
U.D. 3

# 1. - Programmazione in linguaggio C

Struttura
I Tipi semplici
Costanti, Variabili, Operatori
Espressioni omogenee
Operatori aritmetici, logici, bitwise

**Fonti:**
A. Antola - POLI_MI Dispense del Corso
A. Ciampolini – UNI_BO – Lucidi del corso

## INTRODUZIONE

Sviluppo del software:

parte dalle specifiche (descrizione dei requisiti) per arrivare ai programmi (prodotti)

Programmazione:

- analisi del problema
- scomposizione funzionale
- definizione degli algoritmi
- codifica degli algoritmi e del programma: descrizione in un linguaggio di programmazione di alto livello

### Linguaggi di programmazione di alto livello (HHL)

- FORTRAN (FORmula TRANslator) - seconda metà anni 50, versioni successive FORTRAN 70 ...
- COBOL (Common Business Oriented Language) - anni 60

Linguaggi «strutturati» (programmazione strutturata) ALGOL60

- PASCAL
- Modula 2
- C
- C++
- linguaggi basati su altri paradigmi di programmazione

**Linguaggio C** sviluppato negli anni 70 (C standard ANSI). Molto diffuso e adatto ad un ampio spettro di applicazioni

- *scientifiche*
- *gestionali*
- *industriali*: acquisizione dati, controllo di processo..
- *informatiche*: software di base (SO Unix), strumenti (CAD), pacchetti

## Linguaggio artificiale di alto livello (HHL)

E' caratterizzato da:

*elementi*:
alfabeto o vocabolario del linguaggio
*sintassi:*
insieme di regole tramite le quali si compongono gli elementi per costruire frasi eseguibili (= istruzioni)
*semantica:*
significato degli elementi, delle istruzioni, del programma

*Regole sintattiche:*
devono essere univoche sulla composizione delle frasi.
⇒ si deve poter stabilire con certezza e in modo automatico se una frase è sintatticamente corretta,

*Correttezza sintattica*: è condizione necessaria per la corretta esecuzione del programma (è possibile la traduzione da parte del compilatore).

Descrizione formale delle regole sintattiche
- diagrammi sintattici
- Backus Naur Form

aiutano a prevenire errori sintattici

Errori di un programma:
sintattici: rilevati compile-time
di esecuzione: run-time

ESEMPIO 1 - Algoritmo per elevamento a potenza

```
#include <stdio.h>


main ( )

{
 int potenza, base;
 int esponente, i;         /* devono essere >=0*/

 printf("Inserisci il valore della base:");
 scanf("%d",&base);
 printf("Inserisci il valore dell'esponente:");
 scanf("%d", &esponente);

 potenza=1;
 i=0;

 while (i<esponente)
   {
   potenza=potenza*base;
   i=i+1;
    }

 printf("Potenza=%d \n", potenza);
}
```

## ELEMENTI DEL LINGUAGGIO C

*parole chiave* *(riservate)* ⇒ proprie del linguaggio
- sono le istruzioni oppure hanno un significato particolare (tipi)

*identificatori* ⇒ costituiti da sequenze di lettere ...
- rappresentano nomi di variabili, costanti, (tipi nuovi), funzioni, procedure
- definiti dall'utente oppure «di sistema» (di libreria)

*operatori* (unari o binari)
- di assegnamento =
- aritmetici + - * /
- relazionali (confronto) > < <= == ...
- logici ! (not) && ||
- di chiamata di funzione/procedura ( )
- di dereferenziazione &
- dipendenti dal costruttore di tipo * [ ] ....
- altri

*separatori (delimitatori)*
- di identificatori di variabili e costanti ,
- di istruzioni ;
- commento /* */
- di blocco di istruzioni { }
- in espressioni ( )

*direttive* al preprocessore C
- # include (parola chiave) .....

*valori costanti* (cifre o caratteri)

# THE ‘C’ ALPHABET

**The characters in ‘C’ are divided into 3 groups**

- **Digits: 0 - 9**
- **Letters: A - Z, a - z, $, -**
- **Special Characters:**

| | | |
|---|---|---|
| b Blank | / Slash | ‘ Apostrophe |
| + Plus | ! Exclamation | * Asterisk |
| ? Question Mark | ^ Circumflex | ( Let Parent |
| : Colon | \| Stroke | ) Right Par |
| < Less than | , Comma | [ Left Bracket |
| = Equal | % Percent | ] Right Bracket |
| > Greater than | $ Dollar sign | { Left Brace |
| ~ Tilde | . Period | } Right Brace |
| & Ampersand | _ Underscore | # Pound sign |
| - Hyphen | “ Quotation mark | ; Semicolon |
| \ Backslash | | |

# Reserved Words

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

---

## Struttura di un Programma C

Un programma C ha in linea di principio la seguente forma:

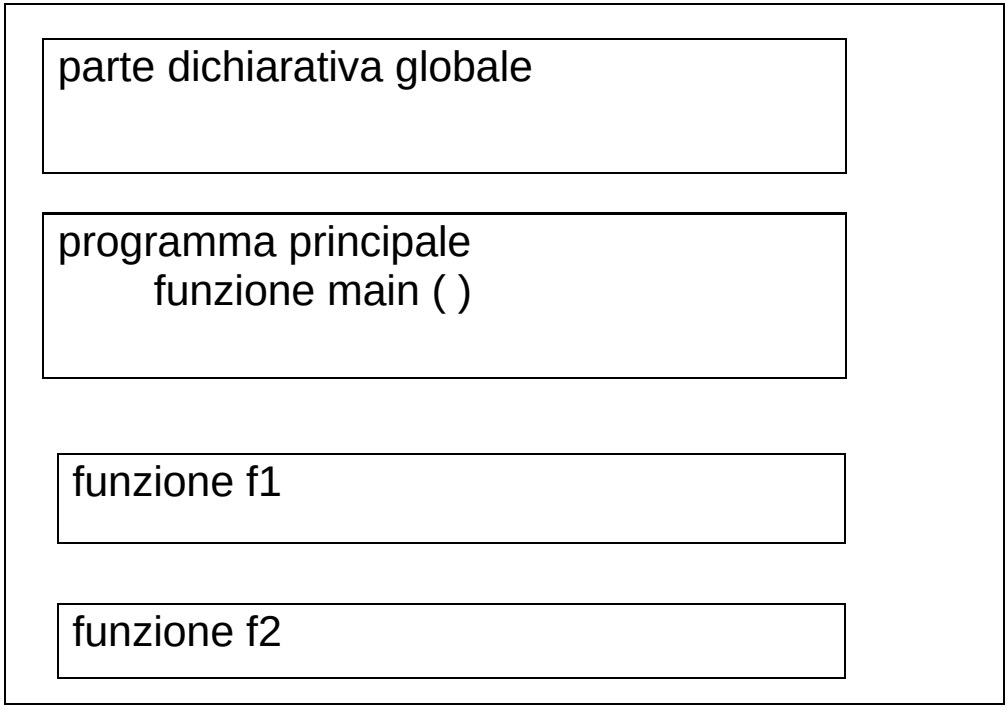
- **Direttive per il preprocessore**
- **Definizione di tipi**
- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e delle variabili passate alle funzioni)
- **Dichiarazione delle Variabili Globali**
- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:

```
Tipo NomeFunzione(Parametri)
{
   Dichiarazione Variabili Locali
   Istruzioni C
}
```

```
#include <stdio.h>

typedef struct point {
            int   x;    int   y;
            };
int       f1(void);
void    f2(int i, double g);


int   sum;


int main(void)
{
      int j;
      double g=0.0;
      for(j=0;j<2;j++)
            f2(j,g);
      return(2);
}

void f2(int i, double g)
{
      sum = sum + g*i;
}
```

## STRUTTURA DI UN PROGRAMMA C - 1

programma C




Stile di scrittura di un programma ⇒ **leggibilità**

- scelta degli identificatori
- indentazione: «struttura grafica» che rispecchia la struttura logica

- uso di commenti

## STRUTTURA DI UN PROGRAMMA C - 2

### Parte dichiarativa globale:

- servizi (funzioni) importate da altri moduli (file), cioè definite e codificate in altri file

- «oggetti» (tipi di dati, variabili, costanti simboliche, prototipi di funzioni) visibili (utilizzabili) da tutto il programma, cioè da main e dalle altre funzioni.

### Programma principale:

```
main ()

{

  parte dichiarativa
  locale



  parte esecutiva

}
```

parola riservata (identificatore di funzione)
appare una e una sola volta nel programma
definisce l'inizio dell'esecuzione
è (formalmente) una funzione

definisce l'insieme di «oggetti» usati dal programma principale per l'esecuzione. sono oggetti visibili (locali) a main.

insieme di istruzioni che costituiscono il programma principale

## STRUTTURA DI UN PROGRAMMA C - 3

### Parte dichiarativa locale

1. dichiarazione di costanti
2. definizione di «nuovi» tipi definiti dall'utente (ridenominazione)
3. dichiarazione di variabili
4. prototipi di funzioni

«Regole» sintattiche sulle dichiarazioni sia locali che globali:

- ogni identificatore usato deve essere prima definito
- ogni variabile usata deve essere prima dichiarata

**Parte esecutiva: istruzioni** (per tipologia)

- istruzioni di assegnamento
- istruzioni composte
- costrutti di (modifica del flusso di) controllo (costrutti condizionali, costrutti ciclici)
- «istruzioni» di ingresso e uscita
- chiamate di sottoprogrammi (funzioni)

ESEMPIO 1 - Dichiarazioni e istruzioni

```
#include <stdio.h>


main ( )

{
 int potenza, base;
 int esponente, i;         /* devono essere >=0*/

 printf("Inserisci il valore della base:");
 scanf("%d",&base);
 printf("Inserisci il valore dell'esponente:");
 scanf("%d", &esponente);

 potenza=1;
 i=0;

 while (i<esponente)
   {
   potenza=potenza*base;
   i=i+1;
    }

 printf("Potenza=%d \n", potenza);
}
```

ESEMPIO 2 - Dichiarazioni e istruzioni

```
#include <stdio.h>

main ( )

{
 int dato, sommatoria;

 sommatoria=0;
 printf("Inserisci il prossimo dato:");
 scanf("%d",&dato);

 while(dato!=0)
  /*ciclo di acquisizione e somma */
 {
  sommatoria=sommatoria+dato;
  printf("Inserisci il prossimo dato:");
  scanf("%d",&dato);
 }

 printf("Il valore della sommatoria e': %d",
sommatoria);
}
```

## 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.

# Costanti

## Numeri interi

Rappresentano numeri relativi (quindi con segno):

| | **2 byte** | **4 byte** |
|---|---|---|
| base decimale | 12 | 70000, 12**L** |
| base ottale | **0**14 | **0**210560 |
| base esadecimale | **0x**FF | **0x**11170 |

## 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, OX(esadecimale)

## 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** |
| tab | **\t** |
| backspace | **\b** |
| form feed | **\f** |
| carriage return | **\r** |
| codifica ottale | **\ooo** (o cifra ottale 0-7)<br>\041 è la codifica del carattere ! |

Il carattere \ inibisce il significato predefinito di alcuni caratteri “speciali” (es. ', ", \, ecc.)
\' \\ \" \0 (carattere nullo)

## Stringhe:

Sono sequenze di caratteri tra doppi apici " ".

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

## Esempio: (printf e` l’istruzione per la stampa)

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

## Effetto ottenuto:

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

# Constant Values

- Integral bases: `68`, `043`, `0x3B`
- Integral suffixes: `-45L`, `88u`, `4ul`
- Floating point suffixes (default is `double`): `45.f`, `.34L`, `0.87E-2`
- Characters: `'a'`, `'\n'`, `'\"'`
- Strings (null-terminated): `""`, `"I am a string"`, `"\n"`, `"tab here:\t"`

Constants values are (obviously) non-addressable.

# Constants

## Integer vs Floating-point Constants

10 33 3.333

## Decimal Constants

10 33L

## Octal

017

## Hex

0X1A 0x1A 0x1a

## Character

‘A’ ‘a’ ‘2’

## String

“A string has more than one character”

## VARIABILI E DICHIARAZIONE DI VARIABILI

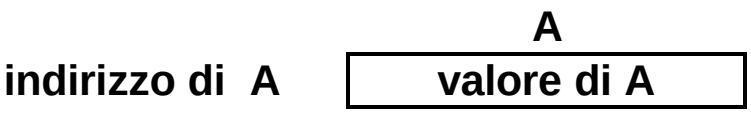
Sono contenitori di informazioni (cioè di valori)

- hanno un **nome** simbolico che rappresenta in modo univoco una locazione di memoria
- hanno un **tipo** che rappresenta il tipo di **codifica** usato per rappresentare i valori, quali **valori** possono assumere, quali **operazioni** sono lecite e come agiscono tali operazioni

**La dichiarazione di una variabile** serve a dire *che* e *come* una variabile verrà utilizzata dal programma.

- definisce **l'identificatore** simbolico (nome)
- definisce il **tipo** adatto ai valori da contenere
- alloca la **quantità di memoria** adeguata a contenere il tipo
- associa in modo univoco **l'indirizzo** di memoria al nome
- consente di rilevare errori sull'uso «improprio» della variabile nel programma in compilazione

Rappresentazione in memoria della variabile **A**




L'indirizzo di A è quello del primo (ed eventualmente unico) byte della porzione di memoria che ne contiene il valore.

Sintassi C

**tipo id_var <, id_var, ..>;**

# Istruzione di Assegnamento

Il concetto di **variabile** nel linguaggio C rappresenta un'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:**

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

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


/*  assegnamento  del  risultato  di  una
espr. aritmetica: */

  Y = 2*3.14*X;
     }
```

# 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 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 e` il dominio
- {+, -, *, /, =, >, <, etc } e` l'insieme di operazioni

# Il concetto di Tipo

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

### Se un linguaggio e` 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 piu` alto della loro organizzazione fisica. 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

Il C e` un linguaggio tipato.

### Classificazione dei tipi di dato in C:

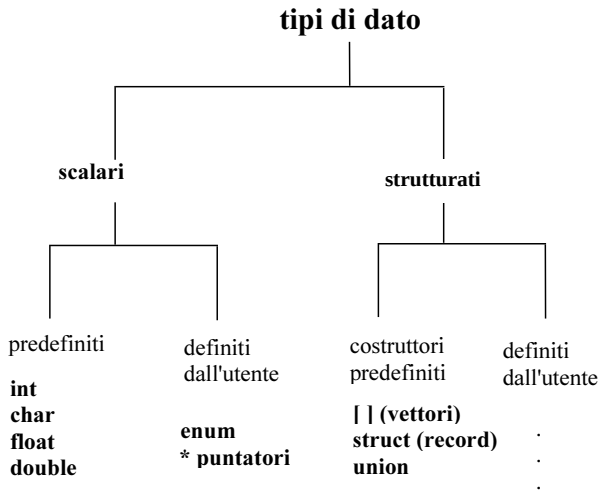
### Si distingue tra:

- 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, 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)

☞ E` 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 e' su 16 bit..*/
long int Y;/*..Y e' 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]*/
```

## TIPI SEMPLICI BUILT-IN DEL C

I nomi di questi tipi sono delle parole chiave del linguaggio:

- **char**: (8 bit - 1 byte) Valori da 0 a 255 che rappresentano la codifica ASCII estesa del carattere corrispondente

- **int** (16bit - 2 byte) Rappresentano gli interi relativi. Valori in complemento a 2 da - 32768 a + 32767

- **float** (32 bit - 4 byte). Rappresentano i razionali espressi in virgola mobile (buona approssimazione dei reali). Valori espressi tramite mantissa e esponente (standard IEEE). Intervallo di valori rappresentabili: da $-10^{38}$ a $+10^{38}$)

- **double** (8 byte). Sono float in doppia precisione.

Si dicono tipi aritmetici (char, int *integral*; float e double *floating*)

L'insieme di valori ammissibili (vmin e vmax) e lo spazio allocato in memoria possono essere modificati tramite **qualificatori** (specificatori) di tipo. I qualificatori sono parole chiave del linguaggio che si premettono al tipo.

**Indirizzo** di una variabile

- operatore: **`&nome_var`**
- valori assunti per gli indirizzi: interi >= 0

**`&nome_var`** rappresenta l'indirizzo di memoria del primo byte allocato per la variabile.

# Data Type Qualifiers

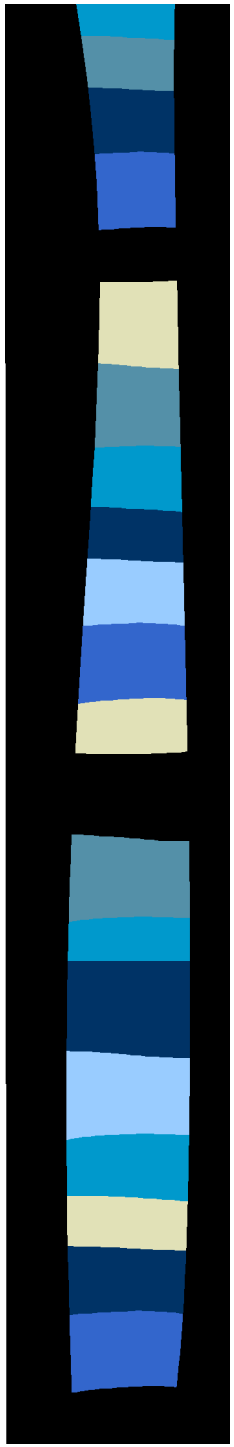
short

long

signed

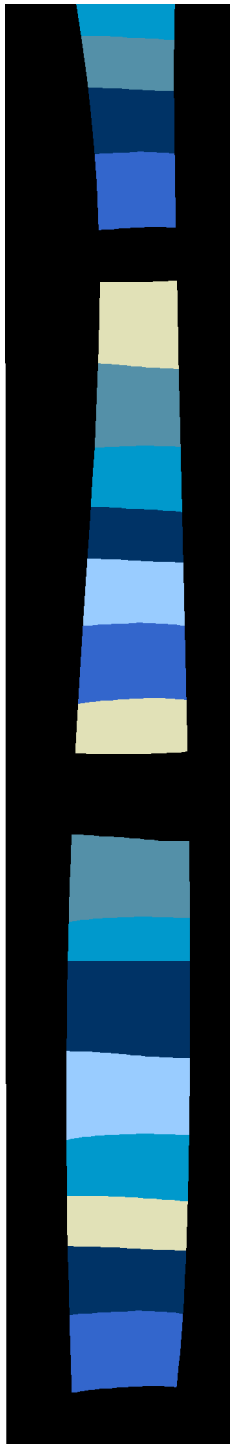
unsigned

short int <= int <= long int



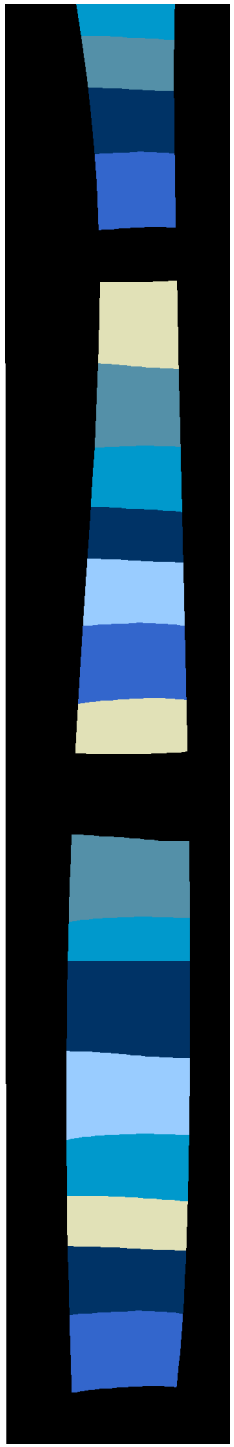
# Classi di variabili #1

- Sono modificatori applicati ai tipi predefiniti che ne alterano le caratteristiche - *aggettivi*.
- Permettono di estendere i tipi predefiniti
- Molto utile associato alla creazione di nuovi tipi definiti dall'utente.



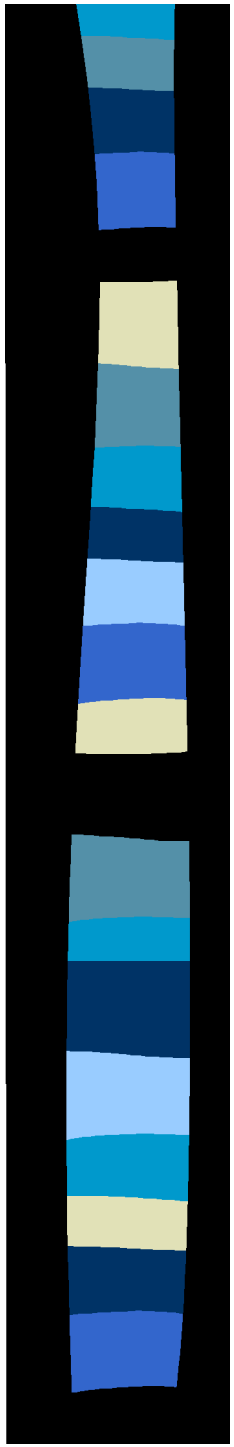
# Classi di variabili #2

- const → variabili il cui valore non può essere modificato.
- extern → variabili che sono dichiarate altrove.
- register → variabili che vengono memorizzate in un registro della CPU.
- signed → variabili che hanno esplicitamente un segno.



# Classi di variabili #3

- static → variabili che non vengono eliminate all'uscita di una procedura.
- unsigned → variabili che sono private del segno.
- volatile → variabili che possono essere modificate in modo non definito dal programma (da hardware pilotato).
- long, short → attributo che varia la dimensione della variabile.

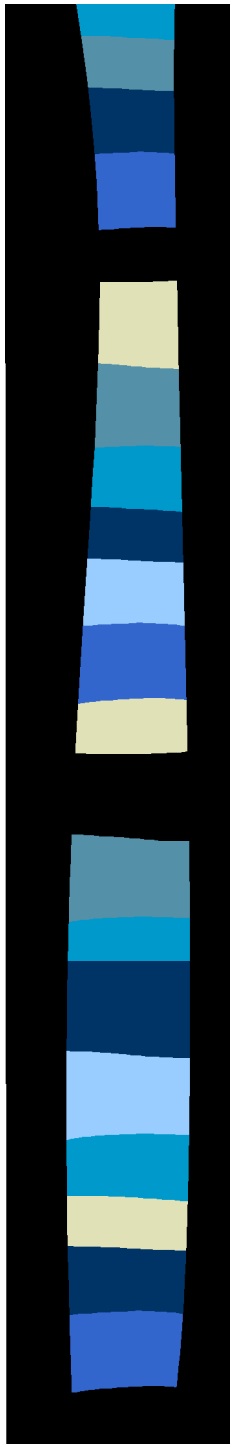


# Sintassi generica

Sintassi generica per dichiarare una variabile:

{classe} [tipo] [nome] {=valore};

- ✓{..} indica parametro opzionale
- ✓[..] indica parametro obbligatorio



# Esempi

long int i=1e50; //aumento il valore massimo

short int i=10; // diminuisco il valore massimo

unsigned char carattere=‘a’;

const double val=123.42134;

long double val =1234.42;

**!!** Alcuni modificatori non hano senso:

```
long char a; /* non ha senso */
```

# Il tipo `int`

## Dominio:

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.

**Esempio**: definizione di una variabile intera

```
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:

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

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

# Int Data Type

Whole numbers and their negatives

. . . -3, -2, -1, 0, 1, 2, 3, . . .

Constants are written as in other languages

```
int vacation_days = 12;
int dependents = 3;
```

Can use with qualifiers:

```
short int mph;      /* defines a short integer */
                    /* 2 bytes, 16 bits, on AXP */
int line_speed;     /* defines regular integer */
                    /* 4 bytes on AXP, 2 bytes on PC */
long int dist_to_moon; /* long int, 4 bytes on AXP */
```

# Il tipo int

## Operatori:

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:<br>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:<br>10==3 ➠ *falso*<br>10!=3 ➠ *vero* |
| **<, >, <=, >=** | minore, maggiore, minore o uguale, maggiore o uguale<br>10>=3 ➠ *vero* |

# 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

E` 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** e` **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

Il dominio associato al tipo **char** e` **ordinato**: l’ordine dipende dal codice associato ai vari caratteri nella tabella di riferimento.

### Definizione di variabili di tipo char: esempio

```
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 (cioe`, **l’intero** che lo indica nella tabella ASCII)

# Special Characters

- Escape sequences are used to represent many special characters in C

| | | |
|---|---|---|
| \n | newline | each escape sequence represents only one character |
| \t | tab | |
| \b | backspace | |
| \f | form feed | |
| \a | audible bell | |
| \0 | null | |

- Can be mixed freely with other characters

  printf("\nA\nB\tC\nDE\aF\n"); /* what does this print? */

  do NOT try to print the null ('\0') character

# Il tipo char: Operatori

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

☞ 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)

'a'>'b' ***falso*** perche` *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 e` *vero*)

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

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

'A' && 'a' ➠ 1

**Uso dei qualificatori:**
e` possibile, come per gli interi, applicare i qualificatori signed, unsigned a variabili di tipo char:

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

# Data Types: Int vs Char

Why is it that given ...

```
char number1;
int    number2;
int    number3;

number1 = 160;              /* assigns values */
number2 = 565;              /* to variables */
number3 = number2;
number2 = number1;    /* this is OK */
number1 = number3;    /* this is NOT? */
```

# I tipi float e double (reali)

**Dominio:**

Concettualmente, e` l'insieme dei numeri reali R.

In realta`, e` un sottoinsieme di R 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 puo` 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;
```



# Tipi float/double

## Operatori

**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:**

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).

# Float Data Type

- Real numbers expressed as decimal fractions

  ```
  float salary, pay_rate;      /* defines two float variables */
  ```

- May also use E-type notation (exponential notation)

  ```
  float term2 = 5.7345E2;  /* equals 573.45 */
  ```

- All three of the following represent the same value

  ```
  float term1 = 123.45;
  float term2 = 1.2345e2;   /* E or e may be used */
  float term3 = 12345.0e-2;
  ```

# Double Data Type

- Also real numbers expressed as decimal fractions

- Double simply gives greater precision (it is a form of a floating point data type)

  float salary, pay_rate; /* defines two float variables */

  double bigger_salary; /* defines a larger ‘float’ value */

- Can also assign value after declaration:

  double dist_in_miles, Avagadro_number;

  Avagadro_number = 6.023e23;

  dist_in_miles = 14.638579237;

**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;

## COSTANTI E DICHIARAZIONE DI COSTANTI

**Oltre alle Costanti esplicite**

esprimono direttamente dei **valori**

| | |
|---|---|
| `23` | costante di tipo `int` |
| `3.1416` | costante di tipo `double` |
| `'A'` | costante di tipo `char` |

**Ci sono le Costanti simboliche**

- sono **nomi** simbolici che il programmatore adotta convenzionalmente per indicare dei valori prefissati
- hanno un **tipo** espresso implicitamente dal valore

**La dichiarazione di una costante**

- definisce il nome
- associa un valore (e tipo implicito)

*In C la «dichiarazione» (definizione) di costante può essere fatta tramite **direttiva al preprocessore** C*

```
#define nmaxp      10
#define vmax       150.0
#define FALSE      0
#define TRUE       1
```

*In compilazione, viene effettuata una **sostituzione letterale** del nome simbolico con il valore associato*.

E' anche possibile una dichiarazione del tipo:

```
const  <tipo> <nome> = <valore>;
```

# Symbolic Constants

```
const int max_buffer = 512;
const float e_approx = 2.718f;
```

- Much preferred to "magic" numbers
  - What does "512" mean?
  - Localized if need to change value
  - What if "512" means a number of things?!
- Memory is allocated
- Must be initialized (because ...)
- Cannot be changed, i.e., is "read-only"
- Preferred to `#defines`

# VARIABILI: ASSEGNAMENTO ed ESPRESSIONI

## Uso delle variabili

- per assegnare loro un valore (**istruzione di assegnamento**).
  il nome di una variabile a sinistra del simbolo «=« indica **dove** memorizzare il valore descritto a destra del simbolo «=«.

- come **operandi** in **espressioni**.
  il nome di una variabile indica che se ne deve usare il **valore** contenuto.

## ISTRUZIONI DI ASSEGNAMENTO

Indicano **l'operazione** che assegna un **valore** ad una **variabile**

**Sintassi C:**

```
<nome_variabile> = <espressione>;
```

- **<nome_variabile>** indica il nome della **variabile** a cui si vuole assegnare un valore, in sostituzione di quello precedentemente contenuto in essa ... è un indirizzo
- **=** è il simbolo di assegnamento
- **<espressione>** descrive come ottenere il **valore** da assegnare alla variabile

**Significato (semantica):**
si eseguono le operazioni descritte in **<espressione>**
il valore ottenuto viene inserito nella posizione di memoria indicata dalla variabile a sinistra del simbolo di assegnamento.

**Compatibilità tra i tipi**: il compilatore controlla la compatibilità tra tipi. In alcune situazioni, risolve la non compatibilità adottando delle regole di conversione implicita e automatica tra tipi.

- il valore generato da **<espressione>** dovrebbe essere dello stesso tipo della variabile da assegnare
- assegnamento tra tipi eterogenei (tipi aritmetici): all'atto dell'assegnamento, il valore di **<espressione>** viene «convertito» in un valore corrispondente appartenente al tipo della variabile da assegnare.

# ESPRESSIONI E OPERATORI

## Sintassi C

**`<espressione>`**: contiene identificatori (di variabili, di costanti, di funzioni), costanti esplicite, operatori, ( )

## Semantica

descrive il modo con cui ottenere dal valore degli operandi e dall'applicazione degli operatori (operazioni) il valore dell'espressione.

Nelle espressioni complesse la **sequenza di esecuzione delle operazioni** è dettata dalla

- **precedenza** predefinita degli operatori
- **forzatura** mediante l'uso delle parentesi tonde

## Operatori

- unari si applicano ad un solo operando
- binari si applicano a due operandi

## TIPI DI OPERATORI

***Operatori aritmetici*** (`+ - * / %`)

- operandi aritmetici e risultato aritmetico
- le operazioni sono eseguite in dipendenza del tipo degli operandi.

***Operatori di confronto*** (`>` `>=` `==` (eguale) `!=` (diverso))

- operandi entrambi dello stesso tipo qualsiasi
- risultato valore logico

**Valori logici**: in C non esiste un tipo «logico» che assume solo i valori False e True, ma per tale scopo viene usato il tipo `int`

- il valore FALSE (**falso**) è associato al valore **zero**
- il valore TRUE (**vero**) è associato ad ogni valore **diverso da zero**

***Operatori logici***

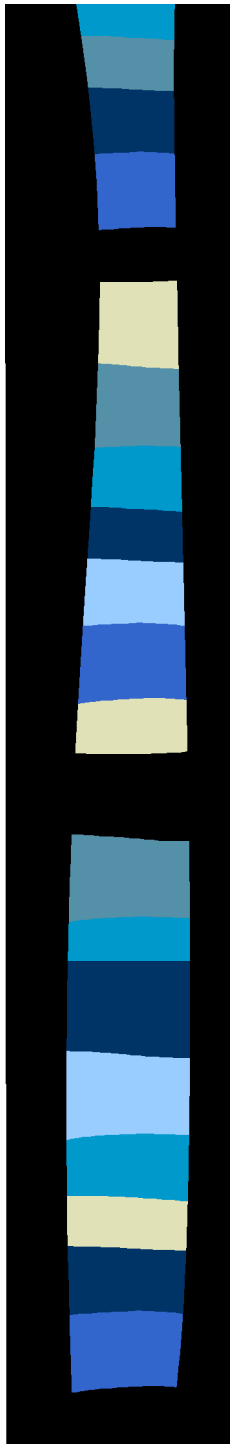
- operandi logici e risultato logico

`&&` (AND)
`||` (OR)
`!` (NOT) operatore unario (con un solo operando)

Esempio

```
(num > valmin) && (num <= valmax)
```

Questa espressione logica vale TRUE solo se il valore di **num** è compreso tra `valmin` (escluso) e `valmax` (compreso).

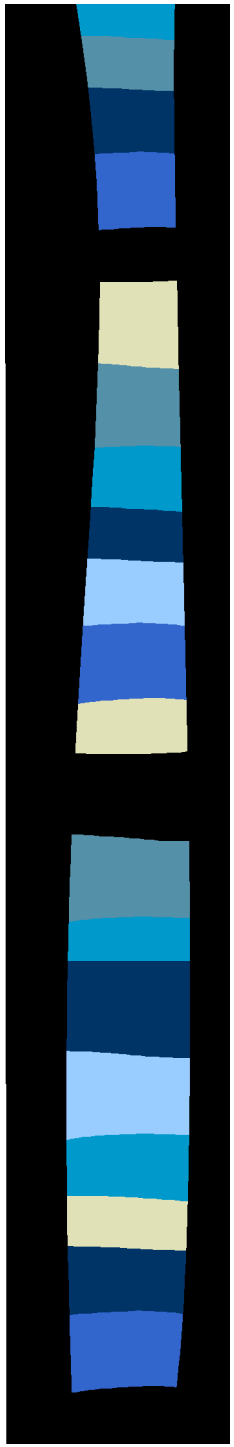


# Operatori aritmetici

- ++ → incremento unitario
- -- → decremento unitario
- * → moltiplicazione
- / → divisione
- % → resto di una divisione fra interi
- + → addizione
- - → sottrazione

# Esempi

- foo1() {
  int x=0, y= 10;
  x = y ++; x = ++ y;
  }
  valore finale → x=10, y=11 x=12, y=12 (*da sx verso dx*)
- foo2() {
  int x=16, y= 5, z=0;
  z = x / y; z = x % y;
  }
  valore finale → z=3 z=1



# Operatori relazionali

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

- **In C != 0 significa TRUE, ==0 significa FALSE.**

# 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 restituiti da espressioni relazionali vengano rappresentati attraverso gli interi {0,1} secondo la convenzione:

- **0 equivale a *falso***
- **1 equivale a *vero***

**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* |

## Operatori Logici in C

**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***.

## Definizione degli operatori logici in C:

| a | b | a&&b | a\|\|b | !a |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | $\neq 0$ | 0 | 1 | 1 |
| $\neq 0$ | 0 | 0 | 1 | 0 |
| $\neq 0$ | $\neq 0$ | 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

# Operatori logici (booleani)

- && → AND

| | | | | |
|---|---|---|---|---|
| | 0 | 0 | 1 | 1 |
| | 0 | 1 | 1 | 0 |
| AND | 0 | 0 | 1 | 0 |

- || → OR

| | | | | |
|---|---|---|---|---|
| | 0 | 0 | 1 | 1 |
| | 0 | 1 | 1 | 0 |
| OR | 0 | 1 | 1 | 1 |

- ! → NOT

| | | |
|---|---|---|
| | 0 | 1 |
| NOT | 1 | 0 |

# ...e anche ...Operatori sui Bit

- ~ → complemento a uno
- << → scorrimento a sinistra
- >> → scorrimento a destra
- & → AND
- ^ → XOR

| | | | | |
|---|---|---|---|---|
| | 0 | 0 | 1 | 1 |
| | 0 | 1 | 1 | 0 |
| XOR | 0 | 1 | 0 | 1 |

- | → OR

# Logical and Bitwise Operators

- Logical (true/false): `&&`, `||`, `!`

```
int n = 7 || 0;    // n == 1
```

- One's complement: `~`

```
n = ~4;    // all bits on except third to last
```

- Bitwise shifts: `<<`, `>>`

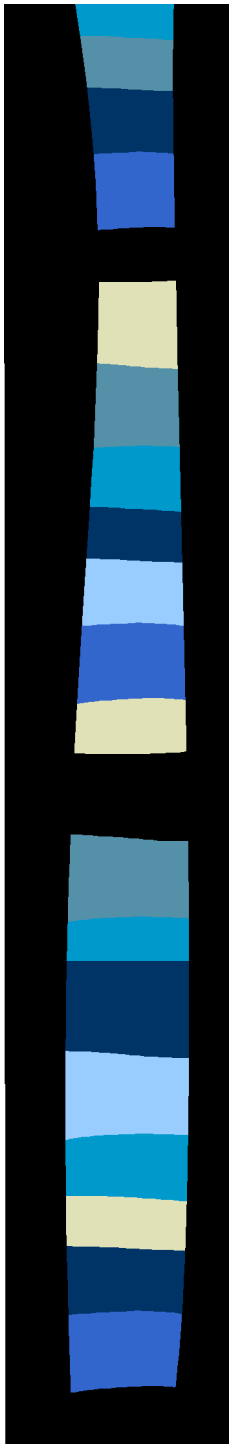
```
n = 12 >> 2;    // n == 3
```

  * standard set for unsigned, integral types only
  * bit-wrap, 0-fill or 1-fill is direction and compiler dependent

- Bitwise masks, logically bit-by-bit: `&`, `|`, `^`

```
n = n & ~017;    // zeros out last 4 bits
```

  Note: better than `n & 0177760` which assumes $\geq$ 16 bits



# Esempi

- ```
  foo1() {
      int x=0, y= 0, z=29;      → z=0001 1101
          x = z << 1;           → x=0011 1010
          x = z >> 1;           → x=0000 1110
          y = ~z; }             → y=1110 0010
  ```

- ```
  foo2() {
      int x=3, y= 5, z=0;       → x=0011, y=0101
          z = x & y;            → z=0001
          z = x | y; }          → z=0111
  ```