Biagio Ianero

Il C a 360 °

Breve manuale per la programmazione in C

Matematicamente.it

BIAGIO IANERO
Il C a 360°

Alcune tabelle provengono da Wikipedia

Foto di copertina: The C on the Clicff by Barry Solow
http://www.flickr.com/photos/baslow/5620208

INDICE

INTRODUZIONE

Ciao caro lettore, mi presento, mi chiamo Biagio Ianero e ti ringrazio di aver scelto questo libro per imparare il Linguaggio C.
In questa guida verranno trattati tutti gli aspetti del linguaggio per una comprensione a 360°.
Ci saranno anche degli esempi pratici per capire meglio ciò che viene spiegato.
Elenco qui di seguito alcune convenzioni che userò nella scrittura:

- Il codice sarà scritto con questo tipo di Font;
- Il codice delle applicazioni di esempio funzionanti verrà scritto "a colori" per una più facile comprensione.
- Box di attenzione: 

Per qualsiasi problema riguardante la comprensione della teoria, del codice od opinioni sul testo in generale contattami all'indirizzo: biagio.ianero@alice.it

Ti auguro una buona lettura ed una facile comprensione...

IL LINGUAGGIO C

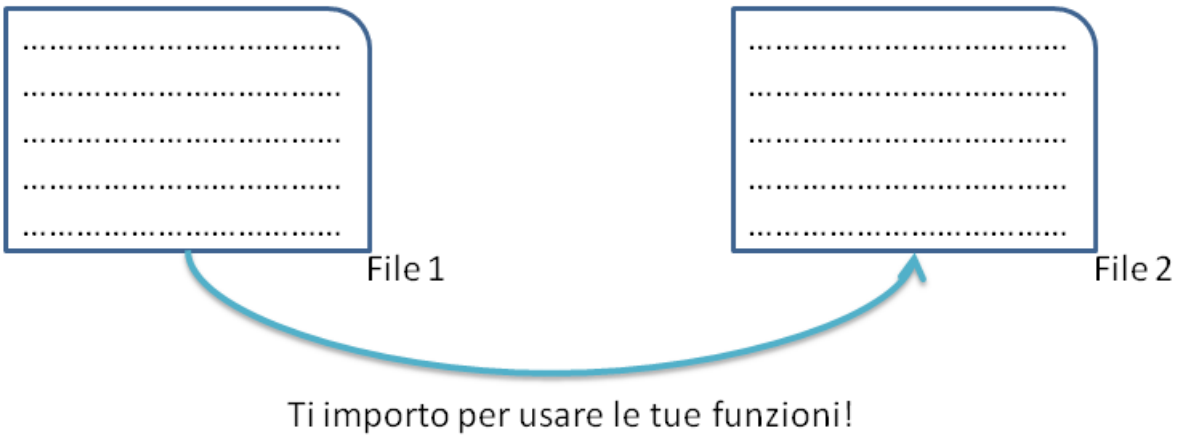
Il linguaggio C, diretto successore dell'assembly, è un linguaggio "di basso livello" perchè permette di svolgere operazioni di I/O molto basilari e con estrema semplicità, non a caso è molto usato per la scrittura di driver e kernel di sistemi operativi. E' un linguaggio di tipo procedurale, il che significa che usa una logica "a blocchi", detti comunemente funzioni. Questi blocchi di codice hanno un nome e sono delimitati da appositi simboli che spiegheremo meglio nei capitoli successivi.
Il C è un linguaggio di tipo Case-Sensitive, quindi "ciao" sarà diverso da "CIAO" e diverso da "cIaO".
Inoltre è un linguaggio Cross-Platform, cioè significa che può essere usato per programmare su qualsiasi S.O.
Cercherò di essere il più generale possibile senza fare esempi riguardanti una piattaforma specifica.
Per quanto mi riguarda lo preferisco a qualunque altro linguaggio di programmazione e spero sia lo stesso per voi.
Iniziamo con alcuni concetti di base prima di addentrarci nella scrittura di codice vera e propria.

LIBRERIE

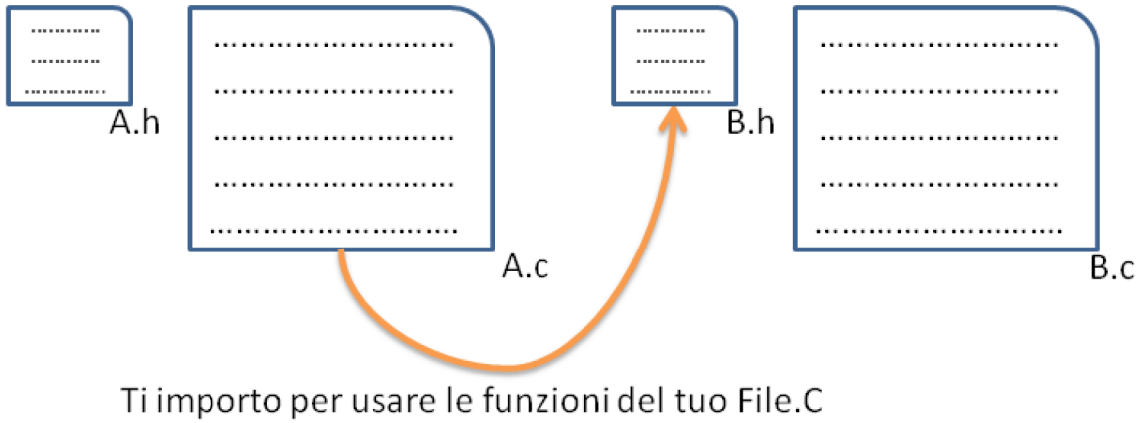
Facciamo un esempio: vogliamo creare una macchina, ma da dove iniziamo?
Chi ci fornisce gli attrezzi? Dove prendiamo il motore? E le gomme?
Dovremmo fare tutto noi da zero, ma è per questo motivo che ci vengono in soccorso **le librerie**.
Nel nostro esempio la macchina è l'applicazione che vogliamo scrivere, ma per scriverla abbiamo bisogno di funzioni basilari scritte già da altri programmatori da "includere" nel nostro codice, altrimenti se dovessimo riscrivere ogni volta tutto da zero, il lavoro dell'app vero e proprio rappresenterebbe lo 0,00000001%.

FILE DI CODICE E COMPILAZIONE

Dove scriviamo il nostro codice C? La risposta è in un editor di testo più rudimentale possibile!
Questo codice sarà salvato con estensione .c al posto del classico .txt ed infine compilato.
La compilazione è un processo tramite il quale dal file.c ricaviamo il file eseguibile: la nostra applicazione vera e propria.
Ci sono varie fasi per portare a termine questo processo che sono del tutto trasparenti al programmatore grazie ai moderni compilatori che si trovano tranquillamente in rete.
La nostra applicazione sarà composta da più file.c, ognuno con un compito specifico, che saranno poi uniti (sempre in fase di compilazione...) per formare un solo file eseguibile.
Un file.c può utilizzare una funzione scritta in un altro file.c, semplicemente importandolo.
Viene quindi da pensare ad una situazione di questo genere:



C'è però un problema: i file.c sono i file in cui viene scritto il codice vero e proprio, dove viene fatto il cosiddetto “lavoro sporco”, sarebbe quindi molto difficile trovare una determinata funzione in un file pieno di istruzioni di quel tipo.
Per rendere la cosa molto più semplice e pulita il C usa un'altro tipo di file, chiamato file di **header** avente estensione .h .
Quindi se esiste un file chiamato A.c, esisterà anche un file che avrà nome A.h, nel quale saranno solo *dichiarate* tutte le funzioni *implementate* in A.c .
Tornando all'esempio della macchina, in un file chiamato Movimento.h troveremo le *dichiarazioni* delle funzioni “sterza”, “accelera” e “frena”, mentre nel file Movimento.c troveremo tutte le istruzioni necessarie per *l'implementazione* della sterzata, dell'accelerazione e della frenata della nostra automobile.
L'accoppiata file.c/file.h forma un **modulo** di codice sorgente.
La situazione precedente si trasforma quindi in:



Come vediamo è il file header ad essere importato, il file grazie al quale riceviamo tutte le informazioni sul file.c dello stesso modulo.
Vedremo più avanti la procedura corretta per l'importazione dei file header.

VARIABILI E TIPI DI DATI
Durante la scrittura del nostro codice avremo bisogno di trattare con tipi di dati diversi in base alle nostre esigenze (numeri interi, decimali, ecc...).
Come sappiamo un bit può essere rappresentato da “0” o da “1” e mettendo insieme più bit si ottengono rappresentazioni di numeri più grandi di 0 e di 1.
Più bit mettiamo insieme, più grande è il limite numerico che riusciamo a rappresentare.
Dove si raggruppano questi bit? E' qui che introduciamo il concetto di **variabile.**
In C qualsiasi tipo di dato è una variabile.
Essa non è nient'altro che un “agglomerato” di bit che ci rappresenta un determinato valore.
I tipi di dati primitivi che il C ci mette a disposizione sono i seguenti:

- variabile di tipo “int”, rappresenta un numero intero e, a seconda della macchina sulla quale si compila il codice, può contenere 16 o 32 bit;
- variabile di tipo “float”, rappresenta un numero decimale ed è formata da 32 bit;
- variabile di tipo “double”, rappresenta un numero decimale ed è formata da 64 bit;
- variabile di tipo “char”, rappresenta un carattere e contiene 8 bit;

Capiamo quindi che una variabile può rappresentare al suo interno un totale di numeri pari a 2 (numeri rappresentabili in un bit) elevato al numero di bit che la formano.
La variabile che contiene meno bit è il char poiché è formata da un solo byte (1 byte = 8 bit).

I più attenti avranno già capito che il char, oltre che per rappresentare caratteri letterali, può essere anche utilizzato come "piccolo int"; esso infatti potenzialmente può rappresentare 2^8 numeri, decisamente inferiore alla variabile "int", ma comunque utilizzabile.
Ovviamente queste variabili comprendono anche numeri negativi, ad esempio l'int va da -2147483648 a +2147483648.
Se volessimo solo rappresentare numeri dopo lo zero, quindi raddoppiare il limite numerico positivo ed azzerare il limite numerico negativo possiamo inserire la parola chiave "unsigned" prima di "int".
A questo punto la variabile "unsigned int" potrà contenere numeri da 0 a 4294967296 (cioè 2147483648 * 2).
Ci sono inoltre parole chiave che possono "allungare" o "accorciare" il campo di bit contenuto nella variabile: esse sono "long" e "short".
Troviamo infine altre due parole chiave che sono "void" e "NULL": la prima sta per "non-tipo" di dato, la seconda per "nessun riferimento in memoria".
Non si possono creare variabili di tipo void, ma questo tipo di dato viene usato per creare puntatori generici e per non ritornare nessun valore da una funzione (esamineremo nello specifico questi due aspetti più avanti).

I nomi delle variabili non possono contenere spazi!

OPERATORI ARITMETICI
Il C mette a disposizione gli operatori aritmetici basilari per compiere operazioni matematiche.
Essi sono:

- "+" , simbolo che ci permette di eseguire l'addizione tra l'elemento a destra del simbolo e quello alla sua sinistra;
- "-" , simbolo che ci permette di eseguire la sottrazione tra l'elemento a destra del simbolo e quello alla sua sinistra;
- "*" , simbolo che ci permette di eseguire la moltiplicazione tra l'elemento a destra del simbolo e quello alla sua sinistra;
- "/", simbolo che ci permette di eseguire la divisione tra l'elemento a destra del simbolo e quello alla sua sinistra;
- "%", simbolo che ci permette di ottenere il resto di una divisione intera tra l'elemento a destra del simbolo e quello alla sua sinistra;

Il simbolo "%" non esegue la percentuale!!!

Abbiamo poi anche altri operatori che io definisco "di comodo" perché servono semplicemente ad incrementare (aumentare di 1) o decrementare (diminuire di 1) il valore di una variabile.
Ad esempio data la variabile "var" la scrittura:

var++ oppure ++var

incrementano di 1 la variabile var, mentre:

var-- oppure —var

decrementano di 1 la variabile var.

Cosa cambia se ++ o -- si trovano prima o dopo del nome della variabile?
Cerchiamo di capirlo con un esempio:
Poniamo per semplicità che “var” valga 10.
Allora...

5 * var++

corrisponde a (5 * var) + 1 e il risultato sarà quindi 51.
Mentre...

5 * ++var

corrisponde a 5 * (var + 1) e il risultato sarà quindi 55.

OPERATORI DI CONFRONTO
Gli operatori di confronto sono un'altra categoria di operatori che possiamo usare per mettere a confronto (chi l'avrebbe mai detto eh?) due elementi.
Semplicemente sono:

- “==”, operatore di uguaglianza (diverso da “=” !!!) che ritorna 1 (VERO) se le due variabili messe a confronto sono uguali, altrimenti ritorna il valore 0 (FALSO);
- “!=”, operatore di disuguaglianza che ritorna 1 (VERO) se le due variabili messe a confronto NON sono uguali, altrimenti ritorna il valore 0 (FALSO);
- “>”, ritorna 1 (VERO) se la variabile a sinistra è maggiore di quella di destra, altrimenti ritorna il valore 0 (FALSO);
- “<”, ritorna 1 (VERO) se la variabile a sinistra è minore di quella di destra, altrimenti ritorna il valore 0 (FALSO);
- “>=”, ritorna 1 (VERO) se la variabile a sinistra è maggiore o uguale a quella di destra, altrimenti ritorna il valore 0 (FALSO);
- “<=”, ritorna 1 (VERO) se la variabile a sinistra è minore o uguale a quella di destra, altrimenti ritorna il valore 0 (FALSO);

“=” non è la stessa cosa di “==”: il primo assegna il valore dell'elemento di destra all'elemento a sinistra, il secondo invece effettua solo un confronto ma non modifica il valore degli elementi.

OPERATORI LOGICI
Gli operatori logici più utilizzati in C sono “AND”, “OR” e “NOT”:

- “&&”, è l'AND e restituisce 1 (VERO) solo se entrambe le *espressioni* messe a confronto sono vere, altrimenti 0 (FALSO);
- “||”, è l'OR e restituisce 1 (VERO) se almeno una delle *espressioni* messe a confronto è vera, altrimenti 0 (FALSO);
- “!”, è il NOT ed inverte il valore di ritorno di un'espressione: se essa ritorna 1 (VERO), il risultato sarà 0 (FALSO) e viceversa.

Possiamo quindi riassumere in questo modo...

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A
0	1
1	0

OPERATORI BIT A BIT
Altro tipo di operatori sono quelli che lavorano sui singoli bit.
Essi non sono molto usati nella programmazione in "User-Space", ma comunque vengono elencati per completezza.
Essi sono:

- "&", and su bit, il risultato è 1 solo se entrambi i bit sono 1;
- "|", or su bit, il risultato è 1 se almeno un bit vale 1;
- "^", xor su bit, il risultato è 1 se solo uno dei due bit vale 1;
- "~", not su bit, inverte ogni singolo bit;
- ">>", shift a destra di bit, sposta a destra i bit di un dato elemento di "n" posizioni senza riporto;
- "<<", shift a sinistra di bit, sposta a sinistra i bit di un dato elemento di "n" posizioni senza riporto.

Per capire meglio la cosa facciamo un piccolo esempio...
Prediamo il numero 7 e il numero 9, che in binario sono rispettivamente "0111" e "1001".
Usiamo tutti gli operatori bit a bit su questi due numeri secondo le regole elencate sopra.

0111 & 1001 =	0111 | 1001 =	0111 ^ 1001 =	~ 0111	>> 0111	<< 0111
0001	1111	1110	1000	0011	1110

Come potrete notare gli ultimi tre operatori operano su un solo valore e non su una coppia di valori, come invece fanno i primi tre.

FUNZIONI

Dove raggruppiamo tutte le istruzioni per il corretto funzionamento della nostra applicazione?
Sarebbe semplicemente folle scrivere tutte le istruzioni una dietro l'altra: non ci orienteremmo più tra tutto il codice nel giro di venti minuti.
Ecco perché il C si scrive suddividendolo in **funzioni**, dette anche sotto-programmi o sotto-routine.
Una funzione è semplicemente un "blocco" di codice delimitato da due parentesi graffe (all'inizio una aperta e alla fine una chiusa) e contrassegnato da un nome che lo identifica.
Ecco un'esempio di funzione:

```
void funzione ()
{


}
```

Il primo elemento della funzione è il valore che ritorna a chi la chiama, o meglio *invoca,* in questo caso "void", quindi la funzione non ritorna nessun valore.
Queste funzioni posso avere anche dei parametri che vanno racchiusi tra le due parentesi tonde dopo il nome della funzione...

```
void funzione (int numero)
{


}
```

In questo nuovo esempio ho aggiunto un parametro alla funzione: una variabile di tipo int.
A questo punto vi sarete fatti questa domanda: Dove e come invoco una funzione?
Abbiamo detto che una funzione è un blocco di codice che svolge determinate operazioni, quindi la risposta al dove è: dipende dalla logica che usate nella costruzione della vostra applicazione! (ovviamente tra poco faremo degli esempi pratici...).
La risposta al come è molto semplice: la funzione si invoca scrivendo il suo nome seguito dalle due parentesi tonde.
Un esempio:

```
funzione();
```

In questo esempio non stiamo passando nessun parametro alla funzione perché non c'è niente all'interno delle parentesi tonde.
Ecco un esempio in cui viene passato un intero alla funzione:

```
funzione( 4 );
```

Non so se avete notato il “punto e virgola”...
In C qualsiasi istruzione deve terminare con il “;”, è una regola di sintassi.
L'invocazione di una funzione è un'istruzione? Si, quindi deve terminare con un punto e virgola, così come tutte le altre istruzioni (dichiarazioni di variabili, assegnazioni, ecc...).
E' molto importante distinguere questi tre momenti:

- dichiarazione di una funzione, è il momento in cui viene semplicemente detto al compilatore “attento che nel mio codice userò una funzione che si chiama <nome_della_funzione> più avanti! Quando la incontrerai non impressionarti!”
- invocazione di una funzione, è il momento in cui la funzione viene chiamata, cioè viene detto al compilatore “esegui tutte le istruzioni della funzione <nome_della_funzione>!”;
- implementazione di una funzione, è il momento in cui la funzione viene “scritta”, quando apriamo e chiudiamo le parentesi graffe e scriviamo il vero e proprio lavoro sporco all'interno di esse!

Facciamo un esempio per ogni situazione...

Dichiarazione

```
void funzione (void);
```

Invocazione

```
void funzione();
```

Implementazione

```
void funzione (void)
{
        .....tutte le istruzioni.....
}
```

Quando vogliamo far ritornare un valore alla funzione chiamante usiamo la parola chiave “return” seguita dal valore da restituire e ovviamente dal famosissimo “;”.

SCOPE
Le variabili che dichiariamo possiamo usarle dappertutto nel nostro codice? Ovviamente la risposta è no, altrimenti creeremmo un casino incredibile.
Una variabile è visibile solo all'interno del suo blocco di codice (parentesi graffa aperta/chiusa) a meno che non si verifichino condizioni particolari.
Quando una variabile viene dichiarata al di fuori di ogni funzione (ad esempio all'inizio del nostro file di codice), essa è globale, cioè può essere usata dappertutto (vi sconsiglio di usare variabili globali tranne in casi di stretta necessità).
A questo punto mi sembra doveroso fare un'elenco di tutti i tipi di variabili che possiamo creare:
-**auto**, sono le classiche variabili automatiche che si dichiarano "tipo" "nomeVariabile" e può quindi essere omessa la “keyword” auto;
-**register**, sono variabili usate per aumentare le prestazioni in quanto esse non vengono memorizzate nella RAM, ma solo nel registro di sistema per garantire un più veloce accesso... Non hanno indirizzo in memoria;

-**static**, sono variabili che hanno vita per tutta la durata dell'app e vengono inizializzate una e una sola volta...tuttavia il loro valore può essere incrementato e decrementato ma senza usare l'operatore '=';
-**extern**, sono le variabili globali e visibili in tutto il file di codice nel quale sono dichiarate (spesso le variabili extern sono anche static per evitare errori, ma non è una legge).
-**volatile**, sono variabili che possono essere anche modificati da altri processi diversi da quello che l'ha dichiarata... diciamo in pratica al compilatore di prendere con le pinzette queste variabili ed evitare qualsiasi tipo di ottimizzazione per non imbattersi in errori;
-**const**, variabili il cui valore, una volta inizializzato, diventa di sola lettura e quindi immodificabile.

Queste parole chiave vanno inserite prima della dichiarazione del tipo, ecco un esempio:

```
static int numero;
```

ho dichiarato una variabile int statica.
Se in una funzione dichiariamo una variabile con lo stesso nome di una globale, il compilatore considererà solo quella interna alla funzione, tralasciando la globale.
E' molto sconsigliato quindi creare variabili locali con nomi uguali a quelle globali.

COMMENTI
E' probabile che quando il codice che scriviamo inizia a diventare corposo ci risulti molto difficile capirlo. Per fortuna ci vengono in soccorso i **commenti.**
Ottima pratica quella di commentare il codice per una semplice comprensione, abitudine poco diffusa soprattutto nei giovani programmatori.
Possiamo commentare in due modi il nostro codice: su una riga, oppure su più righe.
Facciamo degli esempi:

```
int numero: //ho dichiarato una variabile intera
```

Questo sopra è il commento su una riga e, come ogni commento mono-riga, deve cominciare con "//".
Omettendo questo simbolo il compilatore non capirà che ciò che abbiamo scritto è un commento e lo tratterà come se fosse codice, riportando ovviamente errori.
Vediamo invece il commento su più righe:

```
float numeroDecimale; /*ho dichiarato una variabile decimale
                        e l'ho chiamata numeroDecimale...*/
```

Come potete vedere il commento multi-riga inizia con "/*" e finisce con "*/".

HELLO WORLD!
Il miglior modo per capire un linguaggio è applicarlo, quindi aprite il vostro editor di testo preferito, (mi raccomando che sia abbastanza semplice...) e digitate il seguente codice:

```
#include <stdio.h>


int main (void)
{
	printf("Hello World!");
	return 0;
}
```

Iniziamo spiegando riga per riga...
Nella prima riga stiamo includendo nel nostro codice il modulo “stdio”, importandone il file di header (come spiegato nel capitolo 1).
Questo modulo fa parte della libreria standard del C e in esso sono definite le principali funzioni di I/O (“stdio” sta per “standard I/O”), tra cui printf che abbiamo usato adesso.
Capite subito che questo modulo va incluso in tutti i programmi C, perché contiene delle funzioni indispensabili.
Il simbolo “#” indica che stiamo per usare una direttiva al preprocessore, in questo caso la direttiva “include” che serve appunto ad includere altri file di codice nel file corrente.
Segue poi la funzione “main”.
Questa non è un funzione come le altre, essa deve essere presente in tutti i programmi scritti in C.
Voi mi chiederete: perché?
In C tutte le funzioni hanno lo stesso grado gerarchico, non esiste una funzione più importante di un'altra.
Quindi come farebbe il compilatore a capire da quale funzione iniziare ad eseguire il codice?
Ecco il motivo per cui c'è una sola funzione più importante delle altre, la funzione di partenza, chiamata appunto “main”.
Ma non avevamo detto prima che ogni funzione va dichiarata, invocata e implementata?
Noi qui la stiamo solo implementando... strano eh?
Invece no, anche main si compone di questi tre momenti, ma due di essi sono particolari rispetto alle altre funzioni classiche.
Dove è dichiarata main? Chi la invoca?
Abbiamo detto che il compilatore sa solo che una funzione che si chiama main ci sarà di sicuro nel nostro codice, quindi non segnala errori se non viene dichiarata.
Se la dichiarate prima di implementarla non cambia niente e comunque non verrebbero sollevati errori dal compilatore.
Per quanto riguarda l'invocazione di main, abbiamo detto che una funzione viene invocata quando c'è bisogno di eseguire le istruzioni al suo interno.
Quando c'è bisogno di invocare la funzione “madre” di un programma? Quando si vuol far partire il programma stesso!
Capite quindi che main viene invocata dal sistema quando lanciamo il nostro programma.
I più attenti avranno anche notato che la funzione main ritorna un int.
A chi restituisce questo valore e per quale motivo?
La risposta è semplice: con l'istruzione return restituiamo un valore a colui che chiama la funzione, quindi l'int viene restituito al sistema perché è lui che chiama main.
Il perché invece riguarda la gestione degli errori.

Nella nostra semplicissima app non abbiamo eseguito operazioni con la memoria e altre inizializzazioni che di solito vengono fatte in main.
Quando queste inizializzazioni falliscono, un buon programmatore cambia il valore di ritorno di main in modo che sia diverso da zero, segnalando quindi un'errore.
In base al numero che ritorna da main viene interpretato il tipo di errore.
Rimane da spiegare la funzione printf().
Essa fa parte del modulo stdio e serve per stampare a console del testo formattato.
Riprenderemo meglio l'argomento quando parleremo di FILE in C, per adesso ci limitiamo ad usarla.
L'output di questo programmino sarà “Hello World!” e verrà stampato a console.

PUNTATORI E ARRAY
Il C ci permette di pasticciare a volontà con la memoria della macchina, forse anche troppo.
In questo capitolo vedremo cosa sono puntatori e array e le strette relazioni tra loro.

Cos'è un puntatore?
Prima di rispondere ricordiamo che ogni variabile, di qualunque tipo essa sia, ha un proprio indirizzo in memoria.
Detto questo, un puntatore non è altro che una variabile che contiene l'indirizzo in memoria di un'altra variabile (quindi l'unico tipo di variabile che non contiene valori numerici o letterali).
Una variabile puntatore deve essere dello stesso tipo di dato della variabile a cui *punta.*

Facciamo un esempio...
Supponiamo di avere questa variabile:

```
int numero;
```

ora dichiariamo un puntatore e lo facciamo puntare a questa variabile, cioè gli assegnamo l'indirizzo di “numero”.

```
int *puntatore;
puntatore = &numero;
```

Cosa sono tutti questi simboli strani?
Innanzitutto il “*” dopo il tipo della variabile sta proprio ad indicare che non si tratta di una variabile intera, ma di un puntatore ad una variabile intera.
Abbiamo detto poi che ad un puntatore si assegna l'indirizzo della variabile alla quale puntare.

Questa operazione viene fatta grazie al simbolo "&", che messo prima di un nome di una variabile ne restituisce l'indirizzo in memoria.
Questo indirizzo lo assegnamo a "puntatore".
A questo punto le scritture:

```
numero = 5;
```

e

```
*puntatore = 5;
```

hanno lo stesso effetto, perché entrambe modificano il valore contenuto al medesimo indirizzo di memoria.
Ma qual è la differenza tra scrivere la variabile puntatore con e senza il "*"?
Ciò che cambia è il modo in cui viene utilizzata la variabile stessa.
Infatti, se il nome del puntatore è scritto senza "*", significa che ci stiamo riferendo all'indirizzo a cui punta.
Se invece lo scriviamo con "*", vuol dire che ci stiamo riferendo al valore contenuto nell'indirizzo a cui punta.
Ecco perché, come nell'esempio sopra, quando assegnamo al puntatore l'indirizzo della variabile "numero", lo scriviamo senza "*", mentre quando andiamo a modificare il valore contenuto in quell'indirizzo con "5", lo scriviamo con "*".
Esiste un tipo particolare di puntatore, ovvero il puntatore a void.
Esso si dichiara ovviamente così

```
void *puntatore;
```

e a differenza degli altri puntatori, può contenere l'indirizzo di qualsiasi tipo di dato (int, float, double, char).
Esso è detto anche puntatore indefinito.
Un puntatore, di qualsiasi tipo esso sia, occupa sempre lo stesso spazio in memoria, quindi un puntatore a char occuperà lo stesso spazio di un puntatore a double e lo stesso di un puntatore indefinito.

ARRAY
Gli array sono delle variabili un po' particolari.
Essi infatti sono variabili che contengono un insieme ordinato di variabili, tutte dello stesso tipo.
Un'array si dichiara in questo modo:

```
"tipo_di_dato" "nome_array" ["numero_di_elementi"] = {"elementi"};
```

Ad esempio dichiariamo un array di 15 interi...

```
int arrayDiInteri[15] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
```

Notate quindi che nelle parentesi quadre troviamo il numero degli elementi che contiene l'array, mentre nelle graffe (dopo l'uguale) troviamo tutti gli elementi veri e propri.
Ogni elemento dell'array è rintracciabile grazie ad un indice.

Gli indici degli array partono da 0 e non da 1!!!

Quindi il primo elemento dell'array si troverà all'indirizzo 0, il secondo all'indirizzo 1 e così via...
Nel nostro caso l'array è di 15 elementi, ciò vuol dire che gli indici vanno da 0 a 14.
Come prendo gli elementi uno ad uno con questi benedetti indici?
La sintassi per fare questo in C è la seguente:

```
int numero = arrayDiInteri[4];
```

Semplicemente basta scrivere il nome dell'array, seguito dalle parentesi quadre al cui interno l'indice dell'elemento che si vuole prendere in considerazione.
In questo esempio ho preso l'elemento con indice 4, quindi il numero “5” e l'ho assegnato a una variabile intera.
Ora mi chiederete... qual'è questa stretta relazione tra array e puntatori di cui ci hai parlato all'inizio?
Prima di rispondere dobbiamo fare delle considerazioni.
Un array è un insieme *contiguo* di elementi, cioè tutti gli elementi sono in memoria uno dietro l'altro senza spazio tra di essi. Quanto occupa in memoria un array?
Semplice: quanto occupa il tipo di dato contenuto moltiplicato il numero di elementi.
Ad esempio...
un array come quello sopra è formato da 15 interi.
Ogni intero in memoria sappiamo che occupa 16 bit (o anche 32, ma per convenzione facciamo 16), cioè 2 byte.
Gli interi all'interno dell'array sono 15, quindi l'array occupa un spazio in memoria pari a:
15 * 2 = 30 byte.
Ora quando noi scriviamo:

```
arrayDiInteri[4];
```

chiediamo al compilatore di restituirci l'elemento con indice 4 nell'array.
Ma come fa il compilatore a trovarlo?
Ecco qui che entrano in gioco i puntatori!
I puntatori hanno un'aritmetica particolare, infatti una scrittura del genere:

```
puntatore++;
```

incrementa l'indirizzo... cioè sposta l'indirizzo al quale esso punta, di “n” byte, quanti sono i byte occupati dal tipo di dato al quale punta.
Se questo è un puntatore a carattere (un char occupa un byte) l'istruzione precedente sposterà in avanti di 1 byte l'indirizzo puntato.
Se invece fosse stato un puntatore a float (un float occupa 32 bit, cioè 4 byte) l'istruzione precedente avrebbe spostato in avanti l'indirizzo puntato di 4 byte.
Quindi come fa il compilatore a trovare l'elemento alla posizione 4 di un'array di interi?
La risposta a questo punto è: prende l'indirizzo dell'array (che corrisponde all'elemento con indice 0) e lo incrementa di 2 byte per 4 volte.
E' molto interessante notare che il nome dell'array senza essere seguito dalle parentesi quadre è in realtà un puntatore!

Infatti per leggere il primo elemento possiamo benissimo scrivere:

```
*arrayDiInteri
```

al posto di...

```
arrayDiInteri[0]
```

Stessa cosa per il secondo elemento (indice 1) possiamo scrivere:

```
*(arrayDiInteri + 1)
```

al posto di...

```
arrayDiInteri[1]
```

Aggiungo infine che un'array può anche essere dichiarato senza il numero di elementi che esso contiene, a patto che venga subito inizializzato.
E' lecita quindi la seguente dichiarazione:

```
int arrayDiInteri[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
```

Rispetto al primo esempio manca il numero di elementi nella parentesi quadre, ma possiamo ometterlo perché lo stiamo inizializzando nella stessa istruzione.

ARRAY MULTIDIMENSIONALI
Un array può avere anche più di una dimensione: quando ciò si verifica parliamo di array multidimensionali.
L'array con dimensioni maggiori di 1 più usato è l'array a 2 dimensioni, che in matematica rappresenta semplicemente una matrice.
Si dichiara così:

```
int matrice[3][4];
```

Abbiamo dichiarato un array di interi di due dimensioni (matrice di 3 righe e 4 colonne).
Potenzialmente in C un array potrebbe avere infinite dimensioni, ma si consiglia comunque di non superare la terza dimensione per evitare problemi di gestione.

STRINGHE

CARATTERI
Prima di spiegare come trattare le stringhe in C, facciamo dei cenni sulle variabili char.
Esse sono variabili che occupano 1 byte di memoria (8 bit) e sono adatte per contenere un valore letterale.
Un valore letterale non è nient'altro che un numero intero, che viene convertito in carattere dalla macchina. Abbiamo anche detto però che una variabile char può essere usata come un piccolo int, perché al suo interno possiamo "infilarci" dei piccoli numeri interi.
Quindi come facciamo a far capire alla macchina quando il char deve essere interpretato come intero e

quando come lettera?
In C per scrivere un carattere e farlo interpretare come tale dalla macchina non dobbiamo far altro che scriverlo tra due apici ' '.
Ad esempio:

```
char carattere = 'A';
```

Ho dichiarato una variabile char e l'ho inizializzata con il carattere 'A'.
Come ho detto prima una lettera non è altro che un numero particolare, che viene rappresentato come carattere dalla macchina.
In teoria qualsiasi numero potrebbe rappresentare qualsiasi lettera, ma proprio per evitare confusioni è stata stabilita una convenzione, uno standard internazionale, chiamato codice "ASCII".
Riporto la tabella ufficiale delle relazioni "numeri-caratteri" del codice ASCII (Fonte: Wikipedia):

Non stampabili

Binario	Oct	Dec	Hex	Abbr	PR	CS	CEC	Descrizione
000 0000	000	0	00	NUL	$^{N}_{U_L}$	^@	\0	Null character
000 0001	001	1	01	SOH	$^{S}_{O_H}$	^A		Start of Header
000 0010	002	2	02	STX	$^{S}_{T_X}$	^B		Start of Text
000 0011	003	3	03	ETX	$^{E}_{T_X}$	^C		End of Text
000 0100	004	4	04	EOT	$^{E}_{O_T}$	^D		End of Transmission
000 0101	005	5	05	ENQ	$^{E}_{N_Q}$	^E		Enquiry
000 0110	006	6	06	ACK	$^{A}_{C_K}$	^F		Acknowledgment
000 0111	007	7	07	BEL	$^{B}_{E_L}$	^G	\a	Bell
000 1000	010	8	08	BS	$^{B}_{S}$	^H	\b	Backspace
000 1001	011	9	09	HT	$^{H}_{T}$	^I	\t	Horizontal Tab
000 1010	012	10	0A	LF	$^{L}_{F}$	^J	\n	Line feed
000 1011	013	11	0B	VT	$^{V}_{T}$	^K	\v	Vertical Tab
000 1100	014	12	0C	FF	$^{F}_{F}$	^L	\f	Form feed
000 1101	015	13	0D	CR	$^{C}_{R}$	^M	\r	Carriage return
000 1110	016	14	0E	SO	$^{S}_{0}$	^N		Shift Out
000 1111	017	15	0F	SI	$^{S}_{I}$	^O		Shift In
001 0000	020	16	10	DLE	$^{D}_{L_E}$	^P		Data Link Escape

001 0001	021	17	11	DC1	$^{D}_{C_1}$	^Q		Device Control 1 (oft. XON)
001 0010	022	18	12	DC2	$^{D}_{C_2}$	^R		Device Control 2
001 0011	023	19	13	DC3	$^{D}_{C_3}$	^S		Device Control 3 (oft. XOFF)
001 0100	024	20	14	DC4	$^{D}_{C_4}$	^T		Device Control 4
001 0101	025	21	15	NAK	$^{N}_{A_K}$	^U		Negative Acknowledgement
001 0110	026	22	16	SYN	$^{S}_{Y_N}$	^V		Synchronous Idle
001 0111	027	23	17	ETB	$^{E}_{T_B}$	^W		End of Trans. Block
001 1000	030	24	18	CAN	$^{C}_{A_N}$	^X		Cancel
001 1001	031	25	19	EM	$^{E}_{M}$	^Y		End of Medium
001 1010	032	26	1A	SUB	$^{S}_{U_B}$	[^Z		Substitute
001 1011	033	27	1B	ESC	$^{E}_{S_C}$	^[	\e	Escape
001 1100	034	28	1C	FS	$^{F}_{S}$	^\		File Separator
001 1101	035	29	1D	GS	$^{G}_{S}$	^]		Group separator
001 1110	036	30	1E	RS	$^{R}_{S}$	^^		Record Separator
001 1111	037	31	1F	US	$^{U}_{S}$	^_		Unit Separator
111 1111	177	127	7F	DEL	$^{D}_{E_L}$	^?		Delete

Stampabili

Binario	Oct	Dec	Hex	Glifo
010 0000	040	32	20	Spazio
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?
100 0000	100	64	40	@

100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c

110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

Seguendo questo codice però non è possibile rappresentare caratteri particolari di altre lingue come il cinese ad esempio.
Proprio per questo motivo è stata introdotta anche un'altra convenzione chiamata “UNICODE”.
Il C segue comunque l'ASCII.

STRINGHE
Cos'è una stringa? Voi mi direte: una parola, un insieme di lettere!
Risposta esatta, ma come la rappresentiamo?
In C non esiste una variabile di tipo “string” capace di contenere più lettere (caratteri).
Noi sappiamo però che una stringa è un insieme di caratteri, quindi variabili di tipo char, e sappiamo anche che per creare “file contigue” di variabili in memoria bisogna usare gli array...
Ebbene si, le stringhe in C si rappresentano come array di variabili char!
Facciamo subito un esempio:

```
char stringa[] = {'C', 'i', 'a', 'o'};
```

Per scrivere una semplice parola di 4 lettere, abbiamo creato un'array di char ed elencato i singoli caratteri tra apici.
Scomodo vero?
Poiché il C è linguaggio molto virtuoso, abbiamo anche qui delle piccole scorciatoie.
Infatti la scrittura precedente è equivalente a:

```
char stringa[] = “Ciao”;
```

Capiamo quindi che quando vogliamo scrivere più caratteri senza separarli da virgole possiamo semplicemente mettere tutto dentro “ ” (doppie virgolette).
In automatico il compilatore per far capire alla macchina che la stringa è finita, posizionerà alla fine dell'ultima lettera il carattere '\0', detto anche “carattere tappo”.

'\0' non sono due caratteri, ecco perché lo scriviamo tra singoli apici.

Ormai sappiamo benissimo che array e puntatori sono parenti stretti, per questo motivo una stringa può essere rappresentata (oltre che come array di char) anche come puntatore a char.
Ecco un esempio:

```
char *stringa = “Ciao”;
```

“stringa” (senza “*”) contiene l'indirizzo del primo carattere, in questo caso l'indirizzo di “C” in memoria.
“*stringa” (con “*”) sta per il primo carattere, in questo caso “C”.
Allo stesso modo:

```
*(stringa + 1)
```

e

```
stringa[1]
```

corrispondono entrambe al carattere 'i'.
Chiaro no?
E cosa succede se scriviamo stringa[4]? Stiamo dicendo al compilatore di restituirci il quinto char

dell'array ma la parola “Ciao” è formata solo da 4 caratteri...
Come ho accennato prima, troveremo il carattere '\0' che ci dice che la stringa è finita!

ANCORA FUNZIONI

In questo capitolo cercheremo di spiegare alcune delle funzioni principali usate nei programmi scritti in C e capiremo anche dei trucchetti come la ricorsione.
Per i concetti basilari di funzione si rimanda al Capitolo 1.

PARAMETRI DELLE FUNZIONI

Le funzioni possono anche avere parametri, cioè variabili dichiarate nelle parentesi tonde della funzione stessa.
Vediamo un esempio di programma avente due funzioni:

```
#include <stdio.h>

int somma (int primo, int secondo);       //dichiarazione di somma()

int main (void)
{
        printf("Questa è una semplice app che somma due numeri prestabiliti. \n");
        printf("Ora sommeremo questi due numeri: 3 e 6. \n");

        int primoNumero = 3;     //creo una variabile int e la inizializzo a 3
        int secondoNumero = 6;   //creo una variabile int e la inizializzo a 6

        int result = somma(primoNumero, secondoNumero);   //invocazione di somma()

        printf("Il risultato è: %d", result);   //stampo il risultato a console

        return 0;
}

//implementazione di somma()
int somma (int primo, int secondo)
{
        int risultato;  //dichiaro una variabile int
        risultato = primo + secondo;  /*effettuo la somma e salvo il risultato nella
          variabile appena creata*/
        return risultato;  //restituisco il valore contenuto in risultato
}
```

In questo paragrafo introduciamo molti concetti nuovi.
Iniziamo spiegando il codice riga per riga.
Come nell'esempio precedente, nella prima riga troviamo #include <stdio.h> e già sappiamo cosa vuol dire.
Prima dell'implementazione di main() stavolta troviamo la dichiarazione di una funzione, chiamata somma() che ci servirà per effettuare la somma tra due numeri interi.
Quindi con questa riga di codice abbiamo solo detto al compilatore: "carissimo compilatore ti avviso che più avanti troverai una funzione che ho chiamato somma, avrà come parametri due variabili intere, quando la invocherò non fare lo schizzinoso!"
Proseguiamo poi con l'implementazione di main().
Abbiamo due printf() che stampano semplici stringhe di testo a console per far capire all'utente di che applicazione si tratta.
Non fate caso a "\n" alla fine di ogni stringa in printf(), lo spiegheremo nel paragrafo dedicato a questa funzione.
Andando avanti troviamo due dichiarazioni di variabili che sono "primoNumero" e "secondoNumero" e li inizializziamo rispettivamente a 3 e a 6, i due valori che sommeremo.
Piccola parentesi: avete notato (anche nel capitolo precedente...) che la dichiarazione e l'inizializzazione delle variabili avvengono nella stessa istruzione?
Questa è una convenzione del C per abbreviare il codice e renderlo più scorrevole.
Infatti le due istruzioni:

```
int primoNumero = 3;
int secondoNumero = 6;
```

sono equivalenti a:

```
int primoNumero;
int secondoNumero;

primoNumero = 3;
secondoNumero = 6;
```

Andiamo avanti...
Arriviamo finalmente all'attesissima invocazione della funzione somma().
Cosa succede qui?

```
int result = somma(primoNumero, secondoNumero);
```

Ho creato un'altra variabile int chiamata result, ma dopo l'uguale invece di un valore intero trovo l'invocazione di una funzione!
Una scrittura del genere fa capire di certo che quella funzione ritornerà un valore intero.
Dove lo metto questo valore intero? Dentro una variabile intera, ovvio! Questa variabile è result.
Adesso capiamo come ho invocato la funzione, come le passo i suoi parametri e come essa li manipola.
La funziona si invoca semplicemente scrivendo il suo nome seguito dalle parentesi tonde.
Il posto dove infiliamo i parametri da passarle sono proprio queste parentesi tonde.

Infatti è proprio quello che ho fatto: ho passato alla funzione somma le due variabili che ho creato precedentemente (notate che i parametri si separano con una virgola quando sono più di uno, è una regola di sintassi...).
Ma c'è qualcosa che non quadra... perché le due variabili che passo come parametri si chiamano "primoNumero" e "secondoNumero", mentre nelle parentesi ho scritto "primo" e "secondo"?
In C, infatti, quando passiamo dei parametri alle funzioni, non passiamo la variabile in sé per sé, ma solo il valore contenuto in essa.
Quindi io non sto passando le variabili "primoNumero" e "secondoNumero", ma sto passando i valori interi contenuti in esse.
Questi due valori devono essere immagazzinati in due variabili intere ovviamente: quali?
Sono proprio quelle che creo nelle parentesi di somma(): "primo" e "secondo".
Se proviamo infatti a modificare il valore della variabile "primo", il valore di "primoNumero" non verrà intaccato: esse sono due variabili ben distinte e separate, ecco perché ho dato nomi diversi.
Quindi i parametri di una funzione non sono altro che variabili dichiarate nelle parentesi tonde della funzione stessa, le quali ricevono dei valori quando la funzione viene invocata.
Una volta invocata la funzione somma() vengono eseguite le istruzioni al suo interno prima di proseguire con main().
A questo punto "primo" e "secondo" valgono rispettivamente 3 e 6.
Creo quindi una variabile intera che chiamo risultato.
Il suo valore è uguale alla somma tra "primo" e "secondo".
Troviamo ora l'istruzione:

```
return risultato;
```

Questo è il momento in cui viene restituito alla funzione chiamante (main() in questo caso) il valore contenuto nella variabile risultato (attenzione il valore, non la variabile stessa!!! E' lo stesso concetto di prima!!!).
Il valore della variabile risultato, che a questo punto è 9 (6 + 3), dove lo "infilo"?
Poiché siamo usciti dalla funzione somma ritorniamo di nuovo al punto dove essa era stata invocata e cioè a questa istruzione:

```
int result = somma(primoNumero, secondoNumero);
```

Il valore di risultato lo "infilo" in result, l'ho chiamata in modo diverso proprio per far capire il concetto: result e risultato sono due variabili diverse, non si conoscono.
Prima di chiudere l'esecuzione dell'app con "return 0;" stampiamo a console il valore della variabile result con la funzione printf().
Spiegheremo tra un'istante cosa vuol dire %d.

I PARAMETRI DI MAIN
Come abbiamo detto nel capitolo precedente, main() è chiamata dal sistema.
Nell'applicazione Hello World, abbiamo dichiarato main() senza alcun parametro, infatti tra le parentesi tonde nell'implementazione troviamo "void".
In realtà anche main() può avere dei parametri che le possono essere passati da linea di comando quando l'app viene lanciata.
Non sono usati spesso, ma li cito completezza.
Vediamo quali possono essere:

- int argc - Numero degli argomenti della riga di comando, compreso il nome del programma.

- char *argv[] - Indirizzo dell'array di stringhe rappresentanti ciascuna un parametro della riga di comando. La prima stringa è il nome del programma completo di pathname se l'esecuzione avviene in una versione di DOS uguale o successiva alla 3.0, altrimenti contiene la stringa "C". L'ultimo elemento dell'array è un puntatore nullo.

- char * envp[] - Indirizzo dell'array di stringhe copiate dall'environment (variabili d'ambiente) che il DOS ha reso disponibile al programma. L'ultimo elemento dell'array è un puntatore nullo.

Facciamo adesso una panoramica di qualche funzione molto usata in C.

PRINTF
Ecco la famosissima funzione per stampare a console il testo formattato.
Ci vorrebbe un libro intero solo per parlare di questa funzione, ma ovviamente ci limitiamo a descriverla nelle sue parti principali.
La sua dichiarazione ed implementazione si trovano in stdio.h.
printf è dichiarata così:

```
int printf(const char *format, ...);
```

Il testo che vogliamo stampare va incluso nelle virgolette aperte e chiuse “ “ (che come sappiamo rappresenta un puntatore a char, o array di char, come possiamo vedere dal primo parametro di printf()).
Il secondo parametro cos'è? Io vedo dei puntini sospensivi “...” al posto di variabili. Cosa significano?
Significa che la funzione printf() può avere parametri variabili dal secondo in poi, che sono proprio le variabili che noi vogliamo farle stampare a console: più variabili le diciamo di stampare, più saranno i parametri, ovvio!
Ma come diciamo a printf() quante variabili stampare oltre al testo?
E' proprio grazie ad alcuni caratteri speciali che inseriamo nella stringa, che è il suo primo parametro.
Ce ne sono vari che indicano una determinata formattazione del testo e alcuni di essi li abbiamo già incontrati come “\n” e “%d”.
Facciamo adesso un elenco più chiaro e pulito:

- “\n”, sposta accapo il cursore;
- “%d”, inserisce il valore della variabile intera posta dopo le virgolette chiuse, seguite da una virgola (l'esempio è quello del paragrafo precedente);
- “%f”, inserisce il valore della variabile decimale (float o double) posta dopo le virgolette chiuse, seguite da una virgola;
- “%c”, inserisce il valore della variabile carattere (char) posta dopo le virgolette chiuse, seguite da una virgola;
- “%s”, inserisce il valore della stringa (variabile char* oppure char[]) posta dopo le virgolette chiuse, seguite da una virgola;
- “%p”, inserisce l'indirizzo di memoria al quale punta il puntatore posto dopo le virgolette chiuse, seguita da una virgola.

Queste sono le principali, per maggiori informazioni su altre formattazioni vi rimando alla documentazione ufficiale di questa funzione.

SCANF
Esatto opposto di printf(), serve per formattare l'input.
La sua dichiarazione è identica a printf(), a parte il nome della funzione ovviamente e si trova anch'essa in

stdio.h.

```
int scanf(const char *format, ...);
```

Ma cosa vuol dire esattamente formattare l'input?
Quando inseriamo dei valori da testiera, essi vengono sempre interpretati come testo.
Quindi anche ad esempio il numero 8 verrà trattato come testo e non come variabile int.
Formattare l'input vuol dire proprio questo: interpretare ciò che si riceve in un determinato tipo di dato e salvarlo ovviamente in un'apposita variabile.
Come diciamo alla funzione di salvare il numero 8 come int, piuttosto che come float o double?
Semplice, con gli stessi caratteri speciali che abbiamo trovato in printf().
Ad esempio se il valore che riceviamo dalla tastiera vogliamo interpretarlo come int, tra virgolette scriveremo "%d".
Dal secondo parametro in poi cosa ci mettiamo?
In printf() i parametri erano le variabili, il cui valore volevamo far stampare a schermo.
In scanf() invece, i parametri sono le variabili in cui vogliamo salvare il valore ricevuto da tastiera.

scanf() vuole come parametri dei puntatori, cosa che non succede con printf().

Per capire meglio quest'ultimo concetto facciamo degli esempi con entrambe le funzioni.
Con la seguente istruzione stampiamo a console il valore di un'ipotetica variabile int che chiamiamo "numero".

```
printf(" %d ", numero);
```

Come possiamo vedere il parametro di printf() è la variabile, niente di più.
Ora vediamo un'istruzione che serve per convertire in "int" l'input ricevuto da tastiera e salvarlo in un'ipotetica variabile che chiamiamo "variabile".

```
scanf(" %d ", &variabile);
```

Cosa cambia rispetto a printf a livello di parametri?
printf() vuole come parametri delle variabili.
scanf() vuole come parametri dei puntatori a variabili.
Nell'esempio con scanf() avremmo dovuto creare un puntatore, assegnargli l'indirizzo di "variabile" e poi darlo in pasto a scanf() come parametro.
Sarebbe uno spreco di tempo e di variabili, ecco perché, sapendo che un puntatore è una variabile che contiene un'indirizzo di un'altra variabile, accorciamo tutti i passaggi passando come parametro direttamente l'indirizzo di "variabile".
Sarà scanf() poi ad andare a modificare il valore contenuto in quell'indirizzo che gli passiamo come parametro.

Un ERRORE diffusissimo è il seguente:

```
scanf(“ %d ”, variabile);  //SBAGLIATO!!!!!!!!
```

La versione corretta invece è:

```
scanf(“ %d ”, &variabile);   //OK!
```

L'INPUT DIVENTA OUTPUT

In questo paragrafo creiamo una piccola applicazione che stampa a console un carattere che noi scriviamo da tastiera.
Scrivete il seguente codice nel vostro editor di testo...

```
#include <stdio.h>

int main ()
{
	char carattere;	//Dichiaro una variabile di tipo carattere.

	printf("Questa è una semplice app che trasforma l'input in output. \n");
	printf("Digitare il carattere da stampare a console. \n");

	scanf("%c", &carattere);   /* Converto ciò che viene digitato da tastiera in un char
				e lo salvo nella variabile "carattere" */

	printf("Il carattere digitato è: %c \n", carattere);

	return 0;
}
```

Nella prima riga di codice troviamo il solito #import del modulo stdio.
In main() iniziamo col dichiarare una variabile char che chiamiamo carattere.
Seguono due printf() di informazione all'utente.
Ecco che arriva scanf() la quale aspetta che l'utente inserisca un carattere da tastiera e prema invio.
Dopo aver schiacciato quest'ultimo tasto, il carattere digitato dall'utente viene convertito in una variabile char e successivamente, tramite un puntatore che contiene l'indirizzo della variabile “carattere”, scanf() lo immagazzina in quest'ultima.
Infine il carattere viene stampato a console.

EXIT

Due righe per descrivere brevemente la funzione exit().
Semplicemente, questa funzione serve per “uccidere” l'applicazione.
Molto usata quando si verificano errori di qualsiasi tipo; essa infatti ha come parametro un intero che varia a seconda dell'errore ricevuto.

Spetterà poi allo sviluppatore interpretare il numero per capire che tipo di errore c'è stato durante l'esecuzione.
Un esempio:

```
exit(1);
```

Dopo aver eseguito questo codice l'app si chiuderà e verrà restituito il numero “1” come descrizione dell'errore.

RAND E SRAND
Altre due funzioni molto utili ed interessanti.
Sto parlando di rand() e srand(), pensate per l'estrazione di numeri casuali e dichiarate nel modulo “stdlib”.
Una precisazione da fare: *i numeri casuali non esistono!*
Le funzioni infatti, si articolano in particolari calcoli matematici basati su un “seme”, che riescono a garantire comunque una buona casualità del numero estratto.
Il seme è un numero che viene fornito da noi come parametro della funzione srand().
Questo seme verrà poi usato nella funzione rand() per l'esecuzione dei calcoli e l'estrazione del numero.
C'è però un problema: se il seme è sempre lo stesso valore, il numero finale che verrà estratto sarà sempre uguale!
Come facciamo quindi per fornire un seme valido per il nostro scopo?
Ci serve attingere un numero da una fonte che varia continuamente....
Fatemici pensare... ho trovato! E' l'orologio di sistema!
Ma come prendo i valori dall'orologio?
Il C, nel modulo “time”, mette a disposizione una funzione che si chiama time(), alla quale se come parametro viene passato NULL ci restituisce l'ora corrente.
Comodo vero?
Ma come fisso “i paletti”, o limiti numerici, all'interno dei quali estrarre il numero casuale?
In altre parole come scrivo se voglio un numero compreso tra 0 e 100?
Bisogna usare l'operatore “%” (resto di una divisione intera)!
Prima di scrivere un programmino di esempio ricordo che srand() va chiamata *una e una sola volta,* mentre rand() va chiamata ogni volta che si desidera estrarre un numero.
Ora aprite il vostro editor di testo e digitate quanto segue:

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

int main (int argc, const char * argv[])
{
   srand(time(NULL));

   printf("Questa semplice app ci permettera di estrarre un numero casuale compreso tra 0
       e 99. \n");
```

```
    int numeroCasuale = rand() % 100;

    printf("Il numero estratto è: %d", numeroCasuale);

    return 0;
}
```

Come potete vedere, nelle prime 3 linee di codice importo gli header di 3 moduli: stdio, stdlib (serve per rand() e srand()), time (serve per time()).
Chiamo subito srand(), ma cosa trovo all'interno delle parentesi come parametro? L'invocazione di una funzione!
Cosa succede in quell'istruzione? Viene eseguita prima la funzione più interna, quindi time().
Abbiamo detto che time() se ha come parametro NULL ci restituisce l'ora corrente.
Questo numero dovremmo metterlo in una variabile e poi passarlo come parametro a srand(), ma sarebbe uno spreco di tempo: il numero restituitoci da time() viene passato direttamente a srand() in un'unica istruzione, senza buttare via variabili.
Segue una printf() di descrizione dell'app.
Troviamo adesso la dichiarazione di una variabile intera e la inizializziamo con il valore di ritorno di rand().
Come ho detto prima per stabilire "i paletti" bisogna usare l'operatore "%", ed è proprio quello che ho fatto.
Il numero di ritorno sarà compreso tra 0 e 99, provare per credere!
Infine stampiamo il numero estratto a console.

LA RICORSIONE
Una funzione ricorsiva è una funzione che richiama se stessa.
Ogni funzione può richiamare se stessa, anche main(), ma bisogna progettare attentamente l'algoritmo per evitare ricorsioni infinite e non funzionanti.
L'esempio più semplice dove applicare la ricorsione riguarda il calcolo del fattoriale di un numero.
Ricordo che il fattoriale di un numero naturale è il prodotto del numero stesso per tutti i suoi antecedenti.
In altre parole, il fattoriale di 7 si ottiene così:

7*6*5*4*3*2*1

Cerchiamo di capire come eseguire questo calcolo in C applicando la ricorsività...
Aprite il vostro editor di testo e digitate il codice seguente:

```
#include <stdio.h>

int calcolaFattoriale (int numero);

int main (int argc, const char * argv[])
{
    int numeroFinale = calcolaFattoriale( 7 );
```

```
    printf("Il fattoriale di 7 è: %d", numeroFinale);

    return 0;
}

int calcolaFattoriale (int numero)
{
    if (numero < 2) //se numero è minore di 2
        return 1;   //restituiusci 1
    else            //altrimenti
        return (numero * calcolaFattoriale(numero-1));
}
```

Nella prima riga importiamo il solito modulo.
Dichiariamo la funzione calcolaFattoriale() che useremo per calcolare il fattoriale del numero 7.
La invochiamo all'interno di main() e salviamo il valore di ritorno (fattoriale di 7) nella variabile numeroFinale.
Nell'implementazione calcolaFattoriale usiamo un'espressione condizionale, che approfondiremo in seguito: per ora vi basti sapere ciò che è scritto nei commenti.
Nell'ultima riga di calcolaFattoriale() troviamo l'invocazione ricorsiva vera e propria, infatti viene chiamata la funzione calcolaFattoriale() all'interno di se stessa.
Ma la funzione che chiamiamo non è la stessa che stiamo eseguendo adesso: è un'altra funzione ben distinta, con parametri diversi ma che hanno lo stesso nome della funzione corrente.
In pratica si ha una struttura del genere:

main() ---> calcolaFattoriale(7) ---> calcolaFattoriale(7-1) ---> calcolaFattoriale(7-1-1) ---> calcolaFattoriale(7-1-1-1) ...

Ad ogni “return” si procede a ritroso fino a main().
Quindi in questo modo:

main() <--- calcolaFattoriale(7) <--- calcolaFattoriale(7-1) <--- calcolaFattoriale(7-1-1) <--- calcolaFattoriale(7-1-1) <--- ...

Arrivati di nuovo a main() abbiamo il nostro fattoriale bello pronto che mettiamo in numeroFinale e stampiamo a console.

PUNTATORI A FUNZIONE
Ebbene si! Anche nelle funzioni troviamo i puntatori.
Cosa sarà mai un puntatore a funzione? Semplice: una variabile, che invece di contenere l'indirizzo di un'altra variabile, contiene l'indirizzo di una funzione.
Esso si dichiara in questo modo:

“tipo_di_dato” (*ptrFunz)(“parametri_funzione”);

Ad esempio il seguente è un puntatore ad una funzione che ritorna un intero ed ha come parametro un'altro intero:

```
int (*ptrFunz)(int numero);
```

Le parentesi sono obbligatorie!

Se avessi scritto così:

```
int *ptrFunz (int numero);
```

Sarebbe stata una semplice funzione che ritorna un puntatore ad intero a chi la chiama!
L'indirizzo di una funzione si ottiene semplicemente scrivendo il suo nome (quindi senza "&").
Capite benissimo che ad un puntatore a funzione l'indirizzo a cui puntare si assegna in questo modo:

```
ptrFunz = nome_funzione;
```

Il nome della funzione non deve essere seguito dalle due parentesi tonde, altrimenti la invocheremo!!!

Scriviamo una stupidissima ed inutile applicazione come esempio:

```
#include <stdio.h>

void stampaUno (void);  //dichiaro la funzione stampaUno()

int main (int argc, const char * argv[])
{
    void (*ptrFunz)(void);  /* dichiaro un puntatore ad una funzione che non ritorna nessun
                               valore e non ha nessun parametro */

    ptrFunz = stampaUno;    //Assegno a ptrFunz l'indirizzo di stampaUno()
    ptrFunz();              //Invoco la funzione puntata da ptrFunz

    return 0;
}
```

```
void stampaUno (void)
{
   printf("1 \n");
}
```

L'applicazione è talmente stupida che le poche spiegazioni necessarie sono elencate tramite commenti nel codice stesso.

CICLI E CONDIZIONI

In questo capitolo vediamo come far diventare “intelligente” la nostra app inserendo all'interno del codice delle espressioni condizionali e dei cicli.

IF, ELSE IF, ELSE

Nel nostro codice avremmo bisogno prima o poi di inserire un'espressione di questo tipo.
L'if-else è l'espressione condizionale più usata e serve a dire al compilatore: ”se succede una determinata cosa, allora esegui queste istruzioni, se invece ne succede un'altra esegui queste altre istruzioni, altrimenti esegui queste ultime istruzioni”
Si viene a creare questo tipo di schema:

```
if (succede_questo)
{
        ...fai_questo...
}
else if (succede_quest_altro)
{
        ..fai_quest_altro
}
else
{
        ...fai_quest_altro_ancora...
}
```

In questo schema logico ci possono essere infiniti “else if”, oppure anche solo il primo “if” e nient'altro.
Cerchiamo di capire meglio con un esempio...
Scrivete e compilate il seguente codice:

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

```
int main (int argc, const char * argv[])
{
    int numero;

    printf("Ora testeremo la tua intelligenza! \n");
    printf("Inserisci un numero intero compreso tra 0 e 10. \n");

    scanf("%d", &numero);

    if (numero > 0 && numero < 10)
    {
        printf("Bravo! Hai inserito un numero valido! \n");
    }
    else
    {
        printf("Il numero inserito non è valido! \n");
    }

    return 0;
}
```

Come potete vedere, chiediamo all'utente di inserire un numero intero compreso tra 0 e 10.
Immagazzino il numero inserito nella variabile intera "numero".
Ora arriviamo alle espressioni condizionali!
Grazie ad esse "testiamo" il valore di "numero" e ci comportiamo diversamente a seconda del valore di questa variabile.
Se "numero" è maggiore di 0 ***e*** minore di 10 (ho usato l'AND logico...) stampiamo un messaggio di OK all'utente, **altrimenti** diciamo che il numero inserito non corrisponde ai criteri che gli abbiamo indicato.
Una piccola curiosità: quando l'istruzione dopo l'if (o dopo qualsiasi espressione o ciclo) è solo una, le parentesi graffe possono essere omesse.

SWITCH
Lo switch è un'altro tipo di espressione condizionale ideale per testare valori di variabili ed agire di conseguenza.
La struttura logica dello switch è la seguente:

```
switch (variabile)
{
```

```
  case valore1:
    istruzione1
          break;

  case valore2:
    istruzione2
          break;
  ...
  default:
    istruzione_default
}
```

Nelle parentesi tonde mettiamo la variabile che vogliamo analizzare, mentre i “case” i riferiscono a tutti i *casi* che si potrebbero presentare.
Quindi per ogni determinato valore che prevediamo nei vari “case”, eseguiremo istruzioni diverse.
E il break a cosa serve? Semplice: ad uscire dallo switch e procedere con il codice successivo.
Prima di spiegare meglio questo concetto facciamo un esempio:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int numero;

    printf("Ora testeremo la tua intelligenza! \n");
    printf("Inserisci un numero intero compreso tra 0 e 4. \n");

    scanf("%d", &numero);

    switch (numero)
    {
        case 1:
        printf("Hai digitato 1\n");
        break;

        case 2:
```

```
        printf("Hai digitato 2\n");
        break;

        case 3:
        printf("Hai digitato 3\n");
        break;

        case 4:
        printf("Hai digitato 4\n");
        break;

        default:
        printf("Numero non valido!\n");
    }

    return 0;
}
```

Nell'esempio, con lo switch stiamo vedendo il valore della variabile “numero”: se vale 1 (case 1:) informiamo semplicemente l'utente che ha digitato 1, se vale 2 (case 2:) ecc... , se il valore di “numero” non corrisponde a nessuno dei casi previsti (default:) informiamo l'utente che il numero digitato non è valido.
Ora è il momento di rispondere praticamente alla domanda “a cosa serve il break”?
Consideriamo queste linee di codice:

```
case 1:
        printf("Hai digitato 1\n");
        break;
```

Se l'utente digita uno, l'esecuzione del codice entra in questo case.
Stampa il messaggio all'utente con printf() e poi?
C'è uno stop!
Usciamo dallo switch grazie a “break” e ci ritroviamo direttamente qui:

```
 return 0;
```

cioè alla fine di tutto lo switch.
Se non avessimo inserito break dopo il case 1, le istruzioni sarebbero state eseguite a cascata, passando a

case 2: senza uscire dal blocco switch.
Ricordo infine che la parte “default:” di uno switch non è obbligatoria.

WHILE
Con questo paragrafo passiamo dalle espressioni condizionali ai cicli.
Cos'è un ciclo? Possiamo dire che un ciclo è un'insieme di istruzioni che si ripetono continuamente fino a quando è verificata una determinata condizione.
I cicli del C sono il **while, do while** e **for**.
Iniziamo ad analizzare il ciclo while.
La sua struttura è la seguente:

```
while (condizione)
{
        istruzioni
}
```

All'interno delle parentesi tonde va inserita la condizione.
Vengono poi eseguite le istruzioni tra le parentesi graffe (sempre se la condizione è verificata!).
Arrivati alla fine (parentesi graffa chiusa) non si esce dal ciclo, ma si ricomincia da capo (parentesi graffa aperta).
In pratica, arrivati alla fine, il compilatore fa questo ragionamento: “La condizione è ancora verificata?” “Se si, riparto dall'inizio, altrimenti esco dal ciclo while e continuo l'esecuzione del codice successivo”.
Cosa succede se la condizione è sempre verificata? E se non lo è mai?
Si avranno rispettivamente cicli infiniti oppure cicli che non verranno mai eseguiti, quindi fate molta attenzione alla logica che usate nello scrivere la condizione di un ciclo.
Scriviamo un programmino di esempio...

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int numero = 0;

    printf("Questa è un'app che incrementa un numero da 0 a 10 progressivamente. \n");

    while (numero >= 0 && numero <= 10)
    {
        printf("Il numero adesso vale: %d. \n", numero);
        numero++;
    }
```

```
    return 0;
}
```

Tralasciamo le righe di codice iniziali che ormai sapete benissimo cosa fanno ed arriviamo direttamente al while.
Nella condizione ho scritto (usando l'AND logico...) che numero deve essere maggiore o uguale a zero e minore o uguale a 10.
Fino a quando questa condizione sarà verificata verranno eseguite le due istruzioni all'interno del ciclo.
La prima stampa il valore di numero, la seconda lo incrementa, poi se la condizione è vera si ricomincia da capo.

DO WHILE
Il ciclo do while è identico al while, con la sola differenza che le istruzioni, nel do while, vengono eseguite almeno una volta, anche se la condizione non lo permette.
La sua struttura è la seguente:

```
do
{

        istruzioni

} while (condizione);
```

Come vedete infatti, le istruzioni da eseguire si trovano prima della condizione e per questo devono essere eseguite almeno una volta anche se la condizione è sempre falsa.

Questa volta il ";" è obbligatorio dopo while (condizione)! Non dimenticatelo o riceverete un errore dal compilatore.

Riscriviamo l'esempio precedente con il do while:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int numero = 0;

    printf("Questa è un'app che incrementa un numero da 0 a 10 progressivamente. \n");
```

```
    do
    {
        printf("Il numero adesso vale: %d. \n", numero);
        numero++;
    } while (numero >= 0 && numero <= 10);

    return 0;
}
```

Stavolta credo proprio che l'esempio non necessiti di altre spiegazioni.

FOR

Il for è l'ultimo ciclo che andiamo ad analizzare.
Esso ha la seguente struttura:

```
for (assegnazione; condizione; incremento)
{


}
```

Come tutti i cicli, il for esegue le sue istruzioni fino a quando la condizione presente tra le sue parentesi tonde lo permette.
Ma cosa sono assegnazione ed incremento nelle parentesi tonde?
Il for è un ciclo pensato per svolgere operazioni dipendenti da una variabile, come ad esempio scorrere tutti gli elementi di un'array (l'esempio che scriveremo farà proprio questo...).
A questa variabile che creiamo, assegnamo un valore nell'assegnazione di un for (capirete benissimo che questa istruzione viene eseguita solo la prima volta, non ad ogni iterazione del ciclo.
Viene controllata poi la condizione e, se vera, vengono eseguite le istruzioni del for.
Alla fine prima di ripartire da capo viene eseguita l'istruzione di incremento sulla variabile (ho scritto incremento, ma sulla variabile potete fare di tutto , non solo incrementarla).
I più attenti hanno già capito che il for può essere anche riprodotto così:

```
assegnazione


while (condizione)
{
        ...istruzioni...
```

```
        incremento
}
```

Facciamo ora una piccola app di esempio che scorre tutti gli elementi di un array e li stampa a console:

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    int arrayDiInt[] = {1, 2, 3, 4, 5}; //dichiaro ed inizializzo un'array di 5 interi

    printf("Questa applicazione scorrerà tutti gli elemnti di un'array e li stamperà a console. \n");

    int indice;     //dichiaro l'indice che userò per scorrere l'array all'interno del for
    for (indice = 0; indice < 5; ++indice)
    {
        printf("L'elemento alla posizione %d dell'array è: %d. \n", indice, arrayDiInt[indice]);
    }

    return 0;
}
```

Vediamo cosa succede in questo codice...
Dichiariamo inizialmente un'array di interi e lo inizializziamo con valori che vanno da 1 a 5.
Segue la solita printf() "di cortesia".
Dichiaro una variabile int che chiamo "indice" e che mi servirà subito dopo come indice per scorrere l'intero array.
Nel for inizializzo la variabile indice a 0.
Indice è minore di 5? Si vale 0, allora esegue le istruzioni del ciclo.
Essa è una printf() che stampa l'elemento alla posizione indice-esima dell'array "arrayDiInt".
Segue l'istruzione di incremento della variabile indice.
Adesso la variabile indice è minore di 5? Si vale 1: si ricomincia da capo, e così fino a quando la variabile indice non sarà più minore di 5.

GOTO
Questa è un'istruzione usata pochissimo dai programmatori perché in grado di creare il cosiddetto "spaghetti-code", cioè codice molto difficile da capire e pieno di errori ed è per questo che anche io ve la sconsiglio.
Ma cosa fa questa istruzione di così pericoloso? La risposta è: salta!

Goto sta per Go To (vai a), viene sempre usata quando c'è un'etichetta e salta a quest'ultima quando viene eseguita.
Un'esempio è questo:

```
etichetta:
    ....istruzioni varie.....

goto etichetta;
```

Quando goto viene usata l'esecuzione del codice salta letteralmente all'etichetta di riferimento, con la possibilità che alcuni "pezzi" di codice non vengano mai eseguiti.
Fate molta attenzione e soprattutto non usatela! Qualsiasi effetto creato dalla goto è riproducibile tranquillamente anche con le altre espressioni o cicli.

PREPROCESSORE

In questo capitolo spiegheremo le principali direttive che possiamo dare al preprocessore della macchina, ovvero tutti quei comandi che iniziano con "#".

INCLUDE

L'abbiamo incontrata praticamente sempre e sappiamo già cosa fa quando viene usata.
Serve ad includere un modulo nel file di codice corrente.
L'esempio più banale....

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

DEFINE

Questa direttiva serve per la creazione di **costanti manifeste.**
Queste ultime sono costanti non dichiarate con la parola chiave const, ma appunto con una direttiva.
Cosa cambia tra le 2? Io direi in pratica... niente!
La differenza c'è solo a livello concettuale, ovvero, con const dichiariamo "una variabile che non cambia mai il suo valore", mentre con #define informiamo il preprocessore di una costante intrinseca al programma stesso.
Ecco il modo giusto per dichiarare costanti manifeste:

```
#define NOME_COSTANTE        "Ciao"
```

Fornisco un'altro esempio dichiarando una costante intera:

```
#define GIORNI_DELL_ANNO    365
```

Dopo quest'altro esempio credo proprio che non dovrebbero esserci più dubbi riguardo questa direttiva.

IF, ELIF, ELSE, ENDIF
Direttive condizionali per il preprocessore, molto usate quando il codice viene compilato su più piattaforme.
A differenza del normale if else, queste direttive, se non verificate, rimuovono proprio del codice dalla compilazione.
Ogni #if deve obbligatoriamente terminare con un #endif.
Poichè le direttive al preprocessore devono essere rappresentate da una sola parola, else if e end if diventano rispettivamente #elif e #endif.
Ecco un esempio:

```
#define UNITED_STATES 1
#define ENGLAND 2
#define ITALY 3
#define COUNTRY 3

#if COUNTRY==ITALY
        char *valuta = "Euro";

#elif COUNTRY==UNITED_STATES
        char *valuta = "Dollaro";

#else
        char *valuta = "Pound";
#endif
```

IFDEF, IFUNDEF, ENDIF
Queste altre direttive sono strettamente legate con la direttiva #define.
Esse infatti servono a stabilire se una costante manifesta è stata definita o meno, ed agire di conseguenza.
IFDEF restituisce VERO se la costante è definita, IFNDEF restituisce VERO se la costante non è definita.

Ad esempio:

```
#define COSTANTE

#ifdef COSTANTE

...fai qualcosa...

#endif
```

Questo if sarà vero, mentre questo...

```
#define COSTANTE
```

```
#ifndef COSTANTE
```

```
...fai qualcosa...
```

```
#endif
```

...sarà falso.

MACRO
Possiamo definire costanti di qualunque tipo con la direttiva #define, che non si fermano ad un semplice valore.
Anche delle funzioni, infatti possono essere definite come costanti, dette **macro**, ma alla loro chiamata, il compilatore non si comporterà allo stesso come se fosse una normale invocazione di una funzione.
Quando invochiamo una funzione, il compilatore va a cercare in memoria ed esegue tutte le istruzioni appartenenti a quella determinata funzione.
Con una macro, invece, il compilatore non cerca altrove, ma sostituisce tutto ciò che c'è dopo #define, al posto della chiamata a funzione.
Vediamo praticamente...

```
#define somma(a,b) (a+b)

main ()
{
        int numero = somma(2, 3);
}
```

Alla chiamata di somma quindi il compilatore non eseguirà istruzioni presenti altrove in memoria, ma si limiterà a fare un "copia e incolla" producendo questo effetto:

```
int numero = (2+3);
```

Chiaro no?
La definizione di macro velocizza l'esecuzione del codice e risparmia memoria rispetto ad una normale funzione.

PRAGMA
La direttiva #pragma serve ad utilizzare specifiche funzioni che variano da compilatore a compilatore.
Ogni comando va specificato dopo #pragma, ma per avere una visione chiara dei comandi disponibili bisogna consultare il manuale del compilatore utilizzato.

GESTIONE DELLA MEMORIA

Gestire la memoria significa ottimizzare al meglio il codice al fine di migliorare le prestazioni dell'applicazione finale.
Esistono varie funzioni per gestire questi compiti che fanno parte della libreria standard del C.
Analizziamo le 3 più usate per l'allocazione, più ovviamente la funzione che servirà per rilasciare la memoria allocata.

SIZE OF
Prima di iniziare a descrivere le funzioni, c'è bisogno di introdurre un nuovo operatore: sizeof.
Sizeof viene usato per sapere quanto occupa in memoria il dato, o il *tipo di dato,* che gli viene passato.
Facciamo un'esempio per entrambe le situazioni:

```
char variabile;
printf(“La variabile occupa: %d”, sizeof(variabile));
```

Questa scrittura ci restituirà sicuramente 1, perché sappiamo benissimo che la variabile char occupa un byte (8 bit).
Sizeof è molto utile perché possiamo passare anche tipi di dati ad esempio...

```
printf(“Un int occupa: %d”, sizeof(int));
```

Come vedete stavolta ho passato un tipo di dato e non una variabile.
Il tipo di dato restituito da sizeof() non è stato mai incontrato fino adesso; esso è **size_t**.
Questo operatore verrà usato moltissimo nelle funzioni per l'allocazione della memoria.

TYPE-CASTING
Descriviamo brevemente il casting perché anch'esso ci servirà moltissimo per gestire al meglio la memoria.
E' un tecnica molto utile per convertire tipi di dati ed è usata spessissimo per i puntatori a causa della loro aritmetica che varia a seconda del tipo di dato puntato.
Per convertire un tipo di dato in un'altro si usa la seguente sintassi...

```
int numero = 5;
double secondoNum = (double)(numero);
```

quindi scrivendo nella prima coppia di parentesi il tipo di dato nel quale convertire la variabile, scritta nella seconda coppia di parentesi tonde.

MALLOC
Quando nel nostro codice scriviamo un'istruzione come questa:

```
int numero;
```

il compilatore in automatico riserva uno spazio di memoria sufficientemente ampio da contenere una variabile intera.
A questo punto voi mi chiederete a cosa serve allora allocare spazi di memoria se lo fa già il compilatore per noi?
Rispondo a questa domanda facendo un'altro esempio:

```
char *stringa;
*stringa = “Ciao”;
```

Nella prima istruzione dichiariamo un puntatore a carattere, fin qui tutto ok.
Nella seconda, invece, andiamo a modificare il valore contenuto nell'indirizzo a cui stringa punta.
Ma noi non abbiamo assegnato nessun indirizzo a cui puntare alla variabile “stringa”!!!
La probabilità che questo codice crashi è molto elevata!
Evitiamo tutti i problemi riservando uno spazio di memoria libero per la variabile stringa...
La funzione “base” per l'allocazione della memoria si chiama malloc(), vuole come parametro il numero di byte da riservare e ritorna il puntatore indefinito al primo byte di memoria allocato (che dobbiamo convertire in un puntatore a char).
La sua struttura è:

```
void *malloc(size_t size);
```

Riprendendo l'esempio di “stringa”, ci basta fare così...

```
char *stringa;
stringa = (char *) malloc( 100 );
```

A questo punto alla variabile “stringa” potremmo assegnare una stringa lunga fino a 100 caratteri.
Siccome noi abbiamo allocato spazio per i “char” (occupano **un byte** ognuno) non abbiamo usato sizeof (perchè la funzione malloc ha come parametri proprio il numero di **byte**).
Ma se volessimo riservare spazio per tipi di dati più grandi di un byte? Ad esempio un int?
Voi mi direte: un int occupa 2 byte, quindi i byte da allocare sono numeroDiVariabili * 2.
Risposta esatta! Usiamo sizeof(int) al posto di “2” solo per una questione di pulizia e leggibilità del codice, ma potreste passare alla funzione anche direttamente il risultato della moltiplicazione eseguita a mente da voi.
Adesso allochiamo uno spazio in memoria per contenere 100 int...

```
int *numeri =(int *) malloc(100 * sizeof(int));
```

Infatti se con l'esempio precedente avessi usato questa scrittura:

```
stringa = (char *) malloc( 100 * sizeof(char) );
```

Avremmo ottenuto lo stesso risultato, in quanto 100 * 1 = 100.

CALLOC
calloc() è una funzione molto simile a malloc(), con la sola differenza che è stata ideata per allocare spazi contigui in memoria.
Essa accetta 2 parametri: numero di elementi e spazio occupato da ogni singolo elemento.
Come malloc(), ritorna un puntatore indefinito (void*) al primo byte di memoria allocato, quindi è necessario un type-casting per convertire il puntatore indefinito in un puntatore specifico (a seconda del tipo di variabili con cui si vuole riempire la memoria allocata).
La sua struttura è:

```
void *calloc(size_t NumElements, size_t elementSize);
```

Facciamo un breve esempio:

```
char *stringa;
stringa = (char *) calloc( 100, sizof(char) );
```

Ho semplicemente allocato 100 byte in memoria (sizeof(char) = 1 byte) e il puntatore indefinito che mi restituisce la funzione l'ho convertito in un puntatore a char (il tipo di dato della variabile "stringa").

REALLOC
Cosa succede se ho bisogno di ampliare o ridurre una porzione di memoria allocata precedentemente?
Per questa operazione ci viene incontro la funzione realloc()!
Essa ha la seguente struttura:

```
void *realloc(void *ptr, size_t size);
```

Questa funzione modifica la dimensione della porzione di memoria puntata dal primo parametro, nella dimensione contenuta nel secondo parametro, e ne restituisce il puntatore al primo byte di memoria.
Ora voi mi chiederete: a cosa mi serve il puntatore che mi restituisce realloc() (puntatore al primo byte della porzione di memoria...) se io già l'avevo da prima?
La funzione realloc() infatti, se non riesce ad allungare o accorciare la porzione di memoria sposta tutto il suo contenuto in una nuova porzione di memoria (con la nuova dimensione) rilasciando la vecchia porzione.
Capite quindi che il puntatore può cambiare o essere lo stesso di prima, a seconda se la porzione di memoria viene spostata o meno.

FREE
Ecco la funzione più semplice di tutte!
free() serve per liberare la memoria allocata precedentemente con le funzioni sopra descritte (in realtà ne esistono anche altre che svolgono questo compito ma mi sono limitato a descrivere quelle che più utilizzerete).
Essa accetta come unico parametro il puntatore alla porzione di memoria da liberare e non restituisce

nessun valore:

```
void free(void *ptr);
```

Qualunque porzione di memoria allocata manualmente deve essere rilasciata con free()!

Facciamo un breve esempio...

```
char *stringa;
stringa = calloc( 100, sizeof(char) );
*stringa = “Ciao a tutti !”;
//uso “stringa” come mi pare e quando ho finito...
free(stringa);
```

NUOVI TIPI DI DATI

In questo capitolo descriveremo dei tipi di dati più complessi e vedremo come crearne di personalizzati.

STRUCT

In C una **struct** è proprio una struttura, formata da vari componenti.
I componenti di una struct possono essere di qualsiasi tipo, anche altre struct, ma mai la struct stessa (mentre è ammesso un puntatore alla struct stessa...)!
Il miglior modo di capire questo concetto è applicarlo in pratica.
Creiamo una struttura che chiamiamo “rettangolo” e che è composta da due variabili double: l'altezza e la larghezza.

```
struct rettangolo
{
        double larghezza;
        double altezza;
};
```

rettangolo non è una variabile, ma è un nuovo *tipo di variabile*.

Abbiamo infatti definito un nuovo tipo di dato vero e proprio, e quindi possiamo dichiarare nuove variabili di tipo “struct rettangolo”...

```
struct rettangolo myRect;
```

myRect è una variabile di tipo “struct rettangolo” e ha come caratteristiche due double: uno si chiama

altezza e l'altro si chiama larghezza.
Come faccio a questo punto a modificare i valori di una struttura?
Ci serviamo dell'operatore "." (punto).

se un valore di una struttura è un puntatore, non modificheremo il suo valore tramite ".", ma tramite l'operatore "->".
Facciamo un esempio pratico per capire il tutto...

```
#include <stdio.h>

int main (int argc, const char * argv[])
{
    struct rettangolo   //dichiaro una struttura che chiamo "rettangolo"
    {
        double altezza;
        double larghezza;
    };

    struct rettangolo myRect; /* dichiaro una variabile di tipo struct rettangolo che chiamo
                                 myRect */

    myRect.altezza = 10.8; //l'altezza del mio rettangolo è 10.8
    myRect.larghezza = 2.6; //la larghezza del mio rettangolo è 2.6

    printf("L'altezza del rettangolo è: %f.\n", myRect.altezza);
    printf("La larghezza del rettangolo è: %f.\n", myRect.larghezza);

    return 0;
}
```

Nelle prime righe dichiaro la struttura e successivamente creo una variabile di quel tipo.
Ne modifico l'altezza e la larghezza e stampo entrambi i valori a console.
Un piccolo trucchetto: quando dichiariamo una struttura possiamo, nella stessa istruzione, dichiarare anche delle variabili di quel tipo.
Ad esempio:

```
struct rettangolo
{
```

```
    double larghezza;
    double altezza;
} myRect;
```

UNION
Una union è identica ad una struct sia come dichiarazione, sia come accesso ai suoi elementi.
Cosa cambia allora?
Cambia il modo in cui viene usata la memoria!
Mentre una stuct è pensata come una struttura avente più elementi (quindi occupa una porzione di memoria pari alla somma della memoria occupata da ogni singolo elemento al suo interno), una union può essere rappresentata da una sola variabile tra quelle presenti al suo interno (ed occupa una porzione di memoria pari alla memoria occupata dall'elemento più grande presente al suo interno).
Per chiarire le idee immaginiamo che una union sia una faccia che può essere vista con tante maschere, è una variabile che può essere rappresentata da diversi tipi di dati.
Facciamo un esempio...

```
union numero
{
    int intero;
    float decimale;
    double doppia_precisione;
} myNumber;
```

A questo punto la variabile “myNumber” di tipo “union numero” NON contiene tre elementi (come nelle struct), ma **è** uno solo di questi elementi alla volta.
In altre parole myNumber può essere o un int o un float o un double.

ENUM
Le variabili enum sono utilissime quando si vuole assegnare valori non numerici ad una variabile, ma solo dei nomi simbolici.
Se ad esempio parliamo di colori, come facciamo ad esprimere un colore con un numero?
Dovremo crearci una tabella, in cui ogni numero corrisponde ad un colore e ricorrere quindi alla direttiva #define, che verrà usata molte volte.
Le variabili enum ci semplificano tutto ciò.
Ecco un'esempio di dichiarazione di una variabile di questo tipo:

```
enum colore
{
    rosso;
    verde;
    giallo;
    blu;
```

```
        marrone;
        arancione;
};
```

Per comodità non li elenco tutti, ci vorrebbe un libro intero solo per elencare tutti i colori!
Come vedete gli elementi all'interno dell'enum non sono variabili, ma solo dei nomi a cui il compilatore assegnerà dei numeri, ma che noi continueremo a chiamare per nome (di solito il compilatore assegna un ordine numerico crescente ad ogni parola).
La dichiarazione ricorda molto le union e le struct.
Un elemento esclude l'altro, ad esempio in una variabile di tipo "enum colore" possiamo assegnare un colore alla volta...

```
enum colore myColor;
myColor = giallo;
```

E' anche possibile non specificare il nome del nuovo tipo di dato nella dichiarazione di enum.
Questo però comporta l'impossibilità di creare variabili enum, ma sarà possibile solo utilizzare i vari nomi come se fossero stati dichiarati con #define.
Ad esempio:

```
enum
{
        giallo;
        rosso;
        verde;
        blu;
};
```

equivale a...

```
#define         giallo  1
#define         rosso   2
#define         verde   3
#define         blu     4
```

TYPEDEF
In C non possiamo creare veri e propri tipi di dati completamente nuovi, ma comunque possiamo personalizzare quelli che abbiamo.
Con typedef possiamo sostituire un nome personalizzato al nome di un tipo di dato classico, ad esempio:

```
typedef int pippo;
```

Da questa istruzione in poi, dichiare...

```
int numero;
```

o...

```
pippo numero;
```

sarà la stessa identica cosa.
Un'esempio più reale potrebbe essere il seguente:

```
typedef char* string;
```

Da questo momento in poi potremo usare variabili di tipo string (che sono dei char *) per scrivere stringhe, al posto di dichiararle come puntatori a char.
Voi mi direte: Ok, bello; ma dov'è l'utilità di tutto questo?
L'utilità è nell'accorciare il codice! Infatti i typedef vengono usati moltissimo nelle struct e nelle union.
Vediamo un esempio:

```
struct rettangolo
{
	double altezza;
	double larghezza;
} myRect;
```

La variabile myRect è di tipo “struct rettangolo”, ma concorderete con me che è un nome davvero lungo da scrivere ogni volta.
Se invece usiamo typedef...

```
typedef struct rettangolo
{
	double altezza;
	double larghezza;
};
```

Le nuove variabili che dichiareremo non saranno più di tipo “struct rettangolo”, ma solo di tipo

“rettangolo”.
Ecco un esempio:

```
rettangolo myRect;
```

Molto più comodo e pulito.

CAMPI DI BIT

C'è un tecnica in C per risparmiare memoria, cioè dichiarare accanto alla variabile il numero di bit che essa dovrà occupare.
Ovviamente meno bit daremo alla variabile, meno saranno i valori che potenzialmente potrà contenere.
Ad esempio supponiamo di aver bandito un concorso con un basso numero di partecipanti e stiamo scrivendo un programmino in C per la gestione di questo concorso:

```
struct concorso
{
        int partecipanti: 10;
        char *vincitore;
        char *amministratore;
}
```

Come potete vedere basta inserire il numero di bit da assegnare alla variabile dopo i “:” (due punti).
In questo caso alla variabile int ho assegnato 10 bit, mentre il compilatore gliene avrebbe regalati 16 o 32.

LAVORARE CON I FILE

In questo capitolo tratteremo con una struct dichiarata nel modulo stdio chiamata FILE, che ci servirà per salvare e leggere valori su disco.
Vedremo inoltre le funzioni base per l'apertura e la chiusura di un file e 3 struct FILE che il C ci mette già a disposizione.

FILE

FILE è la struttura principale per leggere e scrivere file su disco.
Non ci interessa la sua struttura perché non andremo a manipolarla direttamente, in quanto ci sono due funzioni specifiche che ci nascondono il "lavoro sporco"; ci basti sapere che rappresenta un flusso di byte che stavolta sono presenti nella memoria di massa e non nella RAM.
Le funzioni specifiche per l'apertura e la chiusura sono fopen() e fclose(), le analizzeremo tra un attimo.
Vi informo inoltre che la struttura FILE, dobbiamo sempre dichiararla come puntatore perché è questo il tipo di dato (FILE *) che vogliono le due funzioni sopra citate come parametro.

La creazione di una variabile FILE * non è automaticamente associata alla creazione di un vero e proprio file su disco.

FOPEN

Per leggere o scrivere file su disco dobbiamo usare la funzione fopen().
La sua struttura è la seguente:

```
FILE *fopen(char *file_name, char *mode);
```

Essa restituisce un puntatore a FILE e accetta come parametri il percorso completo del file (file_name) e il modo in cui si vuole trattare quel file (lettura, scrittura, appending, ecc...).
I modi più usati per l'apertura di un file sono i seguenti:

- "r", lettura (il file deve già esistere sul disco!);
- "w", scrittura (il file viene creato, o se esiste sovrascritto);
- "a", appending (il file viene creato, o se esiste la scrittura continua dalla fine del file esistente, senza sovrascrittura).

E' possibile che la funzione fopen() restituisca NULL quando si verifica una delle seguenti situazioni:

- Il file aperto in lettura non esiste;
- Il file è protetto.

FCLOSE

Non c'è molto da spiegare su questa funzione: semplicemente in C ogni file aperto con fopen() deve obbligatoriamente essere chiuso con fclose().
La struttura della funzione è la seguente:

```
void fclose(FILE *);
```

FILE DI TESTO

rattare con i file di testo in C è la cosa più semplice che esista, in quanto abbiamo a disposizione moltissime funzioni che ci semplificano enormemente il lavoro.
Introduciamo le 3 strutture di tipo FILE fondamentali in C:

- stdin, standard input (di default la testiera);
- stdout, standard output (di default lo schermo);
- stderr, standard error (di default lo schermo);

Le funzioni di I/O che abbiamo usato fino adesso, come ad esempio printf() e scanf(), usavano queste 3 strutture base rendendo il tutto completamente trasparente al programmatore.
Tuttavia ci possono essere dei casi in cui c'è bisogno di specificare il FILE sul quale scrivere, diverso da uno dei 3 fondamentali.
Il caso più banale di tutti è la scrittura di testo, non su schermo (stdout), ma su un file presente nella memoria di massa.
Per scrivere testo abbiamo fino adesso usato printf(), ma come facciamo ora a dirle che il FILE su cui scrivere non è stdout?
Esiste una variante di printf(), chiamata fprintf(), la quale ha un parametro in più rispetto alla funzione classica: il FILE su cui scrivere.
Capirete quindi che:

```
printf(“Ciao”);
```

oppure...

```
fprintf(stdin, “Ciao”);
```

hanno la stessa efficacia.
Esiste ovviamente anche la variante di scanf(), chiamata fscanf() che formatta l'input provenente da un'altro file diverso dallo stdout.
Un'ultima cosa: quando apriamo un file di testo possiamo scorrere tutti i caratteri presenti nel file, ma alla fine non troveremo un char, ma un int.
Quest'ultimo si chiama EOF (End Of File) ed è un carattere speciale che indica proprio la fine del testo presente nel file.
Ecco un esempio che copia l'input in output per far capire meglio quest'ultimo concetto:

```
#include <stdio.h>

main()
{
    int character;
    character = getchar();
    while (c != EOF)
    {
```

```
            putchar(c);
            character=getchar();
    }
}
```

Ho usato due nuove funzioni: getchar e putchar, la prima riceve un carattere dallo stdin, mentre la seconda stampa un carattere nello stdout.
Notate che character è dichiarato come int?
L'ho fatto perché EOF è un numero troppo grande per essere contenuto in un char.
Come abbiamo già appurato i char non sono altro che numeri (piccoli int) che tramite l'ASCII vengono trattati come caratteri di testo.
Alla macchina non interessa niente se questi numeri si trovano in un int piuttosto che in un char, quindi possiamo usare tranquillamente degli int come caratteri.

ESEMPIO CONCLUSIVO
Concludo questo capitolo con un'app completamente funzionante per riassumere tutti i concetti spiegati fin'ora sui file.
L'app legge e scrive un numero su un percorso specificato dall'utente.
Poiché l'applicazione è stata scritta su un Mac, i percorsi usati nell'esempio saranno tipici del filesystem di Mac OS X.
Per questione di chiarezza e di spazio l'esempio è riportato nella pagina successiva.

```
/* Scrittura e lettura di un intero da file txt */

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

void write (int num, char *path);
int read (char *path);

int main (void)
{
        int choiche, num;
        char *path = calloc(200, sizeof(char));
        printf("\n\t*** TXT Reader & Writer ***\n\n");
        printf("Quale operazione desideri compiere?\n1.Scrittura su file.txt;\n2.Lettura da
        file.txt;\n");
        scanf("%i", &choiche);
        switch (choiche)
```

```
        {
          case 1:
          printf("Digita il numero che desideri salvare.\n");
          scanf("%i", &num);
          printf("Ora digita il percorso completo sul quale salvare, compreso nome ed estensione
          del file (ES: /Users/nome_utente/Desktop/nome_file.txt).\n");
          scanf("%s", path);
          write(num, path);
          break;

          case 2:
          printf("Digita il percorso completo dal quale leggere, compreso nome ed estensione del
          file (ES: /Users/nome_utente/Desktop/nome_file.txt).\n");
          scanf("%s", path);
          int num = read(path);
          printf("L'intero al percorso %s è: %i.\n", path, num);
          break;

          default:
          printf("La scelta effettuata non è compresa tra quelle disponibili.\n");
          break;
        }
        return 0;
}

void write (int num, char *path)
{
        FILE *fp;
        fp = fopen(path, "w");
        fprintf(fp, "%i\n", num);
        fclose(fp);
}

int read (char *path)
{
        int var;
```

```
    FILE *fp;
    fp = fopen(path, "r");
    fscanf(fp, "%i", &var);
    fclose(fp);
    return var;
}
```

STANDARD C LIBRARY

Descriviamo brevemente i compiti svolti da ogni modulo presente nella libreria standard del C.

Nome	Descrizione
<assert.h**>**	Contiene la macro assert, utilizzata per indentificare errori logici ed altri tipi di bug nelle versioni di debug di un programma.
<complex.h**>**	Un gruppo di funzioni usate per manipolare numeri complessi. (Aggiunto con il **C99**)
<ctype.h**>**	Questo header file contiene funzioni usate per classificare i caratteri in base ai loro tipi o per convertirli da maiuscoli a minuscoli, indipendentemente dal set di caratteri utilizzato (tipicamente ASCII, ma esistono anche implementezioni che usano l'EBCDIC).
<errno.h**>**	Per testare i codici di errore restituiti dalle funzioni di libreria.
<fenv.h**>**	Per controllare l'ambiente in virgola mobile. (Aggiunto con il **C99**)
<float.h**>**	Contiene delle costanti definite che indicano le proprietà specifiche dell'implementazione della libreria in virgola mobile, come ad esempio la minima differenza tra due numeri in virgola mobile (_EPSILON), il massimo numero di cifre significative (_DIG) e l'intervallo di numeri che possono essere rappresentati (_MIN, _MAX).
<inttypes.h**>**	Per effettuare conversioni precise tra i tipi interi. (Aggiunto con il **C99**)
<iso646.h**>**	Per programmare nel set di caratteri ISO 646.

	(Aggiunto con l'**NA1**)
<limits.h**>**	Contiene delle costanti definite che indicano le proprietà specifiche dell'implementazione dei tipi interi, come l'intervallo dei numeri rappresentabili (_MIN, _MAX).
<locale.h**>**	Per setlocale() e le costanti relative. Utilizzato per scegliere il codice locale adatto.
<math.h**>**	Per le funzioni matematiche comuni.
<setjmp.h**>**	Dichiara setjmp/longjmp, utilizzate per salti non locali.
<signal.h**>**	Per controllare varie condizioni d'eccezione.
<stdarg.h**>**	Utilizzato da funzioni che accettano un numero variabile di parametri.
<stdbool.h**>**	Per un tipo di dato booleano. (Aggiunto con il **C99**)
<stdint.h**>**	Per definire vari tipi interi. (Aggiunto con il **C99**)
<stddef.h**>**	Per definire vari tipi e macro utili.
<stdio.h**>**	Fornisce le funzionalità basilari di input/output del C. Questo file include il prototipo delle venerabili funzioni printf e scanf.
<stdlib.h**>**	Per eseguire un gran numero di operazioni, tra cui conversioni, generazione di numeri pseudo-casuali, allocazione di memoria, controllo del processo, variabili d'ambiente, segnali, ricerca ed ordinamento.
<string.h**>**	Per manipolare le stringhe.
<tgmath.h**>**	Per funzioni matematiche di tipo generico. (Aggiunto con il **C99**)
<time.h**>**	Per convertire tra vari formati di data e ora.
<wchar.h**>**	Per manipolare stream o stringhe contenenti caratteri estesi - fondamentale per supportare una lunga serie di lingue con caratteri non occidentali. (Aggiunto con l'**NA1**)
<wctype.h**>**	Per la classificazione dei caratteri estesi. (Aggiunto con l'**NA1**)

Fonte: Wikipedia.

UN GIOCHINO FUNZIONANTE

Per concludere propongo un giochino funzionante scritto in C, anche molto carino.
Si chiama iLucky e potete scaricare la versione per Mac dal mio sito, dove trovate anche le regole del gioco:

www.webalice.it/biagio.ianero

Il codice sorgente ve lo inserisco qui sotto come regalino finale...

```
#include <stdio.h>


int lanciaDadi();


int main (int argc, const char * argv[]) {


        int vuoiContinuare;
        int turni;
        int punteggio;
        int arrayNum[9];
        arrayNum[0] = 1;
        arrayNum[1] = 2;
        arrayNum[2] = 3;
        arrayNum[3] = 4;
        arrayNum[4] = 5;
        arrayNum[5] = 6;
        arrayNum[6] = 7;
        arrayNum[7] = 8;
        arrayNum[8] = 9;
        printf("Benvenuto! \n");
        printf("Buon Divertimento Con iLucky! \n");
        printf("Premere un numero per Iniziare, 0 per Abbandonare. \n");
        scanf("%d", &vuoiContinuare);
        printf("Inizia il gioco! \n");
        sleep(2);
        printf("Questi sono i tuoi numeri: \n \n");
        sleep(1);
        printf("%d  %d  %d  %d  %d  %d  %d  %d  %d \n \n", arrayNum[0], arrayNum[1], arrayNum[2],
arrayNum[3], arrayNum[4], arrayNum[5], arrayNum[6], arrayNum[7], arrayNum[8]);
```

```
while (vuoiContinuare != 0)
{
  int risultato;
  sleep(2);
  printf("Se Vuoi Lanciare i Dadi Premi un numero, 0 per uscire al prossimo turno. \n");
  scanf("%d", &vuoiContinuare);

  printf("Vengono lanciati i dadi. \n");
  sleep(1);
  printf("Aspetto il risultato... \n");
  sleep(4);
  risultato = lanciaDadi();
  printf("Questo è il risultato: %d. \n", risultato);

  if (risultato == 10)
  {
          risultato = 1;
  }
  else if (risultato == 11)
  {
          risultato = 2;
  }
  else if (risultato == 12)
  {
          risultato = 3;
  }
  sleep(2);

  int cancellaNum = risultato-1;

  if (arrayNum[cancellaNum] == 0)
  {
          printf("Questo Numero è già stato cancellato; \n");
          sleep(2);
          printf("Il tuo punteggio non verrà incrementato. \n");
```

```
            sleep(2);
            punteggio = punteggio - risultato;
        }

        else if (arrayNum[cancellaNum] != 0)
        {
            printf("Viene cancellato il numero %d. \n", risultato);
            sleep(2);
            arrayNum[cancellaNum] = 0;
        }

        printf("Rivediamo i tuoi numeri. \n \n");
        sleep(1);
        printf("%d  %d  %d  %d  %d  %d  %d  %d  %d \n \n", arrayNum[0], arrayNum[1],
arrayNum[2], arrayNum[3], arrayNum[4], arrayNum[5], arrayNum[6], arrayNum[7], arrayNum[8]);
        punteggio = punteggio + risultato;
        sleep(2);
        printf("Il tuo punteggio ora è di %d punti. \n", punteggio);
        turni ++;

        if (!arrayNum[0] && !arrayNum[1] && !arrayNum[2] && !arrayNum[3] && !arrayNum[4] && !
arrayNum[5] && !arrayNum[6] && !arrayNum[7] && !arrayNum[8])
        {
            printf("Complimenti! Hai cancellato tutti i numeri disponibili. \n");
            vuoiContinuare = 0;
        }
    }
    sleep(1);
    printf("Hai concluso il gioco con %d turni. \n", turni);
    sleep(2);
    printf("Il tuo Risultato Totale è di %d Punti \n", punteggio);
    sleep(2);
    printf("In Media hai Totalizzato %d Punti per Turno \n", (punteggio / turni));
    sleep(1);
    printf("Cerca di battere il tuo record! \n");
    sleep(1);
```

```
        printf("Gioca ancora con iLucky! \n");
        sleep(1);
        printf("Ciao, A Presto :D \n");
    return 0;
}


int lanciaDadi ()
{
        srand(time(0));
        int dadi;
        dadi = (rand()%13);
        if (dadi == 0) {
          dadi++;
        }
        if (dadi == 1) {
          dadi++;
        }
        return dadi;
}
```

RINGRAZIAMENTI

I soli e unici ringraziamenti vanno a te, caro lettore, che hai scelto di usare il mio libro per imparare il linguaggio C.
Se hai avuto difficoltà nella comprensione di qualche argomento, o per qualunque critica e consiglio, non esitare a contattarmi al mio indirizzo email:

biagio.ianero@alice.it
oppure a:
biagioianero@gmail.com

Sarò ben lieto di aiutarti.