LINGUAGGI DI PROGRAMMAZIONE - La torre di babele

The limits of my language mean the limits of my world

Wittgenstein

1954/57 FORmula TRANslator (Backus-IBM),
1959 LISP,
1960 Algol, COmmon Business Oriented Language,
1962 BASIC
1965/75 *Pascal (didattica), PL/1, C(software di base), Prolog,*
…
1995 Java
1996 … Python

Non esiste il linguaggio migliore

- **high** vs. low level
- **general** vs. specific
- interpreted vs. **compiled**
- **imperative** vs. declarative

Perchè il C e perchè no ?!

Why not C - the importance of the programming style

// from International obfuscated C code Contest –IOCCC Competition 1990 - Best small program – N chess queens

```
#include <stdio.h>
v,i,j,k,l,s,a[99];
main()
{ for (scanf("%d",&s);   *a-s;
   v=a[j*=v]-a[i],k=i<s,j+=(v=j<s && (!k&&!!printf(2+"\n\n%c"
   (!l<<!j),"#Q"[l^v?(l^j)&1:2]) && ++l||a[i]<s&&v&&v-i+j&&v+i-j)) &&
   !(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
}
```

Comments of the authors: it contains no C language that might confuse an innocent reader (No pre-processor statements, only one 'for' statement, no ifs, no breaks, no functions, no gotos …).

⇓

The cost of software maintenance increases with the square of the programmer's creativity.

Il linguaggio C

La sintassi: descrive le regole per la composizione delle espressioni legali del linguaggio

La semantica delle espressioni

La semantica del programma (what program means) deve verificarla il programmatore

La notazione Backus Naur Form (BNF) per le regole della sintassi
Elementi di una regola – compattata per semplicità:

- terminali: parole chiave del linguaggio (if, float, int), simboli di operazioni (+), numeri, stringhe e identificatori, punteggiatura (; {} in bold)
- non terminali: identificatori di altre regole (in corsivo).
  - Metasimboli (italic per lettura): simboli del linguaggio BNF scelta (A | B), opzionalità ($\{A\}_{opt}$) e ripetizione zero o più volte ($\{A\}_{0+}$) oppure 1 o più volte ($\{A\}_{1+}$).

Rule: non terminale::=sequenza di terminali, non terminali e metasimboli.

Sintassi espressioni:

identifier ::= letter { letter | digit }$_{0+}$

letter ::= A |.....

digit ::= 0 |.....

expression::= constant_value | … | relational_expression |
logical_expression | arithmetic_expression

relational expression::= expression rel_op expression

logical_expression::= expression log_op expression

rel_op::= == | != | > | >= | < | <=

log_op::= || | && | !

Nomi per variabili e costanti C

a, x, alfa, a1, xy23, Salario_massimo, SalarioMassimo, salarioMassimo, SalarioMassimoImpiegato (MaxSalarioImp), Domani

Regole semantiche
- lettere maiuscole e minuscole sono considerate diverse.
- impossibilità di avere due identificatori con lo stesso nome nella parte dichiarativa (omonimi);
- le parole chiave del linguaggio (es. float, int) e gli identificatori predefiniti (es. printf,scanf) sono riservati;
- non usare diversi identificatori per lo stesso elemento (sinonimi)

Regole semantiche per espressioni

Notazione infissa

(a+b) * (c+d) ≠ a + b *c +d priorità

a - b – c = (a – b) – c associatività

E in notazione posfissa?

a b + c d + *

Ma in questa?

```
v=a[j*=v]-a[i],k=i<s,j+=(v=j<s && (!k&&!!printf(2+"\n\n%c"
   (!l<<!j),"#Q"[l^v?(l^j)&1:2]) && ++l||a[i]<s&&v&&v-i+j&&v+i-j))
   && !(l%=s),v||(i==j?a[i+=k]=0:++a[i])>=s*k&&++a[--i]);
```

La regola di priorità:

() [] ->

!

/ * %

+ -

< <= >= >

== !=

&&

||

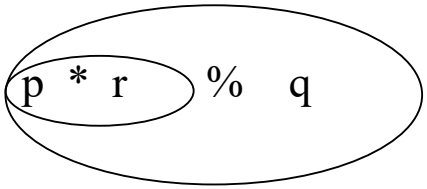
=

Sufficenza della priorità
Z= p*-> temp1

Insufficienza
Z=p*r%q

La regola di associatività:
sx -> dx o il contrario

Es. * / % sx -> dx



Tutto risolto ?

Schema base di un programma

Sintassi:

program ::= *directive_part* { *global_declarative_part* }$_{opt}$

{ *function_definition* }$_{0+}$

int main () { { *local_declarative_part* }$_{0+}$ *executable_part* **}**

directive_part ::= .. | **#include** *identifier*

global-declarative part ::= *constant_declarations* | *type declarations* | *variable_declarations*

variable_declarations ::= { *type_specifier variable_identifier* }$_{1+}$

executable_part ::= *statement_sequence*

statement_sequence ::= *single_statement* | **{** { *single_statement* }$_{1+}$ **}**

sintassi incorpora regola semantica

Regola semantica

- un identificatore può essere utilizzato in un'istruzione (statement) solo se è stato definito/dichiarato in precedenza;

Style

- all'inizio del programma inserire un commento con nome autore, data versione, descrizione scopo programma,....

Le istruzioni di un programma C

executable_part ::= *statement_sequence*

statement_sequence ::= *single_statement* | { *single_statement* }$_{1+}$

⇓

BLOCCO

ISTRUZIONI DI INGRESSO/USCITA

#**include <stdio.h>** nel programma;

printf(messaggio,espressione)

Messaggio:
- stringhe di caratteri tra “”,
- caratteri di controllo (es. \n -> salto riga),
- formato di stampa e conversione: %X, dove X =d (int…), c (char), s (stringa di char), f (float/double formato x.y),…
- espressione da stampare coerente col formato;

scanf (messaggio, indirizzo variabili)

Messaggio: formato di lettura e conversione= %X

Indirizzo della variabile -> **&** nomevariabile

Come funzionano?

Tastiera ⇒ char ⇒ buffer ⇒ scanf ⇒ int/float... ⇒ memoria

⇑ ⇐ fflush()

S.O.

⇓

Video ⇐ char ⇐ buffer ⇐ printf ⇐ int/float... ⇐ memoria

Due istruzioni particolari:

int getchar(void): legge un carattere da stdin e lo restituisce come valore intero (si può usare come carattere)

putchar(int c): visualizza il carattere memorizzato in c

Osservazione

Il corso suppone che l'operazione di lettura di un dato da terminale non riceva mai un valore incompatibile col tipo richiesto (chiedo un numero e si inserisce una stringa), un valore non rappresentabile (numero troppo grande).

Il corso presuppone viceversa che siano effettuati tutti i controlli che permettano di accettare solo i valori necessari al programma da scrivere.

ISTRUZIONI ASSEGNAMENTO

Variabile = espressione;
Esempi:
x = 23; w = 'a'; y = z;
alfa = x+y; x = x+1; ← calcolo + assegnamento
r3 = (alfa*43–xgg)*(delta–32*ijj);

Osservazioni:
- controllo compatibilità
- associatività e priorità
- controllo overflow e approssimazioni nei calcoli

CONTROLLO DELLA SEQUENZA
(a livello di istruzione)

programmazione non strutturata
⇓
Linguaggio simbolico: sequenza, salto condizionato e incondizionato
⇓
Problemi nello sviluppo e nella manutenzione dei programmi

Programmazione strutturata (metà anni ’60)
- togliere salti incondizionati
- codificare alcune restrizioni

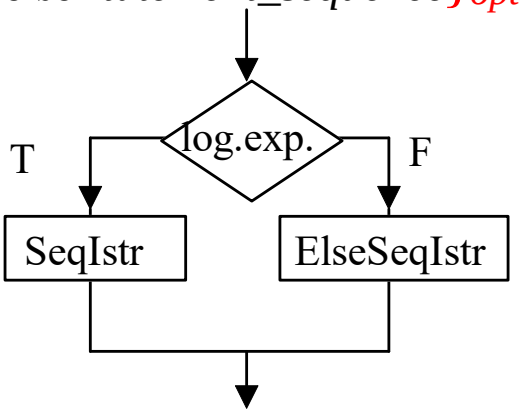
teorema di Bohm-Jacopini (CACM 1966): le 3 strutture di controllo (sequenza, selezione binaria (if) e ciclo (while)) sono sufficienti per realizzare qualsiasi algoritmo.

ISTRUZIONI COMPOSTE (guidate da una condizione logica)

Istruzione condizionale

if (*logic_expr***)** *statement_sequence* { **else** *statement_sequence* }$_{opt}$

precondizione: log.exp deve essere calcolabile prima dell'esecuzione del test.



Esempi:

1. if (x==0) z = x; else {w = y; z = y + 1;}
2. if (x==0) z = x; else w = y; z = y + 1;
3. if (x==0) /*leggibilità*/
   z = x;
   else
   w = y;
   z = y + 1;

4. if (x >0) printf("positivo");
   else if (x<0) printf("negativo"); else printf("zero");

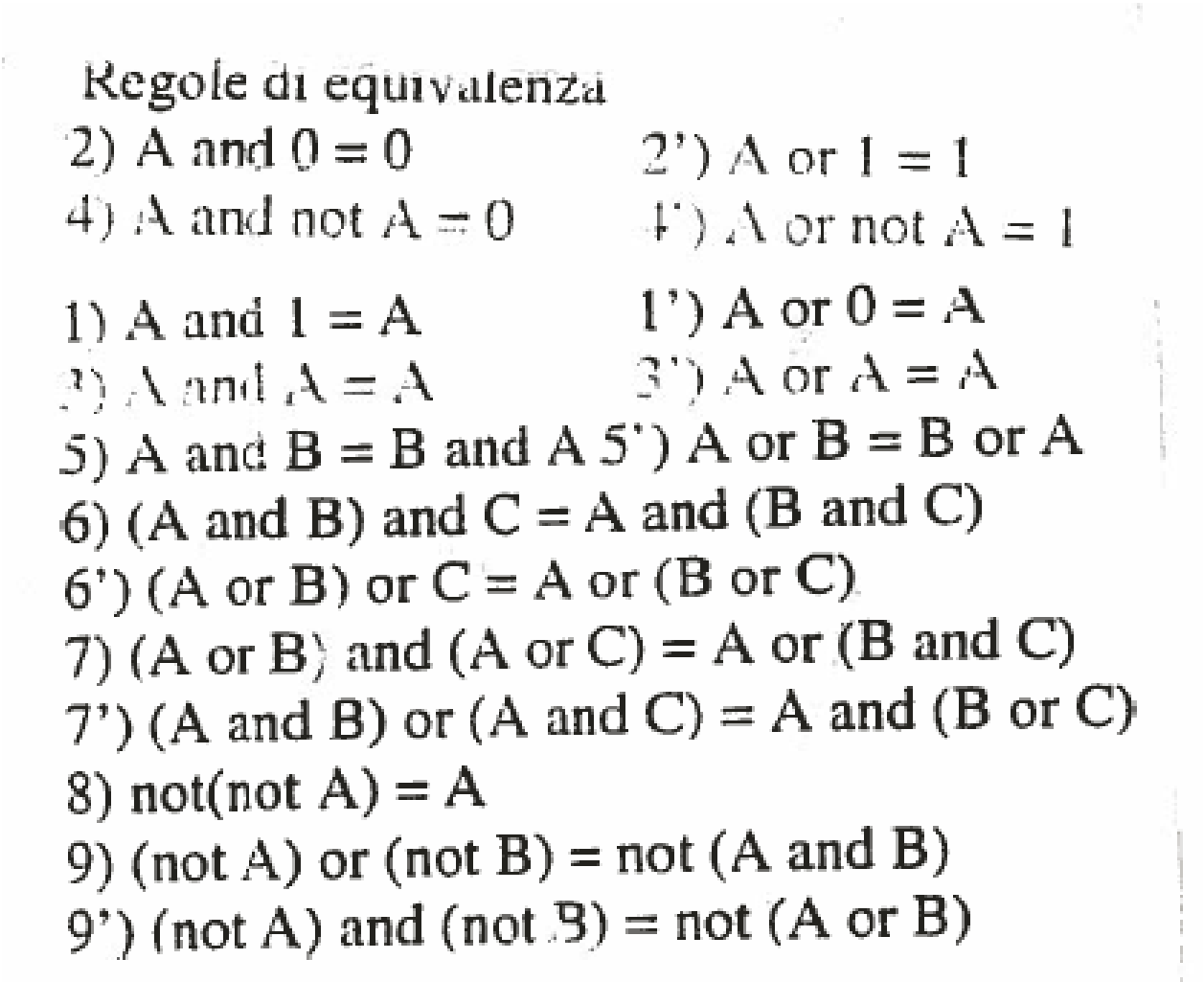
5. /*innesto o sequenziale*/
   if (x==2) blocco A else if (x==5) blocco B
   oppure ?
   if (x==2) blocco A if (x==5) blocco B

6. if(n>0) if (a== c) f=3; else f=5; ambiguità dell'else
   ↓
7. if(n>0) {if (a== c) f=3; else f=5;}
8. if(n>0) {if (a== c) f=3;} else f=5;

9. if(x==0) if (b==1) if (c==0) blocco A;
falsa catena if ((x==0) && (b==1) && (c==0)) blocco A;

10. if (x) BloccoA ≡ if (x!=0) BloccoA - vero (!= 0), falso (=0)

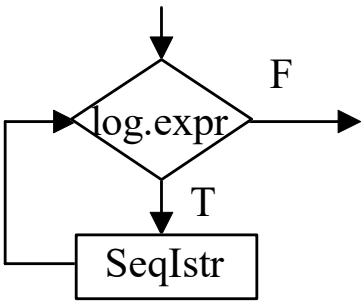
11. side effects: if (x=y) BloccoA ≡ x=y; if (x) BloccoA



Ciclo a condizione iniziale

while (*logic_expr***)** statement_sequence

Semantica:



1. logic_expr deve essere calcolabile prima dell'esecuzione del ciclo.

2. $0 \leq$ numero iterazioni $\leq \infty$.
   while (0) o while (1)

Esempio: leggere sequenza caratteri da terminale terminatore newline (“\n”), visualizzare numero caratteri letti
Attenzione: lunghezza arbitraria da 0 a ∞

```
#include <stdio.h>
  char c; int numero=0;
  int main()
  { printf(“inserire sequenza ”); c=getchar();
    while (c != ‘\n’)
          {numero++; c=getchar(); }
    printf(“\nletti %d caratteri“, numero);
  }
```

oppure

```
  {c=’0’; numero=0;
   while (c != ‘\n’)
     {c=getchar();
      numero++;
     }
  }
```

Esempio.
Leggere 100 interi positivi da terminale e visualizza true se tutti maggiori di 3 altrimenti false.

```
  int n, i;
  int main()
  { i=1;
    while (i<=100)
     { scanf(“%d”,&n);
  ??   if (n>3) printf(“true”); else printf(“false”);
       i++;
     }
  }
```

Strutture non necessarie

Istruzione condizionale multipla

switch (*expression***)**
{ case valore1**:** *statement_sequence*
…..
case valore n**:** *statement_sequence*
{**default:** *statement_sequence* }*opt*
}

Osservazioni:
1. expression deve essere di tipo discreto
2. expression = valore i ⇒ esecuzione case valore i + case che seguono ⇒ istruzione break interrompe cascata
3. expression ≠ tutti i valori ⇒ esecuzione case di default (meglio se esiste), altrimenti no operation
4. esclusività dei valori nei case
5. statement sequence può non avere le {}

Esempio 1:
programma legge opzione utente (1=insert, 2=delete, 3=update) e attiva la corrispondente azione

```
#include <stdio.h>
int main()
{int   opzione;
 printf("Inserisci opzione 1,2,3\n"); scanf("%d", &opzione);
 switch (opzione)
    {case 1: attiva insert        break;
     case 2:  attiva delete       break;
     case 3:  attiva update       break;
     default: printf("\nopzione non valida");
     }
}
```

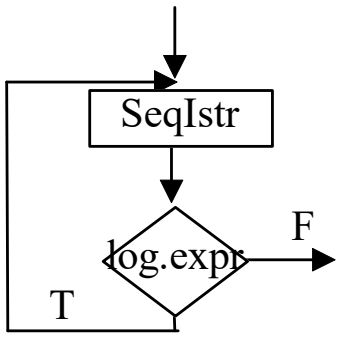
Esempio 2: scandisce un testo, conta quanti caratteri sono cifre numeriche, di separazione (spazi, segni di puteggiatura, a capo) o di tipo diverso.

```
char c;
int n_cifre=0, n_separatori=0, n_altri=0;
 ...
c = ....;
switch (c)
  { case '0': case '1': ...    case '9':
            n_cifre++;       /* ≡ a n_cifre = n_cifre + 1; */  break;
    case ' ': case '\n' : case ';' : case ':' : ...  case ', ':
            n_separatori++;      break;
    default:
            n_altri++;
   }
```

Ciclo a condizione finale

do *statement_sequence* **while (***logic_expr***);**

Semantica:



Osservazioni:

1. logic_expr deve essere calcolabile dopo l'esecuzione del ciclo.
2. 1 ≤ numero iterazioni ≤ ∞.

Esempio: leggi un intero e accettalo se ≥ 0 o se uguale a -99

```
do
    {printf("valore>=0 o -99"); scanf("%d",&numero);}
while ((numero<0) && (numero != -99));
```

Esempio: leggi un intero e accettalo se ≥ 100 e ≤ 200

```
do
    {printf("valore in [100, 200] "); scanf("%d",&numero);}
while ((numero<100) || (numero > 200));
```

Esempio

Legge sequenza interi ≥ 0 terminata da -99 e ne calcola la somma

⇓

- sequenza può essere vuota e controllo che numeri siano ≥ 0 o -99

1. somma = 0
2. leggi/controlla numero
3. while valore != -99 ← seq. vuota
4. { somma=somma+valore
5. leggi/controlla prossimo numero
6. }
7. stampa somma

```
   /* Programma  */
#include <stdio.h>
int somma=0, n;
int main()
2 {do {printf("num"); scanf("%d",&n);}while ((n<0) && (n!= -99));

3   while (n != -99)
4   {somma = somma + n;
5    do{printf("num"); scanf("%d",&n);}while ((n<0) && (n!= -99));
     }
7    printf("\nsomma = %d",somma);
}
```

Esempio

Conta caratteri letti di una sequenza di cardinalità ≥ 1 e terminata con "\n"

```
int numero;
int main()
{c=getchar();
  do {numero++; c=getchar(); } while (c != '\n');
   printf("\nletti %d caratteri", numero);
}
```

Se la sequenza può essere vuota e si vuole mantenere il do while?

```
#include <stdio.h>
  char c; int numero=0;
  int main()
  { c=getchar();
    if (c!='\n')
       do {numero++; c=getchar(); } while (c != '\n');
    printf("\nletti %d caratteri", numero);
  }
```

Ciclo for

for (*expression1; expression2; expression3***)** $\{$*statement_sequence*$\}_{opt}$

Semantica:

```
expression1;
while (expression2)
 { {statement_sequence}opt
    expression3
 }
```

Esempio:
Somma di 100 interi letti da terminale

```
int sum=0, n, i ;
for (i=1; i<=100; i++)
  {printf("numero %d  ", i);    scanf("%d", &n);
   sum = sum + n;
  }
```

Esempio fattoriale:

$$N! = \prod_{i=1}^{N} i$$

per ogni N∈ numeri naturali

con 0! = 1

```
#include <stdio.h>
int fatt=1, N, i;
int main()
{     scanf("%d",&N);
      for(i=1;i<=N   ;i++)  fatt= fatt*i;
      printf("= %d",fatt);          ⇐ limiti rappresentazione
}
```

Perché non usare cicli di questi tipi?
1. for (a=1; v!='\0' ; i++)...;

2. for (; ;)y++;

3. for (; ;) {… if () break; …}

4. for (i=1; i<=100; i++);

istruzioni di salto

break (si applica all'istruzione switch, for, while, do while)
salto strutturato alla prima istruzione che segue l'istruzione

continue (si applica a for, while, do while)
salto strutturato alla prossima iterazione del ciclo

goto X
X: …. MAI!!

L'ASTRAZIONE NEI DATI

Linguaggio binario 0000111100110011

Linguaggio Assembly MEM: RES 1

Linguaggio C: **variabile** caratterizzata da:

proprietà generali

- un nome *identifier*

- un tipo (dominio e operazioni)

- **!!non ha un valore iniziale (programma deve inizializzare)**

proprietà dipendenti dalla posizione di definizione nel programma

- modalità di allocazione e tempo di vita
  Le variabili della *global-declarative part* sono dimensionate a compile time (RES), allocate a inizio esecuzione del programma e deallocate a fine esecuzione del programma.

- campo di validità: l'intero programma

Concetto di TIPO

T = <D,O> dominio D, operazioni O

R(T) Rappresentazione tipo sul calcolatore

Perché tipizzare le variabili?

- controllare l'uso corretto delle variabili nel programma all'atto della compilazione;
- conoscere a priori la quantità di memoria allocata.

Esempi:
float alfa; miotipo x;
al*b*a = alfa + 1; 'alba' undeclared identifier
x=alfa; '=' incompatible types
altrimenti

I tipi disponibili

TIPI SEMPLICI + COSTRUTTORI DI TIPO
⇓
TIPI DERIVATI DAL PROGRAMMATORE
f(applicazione)

Esempi:

tipo semplice: int costruttore ARRAY
⇓
tipo derivato: int A[10]

Capacità di rappresentazione di un linguaggio = f(tipi di dati esprimibili)

I tipi semplici (built-in)

TIPO VOID tipo nullo

TIPO DISCRETO INTERO - short int, int, long int (unsigned)
R(short int) ≤ R(int) ≤ R(long int)
(1 PAROLA)
O = + - * / = == > X++…

TIPO DENSO REALE
float (precisione semplice), double, long double
R(float) ≤ R(double) ≤ R(long double)
O = vedi int (**!!approssimazione … (x/y)*y = x?**)
Es. float salario_medio=0.0;(notazione a v. fissa)
float superficie =7E+20 ≡ $7*10^{20}$ (notazione a v. mobile)

TIPO DISCRETO CARATTERE
char
O = vedi int e la libreria ctype (#include <ctype.h>)
Es.
char car1 =’\153’ (ottale) ≡ 107 (dec) ≡ ‘k’

Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	NUL (null)	32	20	040	SPACE	64	40	100	@	96	60	140	`
1	1	001	SOH (start of heading)	33	21	041	!	65	41	101	A	97	61	141	a
2	2	002	STX (start of text)	34	22	042	"	66	42	102	B	98	62	142	b
3	3	003	ETX (end of text)	35	23	043	#	67	43	103	C	99	63	143	c
4	4	004	EOT (end of transmission)	36	24	044	$	68	44	104	D	100	64	144	d
5	5	005	ENQ (enquiry)	37	25	045	%	69	45	105	E	101	65	145	e
6	6	006	ACK (acknowledge)	38	26	046	&	70	46	106	F	102	66	146	f
7	7	007	BEL (bell)	39	27	047	'	71	47	107	G	103	67	147	g
8	8	010	BS (backspace)	40	28	050	(	72	48	110	H	104	68	150	h
9	9	011	TAB (horizontal tab)	41	29	051	)	73	49	111	I	105	69	151	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	B	013	VT (vertical tab)	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	D	015	CR (carriage return)	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	E	016	SO (shift out)	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	F	017	SI (shift in)	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE (data link escape)	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1 (device control 1)	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2 (device control 2)	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3 (device control 3)	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4 (device control 4)	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN (synchronous idle)	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB (end of trans. block)	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN (cancel)	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM (end of medium)	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB (substitute)	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC (escape)	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS (file separator)	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS (group separator)	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS (record separator)	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US (unit separator)	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

Esempio:

```
#include <stdio.h>
char C;
void main()
{printf(“\ninserisci carattere minuscolo”);
  scanf("%c", &C);                    ←Ipotesi: ins. minuscola
  printf("\nMaiuscola di %c =%c e ASCII=%d",C,C-(‘a’-‘A’),C);
  printf(“fine stampa);                                    ↑
}                                                          32
```

Regole di compatibilità/conversione automatica tra interi e reali

1. Espressione con elementi (costanti, variabili) eterogenee:
   conversione implicita operandi: minor precisione => massima precisione.
   Esempio: int i; float f; i+f ⇒; i diventa float e poi i+f)

2. Assegnamento eterogeneo A = espressione:
   Tipo dell’espressione convertito al tipo della variabile A (perdita informazione tra maggior e minor precisione).
   Esempi: int a, b; float c;
   a=c; conversione e troncamento
   c=a; conversione
   a=a/b; troncamento
   c=a/b come sopra
   a=a/c; conversione di a, divisione, troncamento

3. Conversione esplicita di un tipo (cast) trattato marginalmente.

Scritto come utf8

BuildingsBase__AbstractConstruction.elevation##Amministrazione viabilità__Estesa_amministrativa

Letto come cp1252

BuildingsBase__AbstractConstruction.elevation##Amministrazione viabilitÃ __Estesa_amministrativa

Scritto come cp1252

BuildingsBase__AbstractConstruction.elevation##Amministrazione viabilità__Estesa_amministrativa

Letto come utf8

BuildingsBase__AbstractConstruction.elevation##Amministrazione viabilit�__Estesa_amministrativa

E i valori costanti?

la costante pigreco, aliquota irpef, …

Come trattarle in C

- (NI) tramite variabile
  es. int modello = 10; const int modello = 10;
- (SI) tramite direttiva ⇒ precompilazione
  #define A 12
  #define C ‘a’
  #define D “stringa”
  #define E 12.2 (double)

- (NO) utilizzare direttamente il valore dove serve
  B= 3.14*E+…