteso (256 caratteri). Si indicano tra singoli apici:

'a' **'A'**

Caratteri speciali:

newline	**\n**
carattere nullo	**\0**
tab	**\t**
backspace	**\b**
form feed	**\f**
carriage return	**\r**
'	\'
\	\\
"	\"

Stringhe:

In generale, nell'informatica, una stringa è una sequenza di caratteri.

In C, sequenze di caratteri racchiusa tra doppi apici " ".

"a" **"aaa"** **""** (stringa vuota)

Esempi:

codice:

```
printf("Prima riga\nSeconda riga\n");
printf("\t ciao\"");
```

output:

```
Prima riga
Seconda riga
        ciao"
```

Numeri interi

Rappresentano numeri relativi (quindi con segno):

	2 byte	**4 byte**
base decimale	12	70000, 12**L**

Si possono rappresentare anche in forma binaria, ottale e esadecimale

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)

Identificatori

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

- Deve iniziare con una lettera (o con il carattere '_'), a cui possono seguire lettere e cifre in numero qualunque (dipende dalla realizzazione)

- distinzione tra maiuscole e minuscole (linguaggio **case-sensitive**)

Esempi:

Alfa beta corretti
Gamma1 Gamma2

3X int non sono corretti

In ciascuna realizzazione esistono identificatori standard e non-standard predefiniti (**parole chiave**).

☞ prima di essere usato, un identificatore deve essere gia` stato definito in una parte di testo precedente.

Struttura di un programma C

Nel caso piu` semplice, un **programma C** consiste in:

<parte definizioni globali>
<parte del programma principale (main)>
<eventuali altre funzioni>

Per ora vediamo solo la parte main (programma principale)

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

```
main()
{     <parte definizioni >  → dati!
      <parte istruzioni>  → algoritmi!
}
```

☞ il **<main>** e` costituito da due parti:

- Una **parte di definizioni** (variabili, tipi, costanti, 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 effettua la somma e la stampa */

#include <stdio.h>

main()
{
/* p. di definizioni */
int X,Y,Z;

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

Z=X+Y;

printf("%d",Z);
}
```

Dati + Controllo

Dichiarazione dei dati:

I dati manipolati da un programma possono essere classificati in:

- **costanti**, dati che non cambiano di valore durante l'esecuzione
- **variabili**, dati che cambiano di valore durante l'esecuzione

Controllo:

Istruzione di assegnamento: (variabile come astrazione della cella di memoria)

<identificatore-variabile> **=** <espressione>;

Istruzioni di lettura e scrittura:

scanf(<stringa-controllo>,<sequenza-elementi>);

printf(<stringa-controllo>,<sequenza-elementi>);

Istruzioni di controllo:

- **istruzione composta:** (blocco)

  { <sequenza-istruzioni> **}**

- **istruzione condizionale:** (selezione)

  if <espressione> <istruzione>
  else <istruzione>;

- **istruzione di iterazione:** (ciclo)

  while <espressione> <istruzione>;

Sono in realta` molte di piu`.

ESEMPIO: le equazioni di secondo grado

```
#include<stdio.h>
/* serve quando si vuole fare dell'input-
output, si DEVE DICHIARARE che si farà uso
di istruzioni di input-output */

main()  /* inizia programma principale */

/* dichiarazione dei dati */
float a, b, c; /* variabili di input */
float x1, x2; /* variabili interne */
float det;  /* varibile di uso interno */

/* fase di input */

scanf("%f %f %f", a, b, c);
/* prendiamo dall'input le tre variabili */

/* calcolo determinante */

det = b*b - 4*a*c;
```

```
/* guardiamo com'e' il determinante */

if (det < 0)
{
    printf("Soluzioni immaginarie");
}
else  /* se det > 0 */
{
  x1 = (-b + det)/2*a;
  x2 = (-b - det)/2*a;
  printf("Soluzioni x1=%f x2=%f \n", x1,x2);
}
}
```

ESEMPIO: somma dei primi N numeri naturali

```
#include<stdio.h>


main()  /* inizia programma principale */

/* dichiarazione dei dati */
int N; /* variabile di input */
int somma; /* variabile di output */
float conta;  /* varibile di uso interno */

/* fase di input */
printf("Fino a dove vuoi sommare?\n");*/
scanf("%d", N);

conta = 0;
/* questa variabile ci serve per contare da
1 ad N e ripetere un certo numero di volta
la somma */

somma = 1;
/* facciamo partire la somma da zero e gli
aggiungiamo il dovuto di volta in volta */
```

```
/* ripetiamo la somma un certo numero di
volte */
while (conta <= N)
{
    somma = somma + conta;
    conta = conta + 1;
}

  printf("Somma %d \n", N);

}
```

Definizione di variabili

Una **variabile** rappresenta un dato che puo` cambiare il proprio valore durante l'esecuzione.

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

Esempi:

```
int     A, B, SUM;      /* dati interi */
float   root, Root;     /* dati reali */
char    C , ch;         /* dati carattere */
```

☞ La definizione di una variabile provoca come effetto l'allocazione in memoria della variabile specificata.
Ogni istruzione successiva alla definizione di una variabile A, potra` utilizzare A

Variabile

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

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

Assegnamento:

<identificatore-variabile> **=** <espressione>

Esempi:

```
int a;
float pi;
char ch;
...
a=100;
pi=3.1457;
ch='p';
```

Variabile

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

- **tipo**, definisce l'insieme dei valori che la variabile puo` assumere e degli operatori applicabili.

- **valore**, e` rappresentato (secondo la codifica adottata) nell'area di memoria legata alla variabile;

- **campo d'azione** (scope), e` l'insieme di istruzioni del programma in cui la variabile e` nota e puo` essere manipolata;
  C, Pascal, determinabile staticamente
  LISP, dinamicamente

- **tempo di vita** (o durata o estensione), e` l'intervallo di tempo in cui un'area di memoria e` legata alla variabile;
  FORTRAN, allocazione statica
  C, Pascal, allocazione dinamica

Esempio:

```
#include <stdio.h>
main()
{
/* programma che letto un numero a
terminale stampa il valore della
circonferenza del cerchio con quel raggio
*/
float X, Y; /* variabili locali */

   scanf("%f",&X);
   Y = 2*3.14*X;
   printf("%f",Y);
}
```

Definizione di costanti

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

La dichiarazione di una costante associa ad un identificatore (**nome** della costante) un **valore** (numero o altro identificatore di costante).

Esempi:

```
const   float    pigreco = 3.14;
const   float    pigreco = 3.1415926;  e = 2.7182;
                 menoe = - e;
```

☞ prima di essere usato, un identificatore deve essere gia` stato definito (ad es., e per definire menoe).

Si aumenta la leggibilita` e modificabilita` dei programmi.

Altra soluzione:

☞ **#define** e` una direttiva del precompilatore C: provoca una sostituzione nel testo:

```
#define pigreco 3.14
#define e 2.7182
#define menoe -e
```

(non si alloca spazio in memoria)

Esempio:

```
#include <stdio.h>
main()
{
/* programma che, letto un numero a
terminale stampa il valore della
circonferenza del cerchio con quel raggio
*/

const float pigreco = 3.1415926;
float X, Y;

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

Oppure:

```
#include <stdio.h>
#define pigreco 3.1415926

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

float X, Y;

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

Tipo di dato

Un **tipo di dato** T e` 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 e` definito:
- dall'insieme di valori che variabili di tipo T potranno assumere,
- dall'insieme di operazioni che possono essere applicate ad operandi del tipo T.

Esempio:

Consideriamo i numeri naturali

naturali = [N, {+, *, =, >, < }]

3*4 ---> 12
3>=4 ---> falso

Rappresentazione per un tipo di dato:

descrizione del tipo di dato attraverso le strutture linguistiche fornite dal linguaggio di programmazione scelto.

Il concetto di Tipo

Proprieta`:

- ☞ Ciascun dato (variabile o costante) del programma deve appartenere ad **un solo** tipo.

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

Vantaggi:

Se un linguaggio di programmmazione e` *tipato:*

- ☞ **Astrazione**:L'utente esprime i dati ad un livello di astrazione piu` alto della loro organizzazione fisica. L'insieme di bit che rappresenta un valore di un certo tipo non e` accessibile. Maggior portabilita`.

- ☞ **Protezione:** Il linguaggio protegge l'utente da combinazioni errate di dati ed operatori (**controllo statico** sull'uso di variabili, etc. in fase di compilazione).

- ☞ **Portabilita`:** l'indipendenza dall'architettura rende possibile la compilazione dello stesso programma su macchine profondamente diverse.

Tipo di Dato in C

In C si possono utilizzare:

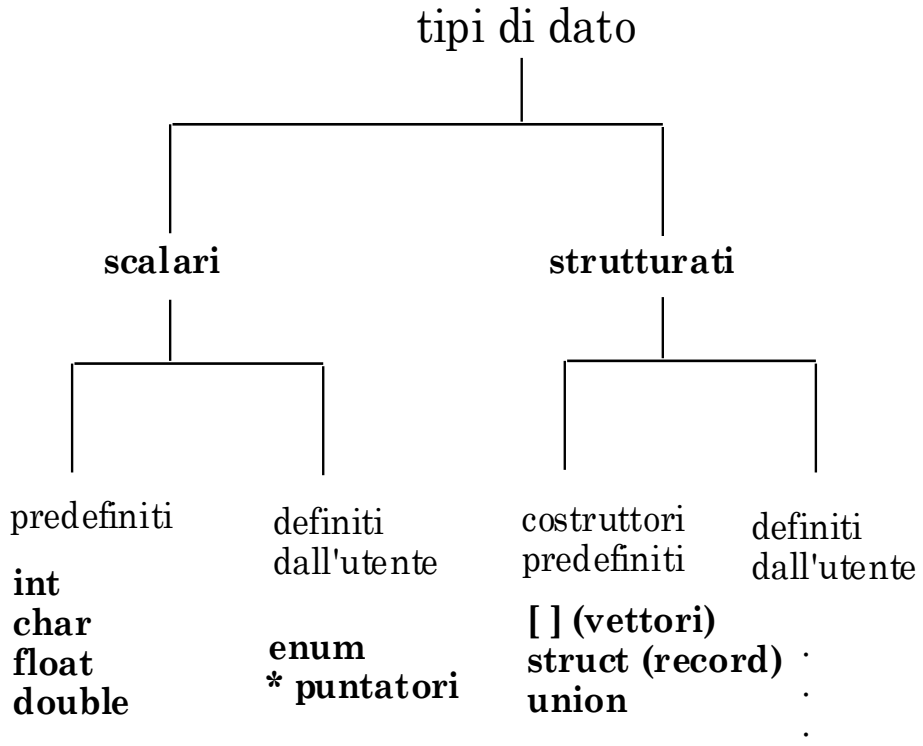
- tipi ***primitivi***: sono tipi di dato previsti dal linguaggio (built-in) e quindi rappresentabili direttamente.
- tipi ***non primitivi***: sono tipi ***definibili dall'utente*** **(**mediante appositi costruttori di tipo, v. *typedef*).

Inoltre si distingue tra:

- tipi ***scalari***, il cui dominio e` costituito da elementi *atomici*, cioe` logicamente non scomponibili.

- tipi ***strutturati,*** il cui dominio e` 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)

Qualificatori e Quantificatori

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

- 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**)

```
int X;
long int Y;
```

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

```
int A;
unsigned int B;
```

Il tipo int

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

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

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

Ad esempio:

se il numero n di bit a disposizione per la rappresentazione di un intero e` 16, allora il dominio rappresentabile e` composto di:

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

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.
  Quindi l'intervallo rappresentabile e`:

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

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

$$[0, (2^n - 1)]$$

Operatori sugli interi

Al tipo **int** (e 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"*** (cioe`, il cui valore puo` 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*

Esempio:

```
/*programma che, letti due numeri a terminale, effettua la stampa del maggiore dei due */

#include <stdio.h>

main()
{
/* p. di definizioni */
int X,Y;

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

if ( X > Y )
     printf("%d",X);
else
     printf("%d",Y);
```

}

Booleani

Sono dati il cui dominio e` 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 vengano rappresentati attraverso interi:

- Ad esempio, l'espressione A == B restituisce:
  - **0**, se la relazione non e` vera
  - **1** , se la relazione e` vera

Operatori logici:

si applicano ad operandi di tipo **int** e producono risultati *booleani*, cioe` 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 e`:

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

Definizione 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*

☞ **In C, gli operandi di operatori logici sono di tipo int:**

- se il valore di un operando e` **diverso da zero**, viene interpretato come ***vero***.
- se il valore di un operando e` **uguale a zero**, viene interpretato come ***falso***.

Quindi, gli 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

Esempio:

```
/*programma che, letti tre numeri a terminale, stampa se il terzo e' compreso o meno nell'intervallo tra i primi due*/

#include <stdio.h>

main()
{
/* p. di definizioni */
int X,Y,Z;

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

if ( X < Z && Z < Y )
  printf("compreso %d tra %d e %d",Z,X,Y);
else
 printf("non comp %d tra %d e %d",Z,X,Y);
}
```

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

Overloading:

Il C (come 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).

I tipi float e double (reali)

Rappresentano l'insieme dei numeri reali.
In realta`, sono un'approssimazione dei reali, sia come **precisione** che come **intervallo** di valori.

Lo spazio allocato (e quindi l'insieme dei valori rappresentabili) dipende dalla rappresentazione adottata.

Uso del quantificatore long:

si puo` applicare a **double**, per aumentare la precisione:

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

Esempio:

Rappresentazione dei reali in C per PC:

- ☞ 4 byte per **float** e 8 per **double** (10 byte per **long double**).

Tipo:	Precisione	Valori
float	6 cifre dec.	3.4×10^{-38} .. $3.4\times10^{+38}$
double	15 cifre dec.	1.7×10^{-308} .. $1.7\times10^{+308}$

Operatori sui Reali

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.

☞ concettualmente, producono un risultato booleano; in pratica, il risultato e`:

0, se la relazione e` falsa

1, se la relazione e` vera

Esempi:

5.0 / 2	›	2.5
2.1 / 2	›	1.05
7.1 < 4.55	›	1
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

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

Per carattere si intende 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.).

Ogni valore viene specificato tra singoli apici.

Ad esempio:

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

Come per i numeri, anche il dominio associato al tipo **char** e` **ordinato**.
L'ordine dipende dal codice utilizzato nella macchina.

Di solito, codice ASCII esteso.

Operatori sui caratteri

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

- ogni carattere viene mappato sull'intero che rappresenta il codice nella tabella dei set di caratteri.

- sui dati **char** e` possibile eseguire tutte le operazioni previste per gli interi. Ogni operazione, infatti, e` 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)

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 e` *vero*)

' " ' + ' # ' › **'C'** (*codice*(")+*codice*(#)=67)

! 'A' › 0 (*codice*(A) e` diverso da zero)

'A' && 'a' › 1

Uso dei qualificatori:

In C il tipo **char** e` visto come dominio di **valori interi**:

Tipo:	Dimensione	Valori
char	1 byte	-128 .. 127
unsigned char	1 byte	0 .. 255

signed char ➔ char

Esempio:

```
/*programma che, letto un carattere a
terminale, stampa se e' compreso tra
altri due caratteri dati come costanti */

#include <stdio.h>

main()
{
/* p. di definizioni */
const char c2='d', c3='h';
char c1;

/*p. istruzioni*/
scanf("%c",&c1);

if ( c2 < c1 && c1 < c3 )
  printf("%c tra %c e %c",c1,c2,c3);
else
 printf("%c non tra %c e %c",c1,c2,c3);
 }
```

Inizializzazione delle variabili

E` possibile specificare un valore iniziale di un oggetto in fase di definizione.

- Variabili scalari, vettori, strutture.

I tipi scalari vengono inizializzati con una sola espressione.

Esempio:

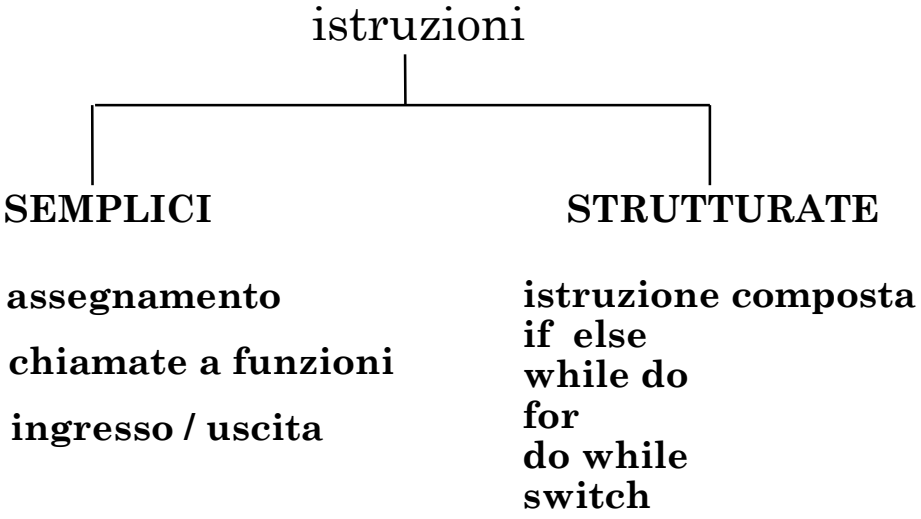
```
int x = 10;
char y = 'a';
double r = 3.14*2;
```

☞ Differisce dalla definizione di costanti, perche` 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). Di questa categoria fanno parte le istruzioni per il controllo del flusso di esecuzione.



Assegnamento

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

Esempio:

```
int A, B;

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

Compatibilita` di tipo ed assegnamento:

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

Esempio:

```
int x,y=35;
char C='z',d;
double r=3.3489,s;

x=y;
d=C;
s=r;
x=r;   /*  conversione  implicita  con
troncamento */
```

Espressioni

Un'espressione e` una regola per il calcolo di un valore. E` una combinazione di operandi tramite operatori.

3 * (17 - 193) % 19

(3 < 7) && (4 <= 5) (X > 10)

☞ A ciascuna variabile che compare in una espressione deve essere gia` stato attribuito un valore: l'espressione utilizza il valore corrente della variabile.

Una espressione C puo` essere un'espressione *aritmetica*, *relazionale, logica, di assegnamento o sequenziale*.

Espressioni Aritmetiche:

A seconda del tipo degli operandi, restituiscono valore intero oppure reale.

- **Espressioni aritmetiche di incremento e decremento:** Si applicano ad operandi di tipo intero (o reale, o carattere) e producono un risultato dello stesso tipo.

```
int A=10;

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

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

Ad esempio:

```
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 */

j = i + k++; /*equivale a:
                  j=i+k; k=k+1;*/
```

☞ Si possono scrivere espressioni il cui valore e` difficile da predire:

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

Espressioni relazionali:

Restituiscono un valore *vero* (in C, uguale a 1) o *falso* (in C, uguale a 0). Sono ottenute applicando gli operatori relazionali ad operandi compatibili.

```
(3 < 7)
x <= y
```

Espressioni logiche:

Sono ottenute mediante gli operatori di:

- complemento (not) !
- congiunzione (and) &&
- disgiunzione (or) ||

Espressione di assegnamento:

☞ Il simbolo = **e` un operatore**
l'istruzione di assegnamento e` una espressione che ritorna un valore:

- il valore ritornato e` quello assegnato alla variabile a sinistra del simbolo =
- il tipo del valore ritornato e` lo stesso tipo della variabile oggetto dell'assegnamento

Ad esempio:

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

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

Valutazione di Espressioni

Nelle espressioni, gli operatori sono valutati secondo una precedenza prestabilita (altrimenti parentesi tonde), e seguendo opportune regole di associativita`.

Regole di Precedenza e Associativita` degli Operatori C:

Operatore	Associativita`
`() [] ->`	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 sinistra a destra
`&`	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

Precedenza ed Associativita` 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

Espressioni

- E` possibile forzare le regole di precedenza mediante l'uso delle parentesi.

- In caso di pari priorita`, l'ordine di valutazione e` spesso da sinistra a destra (quello testuale) ma non sempre.

3*5 % 2 ❯ equivale a: (3*5) % 2
3 < 0 && 3 < 10 ❯ 0 && 1
3<(0 && 3) < 10 ❯ 3 < 0 < 10

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:

(3 < 0) && (X > Y) ❯ solo primo operando
falso && ….(ininfluente)

☞ Bisognerebbe evitare di scrivere espressioni che dipendono dal metodo di valutazione usato ❯ scarsa portabilita` (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:

A<=20 || A>=40	›	vero	{A<20, short cut}
! (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	

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.

In ogni versione ANSI C, esiste una *Libreria Standard* (**stdio**) che mette a disposizione alcune funzioni (dette *funzioni di libreria*) per effettuare l'input e l'output.

Le dichiarazioni delle funzioni messe a disposizione da tale libreria devono essere essere incluse nel programma: **#include** <stdio.h>

- ☞ **#include** e` una direttiva per il **preprocessore** C: nella fase precedente alla compilazione del programma ogni direttiva "#..." provoca delle modifiche testuali al programma sorgente. Nel caso di **`#include <nomefile>`** viene sostituita l'istruzione stessa con il contenuto del file specificato.

Dispositivi standard di input e di output:

per ogni macchina, sono periferiche predefinite (generalmente tastiera e video).

Input/Output

Il C vede le informazioni lette/scritte da/verso i dispositivi standard di I/O come file *sequenziali*, cioe` **sequenze di caratteri** (o stream).

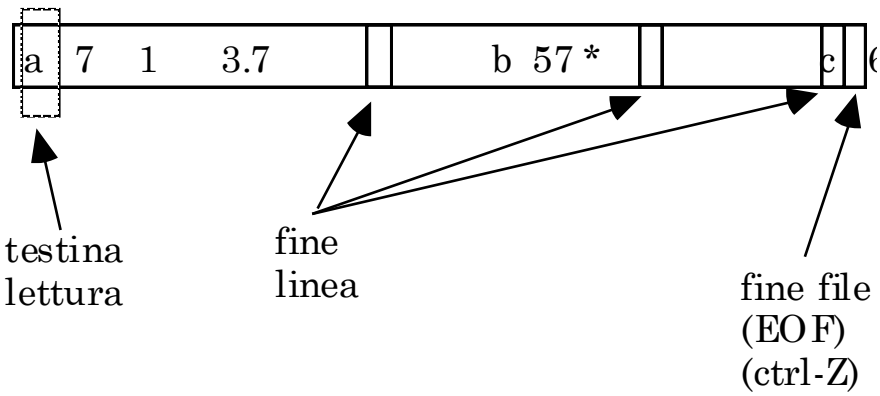
Gli *stream* di input/output possono contenere dei caratteri di controllo:

- End Of File (EOF)
- End Of Line

...

Sono Disponibili funzioni di libreria per:

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



I/O con formato

Nell'I/O con formato occorre specificare il formato (***tipo***) dei dati che si vogliono leggere oppure stampare.

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**)

scanf

Lettura: `scanf`

E` una particolare forma di assegnamento: la scanf assegna i valori letti alle variabili specificate come argomenti (nell'ordine di lettura).

scanf**(<stringa-formato>, <sequenza-variabili>)**

Ad esempio:

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

- legge una serie di valori in base alle specifiche contenute in <*stringa-formato*>: in questo caso "%d%f" indica che devo leggere due valori, uno decimale (%d) ed uno reale (%f)
- memorizza i valori letti nelle variabili (X, Y nell'esempio)
- e` una funzione che restituisce il numero di valori letti e memorizzati, oppure EOF in caso di *end of file*

☞ Gli identificatori delle variabili a cui assegnare i valori sono (quasi) sempre preceduti dal simbolo &.

Ad esempio:

scanf(“%d%d%d”, &A, &B, &C);

=> richiede tre dati interi da leggere

Scrittura: `printf`

La printf viene utilizzata per fornire in uscita il valore di una variabile, o, piu` in generale, il risultato di una espressione.
Anche in scrittura e` necessario specificare (mediante una *stringa di formato*) il formato dei dati che si vogliono stampare.

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

Ad esempio:

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

- scrive 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
- restituisce il numero di caratteri scritti.
- la stringa di formato della printf puo` contenere sequenze costanti di caratteri da visualizzare, ad esempio il carattere di controllo \n (newline).

Esempio:

```
main()
{
int  k;
scanf("%d",&k);
printf("Quadrato di %d: %d\n",k,k*k);
}
```

Input/Output

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**

Caratteri di controllo:

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

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

Esempi:

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

Se in ingresso vengono dati:

```
ABC 3 7.345
```

le variabili

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

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

› viene stampato:

```
Programma scritto da:
A. C.

Fine
```

Esempio:

```
main()
{
  float  x;
  int       ret, i;
  char name[50];
printf("Inserisci un numero decimale,
");
printf("un floating ed una stringa con
meno ");
printf("di  50 caratteri e senza
bianchi");
ret = scanf("%d%f%s", &i, &x, name);
printf("%d valori letti %d %f %s", ret,
i, x, name);
}
```

✏ **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 puo` essere utile adottare una metodologia ***top-down*:**

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

a) dato un problema P ricavarne una soluzione attraverso la sua scomposizione in sottoproblemi piu` semplici

b) eseguire a) 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. Poiche` queste variabili assumono valori tra 0 e 60, possiamo dichiararle di tipo **unsigned int**:

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

```
/*leggi i dati di ingresso*/:
    /*leggi l'orario di partenza*/
    printf("Orario di Partenza (hh:mm:ss)?\n"");

    scanf("%d:%d:%d\n", &Ore, &Minuti, &Secondi);
    /*leggi il tempo di volo*/
    printf("Tempo di volo (in sec.)?\n");
    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("Arrivo previsto alle (hh,mm,ss):");
    printf("%d:%d:%d\n",
                Ore,Minuti, Secondi);
```

Per Ore, Minuti, Secondi si puo` oltrepassare l'estremo superiore dell'intervallo.

```
main()
{
/*dichiarazione dati */
unsigned int        Ore, Minuti, Secondi;
long unsigned int   TempoDiVolo;

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

Dichiarazione di tipo

Associa ad un **identificatore** (arbitrario) un **tipo di dato**. Aumenta la leggibilita` e modificabilita` del programma.

La dichiarazione di tipo serve per introdurre **tipi non primitivi.**

Per dichiarare un nuovo tipo, si utilizza la parola chiave **typedef.**

Tipi scalari non primitivi:

In C sono possibili dichiarazioni di tipi scalari:

- ***ridefiniti***
- ***enumerati***

Tipo ridefinito:

Un nuovo identificatore di tipo viene dichiarato uguale ad un tipo gia` esistente:

typedef **TipoEsistente NuovoTipo;**

Esempio:

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

Istruzioni di controllo

- istruzione composta: (blocco)
- istruzioni condizionali: (selezione)
- istruzioni di iterazione: (ciclo)

Sono alla base della **programmazione strutturata**.