Corso di C con Linux

Iniziamo a programmare
Lezione 21

Prima di scrivere il nostro primo programma, è necessario fare la classica distinzione tra processo e programma.
Un programma è un insieme di istruzioni in codice macchina e dati memorizzati in formato eseguibile ed è come tale un'entità passiva.
Un processo è invece un'istanza del programma in esecuzione ed è quindi un'identità dinamica in costante cambiamento, le cui istruzioni in codice macchina sono eseguite dal processore.
E’ particolarmente utile ricordarci questo anche quando programmiamo in C.
Quando abbiamo di fronte un problema che vogliamo risolvere con un nostro programma, generalmente lo descriviamo tramite delle azioni che la macchina deve compiere per risolverlo.
Una volta compilato il codice, il passo successivo sarà di "darlo in pasto alla macchina" ed ecco che lì il nostro programma diventa uno o più processi.
Come vedremo successivamente, Linux è un sistema operativo multitasking, ovvero permette l'esecuzione di più processi contemporaneamente e in Unix uno stesso programma può generare anche più processi.
In questo capitolo inizieremo veramente a programmare, a partire da esempi banali (alcuni lo sono anche troppo, ma non abbiate fretta e presto vedrete i risultati!).

Breve storia del linguaggio C.
Per comprendere pienamente la filosofia di questo potentissimo linguaggio, è necessario dare una breve occhiata al periodo storico in cui fece la sua comparsa.
I primi embrioni di quello che sarebbe stato il C iniziarono a comparire nel 1969, grazie al lavoro di Brian Kernighan e Dennis Ritchie presso i laboratori della mitica “Bell Research”, contemporaneamente ai primi esperimenti su un nuovo sistema operativo, battezzato UNIX (a tal proposito vi rimando alle lezioni 5, 6 e 7 del corso di Sistemi Operativi).
Il C venne specificatamente creato per permettere ai programmatori di accedere ai registri interni della macchina e agli slot di I/O e di utilizzare indirizzi assoluti, fornendo nel contempo strumenti per la modularizzazione del testo dei programmi, così come veniva richiesto dai progetti sempre più corposi, sviluppati contemporaneamente da più persone, che a quell’epoca facevano la loro prima comparsa.
Durante gli anni ’60 i sistemi operativi per computer cominciarono a diventare sempre più complessi con l’introduzione di funzionalità multi-terminale e multi-processo.

Inizialmente i sistemi operativi venivano pazientemente e laboriosamente realizzati usando il codice assembler, il che faceva lievitare inevitabilmente i tempi e i costi necessari per avere calcolatori commerciabili.
Inoltre il codice assembler non si prestava ad una facile comprensione ed era perciò difficile scoprire (ed eliminare) le cause di eventuali bug che venivano via via segnalati.
La necessità di ridurre tempi e costi di produzione fu la principale motivazione che portò al linguaggio C, implementato per la prima volta in assembler su un PDP-7 della Digital Equipment Corporation.
Una volta pronta una semplice versione assembler divenne possibile riscrivere il compilatore direttamente in C; da allora, eccetto piccole porzioni di codice scritto in assembler per questioni di efficienza, il nuovo linguaggio venne usato per implementare Unix.
Questo è uno dei principali motivi che rendono il C lo strumento principe per la programmazione di sistemi basati su Unix, tra cui il nostro amato Linux.

Iniziamo col conoscere le parole chiave del linguaggio.
Il C è case-sensitive, cioè distingue tra lettere minuscole e lettere maiuscole.

Le parole chiave del C, elencate sotto, sono tutte minuscole:

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

Sul significato di ognuna di esse ci soffermeremo nel seguito; per ora vediamo come è composto un programma scritto in C. La composizione di un programma C può essere riassunta in quattro elementi di base:
- espressioni
- istruzioni
- blocchi di istruzioni
- blocchi di funzioni

A partire dalla prossima lezione cominceremo a conoscere gli elementi base del linguaggio, ovvero le espressioni e le variabili.

Espressioni, variabili e istruzioni
Lezione 22

Un'espressione è composta da operandi e operatori.
All'interno di un'espressione non possono essere usate parole chiave; ad es. non è lecito scrivere:

```
c = auto + 2;
```

Ecco alcuni esempi di espressioni lecite:

1) 2 + 3
2) d + z
3) b < a
4) 300
5) b = c + a

Omettiamo di discutere i primi quattro esempi, che sono banali somme, confronti o valori costanti e passiamo al quinto.
Tale esempio significa: "Prendi il contenuto della variabile c, prendi il contenuto della variabile a, sommali e poni il risultato nella variabile b".
Per capire quanto appena detto occorre la seguente definizione:

Una variabile è una locazione di memoria, ove si possono scrivere e/o leggere dati.

In C una variabile viene rappresentata mediante un identificatore alfanumerico.
Nel seguente esempio, a è una variabile cui assegniamo il valore 10:

```
a = 10;
```

Una variabile è costituita da tre parti:
- il nome;
- il valore;
- l'indirizzo.

Per assegnare il nome ad una variabile occorre dichiararla, cioè dire al compilatore C (v. lezione 2) di riservare uno spazio di memoria per contenerla.
Il compilatore si occuperà di cercare un'area libera di memoria da assegnare alla variabile; quest'area si trova ad un determinato indirizzo.
Per leggere il valore della variabile possiamo utilizzare sia il nome che il suo indirizzo in memoria, ma per il momento utilizzeremo solo la prima possibilità.

Avviamo emacs:

```
$ emacs
```

e scriviamo il seguente esempio:

```
#include <stdio.h>

main()
{
  int a = 10;

  printf("Valore di a      = %d\n", a);
  printf("Indirizzo di a   = %x\n", &a);
  printf("Dimensione di a = %u byte\n", sizeof(a));

  exit(0);
}
```

Salviamolo col nome var.c, compiliamolo con il comando:

```
$ gcc –o var var.c
```

ed eseguiamolo:

```
$ ./var
```

L'output che si ottiene è del tipo:

```
Valore di a      = 10
Indirizzo di a  = bffff8d8
Dimensione di a = 4 byte
```

Tralasciando il significato di alcune istruzioni che impareremo a conoscere nel seguito, il programma stampa il valore di a, che è quello che abbiamo assegnato noi stessi, l'indirizzo di memoria (in notazione esadecimale) nel quale la variabile è memorizzata e la sua dimensione, cioè il numero di byte che il compilatore ha riservato ad a.
Si noti che, poiché l'indirizzo di a è assegnato dal compilatore e dipende dallo stato della macchina (quantità di memoria, numero di processi caricati in memoria, etc.), il valore ottenuto in fase di esecuzione varia da macchina a macchina e anche tra due diverse esecuzioni del programma sulla stessa macchina.
Quello che per il momento è importante sottolineare è il fatto che le variabile non sono altro che contenitori per i dati.

Un'istruzione è un comando C completo.
Tutte le istruzioni terminano con il carattere ";", ad es.:

```
a = 1000;
c = 1;
b = a + c;
```

Supponendo che il compilatore abbia memorizzato le tre variabili in celle di memoria consecutive, il loro contenuto è il seguente:

```
a | 1000 |
b | 1001 |
c |    1 |
```

Le istruzioni possono essere raggruppate in blocchi.
I blocchi di istruzioni sono compresi tra un carattere di inizio '{' e uno di fine '}', ad es.:

```
if (a < b) {
            c = a * b;
            a = a + 1;
           }
```

In questo esempio il blocco di istruzioni è quello che segue la parola chiave if, ovvero:

```
c = a * b;
a = a + 1;
```

Il significato di questa porzione di codice C è: "Se il valore (il contenuto) della variabile a è minore di quello della variabile b, allora poni nella variabile c il valore di a moltiplicato per quello di b, ed aumenta il valore della variabile a di 1".
Le parentesi graffe servono a segnalare al compilatore di trattare tutte le istruzioni del blocco come un'unica entità, cioè come se fossero un'unica istruzione semplice.
I blocchi possono anche essere annidati, cioè avere porzioni di codice del tipo:

```
while (1) {
    b = c + 1;
    if (a < b) {
         b = a + 1;
    }
}
```

Nella lezione successiva impareremo a conoscere le funzioni e i blocchi di funzione.

Funzioni e blocchi di funzione (I)
Lezione 23

Un blocco di funzione è composto da uno o più blocchi di istruzioni e da istruzioni combinate in modo da svolgere un compito specifico.
Durante lo sviluppo dei programmi, istruzioni e blocchi di istruzioni vengono usati dalle funzioni per creare un singolo modulo, che si occupa di risolvere un singolo aspetto di un problema.
Esaminiamo la seguente funzione:

```
 1: int area_triangolo(void)
 2: {
 3:   int base, area, altezza;
 4:   base = 10;
 5:   altezza = 5;
 6:   area = (base * altezza) / 2;
 7:   return area;
8: }
```

Ogni funzione ha una dichiarazione fatta in questo modo:

```
 tipo_valore_restituito nome_funzione(tipo_parametri parametri);
```

La nostra funzione area_triangolo restituisce un valore di tipo int, ovvero un numero intero rappresentante l'area di un triangolo avente base 10.
Esaminiamo ora la linea 3, nella quale troviamo il modo per definire una variabile:

```
 tipo_variabile nome_variabile;
```

Abbiamo già visto questa sintassi nell'esempio della lezione precedente per la dichiarazione della variabile a.
Se le variabili, come in questo caso, sono dello stesso tipo, allora basta utilizzare una notazione ridotta del tipo:

```
 tipo_variabile nome_variabile1, nome_variabile2, nome_variabile3, ...;
```

Più avanti nel capitolo vedremo il significato dei tipi delle variabili.

Per comprendere il significato dell'istruzione return e del parametro void occorre sapere che una funzione in un linguaggio di programmazione ha un significato abbastanza simile alla matematica:

f: A->B
Funzione da A in B

Una funzione riceve da altre funzioni un insieme A di valori e restituisce un insieme B di valori.
Nel caso in cui A sia vuoto (come nel nostro esempio) si specifica void nei parametri della funzione:

```
 int area_triangolo(void)
                    ^^^^
```

Se è B ad essere vuoto, si specifica void prima della definizione della funzione e si omette l'istruzione return, ad es.:

```
 void nome_funzione(void)
 {
   int base;
   base = 0;
}
```

Nel caso in cui le nostre funzioni debbano restituire un valore, occorre dichiarare il tipo del valore che essa restituisce, come abbiamo fatto nella linea 1 della funzione:

```
 int area_triangolo(void);
```

In questo caso occorre usare l'istruzione return prima di chiudere il blocco funzione per restituire il valore elaborato dalla nostra funzione.
A questo punto passiamo a scrivere un primo semplice programma, che chiameremo primo.c:

```
/* Primo programma in C */
#include <stdio.h>

int main(void) {
    printf("Ciao Mondo!\n");
    return 0;
}
```

La prima cosa che notiamo sono i simboli /* e */.
Il primo ha significa “Apri un commento” e il secondo “Chiudi il commento”.

Vediamone un ulteriore esempio:

```
/*
   Questo è un commento insensato:
   secondo voi, chi è più bella:
   Laetitia Casta, Cindy Crawford o Claudia Schiffer?
 */
```

Il compilatore ignora tutto ciò che trova tra /* e */.
Queste coppie di caratteri sono chiamate delimitatori di commento.
I commenti sono molto utili sia durante la stesura del codice, per ricordarci cosa fa il programma, sia nella fase di debug, per cui occorre molta cura nella loro scrittura, perché possono essere un valido aiuto per la comprensione e la scoperta dei bug.
Una cosa importante da notare è che ogni commento aperto con /* deve essere chiuso con */, infatti in ANSI C non è lecito scrivere commenti annidati di questo tipo:

```
/*
Il programma analizza
/* le derivate ennesime */
*/
```

Funzioni e blocchi di funzione (II)
Lezione 24

Continuamo l’esame del codice primo.c, soffermandoci sulla seconda linea dell'esempio:

```
#include <stdio.h>
```

che è una direttiva al preprocessore.
Infatti il codice di un programma scritto in C può includere alcune istruzioni dirette al compilatore, dette appunto direttive al preprocessore, che sono contenute in linee che iniziano con il carattere #.
Il significato della nostra linea di programma è:

"Cerca su disco il file stdio.h e inserisci in questo punto del programma tutto il contenuto di quel file".

I file *.h vengono chiamati header e fanno generalmente riferimento ad una libreria.
Il file stdio.h (stdio = standard input/output) viene usato in tutti i programmi che richiedono una qualche forma di immissione o emissione dati (I/O, input/output), ovvero la quasi totalità dei programmi.

Inserendo il file stdio.h nel programma è possibile utilizzare le funzioni di input/output, in questo si è utilizzata la routine printf.
L’inclusione di un file header può essere fatta in due modi diversi:

```
#include <stdio.h>
#include "stdio.h"
```

Apparentemente le due linee di codice sembrano uguali, se escludiamo le virgolette, ma dal punto di vista del compilatore la differenza è sostanziale.
Infatti:
- Le parentesi angolari (< >) che precedono e seguono il nome di un file header indicano al preprocessore di cercare

il file header richiesto nella directory standard degli header (/usr/include).
- Le virgolette (" ") indicano al preprocessore di cercare il file header prima nella directory corrente e poi, se non viene trovato, nella directory standard.

Se scriviamo:

```
#include "/home/deneb/myio.h"
```

specifichiamo esattamente il percorso dove trovare l'header myio.h.

Alla riga successiva dell'esempio troviamo la funzione:

```
int main(void)
```

che restituisce un valore intero e non riceve nulla in ingresso.
All'interno della funzione main() viene chiamata un'altra funzione:

```
printf("Ciao Mondo!\n");
```

che è una funzione di I/O contenuta nell'header stdio.h, il cui compito è quello di stampare sullo schermo i caratteri compresi tra i doppi apici.
Il carattere \n (newline) indica che non appena viene scritta la stringa ci sia un ritorno del carrello (cioè si vada a capo).
Così come avviene in ogni funzione, per la printf tutto ciò che è compreso tra le parentesi tonde costituisce l'argomento della funzione stessa.

Infine la funzione main() restituisce il valore 0, che indica che il programma è terminato correttamente.
A tal proposito dobbiamo sapere che esistono due macro definite dalla libreria GNU, EXIT_SUCCESS e EXIT_FAILURE, rappresentanti i valori di uscita di un programma rispettivamente nel caso in cui venga eseguito correttamente e nel caso in cui vi siano errori.
In Linux, è buona pratica specificare nella funzione principale (ovvero main) un valore di uscita per il programma.
Per definizione, se il programma ha eseguito tutte le operazioni correttamente uscirà con valore 0, altrimenti con valore maggiore di 0 (generalmente 1).

Vedremo in seguito che valori di uscita diversi da 1 possono essere utilizzati per facilitare il debugging del codice o per comunicare con altri processi.
Le macro EXIT_SUCCESS e EXIT_FAILURE sono state create per migliorare la leggibilità del codice, infatti le istruzioni:

```
return 1;
return EXIT_FAILURE;
```

sono equivalenti.
Per utilizzarle occorre includere nel codice sorgente il file header <stdlib.h>.
Nella tabella seguente sono riassunti i valori delle due macro descritte.

Macro	Valore
EXIT_SUCCESS	0
EXIT_FAILURE	1

Abbiamo visto dunque che una funzione può richiamarne un'altra, ma, perché ciò sia possibile, deve essere visibile alla funzione che la richiama; è per questo motivo che l'header è posto all'inizio del programma e non in fondo.

In generale un programma C è così composto:

1) Comandi per il preprocessore
2) Definizione di tipi
3) Prototipi di funzioni (dichiarazione dei tipi delle funzioni e delle variabili passate alle funzioni)
4) Variabili
5) Funzioni

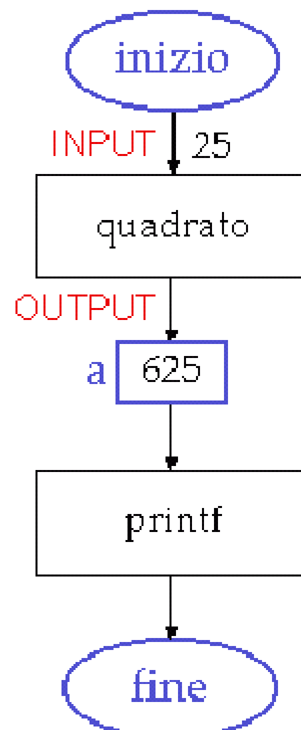
L'esecuzione di ogni programma, al di là della sua struttura, inizia sempre dal main() e prosegue discendendo per ogni funzione e istruzione in esso contenuta fino al termine del blocco funzione del main stesso.

Esempio ed esercizi
Lezione 25

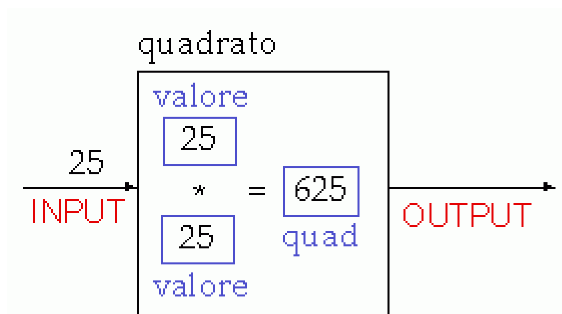
Per riassumere quanto detto finora, ecco un buon esempio, che chiameremo quadrato.c, col quale potersi confrontare e verificare quanto appreso sinora:

```
/* File header */
#include <stdio.h>
#include <stdlib.h>

/* Dichiarazione delle funzioni definite dal programmatore */
int quadrato(int valore); /*
                           * La funzione quadrato prende un valore intero
                           * e ne restituisce il quadrato.
                           */

int main(void)
{
    int a;
    a = quadrato(25);
    printf("Valore del quadrato di 25: %d\n", a);
    return EXIT_SUCCESS;
}


/* Implementazione della funzione quadrato */
int quadrato(int valore)
{
    int quad;
    quad = valore * valore;
    return quad;
}
```

Il funzionamento del programma è schematizzato in figura.



Quando il programma viene avviato viene innanzitutto eseguita la funzione main() che dichiara la variabile a di tipo intero: questa istruzione istruisce il compilatore riguardo all'allocazione di un'area di memoria di dimensione opportuna per contenere un valore intero.
L'istruzione successiva assegna ad a il valore di output della funzione quadrato cui viene dato in input il valore 25.
Dopo che la funzione quadrato è stata eseguita il suo valore di output è posto nella locazione di memoria di a.
In seguito la funzione di libreria printf stampa a video la stringa "Valore del quadrato di 25: ", seguita dall'attuale valore di a (cioè 625), dopodiché va a capo.
Quindi il programma termina indicando il successo dell'esecuzione.
La figura successiva mostra la struttura della funzione quadrato.



Nell'esempio possiamo notare la dichiarazione della funzione quadrato di tipo intero, che ha come parametro una variabile intera e restituisce un valore intero, rappresentato dal quadrato dell'input stesso.
Bisogna ricordare che le funzioni vengono viste in C come scatole nere (black box), cioè tutto il loro contenuto è invisibile all'esterno, perciò tutte le variabili dichiarate al loro interno non hanno visibilità all'esterno.
Nel nostro caso se si usasse la variabile quad all'esterno della funzione quadrato si verificherebbe un errore, a meno che tale variabile non venga dichiarata nuovamente all'interno della funzione main.
Durante l'esecuzione, come è visibile dallo schema, alla variabile a viene assegnato il valore della funzione quadrato e poi tale valore è stampato con l'istruzione:

```
printf("Valore del quadrato di 25: %d", a);
```

che significa: "stampa la stringa Valore del quadrato di 25: e aggiungi ad essa il valore di tipo intero della variabile a."
Il simbolo %d sta appunto ad indicare il valore di tipo intero della variabile a in formato decimale.

Per questo l'output a video del programma è:

```
Valore del quadrato di 25: 625
```

Adesso eccovi alcuni esercizi che siete già in grado di svolgere e nei quali vi potete cimentare in attesa della prossima lezione.

Modificare l'esempio primo.c in modo che l'output sia:

```
Ciao,
quando
vai via?
```

usando prima un'unica chiamata a printf e poi tre diverse chiamate a printf.
Riprendere l'esempio di funzione area_triangolo, modificandola in modo da accettare in ingresso due parametri di tipo intero: uno per la base e uno per l'altezza.
Stampare nella funzione principale l'area del triangolo, avendo assegnato i valori base=10 e altezza=25.
Modificare nell'esempio quadrato.c il corpo della funzione principale (main), affinché alla funzione quadrato sia passata una variabile anziché il valore 25.
Tale variabile deve essere precedentemente dichiarata e inizializzata a piacere.

Indicare riga per riga gli errori presenti nel seguente programma:

```
#include "stdio.h"
*/ Li troverai gli errori? */
int func2(int valore1, int valore2);
```

```
Main()
{
int b;
a=func(b);
b=4;b=b+a;
a=func2(a,b);
return EXIT_SUCCESS;
}
void func(int value)
{
int c;
c=value*4;
return 4;
}
int func2(int valore1, int valore2)
{
int c;
c= valore1+valore2;
}
/* Hai trovato gli errori? /*
```

Due direttive al preprocessore: #define e #undef
Lezione 26

Come già sottolineato, una direttiva al preprocessore è un'istruzione diretta al compilatore, che espande l'ambiente di programmazione stesso.
Le direttive al preprocessore sono dette anche macro e il preprocessore viene indicato più precisamente col nome di preprocessore di macro.
In buona sostanza, il preprocessore C è un programma interno al compilatore, che esamina un file sorgente C e compie determinate modifiche su di esso, basandosi sulle direttive a lui destinate.
Per ora ci interessiamo solo a due macro, #define e #undef , mentre altre le tratteremo in seguito.

Molto spesso durante un programma si ha la necessità di sostituire un valore con un testo.
Ad es. supponiamo di dover creare un programma che utilizzi il valore matematico pi-greco: sarebbe preferibile utilizzare sempre la stringa PI_GRECO ad ogni occorrenza del valore, piuttosto che scrivere di volta in volta il valore 3.1415926536.

E' in casi come questo che si rivela molto utile la macro #define sia per funzionalità che per leggibilità.
La forma generale è:

```
#define nome-macro testo-sostitutivo
```

oppure

```
#define nome-macro
```

Nel caso dell'esempio di pi-greco scriveremo:

```
#define PI_GRECO 3.1415926536
```

L'uso più frequente di #define in Linux è quello di condizionare la portabilità del codice, cioè se viene definita una data macro allora possono essere incluse o meno diverse estensioni della libreria standard C (glibc); in questo caso il nome-macro deve essere definito nella glibc.
L'esempio più classico di utilizzo è il seguente:

```
#define FALSO 0
#define VERO  1
```

Definendo questa direttiva, nel corpo del programma la macro FALSO viene sostituita con il valore 0 e la macro VERO con il valore 1.
Questa situazione è simile a quella vista nei Makefile: "Tutte le volte che una macro viene incontrata, la si sostituisce con il gesto ad essa corrispondente".

Date un'occhiata al seguente esempio che non necessita di ulteriori spiegazioni:

```
#include <stdio.h>
#include <stdlib.h>

#define FALSO 0
#define VERO 1

int main(void) {
   printf("VERO:%d\n", VERO);
   printf("FALSO:%d\nDue:%d\n", FALSO, VERO+1);
   return(EXIT_SUCCESS);
   }
```

Un altro caso di #define che abbiamo già usato, inconsapevolemente, è EXIT_SUCCESS, dichiarato nella libreria standard del C.

utile uso di #define è definire i messaggi di errore che si possono ripetere durante l'esecuzione del programma, ad es.:

```
#define MSG_ERRORE "This is I LOVE YOU: try format to save your disk!"
```

Dopo una tale definizione, una possibile istruzione printf nel corpo del programma è:

```
printf(MSG_ERRORE);
```

Se siete dei buoni osservatori avrete certamente notato che tutte le macro definite sono maiuscole.
Questa è una convenzione, ormai consolidata tra i programmatori, che permette, durante la lettura di un programma, di rendersi conto a prima vista della presenza degli identificatori e quindi delle sostituzioni.

La direttiva #undef cancella la definizione associata alla macro da #define.

```
#define VALORE 1  /* imposto VALORE a 1 */
#undef VALORE     /* ma non mi piace, per esigenze preferisco 2 */
#define VALORE 2  /* imposto VALORE a 2 */
```

In questo caso se non fosse stata usata la #undef, avrei causato un messaggio di errore di "multiple define", ciò significa che le definizioni:

```
#define DISPARI 3
#define DISPARI 5
```

non sono lecite, in quanto la definizione di una macro deve essere unica.
Dopo che una definizione di macro è stata cancellata con #undef, ogni sua istanza non attiverà più la sostituzione corrispondente.
Questa possibilità del preprocessore viene solo usata in caso di compilazione condizionale, che vedremo più avanti durante il corso.

Rappresentazione delle informazioni
Lezione 27

All'interno di un calcolatore digitale ogni informazione viene codificata in bit (acronimo per binary digit), ovvero in sequenze di numeri binari, costituiti dalle cifre 0 e 1.
Di per sé due sole cifre possono sembrare poche per memorizzare l'enorme quantità di dati (con i loro possibili significati) che un computer deve essere in grado di manipolare; infatti essi possono rappresentare stati di acceso/spento, vero/falso, bianco/nero e in genere solo coppie di valori di opposto significato.
In realtà utilizzare un insieme così ristretto di caratteri base semplifica enormemente il lavoro di chi progetta l'hardware del calcolatore, riducendo inoltre la possibilità di errori da parte della macchina (le cifre 0 e 1 vengono rappresentate dall'hardware per mezzo di due valori di tensione, ad es. 0V per il bit 0 e 5V per il bit 1, ma questi valori dipendono dal tipo di microchip utilizzati dal costruttore, nonché dall'ambiente di utilizzo cui il calcolatore è destinato).

Per permetterci di lavorare con informazioni più complesse, impossibili da rappresentare con una sola coppia di valori, i bit vengono raggruppati in modo da formare delle codifiche cui è possibile assegnare dei valori convenzionali.

Innanzitutto vediamo che i bit vengono raggruppati per potenze di 2 ed in genere in informatica si utilizzano le seguenti unità di misura:

Unita di misura
Numero bit
Numero byte
1 byte
8 bit
1 byte
1 KB (Kilo-byte)
(1024 * 8) bit
1024 byte
1 MB (Mega-Byte)
(1.048.576 * 8) bit
1024KB = 1.048.576 byte
1 GB (Giga-Byte)
(1.073.741.824 * 8) bit
1024MB = 1.073.741.824 byte
1 TB (Tera-Byte)
(1.099.511.628.000 * 8) bit
1024GB = 1.099.511.628.000 byte

ome vedremo, nei comuni PC i caratteri vengono rappresentati con un byte, mediante la codifica ASCII, che associa ad ogni byte un carattere dell'alfabeto o un particolare carattere di controllo del computer.
Ad es. in ASCII la stringa binaria 01000001 corrisponde al carattere 'A'.
Supponiamo di voler scrivere dei caratteri su un quaderno di carta a quadretti, dove ciascun quadretto corrisponda ad un carattere: poiché generalmente in una pagina di quadernone vi sono 2500 quadretti in media, quanti fogli consumeremo per riempire 1440 KB, quantità genericamente contenuta in un floppy?
Bene, la risposta è circa 590 pagine di quadernone: questo solo per renderci conto della quantità di informazioni che le nostre periferiche si scambiano e memorizzano.
La rappresentazione delle informazioni è importante ai nostri fini per capire come vengono immagazzinate le variabili in memoria, ovvero il tipo di dato della variabile stessa.

Infatti, fin dai primi esempi tutte le volte che abbiamo usato delle variabili si è sempre indicato, nella loro dichiarazione, il tipo di dato.
Ogni funzione che usi al proprio interno delle variabili deve dichiararne il tipo:

```
int funzione (void)
{
    int ciao;
    float test;

    ciao = 0;
    test = 0.1;

    return (ciao);
}
```

All'inizio di questa semplice funzione troviamo indicati i tipi di due variabili alle quali sono poi effettuati degli assegnamenti; questa prima parte prende il nome di parte dichiarativa.
Come abbiamo già accennato nelle lezioni precedenti, la parte dichiarativa di un programma C comunica al compilatore i nomi di tutti gli identificatori definiti dagli utenti, nonché l'utilizzo di ciascun identificatore.

Inoltre dice anche il tipo di informazione che deve essere memorizzata nella locazione di memoria riservata ad ogni variabile.
In C esistono cinque tipi di dati primari che sono riassunti nella tabella seguente.

Tipo
Parola chiave
Carattere
char
Intero

int
Reale
float
Reale in doppia precisione
double
Void
void

I numeri interi, definiti in C dalla parola chiave int, possono essere rappresentati internamente con 2 metodi diversi:
i) rappresentazione in modulo e segno
ii) rappresentazione in complemento a 2
che vedremo nella prossima lezione.

Rappresentazione di numeri interi e reali
Lezione 28

Nella scorsa lezione abbiamo appreso che i numeri interi possono essere rappresentati all'interno del calcolatore in due modi distinti:
i) modulo e segno
ii) complemento a 2

Per la rappresentazione in modulo e segno si procede nel seguente modo:
1- A partire dal numero decimale si effettua la conversione in binario dividendo ciclicamente il modulo per 2 e prendendo i resti della divisione finché il risultato non è 0;
2- si pone ad 1 il primo bit della stringa di rappresentazione se il numero è negativo, 0 se è positivo;
3- si pongono nella stringa gli altri n bit ottenuti dalla divisione, inserendo degli zeri a sinistra fino a completare la stringa.
Vediamo due esempi.
Supponiamo di utilizzare 8 bit per rappresentare un intero e di dover codificare i numeri +4 e –5.

endo 4 e lo divido per 2, ottengo 2 con resto 0;
- divido il risultato precedente (2) per 2, ottengo 1 con resto 0;
- divido 1 per 2, ottengo 0 con resto 1.
Il numero binario corrispondente al modulo 4 è 100 e poiché il numero è positivo il primo bit sarà 0.
Dunque il numero +4 rappresentato in modulo e segno con 8 bit è 00000100 (in grassetto sono evidenziati il bit di segno e la codifica del modulo).
Analogamente procediamo per il numero -5:
- 5 : 2 = 2 resto 1;
- 2 : 2 = 1 resto 0;
- 1 : 2 = 0 resto 1.
Il numero binario corrispondente al modulo 5 è 101 e poiché il numero è negativo il primo bit sarà 1.
Il numero -5 rappresentato in modulo e segno con 8 bit è 10000101.

Per la rappresentazione in complemento a 2 si procede nel seguente modo.
Sia n la dimensione massima della stringa di bit mediante la quale rappresentiamo un intero (generalmente n vale 16, 32 o 64).

a a un generico intero compreso tra -2n-1 e 2n-1:
- Se a >= 0 si converte in binario;
- Se a < 0 si calcola a + 2n e si converte il risultato in binario senza segno su n bit.
Supponiamo che sia n = 4 (valore improbabile ma utile per semplificare il calcolo) e di dover rappresentare in complemento a 2 il numero a = -5.
Poiché a è negativo, gli sommiamo il valore 24 = 16 e otteniamo il valore 11, che convertiamo in binario con il metodo del modulo e segno:

11 : 2 = 5 resto 1;
5 : 2 = 2 resto 1;
2 : 2 = 1 resto 0;
1 : 2 = 1;

Dunque il valore –5 rappresentato in complemento a 2 con 4 bit è 1011.
In generale con n bit è possibile rappresentare un intero il cui valore è compreso tra -2n-1-1 e +2n-1-1 se si utilizza la rappresentazione in modulo e segno, tra -2n-1 e 2n-1-1 se si utilizza il complemento a due.

Nelle tabelle seguenti sono mostrati i range di valori del tipo di dati int al variare del numero di bit usati per la codifica per entrambe le rappresentazioni.

Numero di bit	Modulo e segno
8	da –127 a +127
16	da –32.767 a +32.767
32	da –2.147.483.647 a +2.147.483.647
64	da –9.223.372.036.854.775.807 a +9.223.372.036.854.775.807

Numero di bit	Complemento a due
8	da –128 a +127
16	da –32.768 a +32.767
32	da –2.147.483.648 a +2.147.483.647
64	da –9.223.372.036.854.775.808 a +9.223.372.036.854.775.807

In C i numeri reali vengono rappresentati mediante la notazione scientifica.
Un numero reale N espresso nella notazione scientifica (detta floating point) assume la forma:

N = m * (b elevato al numero e)

dove b è la base, m la mantissa, ed e l'esponente.
La maggior parte dei calcolatori rappresenta i numeri reali secondo lo standard IEEE 754, che tratta solo valori reali normalizzati, cioè aventi mantissa costituita da un solo bit di valore 1.
Per questo motivo si trascura la parte intera della mantissa e si considera solo la sua parte frazionaria.

In C per i tipi di dati float e double si usano rispettivamente le rappresentazioni in singola e doppia precisione, organizzate secondo quanto descritto in tabella:

Tipo di dato	Numero di bit usati	Segno (S)	Esponente (E)	Parte frazionaria della mantissa (PF)
Float	32	1 bit	8 bit	23 bit
Double	64	1 bit	11 bit	52 bit

Gli esponenti per i tipi float e double sono rappresentati in eccesso-127 ed eccesso-1023 rispettivamente: ciò vuol dire che al valore effettivo dell'esponente viene sommato il valore 127 (o 1023) e lo si converte in un numero binario intero senza segno su 8 (o 11) bit.
Come esempio determiniamo la rappresentazione a singola precisione di N = -423.
Poiché N è negativo il bit di segno è 1.
In binario il valore 423 diventa 110100111 che in formato IEEE 754 si scrive nella forma 1,10100111 * 28.
L'esponente 8 in eccesso-127 diventa 135, che convertito in binario è 10000111.
Dunque la rappresentazione all'interno del calcolatore sarà:
1 10000111 10100111000000000000000

Codifica dei caratteri
Lezione 29

ontinuiamo il nostro viaggio all'interno del calcolatore, esaminando il modo in cui vengono codificati i caratteri.
Normalmente i caratteri sono rappresentati come stringhe di 7 bit mediante una codifica chiamata ASCII (acronimo di American Standard Code for Information Interchange).
Sulle macchine moderne, ciascuno dei 128 caratteri ASCII è dato dai 7 bit meno significativi di un ottetto.
Gli ottetti sono riuniti in parole (word) di memoria, la cui dimensione dipende dall'architettura usata; ad es. con word da 32 bit una stringa di 6 caratteri occupa solamente 2 parole di memoria.
Per visualizzare la mappa dei caratteri ASCII, provate a scrivere al prompt di Linux il comando:

man 7 ascii

Osserviamo che abbiamo utilizzato il termine 'ottetto', che è formalmente corretto ma raramente utilizzato: la maggior parte delle persone si riferisce a un ottetto come ad un byte e ritiene che i byte siano lunghi 8 bit.
Per essere corretti, il termine 'byte' è più generale; ad es. in passato ci sono state macchine a 36 bit con byte di 9 bit, anche se probabilmente ciò non capiterà più in futuro.
Purtroppo non tutto il mondo usa i codici ASCII, a causa del fatto che non contiene molti caratteri accentati e alcuni caratteri speciali necessari per molte lingue.

Diversi sono stati i tentativi di risolvere questo problema.
Attualmente tutti fanno uso dell'ottavo bit non usato dai codici ASCII (il più significativo), cosicché l'intero set di caratteri rappresentabili è esteso a 256, di cui la prima metà (da 0 a 127) è costituita dai codici ASCII originari.
Quello più largamente utilizzato è il set di caratteri 'Latin-1' (o più formalmente ISO 8859-1), che è il set di caratteri predefinito per Linux, HTML e X.
Per visualizzare la mappa dei caratteri Latin-1, potete riferirvi alla corrispondente pagina del manuale Linux scrivendo al prompt il comando:

man 7 iso_8859_1

Microsoft Windows usa una versione mutante di Latin-1, che aggiunge alcuni caratteri come le virgolette destre e sinistre, in posizioni lasciate libere da Latin-1 per ragioni storiche (per una resoconto severo dei problemi che ciò ha provocato, vedere la pagina cliccando qui).
Latin-1 gestisce le principali lingue europee, tra cui inglese, francese, tedesco, spagnolo, italiano, olandese, norvegese, svedese e danese.
Tuttavia non è ancora sufficiente, per cui esistono altre serie di set di caratteri da Latin-2 a Latin-9, per rappresentare il greco, l'arabo, l'ebraico e il serbo-croato (chi è interessato può trovare maggiori dettagli alla pagina ISO alphabet soup).

soluzione definitiva è uno standard (enorme) a 16 bit chiamato Unicode, che è identico a Latin-1 nelle 256 posizioni più basse, mentre nello spazio successivo comprende caratteri per le lingue greco, cirillico, armeno, ebraico, arabo, devanagarico, bengalese, gurmukhi, gujarati, oriya, tamil, telugu, kannada, malese, tailandese, lao, georgiano, tibetano, giapponese kana e il set completo del coreano hangul moderno e un set unificato di ideogrammi cinesi/giapponesi/coreani (CJK).
Per maggiori dettagli vedere la Unicode Home Page .

Detto questo dobbiamo cominciare a vedere come ci possano essere utili tutte queste informazioni in un programma C.
Sappiamo che il nostro calcolatore memorizza i dati al suo interno mediante opportune sequenze di bit, ma noi vogliamo poter lavorare con i dati senza dover conoscere la numerazione binaria e la corrispondenza tra i codici binari e il loro significato.
Ad es. vogliamo vedere a video il carattere 'A' piuttosto che la sua codifica ASCII (01000001), in quanto quest'ultima ci risulta piuttosto difficile e sicuramente non immediata.
Fortunatamente il C non ci costringe a sforzi mnemonici inutili e ci aiuta a lavorare in maniera più semplice, fornendoci degli strumenti opportuni.
A partire dalla prossima lezione vedremo in che modo il C tratta i nostri dati.

I tipi di dati fondamentali
Lezione 30

Nella lezione 22 abbiamo accennato al fatto che i dati che il nostro programma deve elaborare sono contenuti in variabili.
Nonostante il calcolatore memorizzi tutte le informazioni come sequenze di cifre binarie, noi diamo ad ognuna di esse un significato differente: ad es. possiamo dover lavorare con stringhe di caratteri oppure con valori numerici.
Un byte in cui è memorizzata la sequenza binaria 00111111 può contenere la rappresentazione in complemento a 2 dell'intero 63 oppure la codifica ASCII del carattere '?'; lo stesso byte può dunque rappresentare un valore numerico o un carattere dell'alfabeto: può contenere cioè valori di tipo diverso.

In C esistono 4 tipi fondamentali:
1. gli interi (rappresentati con la parola chiave int);
2. i valori in virgola mobile, o floating point (rappresentati con la parola chiave float);
3. i valori in doppia precisione (double);
4. i caratteri (char).

Ognuno di questi tipi rappresenta un certo intervallo (range) di valori che una variabile può assumere, dipendente dalla piattaforma.
Per questa ragione, quando si dichiara una variabile occorre indicarne il tipo, in modo che il compilatore possa allocare lo spazio di memoria necessario a contenere tutti i possibili valori che la variabile può assumere.

Una variabile che assume valori interi può essere dichiarata come:

int
long int
short int
unsigned int
unsigned long int
unsigned short int

a differenza tra int, long int e short int risiede nell'ampiezza dell'intervallo di valori che possono assumere, a sua volta legata al numero di bit utilizzati per rappresentare un valore di quel tipo.
Se si premette la parola chiave unsigned viene specificato che la variabile assume solo valori positivi.
La tabella seguente indica il numero di byte utilizzati dagli attuali PC a 32 bit per memorizzare i tipi interi, e quindi gli intervalli di valori che variabili di quel tipo possono assumere:

Tipo	Dimensione in byte	Range
int	4	da –2.147.483.648 a +2.147.483.647
long int	4	da –2.147.483.648 a +2.147.483.647
short int	2	da –32.768 a +32.767
unsigned int	4	da 0 a 4.294.967.295
unsigned long int	4	da 0 a 4.294.967.295
unsigned short int	2	da 0 a 65535

Possiamo notare che nei PC a 32 bit non esiste alcuna differenza tra i tipi int e long int, ma ciò non è vero se si considerano architetture con un diverso grado di parallelismo.
E' buona prassi di programmazione dichiarare le variabili della minima dimensione sufficiente a contenerle, specificando, quando ciò avviene, se è di tipo unsigned; quest'ultima opzione, del resto, raddoppia il massimo valore

positivo che la variabile può contenere.

Per conoscere quanti byte sono utilizzati dalla nostra macchina per memorizzato un certo tipo di dato, si può utilizzare l'operatore sizeof(arg) che restituisce la dimensione di arg espressa in byte.
Ad es. il seguente programma stampa a video le dimensioni associate ai vari tipi interi del C:

```
#include <stdio.h>

int main(void)
{
        printf("Dimensione int        = %d\n", sizeof(int));
        printf("Dimensione long int  = %d\n", sizeof(long int));
        printf("Dimensione short int = %d\n", sizeof(short int));
        printf("Dimensione unsigned int        = %d\n", sizeof(unsigned int));
        printf("Dimensione unsigned long int  = %d\n", sizeof(unsigned long int));
        printf("Dimensione unsigned short int = %d\n", sizeof(unsigned short int));

        return 0;
}
```

Il cui risultato su un sistema Intel x086 è:

```
Dimensione int        = 4
Dimensione long int  = 4
Dimensione short int = 2
Dimensione unsigned int       = 4
Dimensione unsigned long int  = 4
Dimensione unsigned short int = 2
```

Il risultato ottenuto può non coincidere con quello mostrato sopra nel caso utilizziate un'architettura diversa dalla mia.

Una variabile che può assumere valori in virgola mobile può essere dichiarata come:

float
double

La differenza tra i due tipi precedenti consiste nella precisione con la quale i numeri reali vengono rappresentati; infatti il tipo double utilizza generalmente il doppio dello spazio di memoria del tipo float, fornendo una rappresentazione più accurata (ma anche più complessa da gestire) dei valori reali.
Come esercizio, modificate il programma precedente in modo da visualizzare il numero di byte che la vostra macchina utilizza per questi due tipi di dati.

I tipi di dati fondamentali - Parte II
Lezione 31

Completiamo oggi l'elenco dei tipi di dati fondamentali del C con il tipo char, che permette di memorizzare caratteri.
Per indicare una costante carattere si utilizza il singolo apice, ad es:

```
char sesso, scelta;
...
sesso = 'm';
scelta = 'y';
```

L'insieme dei valori assunti da una variabile di tipo char è costituito da lettere dell'alfabeto, cifre numeriche e segni di interpunzione e dipende essenzialmente dalla codifica adottata dalla macchina (v. lezione 29).

E' importante sottolineare che l'uso di valori numerici interi per la codifica dei caratteri rende le variabili int e char interscambiabili in numerose occasioni ed è una prassi diffusa tra i programmatori, anche se spesso abusata.
Chi sostiene la necessità "morale" di scrivere codice elegante e leggibile può non approvare l'utilizzo di questo stratagemma, tuttavia è bene notare che risulta essere molto utile in varie circostanze, ad es. per gli algoritmi di ordinamento delle stringhe.

A proposito di stringhe, avrete notato l'assenza di un tipo specifico per esse: infatti il C standard tratta le stringhe per mezzo di array di caratteri.

Un array è un vettore di elementi di un dato tipo (che deve essere lo stesso per tutti), ognuno dei quali è indirizzabile per mezzo della sua posizione all'interno dell'array.
Per dichiarare un array in C si utilizza una particolare notazione in cui il nome dell'array è seguito dalle parentesi quadre.
Ad es.

char nome[6];

definisce un array di 6 caratteri, indicato con nome.
Notiamo che anteposto al nome dell'array è il tipo degli elementi che esso deve contenere e che all'interno delle parentesi quadre [ ] è indicato il numero di elementi all'interno dell'array.

Una volta dichiarato l'array, è possibile riferirsi ad un suo elemento specificandone la posizione attraverso un indice all'interno di parentesi quadre, ad es.

nome[1] = 'n';

assegna all'elemento in posizione 1 dell'array il valore 'n'.

Attenzione: è bene ricordare sempre che la numerazione degli elementi di un array parte da 0 fino alla dimensione dell'array-1.
Quindi nome[0] è il primo carattere, nome[1] il secondo e così via fino a nome[5] che è il sesto ed ultimo carattere nell'array.
L'elemento nome[6] non esiste e accedere a tale elemento provoca, in fase di esecuzione, un errore di accesso in memoria e tipicamente la terminazione con errore del nostro programma.

pprofondiremo in seguito il discorso sugli array e per il momento vedremo quello che ci interessa per poter lavorare con le stringhe.

Una stringa è un array di caratteri terminanti con un carattere null (indicato con '\0'), detto carattere terminatore.
Una stringa costante in C viene indicata racchiudendo tra doppi apici i caratteri che la compongono (escluso il carattere terminatore).
Ad es.:

" Questa e' una stringa "

è una stringa, o meglio un array di 22 caratteri (incluso '\0' che il C inserisce automaticamente).
Si noti che '\0' è diverso dallo '0' ASCII, in quanto diverse sono le loro rappresentazioni binarie, 00000000 per il primo e 00110000 per il secondo.
Inoltre 'Q' e "Q" hanno un significato diverso: la prima rappresenta il carattere Q, la seconda la stringa formata dai due caratteri 'Q' e '\0'.

Particolare importanza ha la stringa nulla, rappresentata con due doppi apici consecutivi "", che contiene il solo carattere terminatore.

A questo punto ci si potrebbe chiedere come inserire in una stringa dei caratteri speciali, come il carattere di tabulazione, il backspace e il ritorno a capo.
Il C permette di esprimere questi ed altri caratteri speciali in modi differenti, ma per alcuni di essi esistono delle sequenze di escape, costituite dalla barra rovesciata (o backslash) seguita da una lettera.

Il linguaggio C definisce le seguenti sequenze di escape:

\a
alarm (allarme)
\b
backspace
\f
form feed (salto pagina)
\n
new line (a capo)

\r
return (ritorno carrello)
\t
tab orizzontale
\v
tab verticale
\?
punto interrogativo
\'
apice singolo
\"
apice doppio
\\
backslash

Si noti che il carattere '\' è utilizzato per specificare al compilatore che la lettera che lo segue deve essere interpretata con un significato speciale; per questo motivo, quando la si vuole includere in una stringa come carattere, occorre utilizzare la sequenza di escape corrispondente.

Operatori matematici, relazionali e logici.
Lezione 32

Con la lezione di oggi cominciamo a vedere in che modo è possibile lavorare sui tipi di dati fondamentali visti nelle lezioni precedenti.
Abbiamo già visto che l'assegnamento in C viene effettuato utilizzando il simbolo "=" (uguale), ad es.:
int x = 0; Oltre agli operatori aritmetici standard:

+ (somma)
- (sottrazione)
* (moltiplicazione)
/ (divisione)

esiste l'operatore % (modulo) per gli interi, che ritorna il resto di una divisione intera.
Ecco un semplice es. che riassume l'utilizzo di questi operatori per gli interi:

```
int x = 10;
int y = 3;
int h, j, k, u, v;
h = x + y; /* h contiene il valore 13 */
j = x - y; /* j contiene il valore 7  */
k = x * y; /* k contiene il valore 30 */
u = x / y; /* u contiene il valore 3  */
v = x % y; /* v contiene il valore 1  */
```

Per gli interi il C offre anche gli operatori di incremento (++) e decremento (--), che possono essere preposti o posposti alla variabile.
Se sono preposti il valore incrementato o decrementato è calcolato prima che l'espessione sia valutata, se sono posposti il valore viene calcolato dopo la valutazione dell'espressione, ad es.:

```
int x, z=2;
es. 1)
x = (++z)-1; /* Prima viene incrementato z al valore 3 e poi viene assegnato a x il valore 3-1=2 */
es. 2)
x = (z++)-1; /* Prima viene assegnato a x il valore 2-1=1 e poi incrementato z al valore 3 */
```

Ecco un ulteriore esempio:

```
int x, y, w;
x=((++y) - (w--)) % 10;
```

che equivale alle seguenti istruzioni:

```
int x, y, w;
y++;
```

x = (y - w) % 10;
w--;

È importante sottolineare che un'istruzione del tipo x++ è più veloce della corrispondente x = x+1. L'operatore "%" (modulo) può essere utilizzato solamente con le variabili di tipo integer; la divisione "/" è utilizzata sia per gli integer che per i float. A proposito della divisione riportiamo un altro esempio:
f = 5/2;
in cui f assumerà il valore 1, anche se è stato dichiarato come float. Infatti, di regola, se entrambi gli argomenti della divisione sono integer, allora verrà effettuata una divisione intera. Per avere un risultato di tipo float è necessario scrivere:
f = 5.0/2;
oppure:
f = 5/2.0;
oppure ancora:
f = 5.0/2.0;

In questo modo f assumerà il valore 2.5. Esiste, inoltre, una forma contratta per espressioni del tipo:
expr1 = expr1 op expr2
dove op è un operatore.

Infatti, queste espressioni possono scriversi nella forma:
expr1 op= expr2
Ad es.:
x = x+5;
può essere scritta in forma contratta:
x += 5;
e ancora:
x = x * (y+7);
può diventare:
x *= y+7;

E' importante notare che l'espressione x *= y+3 corrisponde a x = x*(y+7) e non a x = (x*y)+7.
Gli operatori relazionali o di confronto vengono utilizzati per testare delle condizioni di uguaglianza e disuguaglianza e sono utilizzati spesso nei programmi.
Per testare l'uguaglianza si usa:

==
per la disuguaglianza:
>!=
Vi sono poi gli operatori:
< (minore)
> (maggiore)
<= (minore o uguale)
>= (maggiore o uguale)

Gli operatori relazionali hanno una precedenza inferiore agli operatori aritmetici, perciò un'espressione del tipo:

i <= lim-1
viene interpretata come:
i <= (lim-1)

Occorre fare molta attenzione all'utilizzo del test di uguaglianza, perché spesso avviene che ci si dimentica di mettere il secondo segno di uguale, col risultato che invece di testare una condizione si esegue un assegnamento (vedremo meglio questo possibile errore nella prossima lezione).
Infine vediamo gli operatori logici, che sono due:

&& (AND)
|| (OR)

Le espessioni connesse da && e || vengono valutate da sinistra a destra e la valutazione si blocca non appena si determina la verità o la falsità dell'intera espressione.
Come vedremo in seguito questa proprietà risulta molto utile per evitare certi errori durante l'esecuzione dei programmi, come divisioni per zero o superamento dei limiti di un array.
Nella prossima lezione vedremo come utilizzare alcuni di questi operatori per controllare il flusso di esecuzione dei nostri programmi.

Strutture di controllo - Parte I
Lezione 33

Ogni programma che si rispetti necessita, nella maggior parte dei casi, di dover intraprendere operazioni diverse a seconda del verificarsi o meno di determinate condizioni.
Ad es. se l'utente immette un valore non adeguato come risposta ad un prompt del programma, l'esecuzione può non poter procedere regolarmente e si deve prevedere un comportamento opportuno per ripristinare la correttezza delle informazioni, tipicamente riproponendo la richiesta o terminando il programma.
Per poter realizzare operazioni di questo tipo, il C offre diverse soluzioni.

Iniziamo ad esaminare la struttura di controllo condizionale più semplice, ovvero l'istruzione if, il cui uso è il seguente:

```
if (questa condizione è vera)
    esegui questa istruzione;
```

La parola chiave if indica che occorre eseguire un test sulla condizione racchiusa tra parentesi tonde; se la condizione è verificata bisogna eseguire l'istruzione indicata successivamente.
Esaminiamo ad es. il seguente codice:

```
if ( eta >= 100 )
    printf("Complimenti %s!\n", nome);
printf("Attendere prego...\n");
```

Se l'istruzione non è verificata, l'istruzione non viene eseguita e il controllo passa all'istruzione successiva all'if; nell'esempio precedente se l'età dell'utente è inferiore a 100 anni, allora non viene stampata la stringa con i complimenti e l'esecuzione passa direttamente alla stampa della stringa di attesa.

al verificarsi della condizione occorre eseguire più linee di codice, basta inserire un blocco di istruzioni (v. lezione 22), ad es. potremmo voler estendere l'esempio precedente nel seguente modo:

```
if ( eta >= 100 ) {
    printf("Complimenti %s!\n", nome);
    occupazione = "nonno";
}
printf("Attendere prego...\n");
```

Più in generale può essere necessario intraprendere operazioni alternative, specificando le istruzioni da eseguire quando la condizione non è verificata.
In questo caso si può utilizzare la parola chiave else, che permette di costruire strutture di controllo del tipo:

```
if (questa condizione è vera)
    esegui questa istruzione;
else /* altrimenti */
    esegui quest'altra istruzione;
```

Ad es.:

```
if ( eta >= 18 )
    maggiorenne = TRUE;
else
    maggiorenne = FALSE;
```

Se le alternative da gestire sono più di due, è facile estendere l'uso di else, ricorrendo a strutture di tipo else-if del tipo:

```
if (questa condizione è vera)
    esegui questa istruzione;
else if (quest'altra condizione è vera)
    esegui quest'altra istruzione;
else
esegui quest'altra istruzione;
```

Ad es.:

```
if ( a > 0 )
printf("Risultato positivo!\n");
else if ( a < 0 )
    printf("Risultato negativo!\n");
else
    printf("Risultato nullo!\n");
```

La condizione da verificare può essere un'espressione più o meno complessa, ottenuta concatenando più condizioni mediante gli operatori logici visti la scorsa lezione, ad es.:

```
if ( (a > 0) && (a % 2 == 0) )
printf("Risultato positivo pari!\n");
```

Si faccia molta attenzione quando si testa una condizione di uguaglianza.
Infatti un errore molto frequente, soprattutto agli inizi, è quello di scrivere un solo segno '=' invece di due, il che determina comportamenti "strani" difficili da individuare e correggere.
Il motivo principale è che in C non esiste il tipo booleano e il test della condizione è ritenuto vero se il valore dell'espressione è diverso da zero, falso se l'espressione è nulla.
Scriviamo il seguente programma:

```
#include <stdio.h>

int main (int argc, char *argv)
{
        int a;
        if ( a = 0 )
                a++;
        printf("a=%d\n", a);
        return 0;
}
```

Proviamo a compilarlo e notiamo che non ci viene segnalato dal compilatore alcun errore o avvertimento.
L'output di questo semplice esempio è:

```
a=0
```

Se scriviamo come condizione dell'if:

```
if ( a = 1 )
```

l'esecuzione del programma ci darà:

```
a=2
```

Questo dimostra che mentre nel primo caso non viene eseguito l'incremento di a, nel secondo caso invece, poiché l'espressione da testare ha valore diverso da 0, l'incremento viene effettuato.

Vedremo alcuni altri aspetti delicati che riguardano le strutture di controllo nella prossima lezione.

Strutture di controllo - Parte II
Lezione 34

L'assenza di un tipo booleano predefinito nel linguaggio può creare dei problemi di leggibilità del codice, soprattutto per i programmatori alle prime armi.
Quando occorre utilizzare una variabile come flag booleano, sovente la si definisce come short int, assegnandole come valore 1 o 0.
Nel migliore dei casi il programmatore utilizza un accorgimento del tipo:

```
#define FALSE  0
#define TRUE   1
...
```

```
short int test = TRUE;
...
```

curandosi di assegnare a test sempre TRUE o FALSE piuttosto che i corrispondenti valori interi.
Comunque, questi identificatori sono già definiti nell'header standard stdlib.h, che basta includere nei nostri sorgenti per poter utilizzare tranquillamente TRUE e FALSE.

Nel caso in cui occorre testare il valore di un flag booleano, si può essere tentati a scrivere un codice del tipo:

```
if (test == TRUE)
...
```

Tuttavia è più elegante, ed ha il medesimo significato del precedente, scrivere:

```
if (test)
    ...
```

Infatti se test è pari a TRUE (ovvero diverso da 0), il blocco di istruzioni dell'if viene parimenti eseguito.

Accanto all'istruzione condizionale if-else esiste un operatore condizionale, detto anche operatore ternario per via del fatto che lavora su tre operandi, che può essere considerato come una sorta di if-else abbreviato.
La sua forma generale è la seguente:

```
espressione1 ? espressione2 : espressione3
```

ed equivale a: “Se espressione1 è vera (cioè il suo valore è diverso da 0), allora il valore restituito dall'operazione è espressione2 altrimenti il valore restituito è espressione3”.
Ad es. la porzione di codice:

```
int i;
...
printf(“i e' %s\n”, (i >= 0) ? “positivo” : “negativo”);
```

stampa a video la stringa “i e' positivo” se i è maggiore o uguale a 0, oppure, in caso contrario, la stringa “i e' negativo”.
L'analogo esempio con if-else:

```
int i;
...
printf(“i e' ”);
if (i >= 0)
printf(“positivo\n”);
else
    printf(“negativo\n”);
```

è certamente più leggibile; tuttavia spesso si preferisce utilizzare l'operatore condizionale quando si vuole esprimere con un solo valore il risultato della scelta tra due situazioni mutuamente esclusive.

Passiamo adesso ad un’altra importante struttura di controllo, la cui utilità è innegabile quando ci si trova a dover definire delle scelte che non sono basate su due sole alternative.
Stiamo parlando del controllo switch, la cui sintassi è la seguente:

```
switch (espressione) {
    case valore1:
               istruzione1;
               break;
    case valore2:
istruzione2;
               break;
    .
    .
    .
    case valoreN:
               istruzioneN;
               break;
```

```
    case default:
istruzione;
               break;
}
```

Questo costrutto ha lo stesso significato della seguente struttura if:

```
if (espressione == valore1)
    istruzione1;
else if (espressione == valore2)
    istruzione2;
else
    ...
else if (espressione == valoreN)
    istruzioneN;
else istruzione;
```

L'espressione che segue la parola chiave switch può essere una qualsiasi espressione C che restituisce un valore intero o un carattere (dal momento che i caratteri non sono altro che valori interi particolari); tale valore viene utilizzato come criterio di selezione delle alternative disponibili.
La parola chiave case può essere seguita da un valore intero, da costante di tipo carattere o anche da un'espressione costante (ad es. 2*3), che viene valutata in fase di compilazione, ma non può essere una variabile; inoltre è indispensabile che ogni valore (detto anche etichetta o label) sia differente dagli altri.

opo ogni case può essere indicata una sola istruzione o un blocco di istruzioni, oppure anche l'istruzione nulla, costituita solo da un ';' o addirittura non mettendo nulla (nella prossima lezione vedremo un esempio che ci chiarirà meglio questa soluzione).

Non appena l'esecuzione di un programma incontra l'istruzione switch, vengono compiuti i seguenti passi:
1) viene innanzitutto valutata l'espressione intera che segue la parola chiave switch;
2) il valore ottenuto viene via via confrontato con tutti i valori costanti che seguono la parola chiave case;
3) quando l'uguaglianza è verificata, il programma esegue tutte le istruzioni che seguono il case corrispondente e tutti i case seguenti, compresa l'istruzione default;
4) se nessun confronto dà esito positivo, vengono eseguite solo le istruzioni che seguono la parola chiave default.
Il caso default è facoltativo e viene utilizzato proprio per individuare un'azione da compiere se nessuna etichetta eguaglia il valore ottenuto dalla valutazione dell'espressione.

Infine, osserviamo che, a differenza di altri linguaggi, l'esecuzione parte dal caso corrispondente e termina alla fine della struttura di controllo.
Se si vuole terminare lo switch dopo l'esecuzione delle istruzioni relative ad ogni singolo caso, occorre inserire l'istruzione break.

Strutture di controllo - Parte III
Lezione 35

Prima di proseguire con le altre strutture di controllo, vediamo di chiarirci le idee sull'uso di switch per mezzo di alcuni esempi pratici.
Per cominciare compiliamo ed eseguiamo l'esempio seguente.

```
/*
 * File lez35_1.c
 */
#include <stdio.h>

int main() {
    char c = 'b';

    switch (c) {
       case 'a':
               printf("Sono nel caso 'a'\n");
               break;
        case 'b':
               printf("Sono nel caso 'b'\n");
```

```
            break;
        case 'c':
            printf("Sono nel caso 'c'\n");
            break;
    }
    return 0;
}
```

Cambiando il valore di inizializzazione della variabile c possiamo vedere in che modo cambia l'output del programma.
Per esercizio, provate anche ad eliminare i break alla fine di ogni case e a rieseguire il nuovo programma, scrivendo anche il corrispondente Makefile.

Sebbene possa apparire addirittura fastidiosa la necessità di dover riscrivere la parola chiave break alla fine di ogni case, si presentano spesso dei casi in cui questa caratteristica risulta molto utile, ad es. quando si vuole eseguire la stessa istruzione per valori differenti dell'espressione.

Ne è un esempio il programma seguente:

```
/*
 * File lez35_2.c
 */
#include <stdio.h>

int main() {
    int i = 3;

    switch (i) {
        case 1:
        case 3:
        case 5:
case 7:
        case 9:
                printf("i e' dispari e minore di 10\n");
break;
        case 0:
        case 2:
        case 4:
        case 6:
        case 8:
printf("i e' pari e minore di 10\n");
                break;
        default:
                printf("i e' maggiore o uguale a 10\n");
    }
    return 0;
}
```

Notiamo l'assenza di break dopo il caso default.
Come esercizio, provate ad estendere e modificare gli esempi precedenti con altre soluzioni, ad es. utilizzando il costrutto if invece della struttura switch.

Consideriamo adesso altre importanti strutture di controllo, che consentono di eseguire cicli di istruzioni (loop), ovvero di definire una sequenza di operazioni da effettuare un prestabilito numero di volte o finché non si verifica una determinata condizione.

In C, il costrutto fondamentale per i cicli è l'istruzione while, la cui forma è la seguente:

```
while (questa condizione è vera)
    esegui questa istruzione;
```

Si noti la forte somiglianza sintattica con il costrutto if.
Tuttavia esiste una differenza fondamentale: l'istruzione while continua ad eseguire l'istruzione corrispondente finché la condizione rimane verificata.
Quando la condizione è - o diventa - falsa, l'istruzione non viene più eseguita e il controllo passa alla prima istruzione che segue quella del ciclo while.

Ad es., il programma:

```
/*
 * File lez35_3.c
 */
#include <stdio.h>

int main() {
    int x = 3;

    while (x >= 0)
printf("x = %d\n", x--);
    printf("Fine\n");
}
```

produce sullo schermo il seguente output:

```
x = 3
x = 2
x = 1
x = 0
Fine
```

L'istruzione controllata da while può essere una singola linea di codice o un blocco di istruzioni racchiuse tra parentesi graffe:

```
while (questa condizione è vera)
    {
    istruzione_1;
    istruzione_2;
    ...
    istruzione_n;
    }
```

Grazie alle parentesi è possibile avere costrutti while annidati, come nel seguente esempio:

```
*
 * File lez35_4.c
 */
#include <stdio.h>

int main() {
    int x = 3;
    int y;

    while (x > 0)
        {
        printf("x = %d\n", x--);
        y = 0;
        while (y <= 2) {
            printf("\ty = %d", ++y);
        }
        printf("\n");
    }
    printf("Fine\n");
}
```

Si noti che se la condizione è falsa la prima volta che viene valutata, le istruzioni del ciclo non vengono mai eseguite e il controllo salta direttamente all'istruzione successiva al while.

L'istruzione while è in grado di accettare anche espressioni, oltre che condizioni; in tal caso, quando l'espressione è un numero diverso da 0 è considerata vera e le istruzioni del ciclo vengono eseguite, quando vale 0 l'istruzione while termina.

Ad es. il ciclo del primo esempio può essere scritto correttamente anche in questa forma:

```
int x = 4;
while (x--)
    printf("x = %d\n", x);
```

Nella prossima lezione approfondiremo l'uso del ciclo while e conosceremo due importanti funzioni per leggere i caratteri da tastiera e scriverli su video.

Lettura e scrittura di caratteri
Lezione 36

Le operazioni di ingresso/uscita (input/output, spesso indicati con la sigla I/O), ovvero lo scambio di informazioni tra un programma e il mondo esterno al calcolatore, sono generalmente le parti meno eleganti di un algoritmo, a causa della complessità, sia logica che fisico-architetturale, della modalità di interscambio con l'esterno, ivi compresa la necessità di provvedere al recupero di eventuali errori (ad es. il file che si sta cercando di aprire non esiste).

La specifica del linguaggio C non prevede espressamente nessuna funzionalità riguardo le operazioni di I/O, tuttavia, poiché un programma C deve essere eseguito sotto il controllo di un sistema operativo, i programmatori dei vari sistemi hanno previsto un certo numero di funzioni che, pur non previste inizialmente, di fatto sono diventate una dotazione standard di ogni compilatore C.

Le funzioni per le operazioni di I/O, che sono generalmente costituite da procedure codificate in linguaggio assembler e fanno parte del "corredo" di tutti i compilatori, sono raccolte in una libreria standard, contenuta nel file header "stdio.h", che i programmi C devono necessariamente includere ogni qualvolta devono colloquiare con il mondo esterno, inclusi la tastiera e il terminale video.

Tralasciando tutti i dettagli relativi a questa problematica, per il momento diamo un primo sguardo a due semplici funzioni: getchar(), che legge un carattere dalla tastiera, e putchar(), che scrive un carattere sul video.

In realtà queste due funzioni leggono e scrivono, rispettivamente, dallo standard input e sullo standard output, e questi sono associati per default alla tastiera e al video.

La funzione getchar() è così definita:

```
int getchar(void)
```

ovvero, non richiede alcun parametro d'ingresso e restituisce un valore di tipo int, che rappresenta la codifica ASCII del carattere letto.
La funzione putchar(), duale della precedente, è così definita:

```
int putchar(char c)
```

ovvero richiede come parametro d'ingresso il carattere da scrivere e restituisce un valore di tipo int, che rappresenta la codifica ASCII del carattere scritto (tale valore è non significativo e nei programmi viene spesso ignorato).

Come primo esempio, scriviamo un programma che legge un carattere da tastiera e lo stampa sul video:

```
/*
 * File lez36_1.c
 */
#include <stdio.h>

int main()
{
       int c;

       c = getchar();
       (void)putchar((char)c);

 return 0;
}
```

Compilando ed eseguendo questo esempio, si può osservare un comportamento diverso da quello atteso.
Infatti, leggendo il codice sorgente ci si aspetta che l'esecuzione del programma attenda un carattere da tastiera, lo stampi su video ed esca.
Invece possiamo vedere che il programma attende sempre un altro carattere, finché non viene premuto il tasto [INVIO].

Questo comportamento deriva dal modo di gestire l'I/O da parte di Linux (e dei sistemi UNIX in generale).
Linux memorizza in un'area temporanea (un buffer) tutti i caratteri provenienti da tastiera, fino all'arrivo di un carattere di ritorno a capo, dopodiché li passa nello stesso ordine al programma.
Altri sistemi adottano invece una soluzione differente, più in linea, se si vuole, con le nostre attese.

Questo esempio ci permette di sottolineare che le funzioni di I/O non fanno parte del linguaggio C, ma sono fortemente influenzate dalle caratteristiche dell'ambiente di esecuzione.

Sotto questo aspetto, l'esempio successivo è molto interessante:

```
/*
 * File lez36_2.c
 */
#include <stdio.h>

int main()
{
        int c;

        while ((c = getchar()) != 'q')
              putchar((char)c);

return 0;
}
```

Una possibile esecuzione è la seguente (in grassetto sono indicati i caratteri immessi dall'utente, in corsivo quelli stampati dal programma):

```
riga normale
riga normale
altra riga normale
altra riga normale
riga con una q
```

Notiamo che ogni riga immessa viene stampata due volte, mentre l'ultima non viene ripetuta dal programma.
Le righe in grassetto sono mostrate perché la shell di Unix ripete sul video ogni carattere immesso dalla tastiera.
Quando si preme il tasto [INVIO], le funzione getchar() ritorna il contenuto di tutto il buffer, che viene stampato da putchar().
Poiché l'ultima riga contiene il carattere 'q', la condizione dell'istruzione while diventa falsa e putchar() non viene eseguito (per nessuno dei caratteri della riga!).

Notiamo ancora una differenza tra le istruzioni putchar() dei due esempi: il casting al tipo void nel primo programma non è necessario, in quanto non modifica il comportamento della funzione; l'unica utilità di questa operazione è quella di indicare esplicitamente che stiamo ignorando il valore ritornato da putchar().

Infine un esercizio:
Create un programma che esegua un ciclo che, facendo uso delle istruzioni while e if (o switch), stampino a video tutto il set ASCII, fornendo, per i caratteri non stampabili, un'indicazione del loro significato (simile a quanto mostrato dalla pagina man ascii).

Strutture di controllo - Parte IV
Lezione 37

Esaminiamo adesso altre due strutture di controllo, partendo dal costrutto do-while, che ha il seguente formato:

```
do {
    istruzione_1;
    istruzione_2;
    ...
    istruzione_n;
} while (condizione);
```

Com'è immediato rilevare, questa istruzione è molto simile al while, eccezion fatta per la posizione in cui viene eseguito il test sulla condizione.
Infatti nel while la condizione viene verificata prima di eseguire qualsiasi istruzione del ciclo, mentre il costrutto do-while esegue la verifica della condizione dopo aver eseguito una volta le istruzioni del ciclo.
Di conseguenza, in do-while il ciclo viene eseguito sempre almeno una volta, anche se la condizione viene trovata subito falsa.

Per sottolineare questa differenza fra i due costrutti, si esegua il seguente programma:

```
/*
 * File lez37_1.c
 */
#include <stdio.h>

int main()
{
      while (2 < 1)
            printf("Ciclo while\n");

      do
            printf("Ciclo do-while\n");
      while (2 < 1);

      return 0;
}
```

Poiché la condizione 2 < 1 non è mai vera, il ciclo while termina senza eseguire la sua istruzione printf, mentre il ciclo do-while esegue prima l'istruzione printf, poi valuta l'espressione e quindi esce.
A parte questa caratteristica, le due strutture di controllo si comportano esattamente allo stesso modo; anzi è raro trovare cicli do-while nei programmi C, poiché sono veramente poche le occasioni in cui si possa voler eseguire un ciclo almeno una volta indipendentemente dal valore della sua condizione.
Si noti, nell'esempio, che non sono state utilizzate le parentesi graffe per il corpo dell'istruzione do-while: ciò può essere fatto quando il ciclo è costituito da una sola istruzione.

Per esercizio, riscrivere l'esempio “lez35_1.c”, utilizzando il costrutto do-while.

Qualora l'esecuzione di una o più istruzioni debba avvenire un numero definito di volte, si può utilizzare il costrutto for, la cui forma è la seguente:

```
for (inizializzazione contatore; condizione; aggiornamento contatore)
{
    istruzione_1;
    istruzione_2;
    ...
    istruzione_n;
}
```

Questa istruzione, molto usata dalla maggior parte dei programmatori, definisce tre parametri del ciclo:
1) l'inizializzazione di un contatore;
2) la valutazione di una condizione o di un'espressione;
3) l'aggiornamento del valore del contatore tramite un'espressione opportuna;
ed è sostanzialmente equivalente ad un ciclo while del tipo:

```
inizializzazione contatore;
while (condizione)
{
    istruzione_1;
    istruzione_2;
    ...
    istruzione_n;
    aggiornamento contatore;
}
```

In realtà, internamente il C tratta un ciclo for come un ciclo while a tutti gli effetti.

Il primo e l'ultimo parametro non devono necessariamente inizializzare e aggiornare il contatore: ognuno di essi può essere una qualsiasi istruzione C o essere addirittura omesso.
Anche il secondo parametro può essere un'istruzione qualsiasi, ma il compilatore lo interpreta sempre come una condizione da verificare ed il suo risultato viene valutato come vero o falso (si ricordi sempre che il C considera falso lo zero binario e vero qualsiasi altro valore).

A tal proposito, è utile il seguente programma:

```
/*
 * File lez37_2.c
 */
#include <stdio.h>

int main()
{
    for (printf("Inizializzazione contatore\n");
         printf("Valutazione condizione\n");
         printf("Aggiornamento contatore\n"))
    {
         printf("Istruzioni del ciclo\n");
    }

    return 0;
}
```

Compilate il codice precedente con il comando:

```
cc -o lez37_2 lez37_2.c
```

ed eseguitelo con il comando:

```
./lez37_2 > lez37_2.txt
```

Sullo schermo non apparirà alcuna scritta, benché ci si aspetti di vedere almeno qualcuno dei vari printf del nostro sorgente: ciò è dovuto al simbolo '>' che segue il nome dell'eseguibile e che indica al sistema operativo di non stampare l'output del programma sul video ma di redirigerlo sul file “lex37_2.txt” (se il file non esiste viene creato).
Premendo contemporaneamente i tasti [Ctrl] e [C], riapparirà il prompt della shell, segno che l'esecuzione del programma è stata interrotta.

A questo punto troverete nella directory dell'eseguibile il file “lez37_2.txt”: apritelo con un editor di testo qualsiasi e vedrete che esso contiene le seguenti righe (se il file è troppo lungo, potete visualizzare solo le prime 10 righe tramite il comando “head -n 10 lez37_2.txt”):

```
Inizializzazione contatore
Valutazione condizione
Istruzioni del ciclo
Aggiornamento contatore
Valutazione condizione
Istruzioni del ciclo
Aggiornamento contatore
Valutazione condizione
Istruzioni del ciclo
Aggiornamento contatore
...
```

Avete un pò di tempo per riflettere su questo risultato, che commenteremo nella prossima lezione.

Strutture di controllo - Parte V
Lezione 38

Abbiamo terminato la scorsa lezione con un risultato (apparentemente) strano, ottenuto dall'esecuzione dell'ultimo esempio, che mostra il funzionamento del ciclo for.
Infatti, tenendo presente che l'istruzione printf ritorna come valore il numero di caratteri scritti, risulta evidente che l'esecuzione dell'istruzione for avviene con questi passaggi:
1) Viene eseguita l'inizializzazione del ciclo (solo la prima volta);
2) Viene valutata l'espressione: se è vera (o restituisce un valore diverso da zero, come in questo caso), viene eseguito il corpo del ciclo;
3) Al termine del ciclo viene aggiornato il contatore;
4) Viene rivalutata l'espressione e se è vera il ciclo ricomincia.

Per esercizio, riscrivere l'esempio "lez35_1.c", utilizzando il costrutto for.

L'istruzione for può essere utilizzata in maniera più complessa, includendo più di un'espressione sia nel campo di inizializzazione che in quello di aggiornamento del contatore, separandole per mezzo di una virgola.
Analizziamo l'esempio che segue:

```
/*
 * File lez38_1.c
 */
#include <stdio.h>

int main() {
    int x, y;

    for(x = 0, y = 10; ((x < 3) && (y > 9)); x++, y += 2)
              printf("x = %d\ty = %d\n", x, y);

return 0;
}
```

L'esecuzione del programma produce il seguente risultato:

```
x = 0  y = 10
x = 1  y = 12
x = 2  y = 14
```

ovvero:
1) vengono inizializzate le variabili x e y a 0 e 10 rispettivamente;
2) poiché l'espressione è vera viene eseguito il corpo del ciclo, che stampa la prima riga;
3) x e y vengono incrementate di 1 e 2 rispettivamente;
4) poiché l'espressione è ancora vera viene eseguito il corpo del ciclo, che stampa la seconda riga e così via.

Oltre a poter includere più di un'espressione per inizializzare e aggiornare il ciclo for, è possibile anche omettere del tutto uno o più parametri (addirittura anche tutti e tre).
Ad es. l'istruzione:

```
for(;;);
```

è valida e quando il programma arriva nel punto in cui si trova questa istruzione si blocca in esso per sempre: ciò è dovuto al fatto che la condizione da verificare in questo caso non restituisce mai zero e il ciclo viene ripetuto infinite volte.
Ad es.:

```
char c = 'a'
for(;;)
    putchar (c);
```

stampa una linea senza fine di caratteri 'a'.

Questo tipo di ciclo, detto appunto ciclo infinito, può essere utile in svariati casi e, di fatto, non è raro vedere programmi che ne fanno uso.
E' possibile scrivere cicli infiniti anche con i costrutti while e do-while, che in tal caso assumono le forme:

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

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

Generalmente, inserire cicli infiniti nei propri programmi non è opportuno ed occorre sempre prevedere un modo per poter terminare il programma in maniera controllata.
Questo è possibile tramite l'istruzione break, già vista qualche lezione fa.
Quando questa parola chiave viene incontrata all'interno di un ciclo, il controllo passa all'istruzione immediatamente successiva al ciclo.
Accanto ad essa, il C prevede anche un'altra istruzione, continue, il cui effetto è quello di saltare un'interazione del ciclo.
Break e continue sono tipicamente associate ad un'istruzione if.
Chiariamoci le idee con un esempio:

```
/*
 * File lez38_2.c
 */
#include <stdio.h>

int main()
{
        short i, j;

        for (i = 1, j = 100; ; i++, j -= 2)
        {
                if (!(j % i))
printf("j=%d e' divisibile per i=%d", j, i);
                else
                {
                      printf ("j=%d non e' divisibile per i=%d\n", j, i);
                      continue;
                }
                printf("\tIl risultato e' %d\n", j/i);
                if (i >= j)
                      break;
        }
        putchar('\a');

return 0;
}
```

Ad ogni iterazione del loop, viene controllato il resto della divisione tra j e i: se è zero, la condizione dell'if è vera e viene stampato un messaggio che avvisa che j è divisibile per i, altrimenti viene stampato l'avviso che j non è divisibile per i e il controllo passa alla successiva iterazione, per effetto dell'istruzione continue.
Senza di essa, il controllo avrebbe proseguito con la stampa del risultato della divisione.
Il loop viene arrestato non appena i supera j, tramite l'istruzione break che fa saltare il controllo all'istruzione putchar('\a'), che provoca l'emissione di un 'bip' da parte del computer.

Strutture di controllo - Parte VI
Lezione 39

Con questa lezione concludiamo la nostra carrellata sulle strutture di controllo fornite dal linguaggio C.
In particolare ci occuperemo di due strumenti che consentono di ramificare il flusso di esecuzione di un programma, ovvero la tanto discussa (vedremo perché) istruzione goto e le label ("etichette").

Una label è sintatticamente uguale ad una variabile ed è seguita dal carattere due punti, con una forma del tipo:

```
mia_etichetta:
istruzioni;
...
```

Per convenzione, l'etichetta è allineata a sinistra, in una riga di codice a sé stante.
La funzione delle etichette è di permettere al programmatore di indicare una riga di codice alla quale passare durante l'esecuzione, quando viene incontrata la parola chiave goto.
Infatti l'istruzione goto ha la seguente forma:

```
goto mia_etichetta;
```

Vediamo un semplice esempio che ne illustra l'utilizzo:

```
/*
* File lez39_1.c
*/
#include <stdio.h>

int main()
{
int i = 2;
int j = 100;
int x;

for (; i < 10; i++)
while (--j)
{
x = i * j;
if (x < 185)
goto fuori;
printf("i * j = %d\n", x);
}

fuori:
printf("Sono fuori!\n");

return 0;
}
```

Eseguendolo otterremo:

```
i * j = 198
i * j = 196
i * j = 194
i * j = 192
i * j = 190
i * j = 188
i * j = 186
Sono fuori!
```

Il funzionamento dei cicli for e while dovrebbe essere chiaro.
Quello che risulta evidente è che l'utilizzo di goto provoca, una volta verificata la condizione dell'if, l'uscita da entrambi i cicli annidati e l'esecuzione della istruzione seguente all'etichetta fuori.
Se proviamo a mettere break al posto di goto, l'output del programma è ben diverso.
Infatti, l'istruzione break provoca l'uscita dal solo ciclo while, mentre il corpo del costrutto for viene eseguito fino alla fine (cioè finché i è minore di 10).

Notiamo che in entrambi i casi l'istruzione successiva all'etichetta viene eseguita comunque.

L'esempio riportato, seppur banale, mostra uno dei pochi casi che può giustificare l'utilizzo di goto, ovvero l'uscita da cicli annidati al verificarsi di particolari eventi o errori.
Non è, infatti, consigliabile ricorrere all'istruzione goto, sia perché il suo utilizzo rende il codice più difficile da leggere e da mantenere (soprattutto se l'etichetta cui fa riferimento si trova in una porzione molto distante), sia perché qualsiasi segmento di codice contenente un goto può sempre essere scritto in una forma equivalente che non ne fa uso.
Riscriviamo l'esempio precedente senza utilizzare goto:

```
/*
* File lez39_2.c
*/
#include <stdio.h>
#define FALSE 0
#define TRUE 1

int main()
{
int i = 2;
int j = 100;
int x;
short posso_uscire = FALSE;
for (; i < 10; i++)
{
if (posso_uscire)
break;
while (--j)
{
x = i * j;
if (x >= 185)
printf("i * j = %d\n", x);
else
{
posso_uscire = TRUE;
break;
}
}
}
printf("Sono fuori!\n");

return 0;
}
```

In questo caso abbiamo dovuto introdurre una nuova variabile e qualche controllo in più, ma il risultato è il medesimo del caso precedente.

Molti sviluppatori, soprattutto quelli provenienti dall'Assembler, hanno abusato di questa istruzione, che in passato è stata al centro di numerose discussioni.
Attualmente sembrano tutti d'accordo nell'affermare che è preferibile non ricorrere ad essa, tant'è gli autori di uno dei moderni linguaggi di programmazione più usati, ovvero Java, pur prendendo molto dalla sintassi del C++, e quindi anche del C, hanno preferito non includerlo fra le istruzioni disponibili.

Con questa lezione abbiamo terminato l'argomento delle strutture di controllo; nella prossima approfondiremo gli array, cui abbiamo accennato nella lezione 31.