

# Lezione 9

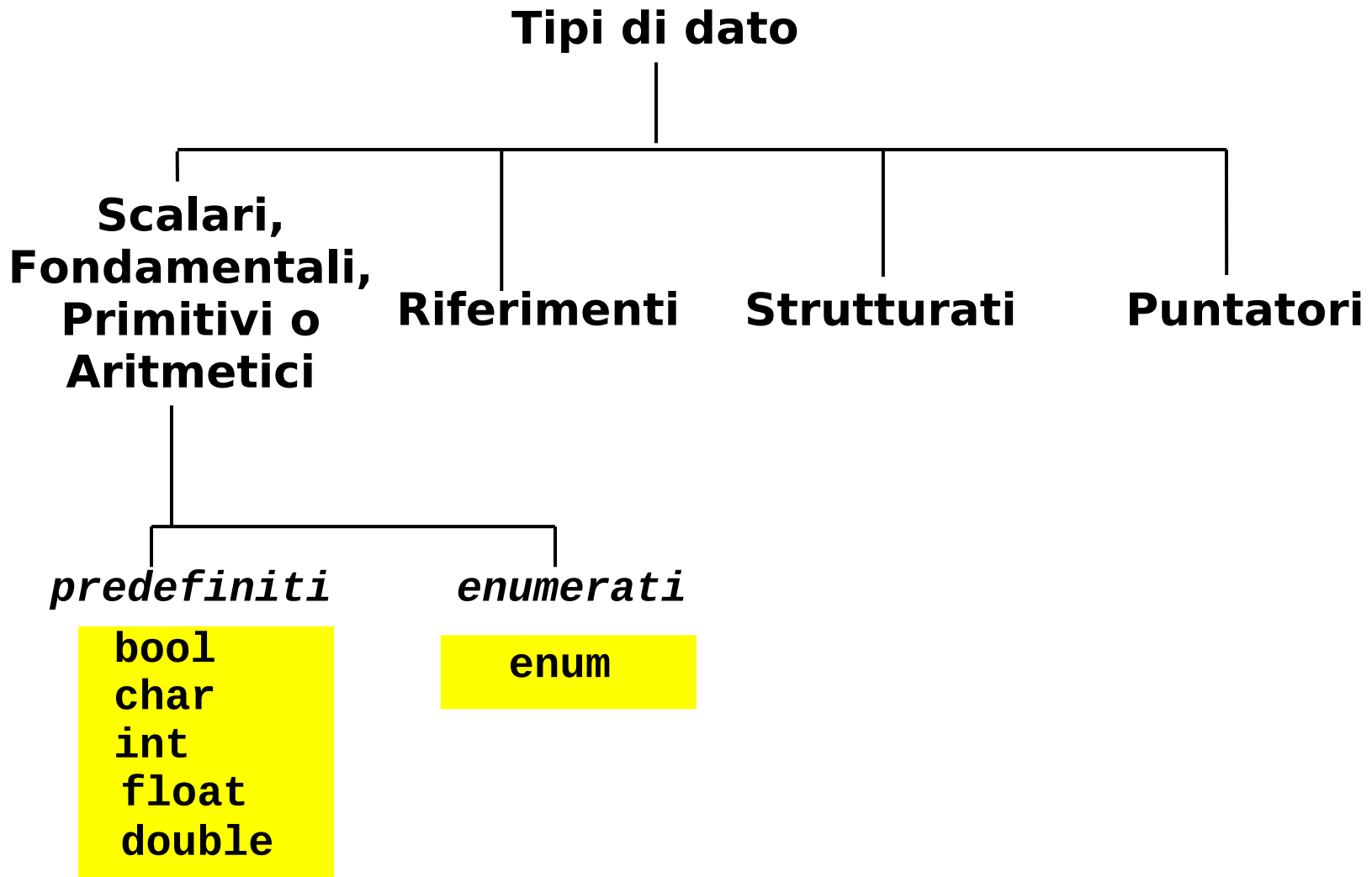
---

Compendio booleani

Tipo char

Conversioni esplicite

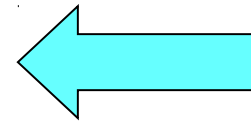
# Tipi di dato



# Tipi di dato primitivi

---

- **Numeri interi (int)**
  - Già trattati
- **Compendio Valori logici**
  - Corto circuito logico
  - Espressione condizionale
- **Caratteri (char)**
- **Conversioni di tipo esplicite**



# Corto-circuito 1/3

---

- Si dice che un operatore logico binario è **valutato in corto circuito** se
  - il suo secondo operando non è valutato se il valore del primo operando è sufficiente a stabilire il risultato
- In C/C++ sono valutati in **corto-circuito** gli operatori logici **&&** e **||**

# Corto-circuito 2/3

---

- Esempi:

`false && x`

Il valore del primo operando è sufficiente per stabilire che l'espressione è falsa, quindi il secondo operando **non è valutato**

`true || f(x)`

Il valore del primo operando è sufficiente per stabilire che l'espressione è vera, quindi il secondo operando **non è valutato**  
Di conseguenza **f(x) non è invocata**

`22 || x`

Il valore del primo operando è sufficiente per stabilire che l'espressione è vera, quindi il secondo operando **non è valutato**

# Corto-circuito 3/3

- Ricordiamo che `&&` e `||` sono associativi a sinistra, per cui, per esempio:

`a && b && c == (a && b) && c`

`a || b || c == (a || b) || c`

- Ne segue che:

`a && b && c`

Se `a && b` è falso, il secondo operando del secondo `&&` (ossia `c`) non viene valutato

`a || b || c`

Se `a || b` è vero, il secondo operando del secondo `||` (ossia `c`) non viene valutato

- Questo esempio con 3 termini si può banalmente generalizzare al caso di  $n$  termini

- Cosa stampa il seguente programma?

```
bool fun() {  
    cout<<"fun invocata"<<endl ;  
    return true ;  
}  
  
main()  
{  
    bool a = true ;  
    if (a && fun())  
        cout<<"programma terminato"<<endl ;  
}
```

- Stampa:

**fun invocata**  
**programma terminato**

- Perché è necessario invocare **fun** per determinare il valore dell'espressione condizionale



- Cosa stampa il seguente programma?

```
bool fun() {
    cout<<"fun invocata"<<endl ;
    return true ;
}

main()
{
    bool a = true ;
    if (a || fun())
        cout<<"programma terminato"<<endl ;
}
```

- Stampa:  
**programma terminato**
- Perché **non** è necessario invocare **fun** per determinare il valore dell'espressione condizionale

# Espressione condizionale

*<condizione> ? <espressione1> : <espressione2>*

- Il valore risultante è quello di *<espressione1>* oppure quello di *<espressione2>*
  - Dipende dal valore dell'espressione *<condizione>*:
    - se *<condizione>* è vera, si usa *<espressione1>*
    - se *<condizione>* è falsa, si usa *<espressione2>*
- Esempi:
  - `3 ? 10 : 20` // vale sempre 10
  - `x ? 10 : 20` // vale 10 se x è vero, 20 altrimenti
  - `(x>y) ? x : y` // vale il maggiore fra x ed y

# Sintesi priorità degli operatori

Fattori

Termini

Assegnamento

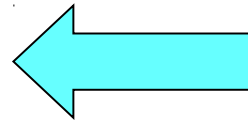
!	++	--	
*	/	%	
	+	-	
>	>=	<	<=
	==	!=	
	&&		
	?:		
	=		

- Svolgere gli esercizi *oper\_cond.cc* e, per casa, *oper\_cond2.cc* della settima esercitazione

# Tipi di dato primitivi

---

- **Numeri interi (int)**
  - Già trattati
- **Compendio Valori logici**
  - Corto circuito logico
  - Espressione condizionale
- **Caratteri (char)**
- **Conversioni di tipo esplicite**



# Prima di iniziare ...

---

- ... un esempio di quello che si può fare con l'opportuna conoscenza del tipo **char**
- <http://asciimation.co.nz/>
- Non vi preoccupate, cominceremo da qualcosa di più semplice ...

# Tipo carattere: char

---

- Rappresenta l'insieme dei caratteri utilizzabili in accordo allo standard del linguaggio C/C++
- Costanti letterali carattere
  - Dato un carattere, la corrispondente costante letterale carattere si ottiene racchiudendo il carattere tra **singoli apici**

'a' 'b' 'A' '2' '@'

- Diverso dal caso dei letterali numerici, che non andavano corredati da simboli aggiuntivi all'inizio ed alla fine



# Caratteri speciali

---

Mediante le costanti letterali carattere si possono però denotare anche:

- caratteri speciali:

'\n'	A capo	
'\t'	Tabulazione	
'\''	Singolo apice	'
'\\'	Backslash	\
'\"'	Doppi apici	"

# Stampa di un carattere 1/2

---

- Se scriviamo

```
cout<<'a'<<endl ;
```

- Cosa stampa?
  - Provare per scoprirlo

# Stampa di un carattere 2/2

---

- Stampa il carattere **a**
- Ne deduciamo che, in qualche modo nel programma è stato memorizzato tale carattere
- Lo stesso carattere è stato poi passato in qualche modo dal programma al terminale
- Ma da quanto abbiamo imparato finora, nella memoria di un elaboratore è possibile memorizzare solo numeri ...

# Rappresentazione caratteri 1/2

---

- Sappiamo che la memoria è fatta solo di locazioni contenenti (solo) numeri
- Come memorizzare un carattere in una locazione che può contenere solo un numero?
- Un problema simile si aveva nelle trasmissioni telegrafiche
  - Si potevano trasmettere solo segnali elettrici
  - Come avevano risolto il problema?

# Rappresentazione caratteri 2/3

---

- Con il codice Morse
  - Associando cioè ad ogni carattere una determinata sequenza di segnali di diversa durata

# Rappresentazione caratteri 3/3

---

- Possibile soluzione per memorizzare caratteri:
  - Associare per convenzione **un numero intero, ossia un codice, diverso a ciascun carattere**
  - Per memorizzare un carattere, si può memorizzare di fatto il numero intero, ossia il codice, che lo rappresenta

# Esempio 1/2

---

- Consideriamo solo tre caratteri:  $a$ ,  $b$  e  $c$
- Decidiamo quale numero intero (codice) associare a ciascun carattere, ad esempio

$a$      1

$b$      2

$c$      3

- Per memorizzare, per esempio, il carattere  $b$  in una locazione di memoria, vi memorizziamo il numero intero 2

# Esempio 2/2

---

- Se sappiamo che in una data locazione è memorizzato un carattere, allora
  - quando abbiamo bisogno di sapere quale carattere contiene
  - controlliamo il numero contenuto nella locazione e dal numero risaliamo al carattere
  - Nel nostro esempio:  $1 \rightarrow a$ ,  $2 \rightarrow b$ ,  $3 \rightarrow c$



# Codifica ASCII 1/2

---

- Generalmente, si utilizza il codice ASCII
- E' una codifica che, nella *forma estesa*, utilizza 1 byte, per cui vi sono 256 codici carattere possibili
- Vi è anche la *forma ristretta* su 7 bit, nel qual caso l'insieme di codici carattere si riduce a 128

# Codifica ASCII 2/2

Codice (in base 10)	Carattere
... (0-31, caratteri di controllo)	
32	<spazio>
33	!
34	"
35	#
...	
48	0
49	1
...	
65	A
66	B
67	C
...	
97	a
...	

La tabella completa è reperibile facilmente in rete, e si trova tipicamente anche nell'Appendice dei libri e manuali sui linguaggi di programmazione

# Tipo char

---

- Nel linguaggio C/C++ il tipo **char** non denota un nuovo tipo in senso stretto, ma è di fatto l'insieme dei valori interi rappresentabili (tipicamente) su di un *byte*
- Il tipo **char** contiene quindi, di fatto, un sottoinsieme abbastanza piccolo di numeri interi

# Intervallo di valori 1/2

---

<i>Tipo</i>	<i>Dimensione</i>	<i>Intervallo valori</i>
<b>char</b>	1 byte	-128 .. 127 (se considerato con segno)  oppure  0 .. 255 (se considerato senza segno)
<b>unsigned char</b>	1 byte	0 .. 255

# Intervallo di valori 2/2

---

- Lo standard non specifica se **char** deve essere considerato con segno o senza
  - La cosa può variare da una macchina all'altra
- Invece **unsigned char** è **sempre senza segno**

# Domanda

---

- Quindi, cosa denota una costante carattere?

# Contenuto costanti carattere

---

- Quindi, le costanti carattere **non denotano altro che numeri interi**
  - Scrivere una costante carattere equivale a scrivere il numero corrispondente al codice ASCII del carattere
    - Ad esempio, scrivere 'a' è equivalente a scrivere 97
  - **Però una costante carattere ha anche associato un tipo**
    - ossia il tipo char

# Stampa di un carattere

---

- Di conseguenza, se scriviamo

```
cout<<'a'<<endl ;
```

abbiamo passato il valore 97 al **cout**

- Ma abbiamo scoperto che **non stampa 97**
- Bensì il carattere **a**
- Come mai?



- Perché l'operatore `<<` ha dedotto dal tipo (**char**) cosa fare!
- Essendo la costante di tipo **char** l'operatore stampa effettivamente il carattere corrispondente alla costante

# Domanda

---

- Dato il seguente frammento di codice:

```
char a = 'd' ;  
cout<<'a'<<endl ;  
cout<<a<<endl ;
```

- Cosa stampa la seconda istruzione?
- Cosa stampa la terza istruzione?

- La seconda istruzione stampa **a**
- La terza istruzione stampa **d**
  - Ossia il contenuto della variabile **a**

- Svolgere *leggi\_stampa\_char.cc* della settimana esercitazione

# Domanda

---

- Che differenza c'è tra quello che stampano le seguenti due istruzioni
  - Supponendo che l'operatore di uscita sia configurato per stampare i numeri in base 10

```
cout<<'1'<<'2' ;
```

```
cout<<12 ;
```

- Nessuna
- Entrambe stampano la sequenza di due caratteri **12** su *stdout*

# Ordinamento 1/2

---

- I caratteri sono ordinati
- In particolare rispettano il seguente ordinamento (detto lessicografico) per ciascuna delle tre classi (cifre, lettere minuscole, lettere maiuscole):

'0' < '1' < '2' < ... < '9'

'A' < 'B' < 'C' < ... < 'Z'

'a' < 'b' < 'c' < ... < 'z'

# Ordinamento 2/2

---

- Ma qual è l'ordinamento **tra** le tre classi
  - Per esempio '1' < 'a'?
- Non è definito!
- Tra le tre classi lo standard del linguaggio non prevede nessuna garanzia di quale dei possibili ordinamenti viene adottato
  - La codifica ASCII ha il suo ordinamento, ma quale codifica deve/può essere utilizzata sulla macchina non è definito dallo standard
  - Lo standard lascia libera la scelta della codifica, purché sia rispettato solo l'ordinamento **all'interno** delle classi
  - L'effettivo ordinamento **tra** le classi dipenderà dalla codifica utilizzata sulla macchina su cui gira il programma



# Cambio di argomento

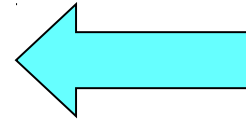
---

- Per poter completare il discorso sul tipo **char**, occorre prima introdurre il concetto di conversione esplicita di tipo

# Tipi di dato primitivi

---

- **Numeri interi (int)**
  - Già trattati
- **Compendio Valori logici**
  - Corto circuito logico
  - Espressione condizionale
- **Caratteri (char)**
- **Conversioni di tipo esplicite**



# Conversioni di tipo

---

- Dato il valore di una costante, di una variabile, di una funzione o in generale di una espressione
  - Tale valore ha anche associato un tipo
  - Esempio:
    - 2 è di tipo **int**
    - 'a' è di tipo **char**
    - 2<3 è di tipo **bool**
- Esiste un modo per convertire un valore, appartenente ad un certo tipo, nel valore corrispondente in un altro tipo?
  - Sì, uno dei modi è mediante una **conversione esplicita**
  - Esempio: da 97 di tipo **int** a 97 di tipo **char** (ossia la costante carattere 'a')

# Conversioni esplicite 1/2

- Tre forme (negli esempi si assuma, ad esempio, che **a** sia una variabile/costante di tipo **char** precedentemente definita):

- **Cast** (C/C++)

*<tipo\_di\_destinazione>* <espressione>

Esempi:      **d = (int) a;**  
              **fun((int) a) ;**

- **Notazione funzionale** (C/C++)

*<tipo\_di\_destinazione>*(<espressione>)

Esempi:      **d = int(a);**  
              **fun(int(a)) ;**

- **Operatore `static_cast`** (solo C++)

**static\_cast**<*<tipo\_di\_destinazione>*>(<espressione>)

Esempi:      **d = static\_cast<int>(a);**  
              **fun(static\_cast<int>(a)) ;**

# Conversioni esplicite 2/2

---

- In tutti e tre i casi, il valore dell'espressione è convertito nel corrispondente valore di tipo *<tipo\_di\_destinazione>*, quale che sia il tipo del valore dell'espressione

# Operatore `static_cast`

---

- L'uso dell'operatore `static_cast` comporta una notazione più pesante rispetto agli altri due
- La cosa è voluta
  - Le conversioni di tipo sono spesso pericolose
  - Bisogna utilizzarle solo quando non si riesce a farne a meno senza complicare troppo il programma
  - Un notazione pesante le fa notare di più
- Se si usa lo `static_cast` il compilatore usa regole più rigide
  - Programma più sicuro
- Al contrario con gli altri due metodi si ha piena libertà (di sbagliare senza essere aiutati dal compilatore ...)

```
int i = 100 ;  
char a = static_cast<char>(i) ;
```

- Supponendo che il valore 100 sia rappresentabile mediante il tipo **char**, e che quindi non vi siano problemi di *overflow*
- Che cosa viene memorizzato nell'oggetto di tipo **char**?

- Esattamente il valore 100
- Ma stavolta il valore sarà di tipo **char**
- Similmente, dopo le istruzioni:

```
char a = 'd' ; // codice ASCII 100  
int i = static_cast<int>(a) ;
```

nella variabile **i** sarà memorizzato il valore 100, ma il tipo sarà **int**



# Domanda

---

- Supponendo che il codice del carattere 'a' sia 97, che differenza di significato c'è tra le due seguenti inizializzazioni?

```
char b = 'a' ;
```

oppure

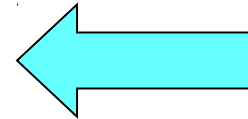
```
char b = static_cast<char>(97) ;
```

- Nessuna, sono perfettamente equivalenti!

# Tipi di dato primitivi

---

- **Numeri interi (int)**
  - Già trattati
- **Compendio Valori logici**
  - Corto circuito logico
  - Espressione condizionale
- **Caratteri (char)**
- **Conversioni di tipo esplicite**



- Dalla settima esercitazione
  - *codice\_car.cc*
  - *car\_codice.cc*
  - *traccia\_car\_codici\_immediato.cc*

# Operazioni

- Sono applicabili tutti gli operatori visti per il tipo `int`
- Pertanto, si può scrivere:

<code>'x' / 'A'</code>	equivale a	<code>120 / 65</code>	uguale a: <code>1</code>
<code>'R' &lt; 'A'</code>	equivale a	<code>82 &lt; 65</code>	uguale a: <code>false</code> (0 in C)
<code>'x' - '4'</code>	equivale a	<code>120 - 52</code>	uguale a: <code>68</code> ( <code>=='D'</code> )
<code>'x' - 4</code>	equivale a	<code>120 - 4</code>	uguale a: <code>116</code> ( <code>=='t'</code> )

- Svolgere la settima esercitazione fino alla prova di programmazione inclusa

# Caratteri speciali

---

- Come costante letterale carattere si può specificare direttamente il codice del carattere, con due diverse possibili notazioni:
  - '\nnn'                      Numero ottale di tre cifre
  - '\0xhhh'                    Numero esadecimale di tre cifre
- Esempi:
  - '\041'                      '\0'                      '\0xfa'
- Questa possibilità va usata con molta attenzione, alla luce di quanto spiegato nella prossima slide

# Portabilità 1/2

---

- Ci interessa la forma in cui il numero è memorizzato per poterci lavorare?
  - No, noi lo usiamo semplicemente come un numero intero, pensa a tutto il compilatore
- Soprattutto: se il codice che scriviamo non fa nessuna assunzione su come sono rappresentati i numeri, allora funzionerà su macchine diverse anche se su tali macchine i numeri sono rappresentati in modo diverso
  - Se invece un certo il codice di un programma fa affidamento sul fatto che i numeri sono rappresentati in un certo modo, allora, quando eseguito su una macchina in cui i numeri non sono rappresentati in quel modo, quel programma non funziona più



# Portabilità 2/2

---

- Nel primo caso si dice che il codice è **portabile** tra diverse macchine (architetture), nel secondo caso si dice invece che il codice **non è portabile**
- Lo stesso accade per i codici dei caratteri
  - Dobbiamo scrivere programmi che non facciano assunzioni su quale codifica è utilizzata, altrimenti cambiando codifica i nostri programmi non funzionano più correttamente!
- E' sensato scrivere programmi non portabili solo quando non vi è altra alternativa per il particolare problema da risolvere o per i vincoli temporali imposti
  - In tutti gli altri casi si è fatto semplicemente un cattivo lavoro se si è scritto un programma non portabile

# Esercizio per casa

---

- Esercizio **propedeutico per le prove d'esame**
- Scrivere una funzione che, dato un carattere passato in ingresso (come parametro formale), restituisca il carattere stesso se non è una lettera minuscola, altrimenti restituisca il corrispondente carattere maiuscolo
- Prima di definire il corpo della funzione, scrivere un programma che, usando SOLO tale funzione, legga un carattere da *stdin* e, se minuscolo, lo ristampi in maiuscolo, altrimenti comunichi che il carattere non è minuscolo
  - Adottiamo cioè, per esercizio, un approccio *top-down*

# Specifiche della funzione

---

- Per adottare in modo efficace l'approccio top-down bisogna definire in modo esatto cosa va in ingresso alla funzione e cosa la funzione restituisce
  - Scriviamo quindi solo la dichiarazione della funzione
  - Inseriamo i dettagli sul comportamento della funzione sotto forma di commenti all'intestazione della funzione stessa

# Prima parte

---

```
/*
 * Dato il carattere in ingresso c restituisce il maiuscolo
 * di c utilizzando solo le proprietà di ordinamento dei
 * codici dei caratteri.
 * Assunzione: se c non è minuscolo, ritorna il
 * carattere inalterato.
 */
char maiuscolo(char c);

main() {
    char minus, maius;
    cin>>minus;
    maius = maiuscolo (minus);
    if (minus==maius)
        cout<<"Il carattere "<<minus
            <<" non è minuscolo"<<endl;
    else
        cout<<"Minuscolo = "<<minus<<" - Maiuscolo = "
            <<maius<<endl;
}
```

# Bozza di algoritmo funzione

---

- Se il parametro formale **c** non contiene una lettera minuscola, restituisci il carattere senza alcuna modifica
- Altrimenti, calcola il corrispondente carattere maiuscolo, sfruttando le proprietà di ordinamento della codifica dei caratteri ASCII:
  - ogni carattere è associato ad un valore intero
    - Che noi assumiamo di **NON CONOSCERE**
  - le lettere da 'A' a 'Z' sono in ordine alfabetico
  - le lettere da 'a' a 'z' sono in ordine alfabetico

# Funzione

---

```
/*
 * Dato il carattere in ingresso c restituisce il maiuscolo
 * di c utilizzando solo le proprietà di ordinamento dei
 * codici dei caratteri.
 * Assunzione: se c non è minuscolo, ritorna il
 * carattere inalterato.
 */
char maiuscolo(char c)
{
    if (c < 'a' || c > 'z')
        return c;
    return c - 'a' + 'A';
}
```