
Inizializzazione delle Variabili

All'atto della dichiarazione di una variabile, è possibile assegnare un valore alla variabile stessa, mediante un'estensione della dichiarazione così fatta:

Tipo NomeVariabile = costante;

Questa inizializzazione si traduce in due fatti diversi a seconda della collocazione della variabile che stiamo dichiarando:

- Se la variabile è una **variabile globale**, il valore viene assegnato alla variabile prima di cominciare l'esecuzione del programma,
- Se la variabile è una **variabile locale** l'inizializzazione viene effettuata nella posizione dello stack in cui viene posizionata la variabile locale quando si entra nel blocco a cui appartiene, e questa inizializzazione viene ripetuta ogni volta che si entra in quel blocco. Fa **eccezione** il caso in cui la **variabile locale** sia **preceduta dallo** specificatore **static**, nel qual caso la variabile non viene messa sullo stack ma in una zona dati, e l'inizializzazione viene effettuata una ed una sola volta, nel momento in cui il controllo entra per la prima volta nella funzione in cui la variabile è stata dichiarata.
- Se la **variabile è un parametro formale** di una funzione, **non può essere inizializzata**.

esempi:

```
int     i = 0;          i inizializzato a zero
double  f = 13.7;       f inizializzato a 13.7

int  j=2, k, m=3;       j inizializzato a zero, k non inizializzato,
                        m inizializzato a 3

#define  NUM 10
int  n = NUM;           n inizializzato a 10

int  p = 0xFF;          p inizializzato a 255
int  q = 011;           q inizializzato a 9   (attenzione, costante ottale)
```

Costanti

Le costanti sono entità che non possono essere modificate durante l'esecuzione del programma, ma mantengono invariato il loro valore per tutta la durata del programma.
Le costanti possono essere:

1. **scritte direttamente nel programma** scrivendo il valore che interessa.
   Ad es. i = 1; f = 137.12; k = 0xFF;
2. **indicate mediante un simbolo tramite una direttiva al preprocessore #define** nel qual caso il preprocessore, sostituendo al simbolo il valore, ci riporta al caso precedente,
   Ad es. #define SIZE 10
   int i = SIZE;
   viene tradotto dal preprocessore in
   int i = 10;
3. **indicate mediante un simbolo** (tramite la keyword **const**) che viene inizializzato al valore che interessa e mantenuto in una locazione di memoria come una variabile, ma che non può essere modificato.
   La dichiarazione di un dato di tipo costante deve essere così:
   const Tipo NomeVariabile = costante;
   oppure
   Tipo const NomeVariabile = costante;
   Ad es. const int i = 10;
   Dichiarare una costante mediante la const impone al compilatore di associare il simbolo ad una certa locazione di memoria separata dal resto dei dati, e di mantenerne in questa locazione il valore assegnato in via di inizializzazione.

Comunque, in tutti i 3 casi, resta sempre il problema di come scrivere un certo valore costante (numerico intero, numerico in virgola mobile, stringa) all'interno in una certa istruzione. Il formato varia al variare del tipo di dato.

Esiste inoltre un caso particolare di uso delle costanti che sarà descritto a proposito delle funzioni e dei loro parametri formali, e che serve a dichiarare che un parametro formale non è modificabile.

Costanti Numeriche Intere

Le costanti numeriche intere: sono espresse da numeri senza componente frazionaria, possono essere preceduti eventualmente da un **segno** (+ o -), e possono essere espresse in notazione decimale (base 10), esadecimale (base 16) o ottale (base 8).

- In notazione decimale il numero è espresso nel modo consueto,
- in notazione esadecimale il numero deve essere preceduta dal segno **0x** ,
- in notazione ottale il numero deve essere preceduta da uno **0** .

Per indicare il tipo di dato con cui rappresentare la costante, si utilizzano dei caratteri da porre immediatamente dopo i caratteri numerici.

- Per default le costanti intere sono considerate int.
- Se la costante intera è troppo grande per essere contenuto in un int, viene considerato long.
- Una costante unsigned int deve essere seguita dal carattere **u** o **U**.
  es: 41234U
- Una costante long deve essere seguita dal carattere **L** o **l**.
  es: 49761234L
- Una costante unsigned long deve essere seguita dai caratteri **UL** o **ul**.

Esempi:

Valore	Tipo	DECIMALE	ESADECIMALE	OTTALE
0	int	0	0x0	00
1	int	1	0x1	01
8	int	8	0x8	010
-8	int	-8	-0x8	-010
17	int	17	0x11	021
-30	int	-30	-0xFD	036
100000	long int	100000**L** 100000**l**	186A0**L** 186A0**l**	-303240**L** -3032040**l**
-100000	long int	-100000**L** -100000**l**	-186A0**L** -186A0**l**	-303240**L** -3032040**l**

Costanti Numeriche in Virgola Mobile

Le costanti numeriche in virgola mobile sono scritte:

- in **rappresentazione decimale** nel modo consueto, ovvero: un eventuale segno, la componente intera, il punto decimale ed eventualmente la componente decimale.

  es: -10.4 37.235**f** 7. .001

- oppure in **formato esponenziale**, cioè nella forma Xe±M con il significato di $X*10^{\pm M}$, dove X è una componente floating point rappresentata come prima indicato, M è un intero

  es: -1.04e+1 0.37235e+2**L** 0.7e+1 1.e-2

Inoltre, il tipo di dato usato per rappresentare la costante sarà:

- il tipo double se non viene specificato un suffisso alla costante.
- il tipo float se invece viene aggiunto un suffisso **f** o **F** ,
- il tipo long double a 16 byte se viene aggiunto un suffisso **l** o **L**.

Costanti di tipo Char

I caratteri, cioè i tipi char (signed e unsigned) servono a rappresentare un insieme di simboli quali i simboli della tastiera a A b B 1 2 ed alcuni caratteri cosiddetti "di controllo" quali Enter, Tab, BackSpace, NewLine.
Lo standard ASCII associa un carattere ai valori da 0 a 127, mentre per i caratteri da 128 a 255 l'associazione varia a seconda del computer.
I caratteri di controllo sono i primi 32 della tabella ASCII, cui seguono i caratteri stampabili, cioè visualizzabili da un comune editor. Qui di seguito un estratto dalla tabella ASCII.

codice ASCII	48	49	50	51	52	53	54	55	56	57
carattere	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'

codice ASCII	65	66	67	68	69	70	71	ecc..		
carattere	'A'	'B'	'C'	'D'	'E'	'F'	'g'	ecc..		

codice ASCII	97	98	99	100	101	102	103	ecc..		
carattere	'a'	'b'	'b'	'd'	'e'	'f'	'g'	ecc..		

Una variabile di tipo carattere quindi memorizza un certo valore, da 0 a 255, che corrisponde ad un certo carattere. In un assegnamento ad una variabile di tipo carattere, potremo assegnare o il codice numerico o il carattere rappresentato da quel codice numerico.

Una costante di tipo carattere quindi potrà essere rappresentata o come valore numerico da 0 a 255, oppure come simbolo corrispondente a quel codice, ad es. ad un certo char potremo assegnare o il carattere '0' o il valore numerico 48.

La **rappresentazione numerica delle costanti di tipo carattere** è fatta mediante la rappresentazione decimale del codice numerico. Ad es assegnando ad un char il valore 97 gli assegnamo il carattere 'a'.

La **rappresentazione simbolica** delle costanti di tipo carattere, **non soffre dei problemi dovuti alla differenze tra le codifiche dei caratteri**, perchè **scrive nel codice** direttamente **il simbolo** che si vuole ottenere, e non una sua rappresentazione numerica.
D'altro canto è necessario delimitare la rappresentazione simbolica di un certo carattere separandola dagli altri elementi del linguaggio C. A questo scopo **la rappresentazione delle costanti di tipo carattere mediante i simboli viene fatta scrivendo il simbolo all'interno di due apici singoli**.

ad es. '**L**' indica il carattere che rappresenta il simbolo L.

Però, poichè non tutti i caratteri sono visualizzabili da un editor (ad es. i caratteri di controllo) deve essere previsto **un modo simbolico per indicare un carattere che non può essere visualizzato**. A tal scopo sono definite le cosiddette "sequenze di escape", ovvero dei simboli stampabili che quando sono precedute dal carattere \ vengono interpretate dal C come un carattere diverso.

Ad es. la sequenza di escape '**\n**' indica il carattere **new line** (1 carattere in Unix, due caratteri in DOS) cioè l'andata a capo di una riga di testo. Le sequenze di escape vanno racchiuse tra apici singoli come i simboli dei caratteri.

Le sequenze di escape vengono introdotte anche per poter indicare simbolicamente alcuni caratteri particolari (' ") che hanno un significato speciale nel linguaggio C

Le principali sequenze di escape disponibili in C sono le seguenti:

\a	suono
\b	BackSpace
\n	New Line, andata a capo
\r	carriage return
\t	il Tab orrizzontale
\\	il BackSlash, che è il delimitatore delle sequenze di escape
\'	il carattere apice singolo ' che in C è delimitatore di carattere
\"	il carattere doppio apice " che in C è delimitatore di stringa

Notare l'analogia tra l'uso delle sequenze di escape \\ \' \" in C e l'uso della coppia di apici singoli " dentro la write per stampare un solo apice singolo '.

Infine è possibile rappresentare i caratteri ancora in forma numerica mediante rappresentazione ottale o esadecimale, scrivendo all'interno della coppia di apici singoli:
- un BackSlash seguito dalla rappresentazione ottale del numero, oppure
- un BackSlash seguito da una x seguita dalla rappresentazione esadecimale.

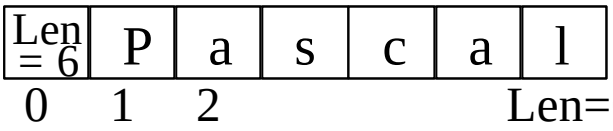
rappr. ottale	'\OOO'	fino a tre caratteri ottali (0:7), (max \377)
rappr. esadecimale	'\xhh'	fino a due (hh) caratteri esadec. (0:9,A:F)

es, sono equivalenti i seguenti assegnamenti, nella stessa colonna:
char ch;

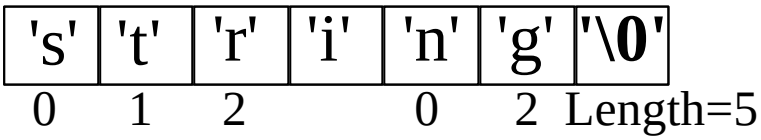
ch='A';	ch=' " ';	ch=' \' ';	ch='\\';
ch=65;	ch=34;	ch=39;	ch=92;
ch='\x41';	ch='\x22';	ch='\x27';	ch='\x5C';
ch='\101';	ch='\42';	ch='\47';	ch='\134';
_____	ch='\042';	ch='\047';	**ch='\';/*errore*/**
	ch='\'";	ch='\047';	
non ammesso	**ch='\0042';**	**ch=' ' ; /*errore*/**	
ch='\400';	**errore,4 ottali**		

Costanti di tipo Stringa

In Pascal, la stringa è un dato primitivo, formato da n caratteri, costituito da un vettore di n+1 posizioni, in cui in prima posizione è contenuto un contatore che indica di quanti caratteri è composta la stringa.



In C invece la stringa vera e propria non esiste, si usa come stringa un vettore di n+1 char , in cui al carattere in ultima posizione è assegnato il valore 0, ovvero il carattere nullo '\0'. Questo carattere particolare rappresenta il delimitatore finale della stringa, ovvero l'indicazione che la stringa è finita.



Quindi, una stringa in C internamente è rappresentata da un vettore di caratteri terminati da un carattere nullo. La lunghezza è nota solo dopo avere trovato il carattere 0 in fondo.

Una costante di tipo stringa, ovvero una costante di tipo vettore di caratteri viene scritta come una **sequenza di caratteri racchiusa tra 2 doppi apici, che non fanno parte della stringa**.

Ciascun carattere nella stringa può essere rappresentato simbolicamente o per mezzo di una sequenza di escape o in forma esadecimale o ottale.

Esempi di costanti stringa sono:

"questA stringa è giusta" 'è' e' un caratt. non ASCII

"quest\x41 stringa e' giusta" ' ' ' è delimitatore di carattere, non di stringa, quindi non c'e' confusione

"quest\x41 stringa e\47 giusta" ' ' ' scritto in ottale con 2 cifre(occhio)

"quest\101 stringa e\047 giusta" ' ' ' scritto in ottale a 3 cifre

"questA stringa " è sbagliata" ERRORE, "" è delimitatore di stringa

"questA stringa \" è stata corretta"

"aa12bb" è uguale a "aa**\061**2bb" ma è diverso da "aa**\61**2bb"

NB: la const a destra dà errore in compilazione,"numeric constant too large"

Operatori in C

Il linguaggio C mette a disposizione degli **operatori**, ovvero dei simboli speciali che indicano al compilatore di generare codice per far eseguire delle manipolazioni logiche, matematiche o di indirizzamento sulle variabili che costituiscono gli **operandi** dell'operatore.

Le funzionalità degli operatori in C sono in linea di massima le stesse di tutti i linguaggi di programmazione evoluti, ad es. il Pascal.

Il C si distingue perchè mette a disposizione alcuni operatori particolari che servono ad effettuare velocemente alcune operazioni semplici.

Le operazioni realizzate da questi operatori sono molto veloci perchè la semplicità delle funzionalità rende possibile implementare l'operazione in una maniera che sfrutta appieno la potenzialità dei registri del calcolatore, limitando il numero degli accessi alla memoria, e lavorando il più possibile coi registri.

Prima di vedere alcuni operatori, è bene introdurre una caratteristica molto particolare del C, il Type Casting, ovvero la modifica del tipo di dato.

Type Casting

Molto spesso il risultato di un'operazione dipende dal tipo di dato che è coinvolto nell'operazione. Ad es. la divisione tra interi ha un risultato diverso dalla divisione tra floating point, anche partendo dagli stessi dati iniziali. 5.0/2.0=2.5 5/2=2

Questo perchè a seconda del tipo di dati coinvolti nelle operazioni, le operazioni sono svolte in modo diverso, o utilizzando registri o porzioni di registri diversi.

Una **caratteristica peculiare del C** è che **permette** durante **l'esecuzione di un'operazione**, o **durante il passaggio di parametri ad una funzione all'atto della chiamata**, **di modificare il tipo del o dei dati coinvolti nell'operazione**.

Ciò non significa affatto che la variabile (o la costante, o il risultato di un'espressione) interessata **cambia le sue caratteristiche** (dimensioni, proprietà, ecc..) di tipo.

Significa invece che se **la variabile** deve essere utilizzata in dei calcoli, **viene copiata in un registro sufficentemente grande** per contenere il **nuovo tipo di dato** e per **svolgere le operazioni di conversione**, ed il **valore del registro caricato viene convertito** secondo le caratteristiche del nuovo tipo di dato.

Solo **dopo questa conversione, l'operazione viene effettuata, secondo le regole proprie del nuovo tipo di dato**, ottenendo quindi come risultato un valore coerente con il nuovo tipo.

La conversione di tipo **type-casting** in qualche caso viene effettuata implicitamente dal compilatore C, ma può anche essere forzata dal programmatore mediante un operatore unario detto **cast**, che opera in questa maniera.

(nome_nuovo_tipo) espressione

dove "nome_nuovo_tipo" è un tipo di dato primitivo o definito dall'utente, ed "espressione" può essere una variabile, una costante o un'espressione complessa. es. (double) i;

"Espressione" viene risolta fino ad arrivare ad un risultato (di un suo certo tipo), poi il risultato viene convertito nel tipo "nome_nuovo_tipo" mediante opportune operazioni più o meno complesse a seconda della conversione.
A questo punto abbiamo come risultato della conversione un dato di tipo "nome_nuovo tipo" con un certo valore, che può essere utilizzato secondo le regole definite per "nome_nuovo_tipo".

Vediamo subito un esempio di cosa succede:

```
int    i=5,    j=2;
double f;
f = i / j;
/* f ora vale 2 */
```

```
int i=5,    j=2;
double f;
f = (double)i / (double) j;
/* f ora vale 2.5 */
```

Un cast quindi equivale ad assegnare l''espressione'' ad una variabile del nuovo tipo specificato, che viene poi utilizzata al posto dell'intera espressione.

L'esempio riportato mostra un caso in cui il type cast modifica fortemente il risultato di un'operazione, che comunque avrebbe potuto essere effettuata, pur con risultati diversi, senza type cast.

Il **casting viene** però **utilizzato spesso anche nel passaggio di parametri a funzioni,** qualora il programmatore abbia necessità di passare alla funzione chiamata un parametro formalmente diverso da quello richiesto dalla funzione. Ad es qualora si debba passare alla funzione un puntatore ad una struttura, di tipo diverso da quella che la funzione si aspetta.
Il compilatore dovrebbe segnalare l'errore. Mediante un type cast del parametro passato all'atto della chiamata (di cui il programmatore si deve assumere la responsabilit relativamente alla correttezza) si può "**imbrogliare**" il compilatore, convincendolo della correttezza formale dell'operazione.

Abbiamo finora parlato di type-casting forzato, ovvero imposto dall'utente.
In realtà il **C effettua automaticamente delle conversioni implicite di tipo**, in particolare quando effettua operazioni matematiche tra operandi di tipo diverso. **Il casting viene effettuato automaticamente dal compilatore C quando due operandi di un'operazione binaria sono tra loro diversi. In tal caso l'operando di tipo più piccolo viene convertito nel tipo più grande, senza perdita di informazioni.**
Quindi, dati due operandi di tipo diverso, il cast automatico si ha secondo queste regole:

Tipo 1° operando	Tipo 2° operando	Tipo Risultato
long double	double	long double
double	float	double
float	(long) int	float
double	(long) int	double
int	char , short	int
long int	(short) int	long int

Attenzione, il casting automatico può dare delle false sicurezze. Riguardare l'esempio precedente per verificare per quale motivo si ha perdita di informazioni (risposta: dipende dall'ordine di esecuzione delle operazioni. Prima viene fatta la divisione, sono due interi quindi nessuna conversione e c'è perdita di informazioni, e solo in un secondo tempo c'e' l'assegnamento al float).

```
int i=5, j=2;      double f;
f = (double) (i /j);          f diventa 2.0
```

N.B. come abbiamo già visto, anche nell'assegnamento il C effettua conversioni di tipo automatiche, convertendo il valore del lato destro nel tipo del lato sinistro, eventualmente perdendo informazioni quando si passa da un tipo ad un tipo più piccolo.

Ora che abbiamo considerato il problema della conversione di tipo, vediamo quali sono e cosa fanno alcuni operatori del C

Operatori Aritmetici + - * / %

Sono operatori presenti in tutti i linguaggi di programmazione:
Sono operatori binari (servono due operandi): + - * / %

Somma	+	x+y	valore del tipo più grande
Differenza	-	x-y	valore del tipo più grande
Cambio Segno	-	-x	stesso tipo
Prodotto	*	x*y	valore del tipo più grande
Divisione	/	x/y	valore floating tra floating valore int tra int, tronca decimali
Resto	%	n%m	resto modulo m, solo tra interi

Operatori Incremento Decremento ++ --

Sono operatori unari (serve un unico operando): ++ --
Aggiungono (tolgono) l'unità (1) all'operando.

- Nel caso l'operando sia un puntatore a qualcosa, incrementano il puntatore della quantità necessaria a farlo puntate all'inizio dell'elemento che segue la struttura (vedremo l'aritmetica dei puntatori).

Equivalgono ad assegnare alla stessa variabile il proprio valore incrementato (decrementato) di uno, ma lavorano velocemente perchè usano bene registri.

```
x++        x=x+1
x--        x=x-1
```

Attenzione, sono previste due modalità di esecuzione, qualora l'incremento sia parte di un'istruzione più complessa.

```
x++;    prima usa x nell'istruzione, poi lo incrementa
++x;    prima incrementa x, poi lo usa nell'istruzione
```

Da isolate non fanno differenza, ma in istruzioni meno semplici cambia.

```
es.  int n, m=0;                    int n, m=0;
     n = m++;                       n = ++m;
     n vale 0,m vale 1              n vale 1, m vale 1
```

Priorità:	++ --	massima
	-	(cambio segno)
	* / %	
	+ -	minima

A pari priorità si valuta a partire da sinistra, a meno di uso di parentesi.

Operatori Relazionali e Logici

Ricordiamo che in C il valore booleano Falso equivale a Zero, e che Vero equivale a diverso da zero. Gli operatori logici (equivalenti ad AND OR NOT del Pascal) e gli operatori relazionali (minore maggiore uguale diverso ecc..) restituiscono perciò 0 quando il risultato è falso e 1 quando il risultato è vero.

Operatori Logici

AND	&&
OR	||
NOT	!

Operatori Relazionali

Maggiore	>
Maggiore o uguale	>=
Minore	<
Minore o uguale	<=
Uguale	==
Diverso	!=

La priorità degli operatori relazionali e logici è minore della priorità degli operatori aritmetici, quindi in un'espressione complessa prima vengono valutate le operazioni aritmetiche, e solo in un secondo momento gli operatori relazionali.

Ad es., l'espressione

```
x <= 5+y
```

viene valutata prima effettuando la somma tra 5 ed y, ed il risultato viene confrontato con x.

Ciò equivale alla valutazione di questa espressione

```
(x  <=  (5+y))
```

E' buona norma (**è fortemente consigliabile !!!!!!!**) abituarsi ad usare **parentesi tonde e spaziature** in abbondanza per delimitare visivamente almeno le espressioni più semplici che entreranno poi in relazioni tramite operatori logici.

Si commettono di solito meno errori di programmazione guidando la valutazione delle espressioni tramite alcune parentesi tonde poste ad hoc, piuttosto che non affidandosi esclusivamente alla precedenza degli operatori.

- Le espressioni sono valutate da sinistra a destra.
- Le priorità all'interno del gruppo degli operatori logici e relazionali sono le seguenti:

```
!                priorità massima (valutato per primo)
> >= <  <=
==  !=
&&
||
```

Ad es. valutiamo l'espressione qui di seguito:

10>5 && !(10<9) || 3<=4

```
10>5 vale 1                    (* vedi slide successiva)
          !(10<9)   vale 1
otteniamo    1   &&  1      ||  3<=4
             1   &&  1  vale 1
otteniamo           1      ||  3<=4
                               3<=4 vale 1
otteniamo           1      ||    1
                      1 || 1  vale 1
```

Il risultato finale dell'espressione è 1.

Non voglio vedere espressioni scritte così, utilizzate le parentesi tonde.

La stessa espressione, scritta in forma più leggibile è riportata qui sotto:

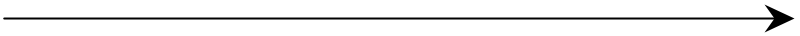
((10>5) && (!(10<9))) || (3<=4)

La sequenza delle parentesi impone l'ordine di valutazione degli operatori, ed evidenzia il criterio logico con cui si deve essere formata l'espressione.

Valutazione Abbreviata di Espressioni connesse da Operatori Logici. (1)

Consideriamo **una espressione formata da espressioni connesse da operatori logici && e ||** come ad esempio

ESPR1 && ESPR2 || ESPR3



direzione di valutazione

Tali espressioni **sono valutate da sinistra verso destra**, quindi, ricordando che l'operatore AND logico && ha priorità maggiore dell'operatore OR logico, l'espressione nel suo complesso può essere interpretato come segue:

(ESPR1 && ESPR2) || ESPR3

nel senso che dovrebbe essere valutata prima l'espressione (ESPR1 && ESPR2) ed in un secondo tempo l'espressione ESPR3, per poi mettere in OR logico i due risultati.

Ciò è abbastanza vero, ma **in realtà il compilatore C è sufficentemente furbo da generare il codice corrispondente alla valutazione di questa espressione, in modo tale da interrompere la valutazione dell'espressione non appena determina la VERITA' o la FALSITA' dell'INTERA espressione.**

Lo stesso concetto vale per porzioni di espressioni, la cui valutazione viene interrotta quando viene determinata la Verità o Falsità della porzione di espressione.

Valutazione Abbreviata di Espressioni connesse da Operatori Logici. (2)

Vediamo come il compilatore C organizza la valutazione di:

(ESPR1 && ESPR2) || ESPR3

- viene valutata ESPR1

- se ESPR1 è falsa (0)
  - NON VIENE VALUTATA ESPR2 perchè in ogni caso l'espressione ESPR1 && ESPR2 sarà falsa
  - viene valutata ESPR3
  - se ESPR3 è falsa (0) l'espressione totale vale falso (0)
  - se invece ESPR3 è vera (!=0) l'espressione totale vale vero.
  - **in ogni caso non abbiamo valutato ESPR2 perchè era inutile**

- se ESPR1 è vera
  - VIENE VALUTATA ESPR2

  - se ESPR2 è vera allora l'espressione ESPR1 && ESPR2 sarà vera, ed anche l'espressione intera ESPR1 && ESPR2 || ESPR3 sarà vera, indipendentemente dal valore di ESPR3,
    - quindi **ESPR3 non viene valutata**, ed il risultato finale è vero.
  - se ESPR2 è falsa allora l'espressione ESPR1 && ESPR2 sarà falsa, e deve essere valutata ESPR3 per conoscere il risultato finale.
    - viene valutata ESPR3
    - se ESPR3 è vera il risultato finale è vero
    - se ESPR3 è falsa il risultato finale è falso

Questo modo di valutare le espressioni, velocizza le operazioni, perchè non esegue operazioni inutili, ma deve essere attentamente valutato, se all'interno delle espressioni sono presenti istruzioni o funzioni che generano qualche effetto secondario, perchè tali funzioni potrebbero essere eseguite oppure no a seconda del risultato booleano delle varie porzioni dell'espressione.

Valutazione Abbreviata di Espressioni connesse da Operatori Logici. (3)
Un esempio di uso ottimale della Valutazione Abbreviata

Vediamo un esempio banale di applicazione della regola di valutazione abbreviata.

Supponiamo di avere un vettore di interi, con SIZE elementi,
vogliamo stampare tutti gli elementi del vettore
o fino all'ultimo
o fino a che non incontriamo un elemento che vale 17, nel qual caso non vogliamo stampare 17 e vogliamo interrompere la stampa.

Possiamo sfruttare la valutazione abbreviata per scrivere un codice cosi fatto:

```
#define SIZE 10;
int Vet[SIZE];
int i;
.... scrittura degli n elementi del vettore Vet
i=0;
while (   (i<SIZE)  &&  (Vet[i]!=17)  )
      printf( "Vet[%d]=%d\n", i , Vet[i++] );
```

l'espressione Vet[i]!=17 viene valutata solo se è stata valutata vera la prima espressione (i<SIZE).
Ciò consente di evitare di accedere alla variabile Vet in una posizione maggiore o uguale della posizione numero SIZE, perchè si cadrebbe fuori dal vettore e si potrebbe causare un eccezione di sistema del tipo segmentation fault che causa l'interruzione traumatica del programma.

Un esempio da non seguire:

Uno degli errori più comuni per chi comincia a programmare in C, consiste nell'ostinarsi a voler scrivere codice "stretto" e poco commentato.

E' sempre un errore, perchè:

1) il codice va modificato nel tempo, ed il codice scritto in forma compatta è più difficile da capire, anche per chi l'ha scritto.

2) L'aggiunta di parentesi tonde e spazi per rendere più visibile e comprensibile il codice non diminuisce le prestazioni.

Come curiosità, vediamo un esempio **DA NON SEGUIRE**, di codice C (estensione Kernighan&Ritchie, e non ANSII) **scritto in forma compatta**, funzionante, che stampa a video una filastrocca inglese.

```
#include <stdio.h>

main(t,_,a)char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')){}#}w')){}{nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/'')
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Non è noto in quale manicomio sia stato internato, colui che ha scritto questo codice C.

Operatori di Assegnamento ed Espressioni aritmetiche: Non Solo Notazione Abbreviata.

Il C mette a disposizione alcuni operatori particolari, che coniugano un'operazione aritmetica all'operazione di assegnamento.

Somma e Assegnamento	x **+=** y	equivale a	x = x+y
Differenza e Assegnamento	x **-=** y		x = x-y
Prodotto e Assegnamento	x **=** **y**		x = x*y
Divisione e Assegnamento	x **/=** y		x = x/y
Resto e Assegnamento	x **%=** y		x = x%y

(Sono disponibili anche operatori di assegnamento ed espressione bit a bit, che vedremo più avanti).

Istruzioni C del seguente tipo (dove op è un'operazione aritmetica)

espressione_1 **op=** espressione_2

equivalgono all'istruzione

espressione_1 **=** espressione_1 **op** espressione_2

in cui l'espressione espressione_1 viene valutata due volte.

Questi operatori composti servono a velocizzare il codice, perchè evitano di valutare ripetutamente una stessa espressione, e in qualche caso riescono a mantenere i valori valutati nei registri, quindi minimizzano l'accesso alla memoria.

Ad es. se noi avessimo un vettore di interi **Vet**, e volessimo aggiungere un 3 all'elemento in posizione i+7*k, dovremmo scrivere (l'accesso alle posizioni di un vettore si effettua mediante l'uso di parentesi quadre) un'istruzione del tipo:

```
Vet[i+7*k] = Vet[i+7*k] +2;
```

dovendo così valutare due volte l'espressione i+7*k

Invece con l'istruzione

```
Vet[i+7*k] += 2;
```

l'espressione i+7*k viene valutata una volta sola.

Operatore sizeof()

Il C mette a disposizione un operatore unario, che restituisce la dimensione della variabile o dello specificatore di tipo passato in input.
Serve a conoscere le dimensioni di alcuni tipi di dati, che potrebbero cambiare al variare dell'architettura su cui il programma deve girare, come ad esempio cambiano le dimensioni degli interi int.

L'operatore sizeof prende in input o una variabile o un identificatore di tipo, e restituisce la dimensione in byte del dato. Il dato può anche essere un dato definito dall'utente, non solo un tipo di dato primitivo.

es:

```
int I;
printf("dimensione di I: %d \n", sizeof(I) );    /* stampa 2 in Windows */
                                                 /* stampa 4 in Linux */

printf("dimensione del float: %d \n", sizeof(float) );      /* stampa 4 */
```

L'operatore sizeof è molto importante per la portabilità del codice, da un'architettura ad un'altra.

Particolarità dell'operatore unario sizeof è che viene valutato non durante l'esecuzione del programma, ma al momento della compilazione

---

Strutture di Controllo del Flusso

In un linguaggio, le strutture di controllo del flusso specificano l'ordine secondo il quale le operazioni devono essere effettuate. Abbiamo già visto informalmente alcune strutture, ora completeremo e preciseremo la descrizione.

Istruzioni e Blocchi di Istruzioni

Ogni **istruzione** in C deve essere terminata da un punto e virgola **;**
Un'espressione qualsiasi come x=0 i++ **1&&23<13==7>>5** oppure una chiamata ad una funzione come ad es. printf(...) diventa un'istruzione quando seguita da un punto e virgola.
Il valore restituito da queste espressioni è il risultato della valutazione dell'espressione considerata.
Un'istruzione costituita dal solo punto e virgola è un'istruzione nulla, che non ha effetto.

Un **Blocco di Istruzioni** è una sequenza di istruzioni C racchiusa tra una parentesi graffa aperta ed una parentesi graffa chiusa.
Un blocco dal punto di vista dei costrutti linguistici (else while ecc..) va considerato come un'unica istruzione.
Il corpo stesso di una funzione è un blocco di istruzioni.
All'interno di un blocco è possibile dichiarare delle variabili locali.
L'unica altra differenza tra un'istruzione singola ed un blocco è che dopo un blocco non deve essere inserito il punto e virgola. Se viene messo viene considerato un'altra istruzione, nulla.

Istruzione if-else

L'istruzione **if** puo' avere due forme di base:

if (espressione) **istruzione**	**if (espressione)** **istruzione** **else** **istruzione2**

Viene valutata l'espressione "espressione", e se risulta vera (cioè diversa da zero) viene eseguita l'istruzione "istruzione". In caso contrario, se esiste la keyword else viene eseguita l'istruzione "istruzione2".

N.B. 1: **istruzioni e blocchi**
con "istruzione" deve intendersi o singola istruzione o blocco di istruzioni.

N.B. 2: **valutazione veloce**
quando viene valutata l'espressione dentro parentesi tonde nel costrutto if, viene effettuato il controllo se il risultato della valutazione è uguale o diverso da zero, cioè se è falso o vero. Quindi scrivere ***if(espressione)*** oppure ***if(espressione!=0)*** è la stessa cosa, ma il primo tipo di scrittura viene codificato dal compilatore in modo più veloce, perchè non è richiesto di caricare un registro con la costante inserita nel programma (lo zero) per il confronto, ma si ricorre ad una istruzione assembler di jump condizionato dall'essere il risultato della valutazione (già contenuto in un registro) diverso o uguale a zero. (Alcuni compilatori ottimizzano situazioni di questo tipo).

N.B. 3: **ambiguità dell'else** in caso di if annidati
Nel caso di costrutti if-else annidati, una keyword **else è relativa al più vicino degli if che lo precedono che manchi dell'else**, ovviamente sempre che sia rispettata la sintassi dell'if-else.

questo codice:	significa:	che è diverso da:

if (x)
  if(y)
  istr1
  else
  istr2
	if (x){
	  if(y)
	  istr1
	  else
	  istr2
	  }
		if (x){
		  if(y)
		  istr1
		  }
		else
		  istr2

Istruzione if-else-if

Il costrutto **if-else-if** è fatto nel modo seguente

```
if (espressione1)
        istruzione1
else if (espressione2)
                istruzione2
        else if  (espressione3)
                        istruzione3
                else
                        istruzione
```

e consente di effettuare una scelta multipla.

- Le espressioni 1, 2, ..ecc. vengono valutate nell'ordine in cui si presentano.
- La prima espressione che si rivela vera fa eseguire la corrispondente istruzione, e fa uscire il controllo dalla struttura di controllo if-else-if.
- L'ultimo else (che può anche non esserci) serve nel caso in cui nessuna delle precedenti condizioni si realizza.

Ecco un esempio, in cui si discrimina secondo il valore di una variabile x, considerando 4 intervalli (-infinito,0) , [0,10) , [10,100) , [100 , +infinito)

```
double x= .... .
if (x<0.)
        istruzione1
else if (x<10.)
                istruzione2
        else if  (x<100.)
                        istruzione3
                else
                        istruzione
```

- La struttura annidata if-else-if per quanto molto potente, ha lo svantaggio di non essere particolarmente veloce, perchè se la prima espressione che risulta vera è molto annidata, prima di poterla verificare è necessario valutare tutte le espressioni precedenti.

Istruzione switch

L'istruzione switch è un'altra struttura di scelta plurima, analoga al case del pascal, che **controlla se un'espressione assume un valore intero in un insieme di COSTANTI intere**, e **fa eseguire una serie di istruzioni in corrispondenza del valore intero verificato**.

- Una limitazione dello switch è che i valori ammessi come possibili scelte (possibili casi previsti) devono essere delle costanti, e non possono essere delle espressioni.
- Questo perchè il costrutto switch vuole implementare una scelta multipla molto più veloce del costrutto if-else-if, ma per fare questo deve limitare le possibili scelte, definendole non in modo run-time mediante un'espressione, ma mediante delle costanti, che il compilatore utilizzerà per scrivere in cdice macchina una sequenza di jump condizionati.

La struttura dello switch è così:

```
switch (espressione)
{
    case espressione_costante1 :
        sequenza di istruzioni1;
        break;
    case espressione_costante2 :
        sequenza di istruzioni2;
        break;
        .... altri case ...
    default :
        sequenza di istruzioni;
        break;
    }
```

un esempio di switch semplice

```
#define NUM 3
int j;
.....
    switch( j )
{
    case 1:
    case 2+NUM:
    case 7:
        sequenza_di_istruzioniA
        break;

    case -34:
    case -34<<2:
        sequenza_di_istruzioniB
        break;

    default
        sequenza_di_istruzioniC
}
```

Ogni possibile caso deve essere etichettato mediante uno (o più) costrutti **case "Espressione_costante":** dove Espressione_costante è un'espressione formata solo da costanti e operatori, ma non da variabili o funzioni.

Possono esserci più etichette per una stessa sequenza di istruzioni, come nel precedente esempio.
Le diverse case rappresentano delle entry point nel costrutto switch.

Se l'espressione valutata nello switch assume uno dei valori indicati nelle espressioni costanti, l'esecuzione passa alla sequenza di istruzioni che seguono la keyword case per quell'espressione costante.

L'esecuzione procede fino a che non si incontra la keyword break, o si oltrepassa la parentesi graffa aperta che termina la struttura switch, ed allora il controllo esce dalla struttura di controllo di flusso switch.

Il caso etichettato come **default** può essere presente o no. Se è presente assume il significato di "**in tutti gli altri casi**" cioè viene eseguito se l'espressione valutata nello switch non ha assunto nessuno dei valori indicati da una delle espressioni costanti di un qualche case.

N.B. la keyword break non è strettamente indispensabile. Se non è presente viene eseguita sequenzialmente ogni istruzione a partire dal case che è stato raggiunto.

Ciclo while

Il loop di tipo while è analogo al while del pascal, e consente di eseguire un loop condizionale. La struttura del while è la seguente:

while (espressione**)**
istruzione

dove "espressione" è una espressione che deve essere valutata ogni volta prima di eseguire "istruzione". Se la valutazione di "espressione" da' come risultato vero allora l'istruzione viene eseguita, altrimenti no, e si esce dal ciclo.
Poichè la valutazione della condizione è effettuata prima delle istruzioni che costituiscono il loop, il loop stesso può essere eseguito zero volte oppure un numero finito di volte oppure ancora può realizzare un loop infinito.

esempio, stampa di una stringa
(modo idiota, ovvero carattere per carattere)

```
char str[]="stringa";
int j=0;
while( str[j] )   printf("%c", str[j++]);
```

loop infinito

```
while(1) { .... }
```

Ciclo do-while

Il loop di tipo while è analogo al repeat-until del pascal, e consente di eseguire un loop condizionale da 1 ad infinite volte. La struttura del do-while è la seguente:

```
do {
    istruzioni
} while (espressione);
```

dove "espressione" è una espressione che deve essere valutata ogni volta dopo avere eseguito "istruzioni". Se la valutazione di "espressione" da' come risultato vero il loop viene ripetuto, cioè si riesegue "istruzioni", altrimenti si esce dal ciclo.

Poichè la valutazione della condizione è effettuata dopo le istruzioni che costituiscono il loop, il loop stesso può essere eseguito da una volta a infinite volte.

es: `int num; do { scanf("%d", &num); } while(num>100);`

Ciclo for

Il loop di tipo for è analogo al for del pascal, ma è più potente.
La forma generale del ciclo for è fatta così:

for (inizializzazione **;** condizione **;** incremento) istruzione

le tre componenti di un ciclo for sono delle espressioni, ovvero possono essere degli assegnamenti o delle chiamate a funzione.

- L'espressione "**inizializzazione**" **viene eseguita una sola volta**, nel momento in cui il controllo viene affidato al costrutto for. Serve per impostare le variabili e quanto necessario per cominciare il loop. Questa "espressione" può anche non essere presente, ed allora dopo la parentesi tonda aperta viene subito il punto e virgola.

- L'espressione "**condizione**" **viene valutata (eseguita) ogni volta prima di eseguire le istruzioni del loop**, ed è quella che stabilisce se il ciclo deve continuare ad essere eseguito ancora una volta, oppure si deve uscire dal costrutto for. Se "condizione" è vera si esegue ancora "istruzione", se invece è falsa (0) si esce dal ciclo for passando alla istruzione successiva del programma.
  Anche l'espressione **"condizione" può non essere presente**, nel qual caso lo spazio tra i due punti e virgola dentro il costrutto for rimane vuoto, ed **il compilatore assume vero come risultato della valutazione della (assente) condizione di proseguimento del ciclo for, e quindi continua ad eseguire le istruzioni dentro al loop**. Quindi un ciclo for fatto così **for(; ;) { .. }** realizza un loop infinito.

- L'espressione "**incremento**" **viene eseguita alla fine di ogni ciclo**, per modificare ad es. le variabili che stabiliscono l'incremento o decremento delle variabili contatore che stabiliscono il numero di cicli del loop. Questa "espressione" può anche non essere presente, ed allora dopo il secondo punto e virgola del for viene subito la parentesi tonda chiusa.

N.B. poichè la condizione di continuazione del loop viene valutata (eseguita) prima di ogni ciclo, il loop for permette anche di non eseguire nemmeno una volta il ciclo.

Ciascuna delle tre **espressioni** del for, cioè inizializzazione, condizione e incremento **può contenere più istruzioni, che dovranno essere separate da virgole**. Vediamone un esempio nel caso in cui si vogliano utilizzare due variabili di controllo.

```
/* con due variabili di controllo */
void main(void)
{
      int x ,y;
      for ( x=0, y=0; x+y<100; x++ , y+=3 )   printf ("%d ", x+y);
}
```

Il ciclo for può anche non contenere un corpo di istruzioni vero e proprio, che può essere limitato ad un'istruzione vuota, il punto e virgola.

```
/* lunghezza di una stringa null-terminata*/
int len;     char str[] = "mia stringa";
for(  len=0;  stringa[len];  len++)  ;
```

Vediamo alcuni altri esempi di utilizzo del ciclo for.

```
/* con incremento */
void main(void)
{     int x;
      for ( x=1; x<100; x++ )   printf ("%d ", x);
}
```

```
/* con decremento */
void main(void)
{     int x;
      for ( x=1; x<100; x-- )   printf ("%d ", x);
}
```

```
/* loop infinito */
      int x;
      for ( ;; )   printf ("?");
```

```
/* con espressioni costituite da funzioni */
      for ( funzA(); funzB(); funzC() )   .....
```

Istruzione break: Uscita da un ciclo

L'istruzione break (che è stata vista già a proposito della struttura switch) serve ad imporre l'uscita da un loop di tipo for, while, do-while, oppure l'uscita dalo switch, e a far riprender il controllo di flusso immediatamente dopo la fine del loop (o switch) da cui si esce.
es.:

```
void main(void)
{     int x;
       for ( x=1; ; x++ )   {
              ....
              if ( x==5) break;  /* esce dal ciclo for */
              ....
       }
       printf ("dopo break ricomincia qui ");
}
```

Nel caso di cicli annidati, con l'istruzione break si esce solo dal ciclo piu interno, entro il quale si trova la chiamata alla istruzione break.
es.:

```
void main(void)
{     int x;
       for ( y=1; ; y++ )
              for ( x=1; ; x++ )   {
                     ....
                     if ( x==5) break;  /* esce dal ciclo for */
                     ....
              }
              printf ("dopo break ricomincia qui%d ", x);
       }
}
```

Ovviamente, in caso di switch dentro un altro ciclo, un break dentro lo switch fa uscire solo dallo switch.

Istruzione continue: ricominciare il ciclo

L'istruzione continue, a differenza della break si applica ai cicli ma non allo switch, come la break interrompe l'esecuzione di un ciclo for while o do-while, ma anzichè uscire dal ciclo definitivamente fa eseguire la successiva iterazione.

- Nel caso di ciclo while o do-while una continue posta nel corpo del ciclo fa saltare alla espressione di controllo, che viene valutata, e se risulta vera viene eseguito ancora il loop, altrimenti si esce dal loop.
- Nel caso del ciclo for invece, un'istruzione continue fa eseguire l'espressione che effettua l'incremento, poi viene effettuata la valutazione della condizione di continuazione del ciclo for, e se risulta vera viene eseguito il ciclo, altrimenti si esce dal for.

Anche per la continue come per la break, in caso di cicli annidati, si interrompe solo il ciclo più interno in cui è avvenuta la chiamata alla continue.

```
void main(void)
{     int x;
       for ( y=1; ; y++ )
              for ( x=1; ; x++ )   {
                     ....
                     if ( x==5) continue;
                       /* si effettua l'incremento di x, x diventa 6,
                          si verifica la condizione, risulta vera perchè assente
                         si riprende dal for interno (quello della x) con x=6 */
              }
       }
}
```

N.B. L'istruzione continue è pochissimo usata.

Istruzione goto: salto

L'istruzione goto, consente di passare il controllo dal punto in cui si effettua la goto ad una locazione del programma individuata da una label (un'etichetta) seguita dai due punti, collocata in una qualche posizione del codice.

- Può essere utile per uscire da più livelli di annidamento di cicli, cosa che la break e la continue non riescono a fare.
- Va usato con attenzione.

```
void main(void)
{     int x;
       for ( y=1; ; y++ )
              for ( x=1; ; x++ )   {
                     for ( z=10; z<1000; z++ )   {
                            .......
                            if ( x==5) goto uscita;
                            ......
                     }
              }
       }
       uscita: ;
       ....
}
```

altro esempio di stranezza consentita dal goto.

```
void main(void)
{     int x=13;
       ......
       goto dentro;
       for ( x=1; ; y++ ) {
              dentro: printf("atterraggio: x=%d\n", x );
       }
}
```

- Il goto non permette di passare da una funzione ad un'altra.

Ecco un' esempio di cosa non è permesso.

```
int funzione1(void){
        .....
        arrivo: ;
        ...
}

void funzione2(int i){
        .....
        goto arrivo;            /* errore in compilazione */
        ...
}
```

Array ad una dimensione

Analogo all'array del Pascal, un array è un insieme di variabili dello stesso tipo cui è possibile accedere mediante un nome comune e referenziare uno specifico elemento tramite un indice.
Il C alloca gli elementi di un array in posizioni adiacenti, associando l'indirizzo più basso al primo elemento, e il più alto all'ultimo.
Gli array ad una dimensione vengono dichiarati come segue:

Tipo NomeVariabile[dimensione_costante] ;

dove Tipo è il tipo degli elementi del vettore, dimensione è una costante che indica quanti eementi deve contenere l'array.

- La dichiarazione serve al compilatore per riservare in memoria spazio sufficente al vettore.
  Lo spazio occupato dal vettore sarà:
  numero_byte = sizeof(Tipo) * dimensione_costante ;

esempi:

int vettore[10];	un vettore di 10 interi
char stringa[100];	un vettore di 100 caratteri, cioè potenzialmenteuna stringa per 99 caratteri più il carattere terminatore '\0

- Il C indicizza gli elementi a partire da 0, quindi possiamo accedere agli element con gli indici da 0 a dimensione_costante-1.
- L'accesso si effettua mediante il nome dell'array seguito dalle parentesi quadre in cui viene racchiuso l'indice dell'elemento cercato.
  vettore[3] accede al quarto elemento del vettore "vettore"
  stringa[0] accede al primo carattere del vettore "stringa"
- Non si puo' assegnare qualcosa al vettore intero, ma solo alle diverse posizioni del vettore (non ha senso scrivere vettore=19; o vettore=stringa).

Un esempio di programma con un semplice vettore.

```
void main(void)
{     char str[100];   int i;
      for ( i=0; i<100; i++)  str[i] = (char) i;
}
```

Inizializzazione di un array.

All'atto della dichiarazione è possibile inizializzare un array come di seguito indicato:

```
int vet [3] =    { 1,2,3 } ;
```

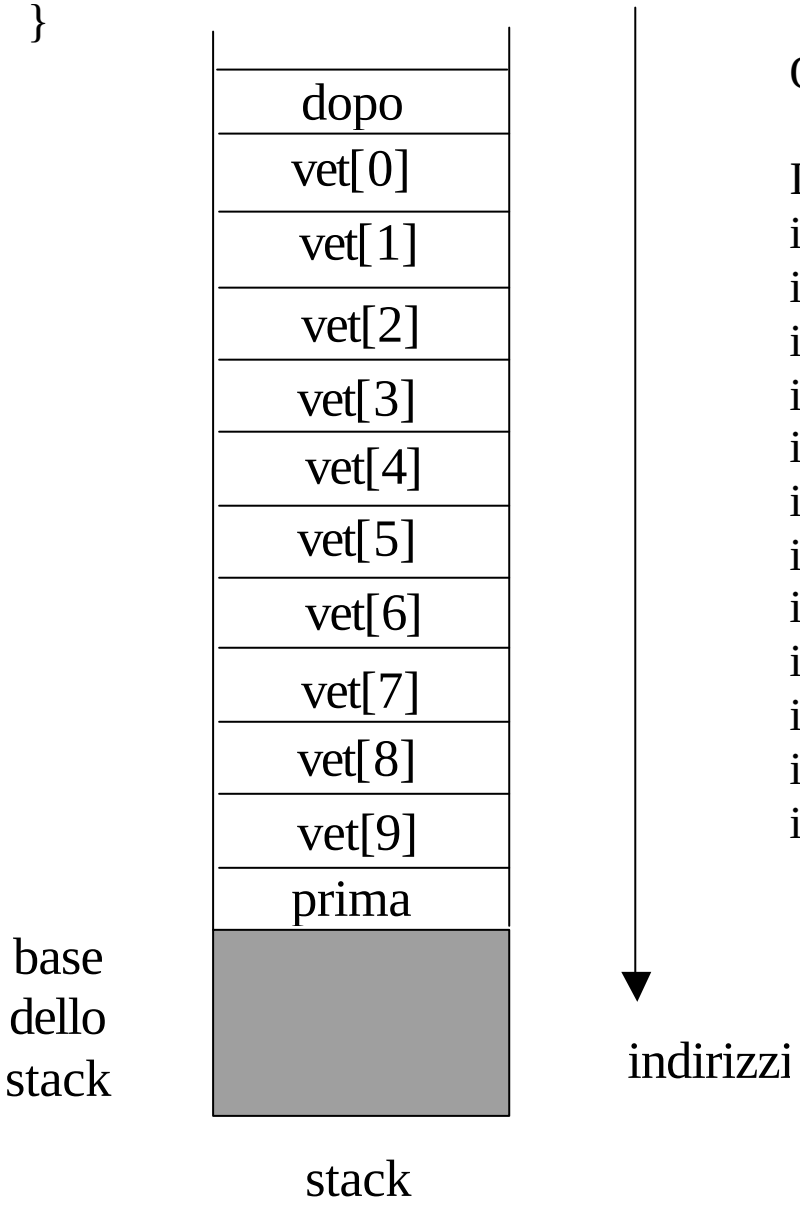
Il problema dello sconfinamento

Il linguaggio C non effettua controlli per verificare che non si acceda a posizioni fuori dal vettore.
Cioè è possibile **(ma è un errore gravissimo anche se comune)** accedere ad una locazione di memoria fuori dal vettore, con vari effetti possibili:

- se si accede in lettura al vettore fuori da esso può accadere, a seconda del sistema operativo:
  - nessun problema apparante, solo errore logico
  - segmentation fault.
- se invece si accede in scrittura al vettore ci saranno danni maggiori, perchè oltre ai due problemi già citati si sovrascriverà una locazione di memoria che potrebbe contenere dati essenziali al programma o peggio ancora al sistema perativo. Nel caso peggiore, ad es. sovrascrittura del vettore degli interrupt, si può arrivare al blocco del sistema.
  Oppure si modificherà il valore delle variabili dichiarate immediatamente sopra o sotto al vettore.

ESEMPIO DI ERRORE di SCONFINAMENTO

```
void main(void)
{
        int prima=0;
        int vet[10];
        int dopo=0;
        printf("INIZIO: prima=%d  dopo=%d\n", prima, dopo );
        for ( i=-1; i<=10; i++) {
                vet[i] = 99;
                printf("i=%d   prima=%d  dopo=%d\n", i, prima, dopo );
        }
}
```



Output del programma

```
INIZIO: prima=0  dopo=0
i=-1   prima=0  dopo=99
i=0   prima=0  dopo=99
i=1   prima=0  dopo=99
i=2   prima=0  dopo=99
i=3   prima=0  dopo=99
i=4   prima=0  dopo=99
i=5   prima=0  dopo=99
i=6   prima=0  dopo=99
i=7   prima=0  dopo=99
i=8   prima=0  dopo=99
i=9   prima=0  dopo=99
i=10   prima=99  dopo=99
```