

# Identificatori

---

- Sequenza (di lunghezza maggiore o uguale a 1) di lettere e cifre che inizia obbligatoriamente con una lettera. E' ammesso anche l'utilizzo dell'underscore \_
- Maiuscole e minuscole sono **diverse** (il linguaggio C è **case-sensitive**)
- Es. corretti: pippo, pippo75, pi88ppo, num\_di\_elementi
- Es. scorretti: 75pippo, pi£ppo, num di elementi

# Parole riservate

---

- **auto, break, case, char, const, continue, default, define, do, double, else, enum, extern, float, for, goto, if, int, long, noalias, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while**
- Tali parole non possono essere utilizzate come identificatori (proprio perché riservate come parole chiavi del linguaggio)

# Commenti

---

- sequenze di caratteri racchiuse fra i delimitatori `/*` e `*/`
- possono estendersi anche su più righe
- non possono essere innestati

## Struttura di un programma C

---

- In prima battuta, la struttura di un programma C è definita nel modo seguente:

```
<programma> ::=  
    {<unità-di-traduzione>}  
    <main>  
    {<unità-di-traduzione>}
```

- *Intuitivamente un programma in C è definito da tre parti:*
  - una o più unità di traduzione,
  - il programma vero e proprio (*main*)
  - una o più unità di traduzione

# Struttura di un programma C

---

- La parte **<main>** è l'unica *obbligatoria*, definita come segue:

```
<main> ::=  
    main() {  
        [<dichiarazioni-e-definizioni>]  
        [<sequenza-istruzioni>]  
    }
```

- *Intuitivamente il main è definito dalla parola chiave **main()** seguita da parentesi graffe al cui interno troviamo*
  - *dichiarazioni e definizioni (opzionali)*
  - *una sequenza di istruzioni (opzionali)*

# Struttura di un programma C

---

## **<dichiarazioni-e-definizioni>**

- *introducono i nomi di costanti, variabili, tipi definiti dall'utente*

## **<sequenza-istruzioni>**

- *sequenza di frasi del linguaggio ognuna delle quali è un'istruzione*

- **Il `main()` è una particolare unità di traduzione (una funzione)**

# Tipi di dato

---

## Costanti e variabili

---

- I **dati** che si usano in un programma possono essere delle **costanti** o delle **variabili**:
  - Le prime possono essere immaginate come dei contenitori di una informazione stabilita una volta per tutte e non più modificabile
  - Le seconde come dei contenitori di una informazione che può essere modificata nel corso del programma
- *NOTA:*  
*Più avanti nel corso ritorneremo in maniera più precisa su questi concetti*

# Tipi di dato

---

- In C (come in tutti i linguaggi di programmazione) a ciascuna variabile e costante è associato anche il **TIPO**, ovvero **la classe di valori** che la costante o variabile può assumere nel corso dell'esecuzione del programma (e quindi **gli operatori applicabili** al valore in essa contenuto).
- L'associazione di un nome (di costante o di variabile) ad un tipo di dato **non cambia mai** durante l'esecuzione del programma.

# Tipi di dato

---

- Il concetto di *tipo di dato* viene introdotto quindi per raggiungere due obiettivi:
  - esprimere in modo sintetico
    - la rappresentazione dei dati in memoria
    - un insieme di operazioni ammissibili
  - permettere di *effettuare controlli statici* (al momento della compilazione) sulla **correttezza** del programma

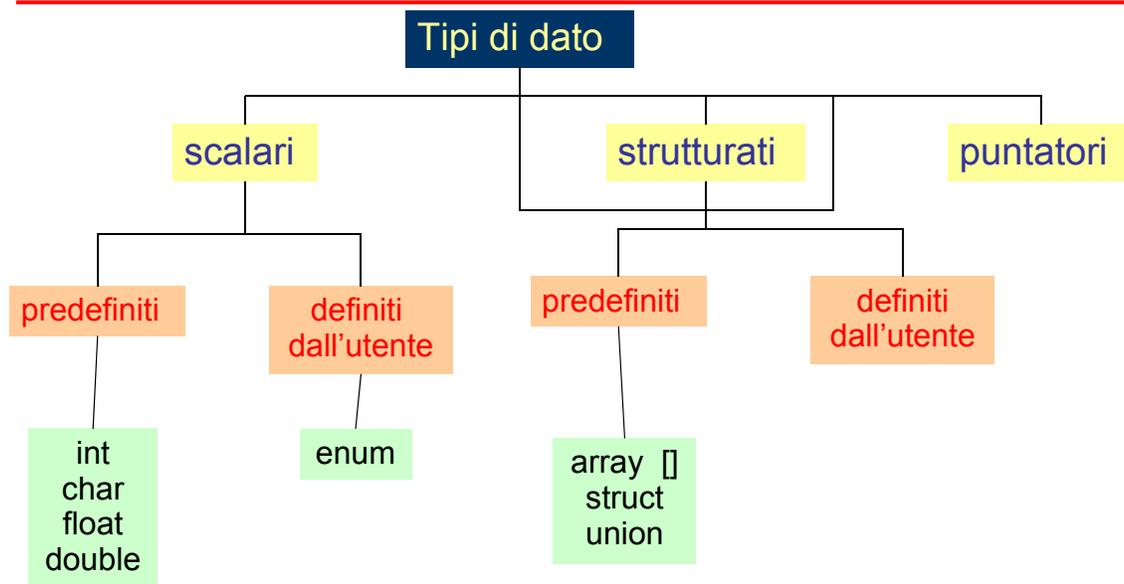
# Tipi di dato

---

Quali sono i tipi di dato ammissibili in C ?

## Tipi di dato in C

---



# Tipi di dato primitivi (predefiniti)

---

## 4 tipi di dato primitivi

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

## 4 qualificatori di tipo

- **signed**
- **unsigned**
- **short**
- **long**
- *signed* e *unsigned* possono essere applicati solo ai tipi *char* e *int*
- *short* può essere applicato solo a *int*
- *long* può essere applicato solo a *int* e a *double*

# Tipi di dato primitivi (predefiniti)

---

In sintesi, questa classificazione dà origine a 12 tipi di dati semplici predefiniti:

- char
- unsigned char
- signed char
- float
- double
- long double
- [signed] short [int]
- [signed] int
- [signed] long [int]
- unsigned short [int]
- unsigned [int]
- unsigned long [int]

# Significato dei qualificatori

---

- **short** e **long** condizionano lo spazio allocato dal compilatore per la memorizzazione delle variabili del tipo definito (*lo spazio effettivamente utilizzato dipende dalla macchina*)
- **signed** e **unsigned** condizionano l'uso che si può fare della memoria allocata
- In particolare, i qualificatori nelle loro combinazioni condizionano:
  - L'insieme dei valori assumibili da una variabile
  - Il valore massimo e minimo

## Tipo **int**

---

- Il tipo “**int**” è ben diverso dal tipo INTERO inteso in senso matematico dove  
INTERO      **Z**       $\{-\infty, \dots, -2, -1, 0, +1, +2, \dots, +\infty\}$
- Ovvero il tipo “**int**” ha un insieme di valori e di operazioni definibili su di essi limitato a priori:
  - L'insieme dei valori dipende dalla macchina
  - Normalmente il tipo “int” è memorizzato in una PAROLA DI MACCHINA (**WORD**), tipicamente lunga 16, 32 o 64 bit

# Tipo int

---

- Per esempio se la macchina è a 32 bit un compilatore può (non è detto che ciò avvenga) prevedere un'allocazione di memoria inferiore per uno SHORT INT:

16 bit  $[-2^{15}, 2^{15}-1]$  ovvero  $[-32768, +32767]$

... e superiore per un LONG INT:

32 bit  $[-2^{31}, 2^{31}-1]$  ovvero  $[-2147483648, +2147483647]$

- Ogni tentativo di assegnare un valore al di fuori del range ammissibile, produce un errore di **OVERFLOW** o **UNDERFLOW**

# Tipo int

---

- Tuttavia non si ha alcuna garanzia sulla dimensione di allocazione. È possibile solo affermare con sicurezza che per ogni compilatore:

Spazio allocato (short int)  $\leq$  Spazio allocato (int)  $\leq$  Spazio allocato (long int)

Ovvero:

$\text{sizeof}(\text{short int}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long int})$

# Tipicamente...

---

- Se una macchina ha una parola di 32 bit
  - SHORT INT 16 bit
  - INT 32 bit
  - LONG INT 32 bit
- Se una macchina ha una parola di 16 bit
  - SHORT INT 16 bit
  - INT 16 bit
  - LONG INT 32 bit
- Se una macchina ha una parola di 64 bit
  - SHORT INT 16/32 bit
  - INT 64 bit
  - LONG INT 64 bit
- Gli **unsigned int** rappresentano valori interi non negativi e quindi guadagnano un bit per ampliare il range

## Limiti

---

- I compilatori che seguono le raccomandazioni dello standard ANSI C mantengono all'interno dello header file `<limits.h>` l'indicazione del range di valori rappresentabili nei tipi interi.
- Ad esempio:
  - SHRT\_MAX 32.767
  - SHRT\_MIN -32.768
  - USHRT\_MAX 65.535
  - INT\_MAX 32.767
  - INT\_MIN -32.768
  - UINT\_MAX 65.535
  - LONG\_MAX 2.147.483.647
  - LONG\_MIN -2.147.483.648
  - ULONG\_MAX 4.294.967.295

# Operatori per il tipo int

---

- Al tipo **int** (e tipi ottenuti da questo mediante qualificazione) sono applicabili i seguenti operatori:

## Operatori aritmetici

+ - \* /

% (resto divisione intera)

## Risultato

int

int

## Operatori relazionali

== !=

< > <= >=

## Risultato

(0 se falso, ≠ 0 se vero)

(0 se falso, ≠ 0 se vero)

# Operatori per il tipo int

---

- + Addizione
- - Sottrazione
- \* Moltiplicazione
- / Divisione intera Es., 5/3=1
- % Modulo (resto dalla divisione intera) Es., 5%3 = 2
  
- == Operatore relazionale di uguaglianza  
(ATT.: diverso dal simbolo = che denota l'operazione di assegnamento!)
- != Operatore relazionale di diversità
- > Operatore relazionale di maggiore stretto
- < Operatore relazionale di minore stretto
- >= Operatore relazionale di maggiore-uguale
- <= Operatore relazionale di minore-uguale

# Operatori per il tipo int

---

- `n++`      Successivo di `n` (notazione postfissa)
- `++n`      Successivo di `n` (notazione prefissa)
  
- `n--`              Precedente di `n` (notazione postfissa)
- `--n`              Precedente di `n` (notazione prefissa)

## Tipi **float** e **double**

---

- Rappresentano (con *diversa precisione*) l'insieme dei numeri reali
  
- In realtà (*così come il tipo int e gli interi*) sono solo una **approssimazione dei numeri reali**,
  - sia come **precisione**
  - sia come **intervallo** di valori rappresentabili

# Tipi float e double

---

- Tipicamente:

- float 4 byte (32 bit)
- double 8 byte (64 bit)
- long double 10 byte (80 bit)

Tipo	Precisione	Valori
float	6 cifre decimali	$3.4 \cdot 10^{-38} \dots 3.4 \cdot 10^{+38}$
double	6 cifre decimali	$1.7 \cdot 10^{-308} \dots 1.7 \cdot 10^{+308}$

## Limiti

- Anche per i numeri float e double esiste una libreria **<float.h>** che indica i limiti nel range ammissibile
- **FLT\_MAX\_10\_EXP +38**  
il massimo valore di esponente per un float con base 10
- **DBL\_MAX\_10\_EXP +308**  
il massimo valore di esponente per un double con base 10
- **FLT\_EPSILON 1.19209290E-07**  
il più piccolo numero float tale che  $1.0+x > 1.0$
- **DBL\_EPSILON 2.22044...E-16**  
il più piccolo numero double tale che  $1.0+x > 1.0$
- **FLT\_MIN\_10\_EXP -37**  
il più piccolo valore di esponente che può essere rappresentato in un numero float con base 10

# Operatori per i tipi float e double

---

## Operatori aritmetici

+ - \* /

## Risultato

float o double

## Operatori relazionali

== !=

(0 se falso, ≠ 0 se vero)

< > <= >=

(0 se falso, ≠ 0 se vero)

# Operatori per i tipi float e double

---

- + Addizione
- - Sottrazione
- \* Moltiplicazione
- / Divisione reale  
(anche se si usa lo stesso simbolo, è diversa da quella intera)
  
- == Operatore relazionale di uguaglianza  
(ATT.: diverso dal simbolo = che denota l'operazione di assegnamento!)
- != Operatore relazionale di diversità
- > Operatore relazionale di maggiore stretto
- < Operatore relazionale di minore stretto
- >= Operatore relazionale di maggiore-uguale
- <= Operatore relazionale di minore-uguale

# Attenzione!

- 
- A causa della rappresentazione su di un numero finito di cifre, ci possono essere errori dovuti al **troncamento** o all'**arrotondamento** di alcune cifre decimali
  - Meglio evitare l'uso dell'operatore `==`
  - **I test di uguaglianza tra valori reali (in teoria uguali) potrebbero non essere verificati.**
  - Es: se  $x$  e  $y$  sono due variabili float, allora non è detto che risulti verificata la seguente `(x / y) * y == x`
  - Meglio utilizzare "**un margine accettabile di errore**":  
`(x == y) → (x <= y+epsilon) && (x >= y-epsilon)`  
(dove ad esempio si è posto: `epsilon=0.000001`)

## Tipo char

- 
- Nel linguaggio C (*a differenza di altri linguaggi*) il tipo **char** non denota un nuovo tipo, ma è **equivalente** al dominio dei valori interi rappresentati su di un byte (range 0-255)
  - Poiché i **char** sono codificati come interi su di un byte valgono gli stessi operatori degli **int**
  - Nelle operazioni, bisogna inoltre considerare che le tabelle ASCII dei caratteri occidentali rispettano il seguente ordinamento (detto lessicografico):  
`'0' < '1' < '2' < ... < '9' < ... < 'A' < 'B' < 'C' < ... < 'Z' < ... < 'a' < 'b' < 'c' < ... < 'z'`
  - Vale la regola di **prossimità** all'interno di ciascuna di queste tre classi ovvero, il carattere successivo di 'a' è 'b', di 'B' è 'C', di '4' è '5', ecc.
  - **Tuttavia le tre classi non sono contigue!**

# Tipo char

---

Tipo	Dimensione	Valori
<b>char</b>	1 byte	-127 .. +128
<b>unsigned char</b>	1 byte	0 .. 255

# Tipo *booleano*

---

- In altri linguaggi di programmazione i valori logici hanno un loro tipo predefinito

**Nel linguaggio C, invece, non esiste un tipo booleano come tipo a sé stante!**

- Pertanto si sfrutta il tipo `int`
  - Il valore 0 (zero) indica FALSO
  - Ogni valore diverso da 0 indica VERO  
(per convenzione, talvolta, si usa 1 per denotare “vero”, ma bisogna ricordare che è solo una convenzione)

# Costanti e variabili

---

## Costanti

---

Una **costante** è una astrazione simbolica di un valore

- **interi** (in varie basi di rappresentazione)

<i>base</i>	<i>2 byte</i>	<i>4 byte</i>
decimale	<b>12</b>	<b>70000, 12L</b>
ottale	<b>014</b>	<b>0210560</b>
esadecimale	<b>0xFF</b>	<b>0x11170</b>

- **reali**

- in doppia precisione    **24.0**    **2.4E1**    **240.0E-1**
- in singola precisione    **24.0F**    **2.4E1F**    **240.0E-1F**

# Costanti

---

- **caratteri**

→ singolo carattere racchiuso fra apici

'A' 'C' '6'

→ caratteri speciali:

'\n' '\t' '\"' '\\' '\"'

## Costanti - stringhe

---

- Una **stringa** è una *sequenza di caratteri* delimitata da virgolette

**"ciao" "Hello\n"**

- In C le stringhe sono semplici sequenze di caratteri di cui l'ultimo, *sempre presente in modo implicito*, è '\0'

**"ciao" = {'c', 'i', 'a', 'o', '\0'}**

# Dichiarazione di costanti

---

- Una dichiarazione di **costante** associa permanentemente un valore ad un identificatore
- Si attribuisce ad ogni costante un **tipo**
- Si utilizza la parola riservata **const**
  
- Sintassi:  
`const <tipo> <identificatore> = <espr> ;`
  
- Esempio
  - `const int N = 100;`
  - `const float pigreco = 3.1415;`
  - `const char sim = 'A';`

# Variabili

---

- Una **variabile** è un'astrazione della *cella di memoria*
- Formalmente, è un simbolo *associato a un indirizzo fisico (L-value) ...*

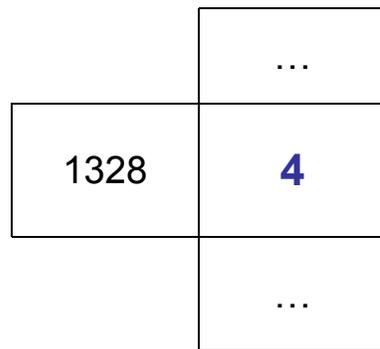
<i>simbolo</i>	<i>indirizzo</i>
x	1328

Perciò, L-value di x è 1328 (**fisso e immutabile!**)

# Variabili

---

- ... che denota un valore (**R-value**)



- ... e R-value di x è *attualmente* 4 (può cambiare!)

## Dichiarazione di variabili

---

- Una variabile utilizzata in un programma deve essere dichiarata
- La dichiarazione è composta da
  - il **nome della variabile** (identificatore)
  - il *tipo* dei valori (R-value) che possono essere denotati alla variabile

# Dichiarazione di variabili

---

- Scopo:

- Elencare tutte le variabili che saranno utilizzate nella parte esecutiva
- Attribuire ad ogni variabile un tipo
- È possibile raggruppare le dichiarazioni di più variabili dello stesso tipo in una lista separata da virgole

# Dichiarazione di variabili

---

- Sintassi:

`<tipo> <identificatore>;`

- Esempi:

- `int x;`            `/* x deve denotare un valore intero */`
- `float y;`        `/* y deve denotare un valore reale */`
- `char ch;`        `/* ch deve denotare un carattere */`
- `int a, b, c;`
- `float z, w;`

# Inizializzazione di variabili

---

- Contestualmente alla *definizione* è possibile *specificare un valore iniziale* per una variabile
- Inizializzazione di una variabile:  
`<tipo> <identificatore> = <espr> ;`
- Esempio
  - `int x = 32;`
  - `double speed = 124.6;`

# Caratteristiche delle variabili

---

- **campo d'azione (scope)**: è la parte di programma in cui la variabile è nota e può essere manipolata
  - in C, Pascal: determinabile *staticamente*
  - in LISP: determinabile *dinamicamente*
- **tipo**: specifica la *classe di valori* che la variabile può assumere (e quindi gli operatori applicabili)

# Caratteristiche delle variabili

---

- **tempo di vita:** è l'intervallo di tempo in cui rimane valida l'associazione simbolo/indirizzo fisico (L-VALUE)
  - in FORTRAN: allocazione *statica*
  - in C, Pascal: allocazione *dinamica*
- **valore:** è rappresentato (secondo la codifica adottata) nell'area di memoria associata alla variabile

## Assegnamento di una variabile

---

# Istruzione di assegnamento

---

- Ad una variabile può essere assegnato un valore nel corso del programma e non solo all'atto della inizializzazione
- L'**istruzione di assegnamento** viene utilizzata per assegnare ad una variabile (non ad una costante) il **valore di una espressione**
- Denotata mediante il simbolo **=**  
(diverso dall'operatore relazionale di uguaglianza che è denotato col simbolo ==)

# Istruzione di assegnamento

---

## SINTASSI:

`<identificatore> = <espr> ;`

## Esempi:

`a = 34;`

`b = 34 + a*4;`

# Istruzione di assegnamento

## Esempio



L'esecuzione di una **dichiarazione** provoca l'allocazione di uno spazio in memoria equivalente a quello necessario a contenere un dato del tipo specificato

N = 238;



L'esecuzione di un **assegnamento** provoca l'inserimento nello spazio relativo alla variabile del valore indicato a destra del simbolo =

# Istruzione di assegnamento

- L'esecuzione di un'istruzione di assegnamento comporta innanzitutto la **valutazione dell'espressione** (a destra dell'assegnamento)
- Es:  

```
c = 3;           /* adesso c vale 3 */  
d = (c+7)/2 -4; /* adesso d vale 1 */  
d = (d-c)*2;    /* adesso d vale -4 */
```
- Dopodiché, si **inserisce il valore risultante nella locazione di memoria** relativa alla variabile (posta a sinistra dell'assegnamento)

## Operatori di assegnamento compatti

---

- Il C introduce una *forma particolare di assegnamento* che **ingloba anche un'operazione**:

*<identificatore>* **OP=** *<espressione>*

è **quasi equivalente** a:

*<identificatore>* = *<identificatore>* **OP** *< espressione>*

dove **OP** indica un operatore (ad esempio: **+**, **-**, **\***, **/**, **%**,...).

---

## Operatori di assegnamento compatti

---

$a = a+b;$                        $\rightarrow$                        $a += b;$

$a = a-b;$                        $\rightarrow$                        $a -= b;$

$a = a*b;$                        $\rightarrow$                        $a *= b;$

$a = a/b;$                        $\rightarrow$                        $a /= b;$

$a = a\%b;$                        $\rightarrow$                        $a \% = b;$