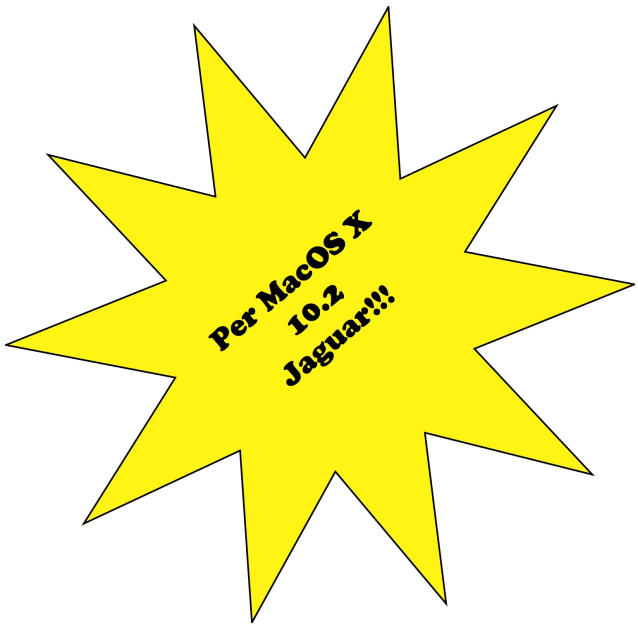
Marco Coïsson

Introduzione al Linguaggio C



Coïsson Editore

Introduzione al linguaggio C

Piano dell'opera

Ho voluto dare questo titolo altisonante a questa prefazione perché mai più mi capiterà di poterlo fare ancora. Molto più modestamente, vorrei qui darvi un'idea di che cosa sia questa *Introduzione al linguaggio C* e con quale scopo sia stata scritta. L'idea è di fornire una specie di trilogia, che parta per l'appunto da un'introduzione al linguaggio C per chi non lo conosce affatto o lo conosce poco; passare poi alla seconda puntata, che riguarderà l'ObjectiveC, un derivato del C che Apple raccomanda come principale linguaggio per scrivere applicazioni che funzionino con MacOS X; e concludere con la terza puntata, che farà vedere come l'ObjectiveC si integri perfettamente con Cocoa, il meraviglioso ambiente di sviluppo che Apple mette a disposizione per scrivere applicazioni native per MacOS X.

Questa prima puntata vuole essere una via di mezzo tra degli appunti di C e un tutorial. Essa è destinata a coloro che non sanno nulla di programmazione e vogliono avvicinarsi ad essa, ma anche a coloro che già sanno programmare in un linguaggio che non sia il C. I primi due capitoli sono "filosofici", nel senso che alcuni concetti di base verranno discussi un po' in astratto e in maniera forse un po' confusa, così, a mo' d'introduzione, giusto per non partire bruscamente con le cose noiose come le definizioni e la sintassi del linguaggio. Chi di voi è completamente a digiuno di programmazione troverà forse vaghe, confuse e inutili le chilate di parole spese in questi primi due capitoli. Ma poi, quando leggerete i capitoli successivi, tornerete a questi due per rileggervi quelle parole così assurde e astratte che in un primo tempo vi avevano lasciati un po' perplessi, e spero che in seconda lettura le troverete utili. Comunque, se preferite partire subito con i dettagli del linguaggio C, potete saltare i primi due capitoli: non ci perdete molto. Promettete però che li leggerete alla fine!

I capitoli dal 3 al 7, invece, introdurranno nel concreto definizioni e sintassi del linguaggio, coprendone tutti gli aspetti di base. Gli esempi saranno sempre brevi ed autocontenuti. Cercherò sempre e comunque di fare esempi che non siano solo degli estratti di programmi più grandi, ma che possano essere scritti ed eseguiti sul momento dal lettore.

Il capitolo 8 rappresenterà invece l'unica eccezione ad una regola che mi sono dato, ovvero quella di non parlare qui delle "estensioni" al linguaggio C; parleremo di due funzioni che torneranno utilissime fin tanto che non avremo a disposizione l'interfaccia utente che MacOS X ci mette a disposizione per comunicare con l'utente, e che verrà toccata solo nella terza puntata di questa trilogia. Si tratta di due funzioni che hanno lo scopo di scrivere caratteri in una finestra e di accettare un input da tastiera da parte dell'utente. Useremo queste funzioni anche nei capitoli precedenti; il capitolo 8 sarà una brevissima guida per l'uso di queste due funzioni limitatamente alle caratteristiche che ci servono negli esempi che faremo.

Che cosa faremo

Questo vuole essere un tutorial di base per l'apprendimento dei fondamenti del C. È adatto a chi sa poco o nulla di programmazione ma è interessato all'argomento. Copre aspetti generali di programmazione e del C, e pertanto quanto si imparerà qui potrà essere utilizzato in una varietà di contesti non necessariamente legati allo sviluppo di applicazioni per un particolare sistema operativo o con un particolare ambiente di sviluppo.

Che cosa non faremo

Non ci addentreremo nella discussione delle funzioni contenute nelle librerie standard del C. Questo perché sarebbe una faccenda troppo lunga e alla fine sarebbe inutile per il fine ultimo che mi propongo con la trilogia di articoli di cui questo è il primo, ovvero fornire un'infarinatura di base per programmare in ObjectiveC / Cocoa in MacOS X. Non ci occuperemo approfonditamente di puntatori, puntatori a funzioni e in generale di allocazione dinamica della memoria.

1. Introduzione – Panoramica sul C

Il C come linguaggio di "basso livello"

Agli albori dell'informatica "domestica", c'era l'abitudine di distinguere i linguaggi di programmazione come di "alto livello" e di "basso livello". Non si tratta di un modo per dire che certi linguaggi sono più belli e altri più brutti, e nemmeno per dire che alcuni di essi sono da preferire rispetto ad altri. Si tratta invece di distinguere che cosa possono fare e a quale scopo essi possano essere usati. Così, l'Assembler era un linguaggio di basso livello: esso permetteva di programmare il computer inserendo direttamente le istruzioni che venivano eseguite dal microprocessore. Si maneggiava direttamente la memoria, i registri del microprocessore e, tramite altre funzioni di basso livello fornite dal sistema operativo, si accedeva direttamente all'hardware (per esempio il lettore di floppy). Al contrario c'erano linguaggi di "alto livello", come il BASIC, il Logo, il C o il Pascal, che trattavano i dettagli riguardanti il computer in maniera astratta: non era più necessario sapere come fare affinché il processore potesse sommare due numeri, ma bastava usare il simbolo +. Non era più necessario sapere in quale locazione di memoria fossero registrati i dati da utilizzare, in quanto era possibile dare loro un nome simbolico e mnemonico (una variabile). Similmente non era necessario conoscere i dettagli del sistema operativo e dell'hardware per leggere o scrivere un file su disco.

Col tempo questa distinzione ha cambiato leggermente significato: oggi è raro che qualcuno usi l'Assembler, per cui i linguaggi di basso livello come li si intendeva una volta sono quasi scomparsi nell'utilizzo comune. Ecco allora che abbiamo un nuovo significato per questi termini. Si definiscono di "basso livello" i linguaggi come il C, che manipolano numeri e caratteri, che vengono organizzati in blocchi, cicli e funzioni. Vengono definiti di "alto livello" quei linguaggi come l'ObjectiveC, il Java, il Python, il PHP, il Perl ed altri che, oltre alle funzioni standard dei linguaggi di basso livello, offrono anche la possibilità di manipolare dati più complessi, come tabelle, vettori, stringhe di caratteri, insiemi, liste e strutture più complesse ancora.

Di fatto, la distinzione tra linguaggi di alto livello e di basso livello è diventata sempre più labile, perché a mano a mano che si sono sviluppati, i linguaggi di basso livello hanno guadagnato sempre più funzioni aggiuntive che li avvicinano ai linguaggi di alto livello. Senza poi dimenticare che, quasi sempre, quando si programma non ci si affida alle sole proprietà del linguaggio che si è scelto, ma si fa riferimento in maniera massiccia a ciò che offre il sistema operativo sul quale si sta sviluppando: e qui, l'approccio che si usa è spesso di "alto livello", nel senso che il sistema opeativo fa di tutto per nascondere il più possibile al programmatore le difficoltà e i dettagli tecnici relativi all'hardware, con ovvi vantaggi in termini di semplicità, velocità di programmazione e "riciclabilità" del codice. Come vedremo nel corso delle prossime puntate, MacOS X offre un livello di astrazione (è talmente di "alto livello") che utilizzarlo come programmatori è non soltanto facile, ma addirittura divertente.

Il C come linguaggio "strutturato"

Ogni volta che ci penso, mi rendo conto che chiamare "strutturato" un linguaggio come il C non vuol dire assolutamente niente. Si tratta di un retaggio dei tempi passati, che però può essere utile mantenere per chiarire un paio di dettagli. Nella notte dei tempi, due linguaggi molto "didattici" e molto usati erano il BASIC e il Pascal. Essi sono per molti versi agli antipodi.

Il BASIC è un linguaggio non necessariamente semplice, ma "piatto" (almeno nelle sue versioni più vecchie, più "tradizionali"): il codice è scritto in un file unico, ed è come se fosse su un unico piano; ogni riga di istruzioni non è né più né meno importante rispetto alle altre. Ogni variabile ha la stessa importanza delle altre. Nulla è nascosto a nessuno. Non c'è gerarchia.

Il Pascal è un linguaggio non necessariamente difficile, ma "strutturato": il codice può essere scritto su più di un file, e anche all'interno di un file esso è raggruppato in "scatole cinesi" o in più livelli; livelli o scatole più interni possono vedere solo alcune delle cose che sono al loro esterno o nelle scatole o livelli racchiusi al loro iterno. Ogni tanto, qualche

collegamento tra scatole o livelli consente una comunicazione bidirezionale tra questi, ma solo per alcuni percorsi ben definiti e solo per alcune cose specifiche. Ogni cosa all'interno del codice di programmazione ha una sua posizione gerarchica, un suo ruolo, è inquadrata in una "struttura". Il Pascal è un linguaggio strutturato.

Il C è un parente molto stretto del Pascal: è un linguaggio strutturato, ma le pareti delle scatole cinesi di cui si compone il codice talvolta hanno delle porticine, oppure sono trasparenti o permeabili a qualche cosa. È molto più facile che col Pascal realizzare un'organizzazione maggiormente "piatta", meno gerarchica, dei pezzi che costituiscono il programma. È un felice connubio tra il BASIC di una volta e il Pascal. È molto meglio del BASIC di oggi, che parte da un antenano non strutturato e cerca di estendersi verso un linguaggio strutturato, dandosi regole astruse e un'organizzazione confusa.

Il C è un buon linguaggio. Se sapete il C sapete programmare in qualunque altro linguaggio. Odierete alcune delle sue caratteristiche, ne amerete alla follia delle altre. Imparerete altri linguaggi, magari li userete più del C; ma quando avrete un dubbio, quando una cosa non vi sarà chiara, finirete per chiedervi: come lo farei in C? Il C sarà il vostro punto di riferimento, sempre e comunque. E poi, se state usando MacOS X, ricordatevi questo: MacOS X è Unix, e Unix e C sono un binomio inscindibile: il modo con cui funziona il C è lo stesso modo con cui funziona Unix. Il modo con cui pensate alla soluzione di un problema in Unix è lo stesso modo con cui affrontate un problema in C: tanti passettini verso la soluzione finale, ogni passettino distinto dagli altri, eppure tutti all'interno di un grande progetto che li coordina e li porta verso la meta finale.

Il C come linguaggio "strongly-typed"

Continuo questa carrellata filosofica dicendo che il C è un linguaggio "strongly-typed", ma che anche non lo è. Come sempre è questione di punti di vista.

"Strongly-typed" (non riesco a darvi una traduzione letterale sensata) vuol dire che se voi avete deciso di lavorare con delle patate, non potete improvvisamente passare a delle carote. Peggio ancora, vuol dire che se lavorerete con patate o con carote è una decisione che dovete prendere non nel momento in cui eseguite il programma, ma nel momento in cui scrivete il codice del programma. Portato alle estreme conseguenze, il programma per fare un puré di patate è diverso da quello che fa il puré di carote, anche se la procedura è esattamente la stessa per entrambi i casi, bisogna solo sostituire un ingrediente con un altro!

Se la mettete in questi termini, il C non è un linguaggio "strongly-typed". È vero che non potrete mai spacciare patate per carote, ma è anche vero che non dovete necessariamente decidere prima ancora di scrivere il programma se il vostro puré sarà di patate o di carote. Potrete tranquillamente fare un programma generico che faccia il puré, e poi l'ingrediente fondamentale lo deciderete all'ultimo momento, quando già il puré lo state preparando. Beh, per lo meno potrete scegliere tra un elenco di possibili ingredienti che avrete identificato all'interno del vostro programma come "ingredienti possibili".

Se la mettete in questi termini, il C è un linguaggio "strongly-typed". Esistono linguaggi come il Perl dove potete prendere delle patate, potete trattarle come carote, spacciarle per cipolle e rivenderle come lavatrici. Poi le ricomprate come case al mare e ne fate un frullato. Il tutto senza che nessuno si scandalizzi. È comodo quando dovete fare qualche "sporco trucco" o quando non volete stare lì a perdere troppo tempo, ma provate a vedere che cosa succede quando il vostro programma diventa sufficientemente grande e complesso e non sapete più con che cosa state lavorando!

Come vedremo nelle prossime puntate parlando di ObjectiveC e di Cocoa, la programmazione ad oggetti e MacOS X si coalizzano per darvi la giusta via di mezzo. E lì, dopo aver visto le delizie e le pene dei linguaggi "strongly-typed" e di quelli che non lo sono, ci commuoveremo.

Vantaggi e svantaggi del C

Io amo il C, quindi vi dico: tutti vantaggi e nessuno svantaggio. Ma in realtà non è vero. Il C è un linguaggio eccellente, formidabile, meraviglioso. Il C è un linguaggio con cui fate di tutto, ma non sempre è facile. Per scrivere una frase di saluto sullo schermo, in BASIC vi basta una linea di codice, in C ve ne servono da due a cinque (a seconda di come mettete le parentesi). Però in C fate delle cose che vi sognate nella maggior parte degli altri linguaggi. In C sapete sempre che cosa state facendo, di che cosa vi state occupando, quali dati state manipolando, dove sono, quanto spazio occupano, dove andranno a finire e perché. Per contro, in C dovete sempre sapere in anticipo di che cosa vi occuperete, quali e quanti dati vi troverete a manipolare, dove dovrete metterli, quanto spazio potranno occupare, dove andrete a metterli quando avrete finito e perché. Se fate un pasticcio, il C non ve lo perdona. Se fate un pasticcio, per lo meno vi rendete conto di averlo fatto (cosa che con molti altri linguaggi non è così scontata!) Il C è molto poco adatto per fare il programmino di prova, quello con cui si collauda un semplice algoritmo di poche linee di codice su un campione di dati fornito a mano. Ma se poi questo algoritmo dovete metterlo in un programma più complesso, allora il C diventa una manna dal cielo. Lo benedirete. Lo amerete…

Varianti del C

…e non lo userete più quando avrete per le mani qualche variante del C.

Oggi va molto di moda il C++. A mio modesto parere è un ottimo linguaggio, ma è un delirio. Obiettivamente fate delle cose eccellenti, ma con una fatica veramente improba! Se dovete programmare "ad oggetti", probabilmente userete il C++ e guarderete con un po' di superiorità e disprezzo il caro vecchio C, troppo "procedurale" per essere utile.

Va di moda il Java, che è una versione semplificata (e più ad "alto livello") del C, con qualche similitudine anche col C++. Non l'ho mai amato, ma se vi piace…

Vanno di moda il PHP (ottimo per la programmazione web, ed è la fotocopia del C, è fantastico!), il Perl (bello e assolutamente il contrario del C, utile finché il programma che fate è sufficientemente piccolo), Python, una miriade di linguaggi di scripting, e vari altri linguaggi più o meno specialistici che non elencherò nemmeno.

E poi c'è l'ObjectiveC. È fratello del C++, eppure è così diverso! Ha tutta la poesia del C, eppure sembra non assomigliargli. È il fratello bello, magro, alto, intelligente, prestante, spiritoso e sensibile, biondo e con gli occhi azzurri del C (magro ma meno aitante e meno spiritoso) e del C++ (intelligentissimo ma noioso e anche un po' grassottello). Quando conoscerete l'ObjectiveC nelle prossime puntate ve ne innamorerete (beh, lo so, l'ho descritto "al maschile" e magari siete dei maschietti come me, quindi non è detto che troviate la cosa molto appropriata, ma adesso non prendetemi troppo alla lettera!) Inizierete ad usare l'ObjectiveC non solo con MacOS X (che ci va a braccetto), ma anche con altri sistemi operativi, perché non potrete più farne a meno. Ma non illudetevi: se non conoscete il C, non conoscete nemmeno l'ObjectiveC. Non potete fare a meno della gavetta in C. Per cui coraggio, e andiamo avanti!

2. Anatomia di un programma

File, funzioni, blocchi e variabili

Dire che un programma scritto in C è organizzato in file sembra una banalità. Invece non è così. I BASIC di una volta imponevano che i programmi fossero scritti in un unico file. Applicazioni realizzate in ambienti particolari come SuperCard (un clone di HyperCard), ad esempio, sono invece costituite da tanti script associati ad oggetti o elementi di interfaccia i quali sono incapsulati in quella che potremmo chiamare l'applicazione vera e propria, che poi è un documento SuperCard (nel nostro esempio). Un'applicazione web è invece un insieme di pagine statiche e dinamiche (scritte in PHP o in Perl, ad esempio) che, eseguite sul web server che ospita il sito e magari appoggiandosi ad un database come MySQL, danno l'impressione di lavorare con un'unica applicazione dedicata ad un certo compito.

E un programma scritto in C? Esso è costituito da uno o più file nei quali viene scritto il codice da eseguire. Se il programma è interamente contenuto in un solo file la questione è un po' più semplice, se no è possibile frammentarlo in più file e poi ci sono tecniche opportune per far sì che i vari file vengano considerati come un tutt'uno. Un apposito programma, detto compilatore, avrà poi il compito di prendere il vostro codice C e trasformarlo in linguaggio macchina (sto un po' semplificando), cosicché sia eseguibile. Un altro programma, detto linker, prenderà gli eseguibili prodotti dal compilatore (uno per file di codice) e li metterà insieme, collegandoli opportunamente al sistema operativo (vedi più avanti il paragrafo "Librerie"), così da produrre un'applicazione contenuta in un solo file che possa essere eseguito dall'utente finale. Come tutto questo avvenga è interessante ma superfluo per imparare a programmare in C. Qui basti sapere che un programma scritto in C può essere scritto suddividendolo su più file (ad esempio raggruppando in un unico file tutte quelle parti di programma che eseguono compiti tra di loro collegati o che cooperano per effettuare una particolare operazione). Ad esempio, un programma scritto in C del gioco dell'oca potrebbe essere costituito da un file che si occupa della parte grafica (disegnare la plancia e i segnalini), di un file che si occupa di lanciare i dadi e muovere i segnalini (senza disegnarli), di un file che analizza le caselle in cui si fermano i segnalini e prende le decisioni del caso, e via discorrendo.

Indipendentemente da quanti file voi abbiate per il vostro programma, questi saranno costituiti da una o più *funzioni*, che rappresentano le unità logiche nelle quali si suddivide il vostro programma. Una funzione è un inseme di linee di programma con un compito preciso, una *funzione* precisa per l'appunto. È individuata da una scrittura di questo tipo:

```
int LanciaIDadi(giocatore)
{
     // varie linee di codice qui
}
```

Iniziamo ad esaminarla con un po' di dettaglio. Essa è dotata di un *nome* (`LanciaIDadi`), che ne dovrebbe identificare lo scopo. Questa funzione, nel nostro ipotetico gioco dell'oca, ha il compito di lanciare i dadi cosicché si possa stabilire di quante caselle debba avanzare il giocatore di turno. Questo, nel nostro esempio è identificato dalla *variabile* `giocatore`, una sorta di contenitore all'interno del quale memorizzate ad esempio il numero del giocatore di turno.

Per chiarire meglio il concetto di variabile, pensate ad un bicchiere: al suo interno potete mettere acqua, vino (di tantissime qualità diverse), spremute, succhi di frutta, tè… Il bicchiere è la vostra variabile (in realtà ad essere variabile è il contenuto, ma in programmazione è il contenitore ad essere chiamato variabile). Nel momento in cui voi avete un bicchiere, potete metterci dentro quello che volete (state "scrivendo" dei dati all'interno della variabile "bicchiere"). Quando volete fare qualche cosa col contenuto della vostra variabile (ad esempio bere il contenuto del bicchiere), la "leggete"; gli effetti potranno essere diversi a seconda di che cosa fosse contenuto nella variabile (un buon bicchiere di vino potrebbe piacervi di più che un po' d'acqua tiepida, ad esempio), ma la cosa fondamentale è che il bicchiere che contiene la

bevanda (la variabile) è sempre lo stesso. Vi basta avere un bicchiere solo, e al suo interno potete mettere tutte le bevande che volete (una per volta!) Così è per la nostra variabile `giocatore`. Essa potrebbe ad esempio essere definita come un recipiente in grado di contenere un numero intero, da 1 al numero di persone che stanno disputando al gioco dell'oca. Quando è il turno del primo giocatore, il numero 1 viene scritto all'interno della variabile `giocatore` (come, è una faccenda che vedremo nel prossimo capitolo). Da adesso in avanti, tutte le volte che una funzione avrà bisogno di sapere chi è il giocatore di turno basterà che vada a guardare che cosa c'è all'interno della variabile `giocatore` (la vada a leggere, anche qui il come è una faccenda che riguarderà il prossimo capitolo). Così, la nostra funzione `LanciaIDadi(giocatore)` si preoccuperà di simulare il lancio di un dado e saprà che il giocatore il cui numero è scritto nella variabile `giocatore` dovrà avanzare di tante caselle quanto sarà il risultato del lancio del dato. Come questo venga fatto è per ora irrilevante, ci arriveremo col tempo.

La cosa veramente eccezionale dell'accoppiata funzioni / variabili è che la funzione `LanciaIDadi(giocatore)` è una sola, ma le linee di codice (omesse) comprese tra le due parentesi graffe sono del tutto indipendenti da quale sia il giocatore di tuno. La funzione tratta in maniera astratta il giocatore di turno, mediante l'utilizzo di una variabile. Quando ha bisogno di fare qualche cosa di concreto (far avanzare il giocatore numero 1 e non il numero 2, ad esempio) non ha bisogno di contemplare separatamente il caso di ogni giocatore della partita, in quanto può riferirsi a turno ad ognuno di essi mediante la variabile `giocatore`.

Qualche altro dettaglio formale: le due parentesi tonde che racchiudono i cosiddetti *argomenti* della funzione (la variabile `giocatore`, nel nostro esempio), fanno parte del nome della funzione stessa, che pertanto più propriamente si chiama `LanciaIDadi()`. La scrittura `int` che precede il nome della funzione indica che la funzione stessa produce un *risultato*: nel nostro caso, ad esempio, potrebbe essere il valore risultante dal lancio del dado (quindi un numero intero tra 1 e 6). Questa è una prima carrellata sulle funzioni, ma è solo un'introduzione "filosofica": torneremo a parlare più nel dettaglio delle funzioni nel capitolo 5.

Di già che ci siamo, vediamo anche che cosa sono quelle due parentesi graffe. Esse delimitano il contenuto di una funzione. All'interno delle due parentesi graffe aperta e chiusa vanno scritte tutte le linee di programma che devono essere eseguite nel momento in cui richiediamo al nostro programma di eseguire la funzione `LanciaIDadi()`. Una funzione deve sempre essere accompagnata da una coppia di parentesi graffe che ne delimitino il contenuto. Alla fine della fiera, un file di un programma scritto in C non è nient'altro che una successione di funzioni, ognuna col suo bravo nome, seguita dal codice ad essa pertinente racchiuso tra parentesi graffe (ci sono alcune cose in più in un programma scritto in C, ma non è ancora il momento di parlarne).

In realtà le parentesi graffe (sempre a coppie aperta-chiusa) non si usano solo per delimitare le funzioni. Sono in realtà generici delimitatori di *blocchi*. Un blocco è una sottounità logica di un programma. Ogni funzione al suo interno può essere suddivisa in più blocchi, ognuno dei quali è delimitato da una coppia di parentesi graffe. Suddividere il codice in blocchi, oltre ad avere una funzione estetica ed organizzativa, può essere indispensabile ad esempio per delimitare un insieme di istruzioni che vanno eseguite più volte in sequenza. Potete scriverle una dietro l'altra per tutte le volte che vi servono, ma potete anche raggrupparle in un blocco e identificare quest'ultimo opportunamente cosicché possa essere eseguito più volte in sequenza. Il vantaggio, oltre al fatto che le linee di programma da eseguire ripetutamente in sequenza le scrivete una volta sola, è che potete anche non sapere a priori quante volte dovrete ripetere in sequenza quel blocco: infatti, il numero di volte potrà essere memorizzato in una variabile! I blocchi hanno anche altre interessanti proprietà, alcune delle quali verranno accennate in questo capitolo. Le altre le scopriremo a partire dal prossimo capitolo, dove abbandoneremo i voli pindarici e inizieremo finalmente a parlare in concreto del linguaggio C.

File di header e file di codice

Vi ho mentito. O per lo meno non vi ho detto tutta la verità. I file di codice, di cui abbiamo parlato finora, non sono gli unici file che troverete in un programma scritto in C. Altrettanto numerosi dei file di codice sono i cosiddetti file di header. Li distinguete gli uni dagli altri perché i file di codice sono in genere identificati dall'estensione .c (.cpp per quelli in C++ e .m

per quelli ObjectiveC), mentre i file di header sono identificati dall'estensione .h.

E che saranno mai i file di header? Anche qui darò una risposta un po' vaga, così, tanto per introdurre l'argomento; poi torneremo a parlare di questa come di molte altre cose nel momento in cui inizieremo a parlare più nei dettagli dei linguaggio C. Dunque, quando scrivete un programma in C, abbiamo detto che questo è organizzato un po' a "scatole cinesi"; ci sono scatole tutte sullo stesso livello, che si chiamano funzioni, al cui interno definite blocchi e variabili. Ma come fanno le varie funzioni ad essere a conoscenza dell'esistenza l'una delle altre? E se per caso voleste definire delle variabili comuni a tutte le funzioni, e non soltanto quelle accessibili solo all'interno della funzione in cui sono definite? E se voleste definire delle costanti? E se voleste…

Qui entrano in gioco di file di header. In essi in genere (ma non è obbligatorio) si scrivono quelle linee di codice necessarie perché tutte le cose elencate nelle righe precedenti possano essere fatte. In particolare: nei file di header si elencano le *variabili globali*, ovvero quelle variabili accessibili dall'interno di qualunque funzione e qualunque blocco, si elencano le funzioni di cui è costituito un programma, cosicché queste si possano *chiamare* l'una con l'altra, ovvero possano essere eseguite da qualunque punto del programma sia necessario eseguirle, si definiscono le costanti che possono fare comodo. Non è necessario creare un file di header per fare questo: potreste includere tutto nel vostro bel file di codice. Ma i file di header hanno un grosso vantaggio, anzi due: innanzitutto il vostro programma potrebbe essere costituito da più di un file, e non avere un file di header comporterebbe dover scrivere all'interno di ogni file di codice tutta questa mole di informazioni comuni a tutti i file del programma (con ovvie ripetizioni, possibilità di errore, scomodità tutte le volte che bisogna fare una modifica che va propagata a tutti i file del programma, e così via); e poi, i file di header contengono tutte le informazioni necessarie per sapere *che cosa fa* un programma *e come farglielo fare* senza aver bisogno di scendere nel dettaglio del codice (cioè senza sapere *come lo fa*). Mi spiego: tipicamente, quando lavorate in C, se il vostro programma inizia ad essere un micropelo complesso, tendete a suddividerlo in unità più semplici, che eseguono un compito limitato e ben preciso, ma lo eseguono bene, in maniera efficiente. In genere, le altre parti del programma non hanno nessun bisogno di sapere esattamente come fa quella parte lì a svolgere così bene il suo compito; a loro basta sapere come fare per farglielo eseguire. È un po' come quando guidate l'auto; non è necessario sapere come funziona il motore a scoppio per andare da qua a lì, basta sapere che il motore è quella cosa che ci permette di andare avanti e per farlo andare più forte bisogna premere l'acceleratore; come questo sia collegato al motore, che cosa comporti nel dettaglio l'averlo premuto, e così via può essere molto interessante, ma non serve per guidare la macchina. Tant'è vero che la tecnica di premere l'acceleratore per andare più veloce ce l'avete con le macchine a benzina e quelle Diesel, con i veicoli elettrici a batterie, a celle a combustibile e a pannelli solari, e così via. In altre parole: il codice C è il dettaglio di come funziona il sistema di propulsione della vostra auto. L'acceleratore è il file di header, la vostra *interfaccia*, quello che vi occorre sapere affinché possiate correttamente interfacciarvi con con il codice C e fargli fare quello per cui è stato scritto.

Messa così, è una cosa molto bella, ma a che cosa serve? Beh, a parte un sacco di belle parole che potrei spendere sulla "riciclabilità" del codice che scrivete, sul fatto che aggiornarne una parte non influirebbe sulle altre, e così via, pensate un po' che cosa succede quando, a lavorare ad un programma, non c'è una persona sola ma c'è un intero staff. Ognuno ha il compito di sviluppare una certa parte del programma; ogni parte deve cooperare con le altre in maniera chiara e unitaria per dare luogo ad un programma facile da usare, il più possibile privo di errori, e che sia utile per l'utenza alla quale verrà proposto. Bene, se non ci fossero i file di header, tutti i programmatori dovrebbero conoscere ogni dettaglio di tutto il programma. Invece, se organizzate bene le cose e create gli opportuni file di header, uno o più per ogni compito specifico che deve essere eseguito dal programma, non dovrete conoscere tutti i dettagli di tutto; vi basterà sapere che le parti di programma che non sono state affidate a voi eseguono correttamente i loro compiti, e per farglieli fare voi dovete interfacciarvi con esse in una certa maniera. Se il giorno dopo quelle parti di programma fossero riscritte da zero, ma senza cambiare il file di header, per voi non cambierebbe nulla. Se fossero riscritte da zero e non ci fosse stata da subito una buona organizzazione (quindi se non aveste avuto i file di header), a voi sarebbe toccato riscrivere da zero anche la vostra parte di programma. Scomodo, invero!

È sorprendente quante cose si riescono a dire su un argomento qualsiasi senza mai scendere nemmeno lontanamente nel concreto, vero? Comunque non temete: a partire dal prossimo capitolo tutti questi discorsi al vento saranno banditi: comincerà il vero "lavoro sporco", quello di scrivere delle linee di codice in C che, volenti o nolenti, dovranno funzionare!

Librerie

Se pensate ad una libreria come a quello che effettivamente è, cioè ad un negozio o ad un mobile nel quale si custodiscono libri (e in essi conoscenza), siete quasi nel giusto. In realtà, quelle che in programmazione si chiamano *librerie*, andrebbero più correttamente chiamate *biblioteche*, perché questo è il significato della parola inglese *library* da cui viene il termine. In ogni caso il senso è quello: una collezione di "cose" (non saranno libri, ovviamente) da cui attingere conoscenza.

Perché mai vi servirebbe tutto ciò? Pensate a quando vi preparate un uovo: che fate? Andate in latteria, comprate delle uova, ne prendete uno e lo cucinate come meglio credete. Ovvio! E se vi dicessi che, per cucinare un uovo, dovreste fare, nell'ordine, le seguenti cose: procurarvi un pezzo di terra, costruirvi una vanga ed una zappa, ararlo, seminarlo a granturco, catturare una gallina allo stato selvatico (ce ne sono tante, vero?), addomesticarla ed allevarla, nutrirla con il granturco che avete coltivato, raccogliere le uova che la gallina vi fa e poi mangiarle (crude, perché mica ancora vi siete costruiti la cucina, e la padella, e mica avete prodotto tutti gli altri ingredienti che vi servono)? Mi direste che sono matto: perché fare tutta questa fatica quando gli esseri umani, da tempo immemorabile, hanno avuto il buon senso di organizzarsi in una società che si suddivite i compiti? Grazie a questa brillante idea, c'è gente che coltiva granturco e gente che compra questo dal coltivatore per allevare galline. Poi c'è gente che compra le uova all'ingrosso dall'allevatore di galline per distribuirle ai vari negozi sotto casa, cosicché voi, quando volete mangiarvi un uovo, andate in latteria e ve lo comprate, senza stare a preoccuparvi di tutto quello che c'è dietro.

Beh, in programmazione le librerie servono a questo: a non farvi re-inventare la ruota tutte le volte. Per quanto innovativa e originale sia l'idea che sta alla base del vostro programma, questo si occuperà certamente di una valanga di operazioni che sono comuni a tutti gli altri programmi esistenti: ad esempio dovrà scrivere messaggi sullo schermo, aprire finestre, disegnare menù, rispondere ai comandi dati dall'utente per mezzo del mouse, leggere e scrivere file su disco, manipolare stringhe di caratteri, e tanto altro ancora. Se ogni volta voi doveste re-inventare ognuna di queste cose passereste la vostra vita a scrivere un solo programma, e il 99.99% di questo sarebbe costituito da linee di codice che non hanno niente a che vedere con il compito specifico che avevate in mente per il vostro programma. Ecco allora che esistono librerie per scrivere caratteri sullo schermo e per accettare input da tastiera; esistono librerie per disegnare finestre e menu, per rispondere ai comandi impartiti dall'utente per mezzo del mouse, per leggere e scrivere file su disco, per accedere alla rete e per far fare qualche cosa più di uno squallido beep all'altoparlante del vostro computer. È comodo, è utile, ed è sicuro. Qualunque programma decidiate di fare, dovrete sempre e comunque appoggiarvi ad una o più librerie che vi aiuteranno nello svolgimento del vostro compito.

È un po' come se qualcuno vi avesse dato a disposizione molti strumenti in più (molte *funzioni* in più) di quelle messe a disposizione dal linguaggio di programmazione che avete scelto di usare. In questo tutorial non ci occuperemo di librerie, ma accenneremo ad esse tutte le volte che le useremo. Esisono libri interi dedicati a descrivere che cosa fanno le decine o centinaia di funzioni disponibili nelle varie librerie che vorreste poter usare. Il C fornisce tutta una serie di librerie standard per l'input/output e per la matematica. Ma poi ogni sistema operativo viene con la sua collezione di librerie (in MacOS X si chiamano framework). Ne riparleremo quando affronteremo nel dettaglio la programmazione di MacOS X.

Scope (è inglese, potremmo tradurlo con "portata", "raggio d'azione")

Introduciamo qui l'ultimo concetto vago e fumoso, prima di passare a discutere nel dettaglio le caratteristiche del linguaggio C. È il concetto dello *scope*. Come già avete visto dal titolo di questo paragrafo, potremmo tradurre questo termine inglese con "portata", "raggio d'azione". Infatti, *scope* indica proprio il campo di validità, il raggio d'azione di un qualche cosa (in genere una variabile) all'interno di un programma scritto in C. Approfondiamo:

abbiamo detto che un programma scritto in C può essere suddiviso in più file di codice, e all'interno di ognuno di questi sussiste una suddivisione in funzioni, ognuna delle quali può a sua volta essere suddivisa in blocchi. Per mantenere le cose semplici parliamo solo di variabili, per il momento. Il loro *scope* è funzione di dove queste vengono definite: esisteranno variabili cosiddette "globali", il cui *scope* sarà l'intero programma, ovvero qualunque funzione in qualunque file del programma potrà leggere e scrivere quella variabile. Esisteranno variabili il cui *scope* sarà il file, ovvero solo le funzioni che appartengono a quel file ma non le altre potranno accedere al contenuto di quelle variabili. Esisteranno variabili cosiddette *locali* il cui *scope* sarà una particolare funzione: solo le istruzioni all'interno di quella funzione lì potranno leggere e scrivere quelle variabili; dall'esterno esse sarà come se non esistessero nemmeno. E poi ci saranno ancora variabili definite all'interno di un blocco situato all'interno di una funzione; il loro *scope* sarà solo quel blocco lì, e le istruzioni all'interno della stessa funzione ma all'esterno di quel blocco non potranno accedere a quelle variabili. È il gioco delle scatole cinesi di cui parlavamo tempo fa.

Perché tutta questa complicazione? Beh, che ci crediate o no, è un fatto di comodità. Fate conto di dover realizzare un programma piuttosto complesso, fatto di un sacco di funzioni (non bisogna mica fare chissà che, basta che ne abbiate una decina, di funzioni, perché il vostro programma sia già tutt'altro che banale). È probabile che all'interno di ognuna di queste funzioni vi servano delle variabili per memorizzare risultati temporanei, roba che non vi serve da nessun'altra parte se non all'interno di quella funzione lì. Se lo *scope* di una variabile fosse sempre tutto il programma (come nei vecchi BASIC, ad esempio), sareste nei guai: oltre alle variabili globali, quelle che effettivamente memorizzano i dati che servono a tutto il vostro programma, vi ritrovereste magari con cinque o sei variabili che servono solo ad una funzione, moltiplicato per il numero di funzioni (magari dieci), e fate in fretta a dover tenere traccia di cinquanta o sessanta variabili il cui *scope* è tutto il programma, rischiando di confondervi, sovrascriverle, utilizzare la variabile sbagliata nella funzione sbagliata, e via dicendo. Se invece potete restringere la zona di validità di una variabile alla sola porzione di codice in cui questa effettivamente vi serve vi semplificate di molto la vita: appena usciti da quella porzione di codice (una funzione o un blocco) la vostra variabile cessa di esistere, è come se non ci fosse mai stata (ci sono delle eccezioni, ovviamente, ma ne parleremo a tempo debito). È comodo, è pulito, e vi permette di modificare, espandere ed aggiornare il codice in maniera molto più semplice che non se tutte le variabili fossero globali e avessero addentellati in ogni parte del programma.

Nei prossimi capitoli parleremo meglio dello *scope* e impareremo ad apprezzarne i pregi e le sottigliezze. Ma ora, basta con i vaniloqui. Dalla prossima pagina si fa sul serio!

3. Tipi, operatori ed espressioni

Tipi di variabile

Pensate ad una variabile come ad un contenitore. Nella vostra cucina avrete sicuramente dei contenitori: quello per il caffè, quello per il sale, quello per lo zucchero, quello per le olive in salamoia e quello per le foglie essiccate di salvia. E anche se in teoria potreste mettere il caffè nel contenitore delle olive in salamoia, preferibilmente non lo fate. Avete vari contenitori, che in base a come sono fatti sono maggiormente indicati per un certo tipo di contenuto piuttosto che per un altro.

Ecco, le variabili sono la stessa cosa: sono contenitori (ad essere variabile è il loro contenuto), il cui *tipo* identifica che cosa potete metterci dentro. Ci saranno così variabili di tipo:

`int`, per memorizzare numeri interi compresi tra -2147483648 e +2147483647;

`long`, per memorizzare numeri interi in un intervallo che potrebbe essere più ampio;

`float`, per memorizzare numeri reali (*in virgola mobile* si dice nel gergo informatico);

`double`, per memorizzare numeri reali con una precisione maggiore rispetto a `float`;

`char`, per memorizzare un singolo carattere.

In aggiunta, il C vi permette di specificare tipi di variabili denominati `short`, che spesso coincide col tipo `int`, `long double`, che può o no coincidere con `double`, e `unsigned`, che si applica a tutti i tipi visti fin'ora e ne elimina il segno (`int` va da -2147483648 a +2147483647, `unsigned int` va da 0 a 4294967295, come vedete l'intervallo di numeri memorizzabili ha la stessa ampiezza, ma togliendo il segno si parte da zero).

Questi sono i *tipi* delle variabili; ma esse sono anche identificate da un *nome*; la linea di codice

```
int  pippo;
```

indica che la variabile il cui nome è `pippo` è di tipo `int`, pertanto può contenere solo numeri interi compresi tra -2147483648 e +2147483647. Notate il punto e virgola alla fine: questa è una regola generale del C (con le dovute eccezioni, naturalmente, se no che gusto c'è?): ogni istruzione va terminata con un punto e virgola. Se non lo mettete, il compilatore vi segnalerà un errore.

Siete pronti? È il momento del vostro primo programma! Con questo esempio cercheremo di capire che cosa sono i tipi di variabili. Allora, aprite il vostro editor di testo preferito (BBEdit Lite o TextEdit vanno benissimo, accertatevi comunque di usare un programma che possa salvare in formato solo testo o .txt) e digitate il seguente programma:

```
#include <stdio.h>

int main(void)
{
     char      carlo;
     int       irene;
     long      luigi;
     float     francesca;
     double    daniela;
     short     salvatore;

     printf("MacOS X\n");
     printf("\n");

     carlo='c';
     irene=-1460;
     luigi=140721;
```

```
    francesca=-3.18;
    daniela=12.774e-65;
    salvatore=19;

    printf("char ha dimensione di %d byte\n",sizeof(char));
    printf("int ha dimensione di %d byte\n",sizeof(int));
    printf("long ha dimensione di %d byte\n",sizeof(long));
    printf("float ha dimensione di %d byte\n",sizeof(float));
    printf("double ha dimensione di %d byte\n",sizeof(double));
    printf("short ha dimensione di %d byte\n",sizeof(short));

    printf("carlo = %c\n",carlo);
    printf("irene = %d\n",irene);
    printf("luigi = %d\n",luigi);
    printf("francesca = %g\n",francesca);
    printf("daniela = %g\n",daniela);
    printf("salvatore = %d\n",salvatore);

    return(0);
}
```

Registratelo dove volete (ad esempio in una cartella che creerete apposta per gli esempi di questo tutorial, e dategli il nome Esempio1.c. Fate molta attenzione a scrivere tutto così come lo vedete nel listato e prestate attenzione alle virgolette (semplici e doppie): che siano quelle dritte ("), non quelle curve (“”)!

Compilate il codice ed eseguitelo. Come? Non sapete come si fa? Allora è il momento del nostro primo *box di approfondimento*:

Compilare ed eseguire programmi C

Aprite l’applicazione Terminal, la trovate nella cartella Utilities dentro la cartella Applicazioni. Vi comparirà una scritta del tipo:

`[computer ~] utente%`

dove al posto di `computer` ci sarà il nome del vostro computer e al posto di `utente` ci sarà il nome dell’utente col quale avete fatto il login (probabilmente il vostro nome o cognome). Il segno “~” indica dove vi trovate nell’albero delle cartelle sull’hard disk, ovvero nella vostra home folder (/Users/vostronome/).

Andate nella cartella in cui avete salvato Esempio1.c: assumendo che abbiate creato una cartella EsempiC all’interno della vostra home folder, digitate:

`cd EsempiC`

e premete invio oppure return. La scritta diventerà allora

`[computer ~/EsempiC] utente%`

Digitate

`ls`

per vedere che cosa c’è all’interno della cartella. Il solo file Esempio1.c dovrebbe essere visibile.

Compilatelo, ovvero trasformatelo in un qualche cosa di eseguibile dal computer, digitando:

`gcc Esempio1.c`

e premete invio o return alla fine. Se non avete fatto errori, dopo pochi istanti il compilatore avrà finito. Digitate

`ls`

e vedrete che, oltre al file Esempio1.c che contiene il codice sorgente in C del vostro programma, ora c’è anche un altro file, a.out, che contiene la versione eseguibile del vostro

programma. Di già che ci siete, digitate

```
mv a.out Esempio1
```

per rinominare il file a.out in Esempio1 (che forse è più intuitivo). Ora eseguite il vostro programma digitando:

```
./Esempio1
```

e, come al solito, premendo invio o return alla fine. Che cosa succede a questo punto lo vedrete nel testo principale. Questa, comunque, è la procedura da seguire per compilare ed eseguire tutti gli esempi contenuti in questo tutorial (di volta in volta, naturalmente, cambieranno i nomi dei file o delle cartelle, ma questo l'avevate già capito!)

Se avete fatto tutto correttamente, vi apparirà un output nella vostra finestra del terminale di questo tipo:

```
MacOS X

char ha dimensione di 1 byte
int ha dimensione di 4 byte
long ha dimensione di 4 byte
float ha dimensione di 4 byte
double ha dimensione di 8 byte
short ha dimensione di 2 byte
carlo = c
irene = -1460
luigi = 140721
francesca = -3.18
daniela = 1.2774e-64
salvatore = 19
```

Cerchiamo di capire che cosa abbiamo fatto.

Il programma inizia con l'istruzione `#include <stdio.h>`; ricordate quando parlavamo di librerie? Beh, `stdio.h` è un *file di header* che si trova già da qualche parte sul vostro hard disk e che a sua volta si occupa di rendere disponibili al vostro programma tutta una serie di funzioni definite nella libreria di nome `stdio`. Questo nome astruso non vuol dire altro che "standard input/output" ed è una libreria standard del C che vi permette di fare input/output di testo dalle finestre del terminale. Qui ci serve per poter scrivere tutte quelle belle cose del tipo `char ha dimensione di 1 byte` sulla finestra del terminale. Il comando `#include` ci permette di includere tutti i file di header che ci servono. Qui ci serve solo questo. Lo analizzeremo meglio più avanti. Notate che questo comando non ha bisogno di un ; alla fine.

Subito dopo c'è una riga vuota. Questa è una lezione importante: in C potete inserire tutte le righe vuote, gli spazi e i tabulatori che volete tra un'istruzione e l'altra e tra un comando e l'altro all'interno della stessa istruzione. Non importa quanti ne mettete. Quindi, usateli per dare al codice del vostro programma un aspetto pulito e leggibile, come nel caso dell'Esempio1.

Subito dopo, la riga `int main(void)` indica che il blocco che segue, raggruppato tra due parentesi graffe, è una funzione di nome `main()`, che non accetta nessun tipo di argomento (`void`), e restituisce un numero di tipo `int`. Non preoccupiamoci per ora di che cosa voglia dire, parleremo di funzioni nel capitolo 5. Qui ci basti sapere che *ogni* programma scritto in C deve *sempre* avere una funzione di nome `main()` che viene eseguita per prima quando lanciate il programma. Ovvero, quando da terminale, dopo aver compilato il file Esempio1.c e aver rinominato il file a.out in Esempio1, avete eseguito il vostro programma col comando ./Esempio1, il computer, come prima cosa, ha cercato nel vostro programma una funzione di nome `main()` e ha iniziato ad eseguire il programma dalla prima riga di codice all'interno della funzione `main()`.

Le sei righe che seguono la parentesi graffa aperta costituiscono la cosiddetta *dichiarazione* delle variabili locali all'interno della funzione `main()`. Come vedete la sintassi è quella che abbiamo descritto all'inizio di questo capitolo. Tutte le variabili che intendete usare all'interno di una funzione vanno *dichiarate* all'inizio della funzione stessa, ovvero dovete dichiarare ufficialmente che *tipo* di variabili volete usare e quale *nome* volete loro assegnare. Nel nostro programma abbiamo deciso di usare una variabile di nome `irene` e di tipo `int`, ad esempio. Qualche nota estetica: per i nomi delle variabili potete usare qualunque sequenza di lettere e numeri che inizi con una lettera; potete usare il carattere "underscore" _; potete usare lettere maiuscole e minuscole. Non potete usare punti, virgole, simboli strani, parentesi o lettere accentate. E, badate bene, lettere maiuscole e lettere minuscole *sono diverse*: la variabile di nome `Irene` sarebbe diversa dalla variabile di nome `irene`. Abbiamo scelto, come consuetudine, di identificare le variabili con dei nomi che iniziano con una lettera minuscola. In genere le funzioni, ad eccezione della funzione `main()`, si identificano con nomi che iniziano con una lettera maiuscola. Siccome qui le nostre sei variabili fanno abbastanza poco, non c'erano nomi furbi che identificassero in maniera chiara ed intuitiva lo scopo per cui uno le ha create ed intende usarle; e allora, come i più attenti di voi avranno notato, ho usato il trucchetto di dare alle variabili dei nomi di persona che abbiano la stessa iniziale del *tipo* a cui appartengono! Così, per sfizio.

A seguire ci sono due righe con l'istruzione `printf()`; essa verrà descritta sommariamente nel capitolo 8. Qui la introduciamo solo brevemente. Essa non è un comando del C, bensì fa parte della libreria `stdio`, quella che abbiamo detto di voler usare quando abbiamo usato il comando `#include` all'inizio del programma. Senza quell'`#include`, non saremmo riusciti a compilare il programma (il comando `gcc` digitato nel terminale avrebbe riportato un errore). L'istruzione, o meglio la *funzione* `printf()` è un realtà un sistema per scrivere del testo sulla finestra del terminale. Come potete vedere, infatti, ciò che è racchiuso tra virgolette nelle varie funzioni `printf()` ve lo siete ritrovato anche nell'output del programma. Il simbolo `\n` identifica il carattere "new line", e indica che al termine della scritta tra virgolette bisogna andare a capo.

A seguire ci sono sei istruzioni di *assegnazione*; ne parleremo ancora in questo capitolo, ma il loro significato è abbastanza chiaro: a sinistra del segno di uguaglianza mettete il nome della variabile, e a destra mettete il valore che volete memorizzare nella variabile stessa. Notate che per le variabili intere abbiamo usato numeri interi, per quelle reali numeri reali e per la variabile di tipo `char` abbiamo usato una singola lettera racchiusa tra virgolette singole.

Le sei istruzioni `printf()` successive hanno lo scopo di scrivere quanti byte di memoria occupa ciascuna variabile di un certo tipo. Ad esempio, l'output del programma ha riportato che le variabili ti tipo `int` occupato 4 byte di memoria. 4 byte sono 32 bit, perciò in una variabile di tipo `int` si possono memorizzare numeri nell'intervallo da $(-2^{32})/2$ a $(+2^{32})/2-1$, cioè da -2147483648 a +2147483647, e in una variabile di tipo `unsigned int` numeri da 0 a $2^{32}-1$, ovvero da 0 a 4294967295. Come vedete, in MacOS X `int` e `long` sono la stessa cosa, `short` va da -32768 a +32767, `unsigned short` da 0 a 65535, `float` richiede 4 byte e `double` è a *doppia precisione*, perché richiede 8 byte.

Le successive sei istruzioni `printf()` scrivono sull'output il contenuto delle sei variabili che abbiamo usato nel nostro programma. E, per finire, l'istruzione `return(0)` al termine della funzione `main()` avverte il computer che il programma è terminato.

Beh, per essere il nostro primo programma non c'è male. Compiti per casa: modificate il programma Esempio1.c, assegando alle sei variabili dei valori diversi. Provate a vedere che cosa succede se assegnate alle variabili intere dei numeri decimali, provate a vedere che cosa succede se assegnate dei valori troppo grandi o troppo piccoli (ad esempio il valore 100000 alla variabile `salvatore`), e così via. Pacioccate un po', ogni volta ricompilate il file Esempio1.c, rinominate a.out in Esempio1 ed eseguite il programma col comando ./Esempio1. Buon divertimento!

Costanti

Ci sono delle volte che una variabile non vi serve, in quanto il valore che volete memorizzare non è destinato a cambiare mai nel corso del programma. Si tratta, per l'appunto,

di una *costante*. In teoria, potreste benissimo definire una variabile ed assegnarle il valore che vi interessa e poi non toccarla mai, ma limitarvi solo a leggerla. Sarebbe una specie di costante. Lo svantaggio (a parte questioni subdole di tempi di compilazione ed esecuzione del programma) è che per sbaglio da qualche parte nel codice vi sbagliate e modifichiate il valore della variabile (che così non sarebbe più costante): vi immaginate il guaio?

Ma il C, dicevamo, non è roba da dilettanti: e allora vi mette a disposizione una bellissima maniera di definire delle vere costanti, roba che se cercate di modificarne il valore il compilatore vi copre di insulti. Vediamo come si fa.

Aprite il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

// qui ci sono le costanti
#define   kNumeroDiDitaDiUnaMano          5
#define   kNumeroDiMani                   2

int main(void)
{
     int           numeroDiDita;

     printf("Numero di dita delle mani: ");
     numeroDiDita=kNumeroDiDitaDiUnaMano*kNumeroDiMani;
     printf("%d\n",numeroDiDita);
     return(0);
}
```

Salvate il file come Esempio2.c, quindi compilatelo ed eseguitelo. L'output del programma è assolutamente ovvio. Vediamo ora come funziona.

Si comincia col solito `#include <stdio.h>` perché vogliamo scrivere del testo sulla finestra del terminale e quindi ci serve la libreria di input/output. Poi c'è una riga che incomincia con due sbarre //. Righe che incominciano in questa maniera sono *commenti*, ciò che segue è di utilità per il programmatore ma non viene né compilato né eseguito. Potete inserire quanti commenti volete nel vostro programma. Essi possono essere ad inizio riga, come in questo caso, oppure al termine di una riga di programma (in quest'ultimo caso tutto ciò che c'è dalle due sbarre fino all'a-capo al termine della riga verrà ignorato). I commenti possono anche essere fatti con la sequenza di caratteri `/*` e allora in questo caso il commento diventa multi-riga, ovvero prosegue anche dopo che siete andati a capo. Potete interrompere l'effetto di questo modo di fare commenti digitando la sequenza inversa `*/`, che annulla l'effetto della prima. Tutto ciò che c'è tra `/*` e `*/` è un commento e verrà ignorato dal compilatore.

E veniamo alle costanti: le istruzioni `#define` sono caratterizzate da una sintassi molto semplice: la prima sequenza alfanumerica che segue l'istruzione è il *nome* della costante (come il nome delle variabili), la seconda è il *valore* immutabile che questa avrà per tutto il programma. I nomi delle costanti possono essere qualunque sequenza alfanumerica compreso l'underscore, devono iniziare con una lettera e lettere maiuscole e minuscole sono diverse (come per le variabili). Le costanti possono essere numeriche, come nel nostro esempio, oppure sequenze di caratteri (in questo caso vanno racchiusi tra virgolette, ci torneremo tra poco). Esistono varie scuole di pensiero su come si scrivano i nomi delle costanti: molti suggeriscono di scriverli tutti in maiuscolo. Io personalmente trovo questa pratica scomoda sia in scrittura (odio il tasto shift e ancora di più il caps-lock) sia in lettura (in codice sorgente pieno di parole tutte in maiuscolo è di difficile lettura). Preferisco far iniziare i nomi delle costanti con una k minuscola, così, giusto per ricordarmi, ovunque sia, che trattasi per l'appunto di costanti.

All'interno della solita funzione `main()` abbiamo un'interessante novità: oltre alle due funzioni`printf()`, c'è una linea in cui viene fatta una moltiplicazione tra le due costanti: abbiamo anticipato concetti (banali, come vedete) coperti nei prossimi paragrafi, ovvero operatori ed espressioni. Qui ci serviva questa linea di codice per far vedere come si possono utilizzare

delle costanti all'interno di un programma.

Le costanti di tipo intero non sono le uniche che possono essere utilizzate. Vediamo qui di seguito un esempio un po' più articolato:

```
#include <stdio.h>

// qui ci sono le costanti
#define   kNumeroDiDitaDiUnaMano      5
#define   kNumeroDiMani               2
#define   kSpecie                     "Homo sapiens sapiens"
#define   kDitaTotali                 kNumeroDiDitaDiUnaMano*kNumeroDiMani
#define   kAltezzaMedia               1.75

int main(void)
{
     printf("La specie %s ha di norma %d mani ",kSpecie,kNumeroDiMani);
     printf("con %d dita cadauna.\n",kNumeroDiDitaDiUnaMano);
     printf("Questo da' un totale di %d dita.\n",kDitaTotali);
     printf("\n");
     printf("Inoltre, l'altezza media e' di %g metri.\n",kAltezzaMedia);
     return(0);
}
```

Salvatelo come Esempio3.c, compilatelo ed eseguitelo. L'output è nuovamente scontato.

Vediamo il codice più nel dettaglio: in aggiunta rispetto all'Esempio2 c'è, ad esempio, la costante `kSpecie`: essa è una costante *stringa*, ovvero una sequenza di caratteri, delimitata dalle virgolette (dritte, non curve!), che *non* fanno parte della stringa stessa. Poi, la costante `kDitaTotali` è definita come il prodotto di altre due costanti. Questo è interessante: una costante può essere funzione di altre costanti, purché queste siano già state definite, ovvero l'istruzione `#define` che le definisce deve essere scritta prima. L'ultima costante è un numero decimale ti dipo `double`.

All'interno della funzione `main()` c'è il solito uso della funzione `printf()`, per i cui dettagli vi rimando al capitolo 8. Il programma non fa nient'altro che usare queste costanti per mostrare interessanti dati sulla specie umana. Perché abbiamo fatto questo esempio così banale? Beh, innanzitutto perché non abbiamo ancora fatto molto, del C, per cui tanto do più non possiamo fare. E poi per far vedere una cosa importante: definire delle costanti è di importanza fondamentale. Nessuno, infatti, vi avrebbe vietato di inserire il nome della specie, il numero di dita, il numero di mani e l'altezza media direttamente all'interno del codice del programma. Però questo avrebbe comportato vari problemi. Il primo si presenta quando, alcuni mesi dopo che avete fatto il programma, ne riprendete in mano il listato per modificarlo; vi garantisco che non vi ricorderete più niente, e vedere scritto da qualche parte il numero 5 non vi avrebbe di certo aiutato a ricordare; invece, la costante `kNumeroDiDitaDiUnaMano` per lo meno ha un nome che dice da solo di che cosa si tratta. Il secondo problema viene nel momento in cui volete adattare il programma ad una mutazione genetica che è avvenuta nella specie umana, che da un certo momento in avanti ha 3 mani anziché due. Il vostro programma potrebbe essere ben più complesso di questo, fatto magari di migliaia e migliaia di linee di codice, diviso magari su più file, redatti magari da più programmatori diversi. Andare a rintracciare tutti i 2 (il numero di mani) all'interno del codice, stabilire se rappresentino il numero di mani oppure qualche cos'altro, e in caso affermativo sostituirli con dei 3 è un lavoro allucinante e che porta a commettere un sacco di errori. Se invece ci si riferisce al numero di mani in maniera simbolica attraverso una costante, allora è sufficiente cambiare il valore di questa là dove è definita per far sì che la modifica si ripercuota senza errori né dimenticanze per tutto il programma. È una questione di un attimo, e fa risparmiare dei mal di testa che vi raccomando!

Operatori

Abbiamo già visto degli operatori negli esempi precedenti, qui elenchiamo tutti i principali. Abbiamo operatori aritmetici:

`+` per effettuare somme,
`-` per effettuare sottrazioni,
`*` per effettuare moltiplicazioni,
`/` per effettuare divisioni,
`%` per determinare il resto di una divisione tra numeri interi.

Vi faccio notare l'assenza di un operatore per l'elevamento a potenza. Da notare che l'operatore `/`, se opera tra variabili intere (tipi `int` e `long`), tronca il risultato ad un numero intero (quindi vi perdete la parte decimale). Se invece coinvolge almeno un numero decimale (tipi `float` e `double`), il risultato è un numero decimale. Questo è un caso particolare di quelle che vengono chiamate *regole di promozione*, che discutiamo qui di seguito nel nostro secondo *box di approfondimento*:

Regole di promozione

Gli operatori, ad esempio quelli aritmetici appena visti, operano (guarda un po') con *due* grandezze, siano esse costanti o variabili. Esse non devono essere necessariamente *omogenee*, purché siano *simili*. Che cosa vuol dire? Supponiamo di avere due variabili, che chiameremo `x` e `y`, e supponiamo di voler calcolare il loro quoziente `x / y` (con `y` diverso da zero). È ovvio che entrambe le variabili (ma una o entrambe potrebbero essere delle costanti!) devono rappresentare dei numeri. Tuttavia, non è necessario che siano entrambe dello stesso *tipo*. È sufficiente che ciascuna delle due sia di tipo numerico (`int`, `long`, `float` o `double`, eventualmente `unsigned`). Se sono entrambe dello stesso tipo, il risultato dell'operazione (una divisione nel nostro caso) sarà dello stesso tipo: così una divisione tra due variabili `double` sarà un numero di tipo `double`, una divisione tra variabili di tipo `int` darà un risultato di tipo `int` (e pertanto il numero risultante sarà troncato alla parte intera, scartando l'eventuale parte decimale). Se invece le due variabili in questione sono di tipo diverso, una delle due viene convertita automaticamente nel tipo dell'altra e il risultato sarà dello stesso tipo. Come avviene questo? La variabile più *debole* viene convertita nel tipo della variabile più *forte*. In particolare:

`double` è più forte di `float` che è più forte di `long` che è più forte di `int` che è più forte di `short`. Una divisione tra una variabile `int` e una variabile `double` comporterà che la variabile `int` sia *promossa* a `double` e il risultato sarà di tipo `double`. La regola di promozione verso il tipo più forte è vera sempre e viene applicata tutte le volte che un operatore, non necessariamente di tipo aritmetico, opera su variabili di tipo diverso.

Vediamo un esempio: aprite il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

int main(void)
{
    int           intero1,intero2;
    int           risultatoIntero;
    double        decimale1,decimale2;
    double        risultatoDecimale;
    double        risultatoDecimaleForzato1,risultatoDecimaleForzato2;
    double        risultatoDecimaleMisto;
```

```
    intero1=5;
    intero2=2;
    decimale1=5.0;
    decimale2=2.0;

    risultatoIntero=intero1/intero2;
    risultatoDecimale=decimale1/decimale2;
    risultatoDecimaleForzato1=intero1/intero2;
    risultatoDecimaleForzato2=(double)intero1/intero2;
    risultatoDecimaleMisto=intero1/decimale2;

    printf("Risultato intero = %d\n",risultatoIntero);
    printf("Risultato decimale = %g\n",risultatoDecimale);
    printf("Risultato decimale forzato di tipo 1 =
%g\n",risultatoDecimaleForzato1);
    printf("Risultato decimale forzato di tipo 2 =
%g\n",risultatoDecimaleForzato2);
    printf("Risultato decimale misto = %g\n",risultatoDecimaleMisto);
    return(0);
}
```

Salvatelo come Esempio4.c, compilatelo ed eseguitelo. L'output che vi verrà mostrato sarà il seguente:

```
Risultato intero = 2
Risultato decimale = 2.5
Risultato decimale forzato di tipo 1 = 2
Risultato decimale forzato di tipo 2 = 2.5
Risultato decimale misto = 2.5
```

Che cosa è successo? Abbiamo definito due variabili intere e due variabili decimali, allo scopo di dividere la prima per la seconda. Abbiamo scelto intenzionalmente due numeri che non siano divisibili l'uno per l'altro (5 e 2) così da far vedere come funzionano i troncamenti e le regole di promozione. Quando dividete i due numeri interi e scrivete il risultato in un'altra variabile intera `risultatoIntero`, ciò che ottenete è un numero intero che corrisponde al risultato vero della divisione troncato (5/2 = 2.5 che, troncato, fa 2). Quando invece dividete tra di loro due numeri decimali (notate che li abbiamo definiti come 2.0 e 5.0 per far capire al compilatore che sono proprio numeri decimali e non numeri interi), il risultato non viene troncato ed è pari a 2.5.

Molto interessante è quello che capita con le successive due operazioni: nel primo caso effettuate la divisione tra interi e forzate il risultato all'interno di una variabile decimale (`risultatoDecimaleForzato1`, che è di tipo `double`). Tuttavia, poiché la divisione coinvolge interi, il risultato sarà un intero, ovvero troncato a 2, e solo successivamente questo valore verrà promosso a 2.0 per essere memorizzato nella variabile di tipo `double` `risultatoDecimaleForzato1`. Nel secondo caso, invece, anche se state facendo una divisione tra interi, aggiungete l'espressione `(double)`: state facendo quello che si chiama un *typecasting*, ovvero una conversione di tipo; avete due variabili intere, le state dividendo l'una per l'altra, ma obbligate il programma a darvi un risultato di tipo `double`, affinché possiate memorizzarlo nella variabile di tipo `double` chiamata `risultatoDecimaleForzato2`, che infatti vale 2.5 (ricordate quando deliravamo sul concetto di linguaggio "strongly-typed"? Beh, ci siamo!)

Interessante è anche l'ultimo caso: dividete una variabile di tipo `int` per una di tipo `double`. I tipi non sono omogenei, così quella di tipo `int` viene promossa a tipo `double` e il risultato è già di tipo `double`, ovvero pari a 2.5 nel nostro esempio.

Giocate con l'Esempio4 inserendo numeri diversi, inventandovi altre operazioni e verificando che cosa succede quando avete a che fare con risultati negativi: le regole di troncamento potrebbero cambiare!

Gli operatori aritmetici, dicevamo, non sono gli unici. Esistono anche gli operatori logici, quelli di incremento e decremento, quelli che operano sui bit e quelli di assegnazione. Vediamoli brevemente.

Gli operatori logici sono quelli che stabiliscono l'uguaglianza o diversità tra due grandezze (variabili, costanti o espressioni, vedi oltre). Essi sono:

== per stabilire se ciò che c'è a destra è uguale a ciò che c'è a sinistra,
!= per stabilire se ciò che c'è a destra è diverso da ciò che c'è a sinistra,
> e >= per stabilire se ciò che c'è a sinistra è maggiore (o uguale) a ciò che c'è a destra,
< e <= per stabilire se ciò che c'è a sinistra è minore (o uguale) a ciò che c'è a destra,
&& AND logico,
|| OR logico,
e qualche altro operatore che qui trascuriamo.
Vedremo gli operatori logici in azione nel prossimo capitolo.

Gli operatori di incremento e decremento sono solo delle scorciatoie: se avete la variabile numerica `i` e volete incrementarla di una unità, potete scrivere `i = i+1;`

Altolà, fermi tutti! Da quel po' di algebretta che ricordo, questa equazione è una follia!!! Infatti non è un'equazione. Il segno di uguale è un *operatore di assegnazione*, ovvero state dicendo che volete scrivere come contenuto della variabile `i` quello che c'è a destra dell'uguale; caso vuole che a destra dell'uguale ci sia un'espressione che coinvolge la variabile `i` stessa; ciò che succede allora è che il *nuovo* valore della variabile `i` è pari al *vecchio* valore della variabile `i` *più 1*. Naturalmente, al posto del `+1` potreste avere qualunque altra cosa. Allora, nel caso qui descritto avete una scorciatoia: potete scrivere `i++;` oppure `++i;` e il contenuto della variabile `i` sarà incrementato di una unità. Le due espressioni non sono equivalenti, come vedremo nel prossimo esempio. Un giochetto analogo lo potete fare per decrementare di una unità una variabile, scrivendo `i--;` oppure `--i;`

Infine avete un'altra scorciatoia: se alla variabile `i` volete aggiungere 5, anziché scrivere `i=i+5;` potete scrivere `i+=5;` L'espressione *operatore aritmetico* seguito immediatamente (cioè senza spazi in mezzo) dall'*operatore di assegnazione* (l'uguale) indica che la variabile a sinistra viene modificata dall'operatore aritmetico in ragione di quanto c'è a destra. Confusi? Chiaritevi le idee con l'Esempio5:

```
#include <stdio.h>

int main(void)
{
    int           i;
    int           n;

    i=4;
    n=i;
    printf("Valore di n all'inizio: %d\n",n);
    n=i++;
    printf("Valore di n dopo i++: %d\n",n);
    n=++i;
    printf("Valore di n dopo un ulteriore ++i: %d\n",n);

    n+=i*(n-4);
    printf("Valore finale di n: %d\n",n);
```

```
    return(0);
}
```

Salvatelo come Esempio5.c, compilatelo ed eseguitelo. Otterrete il seguente output:

```
Valore di n all'inizio: 4
Valore di n dopo i++: 4
Valore di n dopo un ulteriore ++i: 6
Valore finale di n: 18
```

Che cosa è successo? La variabile `n` è inizialmente uguale ad `i`, ovvero vale 4. Poi, `n=i++;` ma siccome l'operatore `++` *segue* il nome della variabile, *prima* il valore di `i` viene memorizzato in `n`, *poi* il valore di `i` viene incrementato. Ecco così che `n` vale ancora 4, mentre `i` vale 5. Successivamente`n=++i;` in questo caso l'operatore `++` *precede* il nome della variabile, quindi *prima* il valore di `i` viene incrementato (era 4, era stato incrementato a 5 dall'istruzione `i++`, ora diventa pari a 6) e *poi* il risultato viene memorizzato in `n` (che quindi vale 6). Infine, l'espressione `i*(n-4)` ovvero 6*(6-4)=12 viene *sommata* al *precedente* valore di `n` (per via dell'operatore `+=`), che era 6, quindi 12+6=18.

Provate a modificare l'Esempio5 usando l'operatore di decremento ed operatori come `-=`, `*=` e `/=` e vedete un po' che cosa succede.

L'operatore di assegnazione `=` l'abbiamo già usato più volte e non mi soffermerò: il risultato di quello che c'è a destra dell'uguale viene memorizzato nella variabile che c'è a sinistra.

Infine ci sono gli operatori sui bit:
`&` AND
`|` OR
`^` OR esclusivo
`<<` sposta a sinistra
`>>` sposta a destra
`~` complemento ad uno
Non ci occuperemo qui degli operatori sui bit, li ho citati solo per completezza.

Infine, non dimentichiamoci degli operatori `+` e `-` che *precedono* un numero od una variabile: essi indicano di *preservare* o *invertire* il valore di quel numero o di quella variabile. Così, se la variabile `n` vale -4, `+n` vale -4 e `-n` vale +4.

Espressioni

Beh, a dire la verità di *espressioni* ce ne siamo già occupati, per lo meno di quelle aritmetiche ed algebriche. Tutte le volte che avete un operatore di assegnazione, infatti (nome variabile, seguita dal segno uguale), ciò che sta a destra dell'uguale è, tecnicamente, un'espressione. Essa è composta da variabili, costanti, funzioni che operano su di esse, operatori aritmetici e così via. In realtà, come vedremo quando discuteremo delle espressioni *logiche* nel prossimo capitolo, la presenza di una variabile e dell'operatore di assegnazione non è affatto necessaria (perché il C può fare la cosiddetta *assegnazione implicita a target nullo*, vedremo di che cosa si tratta).

Sono espressioni anche quelle che possono comparire nelle definizioni delle costanti: ad esempio quando nell'Esempio3 abbiamo definito una costante come il prodotto di altre due costanti (ecco qui un'espressione), quelle che possono comparire persino ad argomento di funzioni (lo vedremo meglio nel capitolo 5). In generale, l'espressione più semplice è costituita da un numero o da una variabile o da una costante, posta a destra dell'operatore di assegnazione, o anche in assenza di questo (e della rispettiva variabile in cui si va a scrivere il risultato dell'espressione a destra dell'uguale).

Seguendo un ordine logico avrei dovuto parlare prima delle espressioni, ma mi sembrava un concetto un po' troppo astratto. Così, a cose già fatte, credo che lo possiate apprezzare con maggiore tranquillità.

4. Blocchi

In questo capitolo, come dice il titolo, ci occuperemo di *blocchi*. Ce ne siamo già occupati saltuariamente in passato, ora li affronteremo in maniera più completa e sistematica. L'unica eccezione che faremo sarà per le *funzioni*, un tipo particolare di blocchi che tratteremo a parte nel capitolo 5.

Un *blocco* in generale è qualunque porzione di codice compreso tra una coppia di parentesi graffe (la prima aperta, la seconda chiusa). Un blocco può contenere al suo interno altri blocchi, come le scatole cinesi o le matrioske russe, purché il blocco più interno venga chiuso dalla sua parentesi graffa chiusa *prima* che venga chiuso il blocco che lo contiene. A differenza del Pascal, che è un linguaggio completamente strutturato, il C impone però delle limitazioni nell'incapsulamento dei blocchi, la più evidente delle quali è che all'interno di un blocco non possono venire definite funzioni o costanti, ma solo variabili e altri tipi di blocchi. Detto in altre parole, i blocchi del C sono permeabili alle funzioni e alle costanti, quelli del Pascal no. Questo è più un limite concettuale che pratico, anzi, questo rende il C un linguaggio meno macchinoso e più snello (e quindi più versatile) rispetto al Pascal.

Blocchi senza nome

I *blocchi senza nome* sono quelli contenuti semplicemente tra due parentesi graffe, la prima aperta, la seconda chiusa. Anche se sono usati raramente, essi possono tornare utili per organizzare meglio il codice all'interno di una funzione particolarmente complessa ed articolata, oppure per definire al loro interno delle variabili locali che non abbiano per *scope* l'intera funzione all'interno della quale si trova il blocco. Forse la maniera migliore per capire di che cosa stiamo parlando è un esempio. Prendete il vostro editor di testo preferito e digitate il seguente codice sorgente:

```
#include <stdio.h>

int main(void)
{
     double             variabileDelMain;

     variabileDelMain=1.42;

     {
          // un blocco senza nome all'interno di una funzione
          int       variabileDelBlocco;

          variabileDelBlocco=-4;

          printf("Dall'interno del blocco si possono usare\n");
          printf("tutte le variabili esterne, come\n");
          printf("variabileDelMain = %g\n",variabileDelMain);
          printf("e tutte quelle interne al blocco, come\n");
          printf("variabileDelBlocco = %d\n",variabileDelBlocco);
     }
     // fuori dal blocco la variabile variabileDelBlocco non è definita
     printf("Fuori dal blocco, solo la variabile\n");
     printf("variabileDelMain = %g\n",variabileDelMain);
     printf("e' definita, perche' appartiene allo stesso blocco\n");
     printf("della funzione. Invece, variabileDelBlocco,\n");
     printf("che appartiene al blocco senza nome interno,\n");
     printf("non e' definita.\n");
```

```
     return(0);
}
```

Salvatelo come Esempio6.c, compilatelo ed eseguitelo. Questo programma non fa niente di particolarmente utile o intelligente, se non farci osservare nella pratica il concetto di *scope* introdotto nel capitolo 2. Abbiamo una funzione `main()`, all'interno della quale è definita una variabile di nome `variabileDelMain`. Questa variabile, essendo definita all'inizio del blocco della funzione, è utilizzabile all'interno di tutta la funzione `main()`. Tuttavia, all'interno della funzione, c'è anche un blocco senza nome, all'inizio del quale è definita una variabile di nome `variabileDelBlocco`; anche questa è utilizzabile *solo all'interno del blocco in cui è definita*, e pertanto non può essere usata al di fuori di esso. Se tra le funzioni `printf()` che concludono il programma ne avessimo messa una che cercava di visualizzare il contenuto della variabile `variabileDelBlocco`, il compilatore non ci avrebbe lasciato compilare il programma (e quindi non avremmo avuto un file da eseguire). Capite? Un blocco, sia esso senza nome o dotato di nome (come le funzioni e quelli che vedremo nel resto del capitolo), è una specie di sottoprogramma, che può utilizzare tutte le variabili definite nei blocchi che lo contengono, nonché tutte quelle definite al suo interno; però non può usare le variabili definite nei blocchi interni ad esso. Un esempio più articolato ci permetterà di illustrare meglio questo concetto. Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

int main(void)
{
     int       alice;

     alice=4;
     {
          // primo blocco senza nome all'interno di main()
          int       bruno;

          bruno=6-alice; // OK, alice qui è definita
     }
     alice=-2;
     // bruno = 5; questa istruzione è commentata, perché illecita
     {
          // secondo blocco senza nome all'interno di main()
          int       carlo;

          carlo=alice*3; // OK, alice qui è definita
          // bruno=carlo/2; questa istruzione è commentata perché illecita
     }
     // carlo = 2; questa istruzione è commentata perché illecita
     return(0);
}
```

Salvatelo come Esempio7.c, compilatelo ed eseguitelo. Come dite? Non è successo niente? Meno male! È esattamente quanto ci aspettavamo. Questo programma non ha nessun altro scopo se non farvi vedere meglio come funziona il concetto di *scope*, ovvero dove sono definite e dove no le variabili locali. Come vedete, `alice` è definita all'inizio della funzione `main()`, e pertanto è utilizzabile anche all'interno di entrambi i blocchi senza nome al suo interno. Viceversa, `bruno` e `carlo` sono utilizzabili *solamente* all'interno dei blocchi in cui sono rispettivamente definite. Se provate a togliere il commento (le due sbarre `//`) dalle istruzioni dichiarate come illecite scoprirete che non sarete più in grado di compilare il programma.

Questa è una delle caratteristiche migliori e peggiori del C: tutte le volte che avete bisogno di una variabile, fosse anche per una cosa banale e veloce, dovete definirla all'inizio del blocco *più esterno* nel quale la volete usare. Così, una variabile che volete usare per tutto il programma dovrete definrirla *all'esterno di qualunque funzione*, una variabile che vi serve all'interno di tutta una funzione dovete definirla *all'inizio di questa*, e così via. Un blocco senza nome può servire quando avete a che fare con una funzione piuttosto complicata e non volete affollare troppo l'elenco delle variabili definite al suo interno. Creare un blocco, anche senza nome, può servire per spezzettare un po' il codice e renderlo più facile da maneggiare (e quindi meno soggetto ad errori di programmazione). Questa è una bella cosa, perché il C vi avverte tutte le volte che, per errore, cercate di usare una variabile in un contesto in cui non è definita; ma è anche una barba, perché siete sempre e comunque costretti a definire tutte (e ripeto, tutte) le variabili che intendete usare, e dovete definirle nel posto giusto. Questa è una cosa che sconcerta coloro che provengono da linguaggi di programmazione come BASIC, Perl, PHP ed altri, dove questa restrizione non è imposta. Obiettivamente sembra una perdita di tempo; ma quando il vostro programma si allunga, ringrazierete il cielo che il compilatore vi protegga dall'errore di usare una variabile che non avete definito: pensate che basta un banale errore di battitura per sbagliare a scrivere il nome di una variabile; se il compilatore del C non facesse questo controllo, una variabile dal nome scritto male (e quindi non corrispondente a nessuna definizione) verrebbe presa per buona, e il suo valore (pari a zero se siete fortunati) sarebbe ben diverso da quello della variabile col nome scritto giusto, che avreste voluto usare voi. Si tratta di errori molto difficili da scovare; il compilatore C vi mette al riparo da tutto questo, al prezzo di definire sempre e comunque tutte le variabili che intendete utilizzare. A poco a poco prenderete l'abitudine di aggiungerle automaticamente all'elenco delle variabili globali o a quelli delle variabili locali (della funzione o del blocco) a mano a mano che, scrivendo il programma, vi capiterà di doverne usare una nuova.

Blocchi if

Si tratta del nostro primo blocco *con nome*, dove il nome, abbastanza ovviamente, è *if*. Se sapete un po' d'inglese, sapete anche che *if* vuol dire *se*, la particella che si usa per esprimere i periodi ipotetici; e questa è la ragione principale per cui i blocchi *if* vengono anche chiamati *blocchi condizionali*.

Un blocco *if* ha il compito di permettervi di far intraprendere al vostro programma strade diverse (il cosiddetto *branching*) a seconda del risultato di una certa operazione logica. La tipica struttura di un blocco *if* è del tipo: "se una certa condizione è vera (o falsa) allora fai questo"; opzionalmente, potete aggiungerci un "se no, fai quest'altro". Messa così, è facile. Ma ci sono un certo numero di sottigliezze, per apprezzare le quali dobbiamo tornare a delle nostre vecchie conoscenze: le espressioni; questa volta ci occuperemo di espressioni che coinvolgono operatori logici (ma, come vedrete, anche l'operatore di assegnazione verrà chiamato in causa).

Una tipica espressione logica consiste nel confronto tra due quantità, siano esse variabili o altre espressioni. Vediamo qualche esempio: mettiamo che abbiate una variabile di tipo `int` chiamata `altezza`; per ragioni note solo agli dei, volete verificare se la persona di cui avete memorizzato l'altezza in questa variabile è più o meno alta di 180 cm. L'espressione logica che userete sarà allora `altezza > 180`, e questa vi darà un risultato logico, ovvero un valore vero o falso. Al posto dell'operatore > avreste potuto usare anche >=, < o <=, con ovvio significato dei simboli. Se aveste voluto controllare che `altezza` fosse *esattamente* uguale a 180 cm, avreste dovuto usare l'operatore di *uguaglianza logica* ==, nell'espressione logica `altezza == 180`, che vi avrebbe restituito il valore *vero* nel caso di uguaglianza, il valore *falso* nel caso di diversità (per verificare la diversità piuttosto che l'uguaglianza logica userete l'operatore `!=`, che richiama un uguale barrato, ovvero il segno di *diverso*). Vi prego di esaminare con attenzione quanto abbiamo detto: l'operatore di *uguaglianza logica* == verifica che due quantità siano uguali; l'operatore di *assegnazione* =, invece, assegna il valore a destra alla grandezza a sinistra.

Forse è meglio fare un esempio pratico. Prendete il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

#define   kMiaAltezza        181

int main(void)
{
     int       altezza;

     printf("Inserisci la tua altezza in cm: ");
     scanf("%d",&altezza);

     if(altezza>kMiaAltezza)
     {
          printf("Sei piu' alto di me!\n");
     }
     else if(altezza==kMiaAltezza)
     {
          printf("Sei alto quanto me.\n");
     }
     else
     {
          printf("Sei piu' basso di me!\n");
     }

     // ora facciamo un esperimento
     if(altezza=kMiaAltezza)
     {
          printf("Essere alti %d cm e' una bella cosa.\n",altezza);
     }
     else
     {
          printf("Purtroppo, non sei alto quanto me.\n");
     }

     return(0);
}
```

Salvatelo come Esempio8.c, compilatelo ed eseguitelo. Alla richiesta di inserire la vostra altezza, digitate 175. L'output che ne verrà fuori sarà il seguente:

```
Inserisci la tua altezza in cm: 175
Sei piu' basso di me!
Essere alti 181 cm e' una bella cosa.
```

Va beh, direte voi, che schifezza di esempio. E invece è molto istruttivo; vediamo perché. Il programma inizia nella solita maniera: la solita libreria di input/output per poter leggere e scrivere testo dalla finestra del terminale, la definizione di una costante (la mia altezza in centimetri), la solita funzione `main()`, con una variabile locale, `altezza`, nella quale viene memorizzata mediante la funzione `scanf()` (vedi capitolo 8) la vostra altezza che avrete digitato da tastiera (175 cm se avete seguito il mio suggerimento).

Poi incomincia il gioco degli `if`: innanzitutto notate la sintassi: l'espressione logica che volete valutare è racchiusa in parentesi tonde. Si incomincia verificando se `altezza>kMiaAltezza`. Se così fosse, la o le istruzioni racchiuse tra le parentesi graffe

immediatamente successive verranno eseguite (nel nostro esempio verrebbe mostrato un messaggio di disappunto per il fatto che sareste più alti di me). Terminata l'esecuzione delle istruzioni racchiuse tra il blocco di parentesi graffe immediatamente successive all'istruzione `if`, il programma continuerebbe la sua esecuzione *al termine di tutto* il blocco `if`, ovvero là dove c'è la riga commentata `// ora facciamo un esperimento`.

Siccome noi abbiamo detto di essere alti 175 cm, l'espressione logica `altezza>kMiaAltezza` restituirà un valore *falso*, pertanto le istruzioni racchiuse nel blocco di parentesi graffe immediatamente successive all'istruzione `if` saranno saltate a pie' pari, e si passerà al controllo della condizione successiva `else if (altezza==kMiaAltezza)`; qui si verifica se l'altezza da voi digitata durante l'esecuzione del programma sia uguale alla mia (181 cm), e in caso affermativo il programma eseguirebbe il blocco di istruzioni racchiuse tra parentesi graffe immediatamente successive all'istruzione `else if`, in questo caso un messaggio di compiacimento per essere alti quanto me. Terminata l'esecuzione di questo blocco di istruzioni, l'esecuzione riprenderebbe al termine di tutto il blocco `if`, ovvero là dove c'è la riga commentata.

Siccome però abbiamo inserito 175 come nostra altezza, anche l'espressione logica di prima restituisce un valore falso, e allora si passa a valutare tutti gli `else if` successivi. Come? Non ce ne sono? Allora, se c'è, viene eseguito il blocco di istruzioni che segue l'istruzione `else` (altrimenti). Come dire: se una delle espressioni logiche che seguono le istruzioni `if` oppure `else if` dà un risultato *vero*, allora le istruzioni racchiuse tra le parentesi graffe immediatamente successive all'espressione logica risultata vera vengono eseguite. Se nessuna delle espressioni logiche che seguono le istruzioni `if` oppure `else if` risulta vera, allora vengono eseguite le istruzioni racchiuse tra le parentesi graffe immediatamente successive all'istruzione `else`, ammesso che ci sia. Infatti, solo la prima delle istruzioni `if` è obbligatoria. Tutti i vari `else if` e l'`else` finale sono facoltativi. Nel nostro caso, è proprio il blocco che segue l'`else` finale ad essere eseguito, mostrando un messaggio di compiacimento perché siete più piccoli di me. Se non ci fosse stato, l'esecuzione del programma sarebbe comunque saltata al termine di tutto il blocco `if`, ovvero alla riga commentata. Come dire: se l'espressione logica che segue l'istruzione `if` obbligatoria è vera, allora le istruzioni che seguono l'`if` vengono eseguite; se no, in presenza di istruzioni `else if` oppure `else` queste vengono valutate ed eventualmente eseguite. Alla fine, l'esecuzione riprende al termine dell'intero blocco `if`, cioè dove sono finalmente finiti tutti gli `else if` e l'eventuale `else` finale.

Già, qui c'è un'altra sottigliezza notevole. Avrete notato che le espressioni logiche riportate nell'esempio precedente sono mutualmente esclusive, perché la vostra altezza non può essere simultaneamente maggiore della mia, minore ed uguale. Provate allora a fare questa modifica: sostituite `altezza>kMiaAltezza` con `altezza>=kMiaAltezza`, ricompilate ed eseguite di nuovo il programma, digitando questa volta 181 come vostra altezza. Che succede? Il programma esprimerà il suo disappunto perché siete più alti di me (questa volta l'espressione logica `altezza>=kMiaAltezza` è vera, pertanto il blocco successivo all'istruzione `if` viene eseguito), ma non c'è traccia del messaggio riguardante il fatto che siete alti esattamente quanto me. Perché? Perché quando un'espressione logica viene trovata vera, il blocco di istruzioni successive alla stessa viene eseguito, dopo di che si salta immediatamente al termine del blocco `if`; tutti gli altri `else if` e l'eventuale `else` finale *non vengono nemmeno presi in considerazione*. Quindi, malgrado l'espressione logica `altezza==kMiaAltezza` fosse vera pure lei, essa non è mai stata valutata, perché ne era già stata trovata una in precedenza che fosse vera. Se volete eseguire bocchi di codice che corrispondono a istruzioni `if` che valutano espressioni logiche che potrebbero tutte essere vere, dovete inserirle in blocchi `if` distinti, e non come `else if` in un blocco solo.

Andiamo avanti: dopo l'istruzione commentata, c'è un altro blocco condizionale. Avete notato che ho usato l'operatore di assegnazione anziché quello di uguaglianza logica? No, non è un errore. È per farvi vedere una cosa. Quando avete eseguito l'esempio la prima volta, inserendo 175 come vostra altezza, dopo il messaggio di compiacimento per il fatto che siete più piccoli di me è apparso un messaggio un po' strano: `Essere alti 181 cm e' una bella cosa`. Che cos'ha di strano questo messaggio? Beh, sta all'interno di un blocco `if` che è stato

eseguito anche se `altezza` non era affatto uguale a `kMiaAltezza`. Infatti, l'operatore usato non è quello di uguaglianza logica `==`, ma quello di assegnazione `=`. Quello che succede in questo caso, è che il valore `kMiaAltezza` viene *assegnato* alla variabile `altezza` (d'altro canto, è questo il compito dell'operatore di assegnazione), *poi* viene valutato il valore logico di questa espressione; siccome essa è andata a buon fine (il valore `kMiaAltezza` è stato effettivamente assegnato alla variabile `altezza`), il valore logico risultante sarà *vero*. Quindi l'espressione logica nel blocco `if` è vera, e il codice incluso nelle parentesi graffe successive viene eseguito. Notate infatti come il messaggio che viene visualizzato a schermo prenda il valore di 181 cm *non* dalla costante `kMiaAltezza`, ma dalla variabile `altezza`, che ha assunto il nuovo valore (diverso da quello di 175 che avevate impostato voi) nel momento in cui avete eseguito l'assegnazione commentata sopra.

Tutto questo per dire che un'espressione logica non è necessariamente fatta mediante l'uso *esplicito* degli operatori logici, ma anche attraverso la cosiddetta *assegnazione implicita a target nullo*: l'assegnazione di un valore ad una variabile è uno di questi casi. Vedremo in futuro ulteriori usi di questa tecnica (che qui, obiettivamente, non è servita a molto) quando parleremo ad esempio di funzioni.

Torniamo alla sintassi: abbiamo dispensato parentesi graffe a gran profusione nell'Esempio 8, ma in realtà in questo caso ne avremmo potuto fare a meno. Infatti, la sintassi del C prescrive che se l'istruzione da eseguire *nel caso in cui l'espressione logica che viene valutata sia vera sia solo una* (e non più di una), è possibile evitare di scrivere le parentesi graffe. Il nostro blocco `if` poteva essere scritto più brevemente così:

```
if(altezza>kMiaAltezza)
          printf("Sei piu' alto di me!\n");
     else if(altezza==kMiaAltezza)
          printf("Sei alto quanto me.\n");
     else
          printf("Sei piu' basso di me!\n");
```

Notate come l'indentazione aiuti a capire quali sono le istruzioni che verranno eseguite nei vari casi. È una scorciatoia comoda ma pericolosa. Mettiamo di avere questa situazione:

```
if(a>b)
     if(c>d)
          printf("Ohibo'.\n");
else
     printf("Meno male!\n");
```

Se `a` è maggiore di `b`, allora il programma verifica se `c` è maggiore di `d`, e in caso affermativo esclama il proprio rammarico. Tuttavia, a chi compete l'`else`? L'indentazione suggerisce che esso esprima la condizione *alternativa* al caso in cui `a` sia maggiore di `b`. Ma l'assenza di parentesi graffe pone l'`else` come la condizione alternativa al caso in cui `c` sia più grande di `d`. Insomma, ci si può confondere. Se siete nel dubbio, mettete le parentesi:

```
if(a>b)
{
     if(c>d)
          printf("Ohibo'.\n");
}
else
     printf("Meno male!\n");
```

Tra l'altro, questa divagazione vi fa vedere che potete tranquillamente inserire blocchi condizionali l'uno all'interno dell'altro; occhio solo a non sbagliarvi con le parentesi. Ecco

perché prendere la buona abitudine di indentare il codice col tasto tabulatore aiuta a non perdersi: provate voi a tenere il conto delle cose quando iniziate ad avere tre o quattro blocchi condizionali incastrati l'uno nell'altro!

Un'ultima nota: vi avevo detto che tutte le istruzioni C vanno terminate con un punto e virgola, salvo alcune eccezioni. Beh, come vedete `if`, `else if` ed `else` fanno eccezione. Se vi sbagliate e inserite un punto e virgola dopo un'istruzione if, potreste incorrere in spiacevoli equivoci:

```
if(a>b);
{
     if(c>d);
          printf("Ohibo'.\n");
}
else;
     printf("Meno male!\n");
```

La condizione `a>b` sarebbe valutata. In caso di risposta vera, l'istruzione successiva (il punto e virgola! ovvero l'istruzione che non fa niente) sarebbe eseguita (non facendo niente non succederebbe niente). In caso di risposta falsa, il programma riprenderebbe al termine del blocco `if`, ovvero *là dove si apre la parentesi graffa*, che sarebbe interpretata come un blocco senza nome. Questo sarebbe eseguito sempre, così come l'espressione di ambascia sarebbe sempre scritta, per gli stessi motivi (il punto e virgola dopo il secondo `if`). In compenso, l'istruzione `else` genererebbe un errore (già in fase di compilazione), perché se ne starebbe lì senza un `if` che la precede (il blocco `if` associato alla condizione `a>b` è già stato dichiarato chiuso perché l'istruzione nulla `;` è quella che viene eseguita se la condizione è vera – non essendoci parentesi subito dopo l'`if` solo essa verrebbe eseguita – e nel caso di condizione falsa non è stato trovato un `else` *immediatamente successivo* al punto e virgola, quindi vuol dire che *non c'è*). Quindi, non mettete *mai* i punti e virgola dopo gli `if` e gli `else` se non volete cacciarvi nelle grane.

Beh, è stato faticoso, ma il resto del capitolo sarà più facile, perché già abbiamo parlato di espressioni logiche.

Blocchi switch

Esiste un'alternativa ai blocchi `if`, e sono i blocchi `switch`. Non è però che siano perfettamente sostituibili l'uno all'altro, per cui non illudetevi, dovete impararli entrambi.

Un blocco `switch` è sostanzialmente come un blocco `if` dotato di vari `else if` e di un `else` finale; ognuna delle condizioni che vengono testate è in effetti governata da un operatore logico di uguaglianza `==`. Non potete usare nessuno degli altri operatori logici né espressioni logiche complesse; ma se state facendo una casistica dei valori che può assumere una variabile, allora il blocco `switch` potrebbe fare per voi. Vediamo un esempio. Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

#define   kVocaleNormale          0
#define   kVocaleAnomala          1
#define   kConsonante             2

#define   kParolaPerA             "assurgere"
#define   kParolaPerE             "escoriazione"
#define   kParolaPerI             "impunita'"
#define   kParolaPerO             "ostracismo"
#define   kParolaPerU             "ungulati"
```

```
int main(void)
{
     char     lettera;
     int           risposta;

     printf("Questo e' un test di intelligenza.");
     printf("Digita una vocale, poi premi \"a-capo\": ");
     scanf("%c",&lettera);

     switch(lettera)
     {
          case 'a':
               printf("%c come %s\n",lettera,kParolaPerA);
               risposta=kVocaleNormale;
               break;
          case 'e':
               printf("%c come %s\n",lettera,kParolaPerE);
               risposta=kVocaleNormale;
               break;
          case 'i':
               printf("%c come %s\n",lettera,kParolaPerI);
               risposta=kVocaleNormale;
               break;
          case 'o':
               printf("%c come %s\n",lettera,kParolaPerO);
               risposta=kVocaleNormale;
               break;
          case 'u':
               printf("%c come %s\n",lettera,kParolaPerU);
               risposta=kVocaleNormale;
               break;
          case 'j':
          case 'y':
               printf("Facciamo i difficili, eh?\n");
               risposta=kVocaleAnomala;
               break;
          default:
               printf("%c non e' una vocale!\n",lettera);
               risposta=kConsonante;
               break;
     }

     if((risposta==kVocaleNormale) || (risposta==kVocaleAnomala))
          printf("Complimenti, hai passato il test!\n");
     else
          printf("Sei un po' duro di comprendonio, vero?\n");

     return(0);
}
```

Salvatelo come Esempio9.c, compilatelo ed eseguitelo. Divertitevi ad eseguirlo un po' di volte, dando risposte diverse. Quando vi sarete stufati, continuate la lettura.

Come funziona questo programma? Inizia come al solito, con l'inclusione della libreria di input/output e la definizione di alcune costanti di tipo stringa di caratteri. Poi, dopo il messaggio introduttivo, vi viene chiesto di digitare una vocale. A questo punto parte il blocco `switch`: come vedete tra paretesi tonde c'è il nome della variabile su cui eseguire un test. Ognuna delle istruzioni `case` che seguono verificano se la variabile contenuta tra le parentesi successive all'istruzione `switch` sia uguale a ciò che segue l'istruzione `case`. È un po' come scrivere:

```
if(lettera=='a')
{
     printf("%c come %s\n",lettera,kParolaPerA);
     risposta=kVocaleNormale;
}
else if(lettera=='e')
{
     printf("%c come %s\n",lettera,kParolaPerE);
     risposta=kVocaleNormale;
}
...
else
{
     printf("%c non e' una vocale!\n",lettera);
     risposta=kConsonante;
}
```

Attenzione però ad alcune sottigliezze: ognuna delle istruzioni `case` termina con un due punti. Non starò qui a dirvi perché (di fatto è una sorta di *label*, una di quelle robe a cui si salta con un'istruzione `goto`, ma sono cose brutte, di cui è meglio non parlare in questa sede). Questo ha delle conseguenze: quando la variable esaminata (`lettera` per noi) corrisponde ad uno dei `case`, il codice immediatamente successivo viene eseguito (non è il caso di racchiuderlo tra parentesi graffe, se lo fate, niente di male, state semplicemente creando un blocco senza nome). L'esecuzione del codice va avanti *finché non viene incontrata l'istruzione* `break`; a questo punto l'esecuzione si interrompe e riprende all'istruzione successiva alla parentesi graffa che chiude il blocco `switch`. Perché questa precisazione è importante? Perché se non mettete l'istruzione `break` succede una cosa carina: quando volete testare il caso che abbiate digitato la lettera j oppure la lettera y fate così: scrivete `case 'j':` come al solito, e poi potreste mettere il codice da eseguire in questo caso terminato con un `break`; però, il codice da eseguire se la lettera che digitate è la y è lo stesso; e allora, perché ripeterlo due volte? Così, dopo `case 'j':` non mettete il `break`, e lasciate che il programma continui l'esecuzione *anche quando incontra l'istruzione* `case` *successiva,* che viene scavalcata a pie' pari e il programma va avanti *finché non trova un* `break`; è un po' come scrivere:

```
if((lettera=='j') || (lettera=='y'))
{
     ...
}
```

State quindi attenti: non mettere il `break` dopo un'istruzione `case` può essere utile, ma se lo "dimenticate" potreste ritrovarvi con un errore anche molto difficile da andare a scovare.

Infine, il programma termina con un messaggio che vi dice se avete passato o no il test di intelligenza. Qui, cogliamo l'occasione per mostrare come un'espressione logica non debba necessariamente essere semplice come quelle che abbiamo usato finora. Infatti, il test che viene

eseguito (sempre sulla stessa variabile, in questo caso), comporta il valutare se questa sia uguale (operatore `==`) ad una costante *oppure* ad un'altra. Le parentesi tonde, superflue in realtà in questo caso per ragioni di *precedenza degli operatori* (di cui però non abbiamo parlato), mostrano l'ordine di priorità con cui vengono eseguite le operazioni: dapprima la variabile `risposta` viene confrontata con `kVocaleNormale` e con `kVocaleAnomala` (in quale ordine non è dato sapere, il compilatore farà di testa sua e voi non potete farci niente), *poi* si valuta se *almeno uno dei confronti* ha dato esito positivo (l'operatore OR logico `||`). In caso affermativo, il test è superato e il messaggio di complimenti viene mostrato, se no si ripiega sul messaggio un po' polemico.

Blocchi while

Iniziamo qui a parlare di quei blocchi altrimenti noti come *cicli* (*loop* in inglese). Sono blocchi dotati di nome (`while`, `for`, `do-while`) che eseguono le istruzioni racchiuse tra le loro parentesi graffe un certo numero di volte. Quante volte? Beh, questo è deciso, indovinate un po', dal risultato della valutazione di un'espressione logica (vero o falso). Iniziamo dal blocco `while`. Prendete il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

#define   kMassimoNumeroDiTentativi             3
#define   SI                                    1
#define   NO                                    0
#define   NON_SO                                -1

int main(void)
{
     int           numero;
     int           tentativi;
     int           vaBene;

     tentativi=0;
     vaBene=NON_SO;
     printf("Questo e' un test di intelligenza.\n");
     while((tentativi<kMassimoNumeroDiTentativi) && (vaBene!=SI))
     {
          tentativi++;
          printf("Digita un numero tra 1 e 5, poi digita \"a-capo\": ");
          scanf("%d",&numero);
          if(numero>=1 && numero<=5)
               vaBene=SI;
          else
          {
               vaBene=NO;
               printf("Sbagliato!\n");
          }
     }
     if(vaBene==SI)
          printf("Complimenti!   Ci   sei   riuscito   in   %d
tentativi!\n",tentativi);
     else
          printf("Peccato. Non ce l'hai fatta!\n");
     return(0);
}
```

Salvatelo come Esempio10.c, compilatelo ed eseguitelo tutte le volte che volete. Poi chiedetevi come funziona. Ed ecco qui la risposta. Dopo il solito inizio barboso, trovate il famoso ciclo `while`: esso consta di una coppia di parentesi tonde entro le quali viene inclusa un'espressione logica che viene verificata. Se essa risulta essere vera, allora vengono eseguite le istruzioni racchiuse tra le parentesi graffe immediatamente successive all'istruzione `while`. Nel nostro caso, all'avvio il programma imposta a zero la variabile `tentativi` (che quindi risulta minore di `kMassimoNumeroDiTentativi`, e la prima condizione è vera), e imposta a `NON_SO` la variabile `vaBene` (che quindi è diversa da `SI`, e anche la seconda condizione è vera). Poiché tra le due condizioni sussiste un operatore di AND logico `&&`, è necessario che *entrambe* siano vere affinché l'espressione logica completa sia vera. Poiché lo sono, le istruzioni comprese nel blocco `while` vengono eseguite. È da notare che se già all'inizio l'espressione logica fosse risultata falsa (ad esempio perché per errore di battitura del codice la variabile `vaBene` potrebbe essere uguale a `SI`), il blocco while non verrebbe *mai* eseguito.

A questo punto vi viene chiesto di digitare un numero tra 1 e 5 (non vi consiglio di digitare lettere, il programma è piuttosto rozzo e "impazzirebbe"). Ogni volta che ci provate la variabile `tentativi` viene incrementata. Se digitate un numero che va bene, la variabile `vaBene` diventa uguale a `SI`, se no diventa uguale a `NO`. Giunti alla parentesi graffa che chiude il ciclo `while`, il programma ritorna all'istruzione `while` stessa, e l'espressione logica viene *nuovamente valutata*. Se avevate digitato un numero tra 1 e 5, allora la variabile `vaBene` è `SI`, l'espressione logica è falsa e il ciclo `while` si interrompe e il programma ricomincia dalla prima istruzione *successiva* alla parentesi graffa di chiusura del ciclo `while`. Se invece `vaBene` è `NO` e non avete ancora esaurito i vostri tentativi, il ciclo `while` vi darà un'altra possibilità di provarci. Alla fine, a seconda dell'esito della vostra prova, vi verrà detto se l'avete superata oppure no.

Un'ultima nota: anche l'istruzione `while` *non vuole* il punto e virgola dopo di sé. Se lo mettete, il punto e virgola (l'istuzione che non fa niente) verrà preso come l'istruzione da eseguire nel caso in cui l'espressione logica sia vera (qui, come per gli `if`, se mancano le parentesi graffe si assume che il blocco di istruzioni da eseguire sia costituito da un'istruzione sola). Se l'espressione logica è vera, l'istruzione nulla non farà nulla (scusate!) per renderla falsa, e avrete realizzato un fantastico ciclo infinito che non fa assolutamente niente! Per uscire da questa situazione imbarazzante, non potrete fare altro che interrompere il vostro programma con control-C.

Blocchi do-while

Il programma dell'esempio precedente funziona, ma non è ottimizzato. Infatti, la prima volta che l'espressione logica del ciclo `while` viene testata siete già sicuri che il test verrà passato. Infatti, se così non fosse, l'utente del vostro programma non avrebbe l'opportunità di misurare la propria intelligenza. In altre parole, un ciclo `while` potrebbe anche non venire eseguito *mai*. Voi, invece, in questo caso avete bisogno che il ciclo venga eseguito *almeno una volta*. Sostanzialmente avete bisogno di spostare la valutazione dell'espressione logica al termine del ciclo anziché all'inizio. Il C vi viene in aiuto con il ciclo `do-while`. Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

#define   kMassimoNumeroDiTentativi             3
#define   SI                                    1
#define   NO                                    0

int main(void)
{
     int            numero;
     int            tentativi;
```

```
    int           vaBene;

    tentativi=0;
    printf("Questo e' un test di intelligenza.\n");
    do
    {
        tentativi++;
        printf("Digita un numero tra 1 e 5, poi digita \"a-capo\": ");
        scanf("%d",&numero);
        if(numero>=1 && numero<=5)
            vaBene=SI;
        else
        {
            vaBene=NO;
            printf("Sbagliato!\n");
        }
    } while((tentativi<kMassimoNumeroDiTentativi) && (vaBene!=SI));

    if(vaBene==SI)
        printf("Complimenti!   Ci   sei   riuscito   in   %d
tentativi!\n",tentativi);
    else
        printf("Peccato. Non ce l'hai fatta!\n");

    return(0);
}
```

Salvatelo come Esempio11.c, compilatelo ed eseguitelo a piacimento. Ma come funziona? Beh, è quasi uguale all'Esempio 10. Solo che, con il ciclo `do-while`, abbiamo spostato la valutazione dell'espressione logica al fondo del ciclo anziché all'inizio. Questo ci assicura che il ciclo venga eseguito *almeno una volta*. Va notato che con questa miglioria abbiamo anche eliminato la costante `NON_SO`, che ora non ci serve più dal momento che non abbiamo bisogno di assegnare un valore alla variabile `vaBene` *prima* che inizi il ciclo.

Notate inoltre che adesso il punto e virgola *dopo* l'istruzione `while` *ci vuole*! Questo serve per far capire al compilatore che quel `while` lì sta in realtà chiudendo un ciclo `do-while` e non è l'apertura di un ciclo `while` normale.

Blocchi for

Ultimi ma non meno importanti sono i blocchi `for`, che funzionano in maniera simile agli altri cicli già visti. In effetti, avere a disposizione cicli `while`, `do-while` e `for` è sovrabbondante: potreste fare tutto con uno solo di questi tre tipi di cicli, che infatti sono sempre trasformabili l'uno nell'altro. Tuttavia, le loro caratteristiche leggermente diverse li rendono più o meno comodi a seconda delle circostanze. Il C ve li mette allora a disposizione tutti e tre (ci sono linguaggi più generosi, come il Pascal, in cui avete ancora più sovrabbondanza di tipi di cicli!), per vostra comodità, affinché possiate scegliere quello che maggiormente si confà alle vostre esigenze di programmazione.

Un uso tipico di un ciclo `for` è mostrato nell'esempio seguente, che digiterete usando il vostro editor di testo preferito:

```
#include <stdio.h>

int main(void)
{
```

```
    int             numeriDaSommare;
    double          totale;
    int             i;

    totale=0.0;
    printf("Quanti numeri vuoi sommare tra di loro? ");
    scanf("%d",&numeriDaSommare);

    for(i=1;i<=numeriDaSommare;i++)
    {
        double          numero;

        numero=0.0;
        printf("Digita il %do numero: ",i);
        scanf("%lg",&numero);
        totale+=numero;
    }

    printf("Il totale vale: %g\n",totale);

    return(0);
}
```

Salvatelo come Esempio12.c, compilatelo, eseguitelo e giocateci. Poi chiedetevi come fuziona.

A parte le solite cose, la prima cosa che vi viene richiesta è di inserire quanti numeri volete sommare (contenetevi! Non sommatene 12741!). A questo punto incontrare la famosa istruzione `for`: essa consta di tre pezzi racchiusi tra parentesi tonde e separati da un punto e virgola, più una coppia di parentesi graffe che racchiude le istruzioni da eseguire ciclicamente. Il primo pezzo è *l'istruzione di inizializzazione* della *variabile di controllo*, nel nostro caso `i`: appena si giunge all'istruzione `for`, la variabile di controllo (`i` nel nostro caso) assume il valore assegnatole nel primo pezzo (1 nel nostro caso). Quindi l'espressione logica *di controllo* viene valutata, ed è quella che sta nel secondo pezzo, tra i due punti e virgola; nel nostro caso è `i<=numeriDaSommare`. Se l'espressione logica è vera, allora viene eseguito il blocco di istruzioni racchiuse tra le parentesi graffe immediatamente successive all'istruzione `for`. Al termine del blocco, viene eseguita l'istruzione di *variazione della variabile di controllo*, ovvero il terzo pezzo: nel nostro caso `i++`. A questo punto l'espressione logica di controllo viene di nuovo valutata (l'istruzione di inizializzazione viene ora ignorata perché viene presa in considerazione solo alla prima esecuzione del ciclo), se è vera il blocco di istruzioni viene eseguito, l'istruzione di variazione della variabile di controllo viene eseguita, l'espressione logica di controllo viene valutata e via discorrendo. Quando l'espressione logica di controllo risulterà falsa (il che potrebbe anche non avvenire mai), il blocco di istruzioni racchiuse tra le parentesi graffe verrà saltato e il programma riprenderà dalla prima istruzione successiva alla parentesi graffa che chiude il ciclo `for`. È da notare che anche per il ciclo `for` vale la solita questione del punto e virgola: se lo mettete subito dopo le parentesi tonde che seguono il `for`, allora state dicendo che l'unica istruzione da eseguire è quella nulla (mancando delle parentesi graffe), e il ciclo non realizzerà niente di utile. Se non mettete le parentesi graffe, state implicitamente dicendo che il blocco di istruzioni da eseguire è costituito da *una sola* istruzione, la prima che c'è dopo l'istruzione `for`.

Notate che all'interno delle parentesi graffe del ciclo `for` abbiamo dichiarato una variabile locale `numero`. Essa non esiste al di fuori del blocco `for`, ma tanto non la useremmo, per cui sarebbe stato inutile se non addirittura scomodo dichiararla all'inizio della funzione `main()`. È la vecchia regola di cui abbiamo già parlato: limitate il più possibile lo *scope* delle variabili, e ne guadagnerete in salute.

È utile ricordare che l'espressione logica di controllo del ciclo *non deve necessariamente coinvolgere* la variabile di controllo, così come l'istruzione di variazione della variabile di controllo non deve necessariamente essere di incremento e non deve necessariamente riguardare la variabile di controllo. Tuttavia, se eliminate la variabile di controllo dal secondo e dal terzo pezzo costitutivo dell'istruzione `for`, state facendo qualche cosa di più raffinato che esula dai confini che ho deciso di porre a questa *Introduzione al linguaggio C*.

Uscire dai cicli

I cicli `for`, `while` e `do-while`, dicevamo, possono avere una fine, ma possono anche non averla. Può darsi che, scrivendo il programma, siate in grado di decidere come e quando dovrà terminare il vostro ciclo, ma può darsi che non lo sappiate. Può anche darsi che durante l'esecuzione di un ciclo si verifichi un errore da qualche parte (che so, l'utente immette un dato sbagliato o altro) e volete uscire dal ciclo. Che fate? Usate due istruzioni: `break` e `continue`.

La prima l'abbiamo già usata con l'istruzione `switch`, e ci serviva, guarda caso, per *uscire* dal blocco switch. La seconda, invece che uscire da un ciclo, ne forza l'esecuzione dell'iterazione successiva. Vediamolo meglio con un esempio. Prendete il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

int main(void)
{
     double        totale;

     totale=0.0;
     printf("Questo programma somma numeri esclusivamente positivi!\n");
     for(;;)
     {
          double        numero;

          numero=0.0;
          printf("Digita un numero d aggiungere al totale (0 per uscire):
");
          scanf("%lg",&numero);
          if(numero>0)
               totale+=numero;
          else if(numero<0)
               continue;
          else
               break;

          printf("Il totale vale: %g\n",totale);
     }

     return(0);
}
```

Salvatelo come Esempio13.c, compilatelo, eseguitelo e giocateci per un po'. Ora parliamone insieme. Dopo le solite cose iniziali, inizia un ciclo `for` *infinito*! Lo notate subito perché non è presente nessuno dei tre pezzi che abbiamo commentato prima, *ma sono presenti i punti e virgola*, che sono obbligatori, in quanto sono parte integrante dell'istruzione `for` stessa. Poiché trattasi di un ciclo infinito, dobbiamo trovare una maniera per uscirne. La maniera ci è offerta dall'utente: tutte le volte che questi digita un numero maggiore di zero, esso viene

aggiunto al totale che viene aggiornato a schermo; tutte le volte che digita un numero minore di zero, esso viene ignorato (l'istruzione `continue` forza l'interruzione dell'iterazione corrente e l'inizio di una nuova, quindi il totale aggiornato non viene mostrato); se digita zero, l'istruzione break ci fa uscire dal ciclo e, a quel punto, il programma è finito.

5. Funzioni

La funzione main

Ne abbiamo parlato brevemente nel capitolo 2, ma allora stavamo più che altro speculando. Ora è giunto il momento di parlare di funzioni. E la prima funzione di cui parliamo è la funzione `main()`, che abbiamo usato finora in tutti gli esempi fatti. La ragione per cui l'abbiamo sempre usata è che essa è obbligatoria: tutti i programmi C devono averne una e solo una.

La funzione `main()`, comunque, ci serve da utile esempio per vedere di che cosa è costituita una funzione. Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

int      variabileGlobale;

int main(void)
{
     int      variabileLocale;

     variabileLocale=1;
     variabileGlobale=24;
     printf("La funzione main() e' obbligatoria.\n");
     printf("Potete accedere alle variabili locali: %d\n",variabileLocale);
     printf("e a quelle globali: %d\n",variabileGlobale);

     return(0);
}
```

Salvatelo come Esempio14.c, compilatelo ed eseguitelo. Ormai ne sapete abbastanza per capire perfettamente quello che succede. Ma qui facciamo un ripassino. La *funzione* `main()` è identificata dal suo nome (`main`, per l'appunto) *e dalla coppia di parentesi tonde che seguono*. Questa regola vale in realtà per tutte le funzioni. Essa è preceduta da una dichiarazione di tipo, `int` in questo caso, di cui parleremo tra poco. È poi seguita da un blocco delimitato da due parentesi graffe all'interno delle quali si trova il contenuto della funzione stessa. Quivi possono trovarsi variabili locali, blocchi, cicli, istruzioni condizionali. Al termine, la funzione main() si chiude con un'istruzione `return()`, che tra parentesi tonde contiene un numero (o una costante, o una variabile). L'istruzione `return()` segnala al computer che l'esecuzione del vostro programma è terminata. Il valore del numero, della costante o della variabile contenuti tra le due parentesi tonde vengono trasmesse al programma dal quale avete lanciato l'Esempio14 (tipicamente l'applicazione Terminale), ed in genere indica come il vostro programma si è concluso, se regolarmente (il valore restituito vale zero come nel nostro esempio) o se con qualche errore (valori diversi da zero). È importante, anzi fondamentale, che il numero, la costante o la variabile che mettete all'interno delle parentesi nell'istruzione `return()` sia dello stesso *tipo* dichiarato per la funzione `main()`, ovvero `int`. L'istruzione `return()` non deve necessariamente essere l'ultima istruzione della funzione; tuttavia, essa è l'ultima ad essere eseguita. Lo vedremo meglio in un prossimo esempio.

Le altre funzioni

Quanto abbiamo detto per la funzione `main()` è in realtà vero anche per le altre funzioni. Ovvero: esse sono identificate da un nome (unico per ogni funzione di un programma), dalla coppia di parentesi che seguono il nome, da un'indicazione di tipo posta prima del nome della

funzione e dalla coppia di parentesi graffe che ne identificano il contenuto. All'interno della funzione troveranno posto variabili locali e blocchi di ogni tipo. Le uniche variabili locali accessibili dall'interno di una funzione saranno quelle definite nella funzione stessa, e saranno distrutte non appena il programma "uscirà" da quella funzione. Dal suo interno si potrà comunque accedere a tutte le variabili globali. Vediamo meglio qualche altra caratteristica fondamentale in un primo semplice esempio. Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

#define   kNumeroMassimoDiTentativi        3
#define   SI                               1
#define   NO                               0

int InserisciUnNumero(void);
int  TestDelNumero(int numero);
void MostraRisultato(int tentativi);

int main(void)
{
     int      tentativi;

     tentativi=0;
     printf("Questo e' un test di intelligenza");
     do
     {
          int      numero;

          tentativi++;
          numero=InserisciUnNumero();
          if(TestDelNumero(numero))
          {
               printf("Complimenti!!!\n");
               break;
          }
          else
               printf("Non va bene!\n");
     } while(tentativi<kNumeroMassimoDiTentativi);

     MostraRisultato(tentativi);

     return(0);
}

int InserisciUnNumero(void)
{
     int      num;

     printf("Inserisci un numero intero tra 1 e 5: ");
     scanf("%d",&num);
     return(num);
}
```

```
int TestDelNumero(int numero)
{
     if((numero>=1) && (numero<=5))
          return(SI);
     else
          return(NO);
}

void MostraRisultato(int tentativi)
{
     printf("In totale hai tentato %d volte\n",tentativi);
}
```

Salvatelo come Esempio15.c, compilatelo e divertitevi ad eseguirlo. Avete notato le molte novità? Adesso le esaminiamo una per una.

Innanzitutto il programma inizia in maniera diversa dal solito: dopo la solita libreria di input/output e le definizioni delle costanti ci sono tre righe nuove: esse ricordano le definizioni delle funzioni, solo che mancano del blocco della funzione stessa e sono seguite da un punto e virgola. Che cosa sono? Sono i *prototipi* delle funzioni. Il compilatore C ne ha bisogno per sapere quali e quante funzioni usa il vostro programma e quali tipi restituiscano *prima* di compilare il programma stesso. Una delle ragioni per cui gli servono queste informazioni è per controllare che voi facciate tutto per benino, segnalandovi eventuali errori derivanti da un errore nello scrivere il nome di una funzione o nel tipo di una variabile in qualche maniera associata ad una funzione (sotto forma di valore restituito o di *argomento*, ci ritorneremo tra poco). Dovete elencare i prototipi di *tutte* le funzioni che usate nel vostro programma, con l'unica eccezione (facoltativa) della funzione `main()`, purché essa sia priva di argomenti e restituisca un tipo `int` (come nel nostro esempio). Il tipo restituito da una funzione può essere un qualunque tipo di variabile, con l'aggiunta del tipo `void`, che indica che la funzione in questione non restituisce alcunché. Tra le parentesi tonde dei prototipi delle funzioni vanno indicati i loro *argomenti*, cioè le variabili (con il loro tipo) che *passate* alle funzioni acciocché queste possano farsene qualche cosa. La faccenda, qui, è piuttosto delicata. Sapete che una funzione può accedere solo alle sue variabili locali (non a quelle delle altre funzioni!) e alle variabili globali. Che fate se una funzione ha bisogno di conoscere il valore di una variabile locale di un'*altra* funzione? Avete due possibilità: modificare lo *scope* di quella variabile, rendendola globale, cioè definendola al di fuori di ogni funzione, subito dopo la definizione delle costanti; oppure *passarla come argomento*. Adesso vediamo come.

La funzione `main()` fa cose già viste e riviste, ma questa volta con un approccio un po' diverso e più "elegante". Definisce la variabile locale `tentativi`, poi all'interno del ciclo `do-while` definisce la variabile `numero`, accessibile solo all'interno del ciclo stesso. A questo punto, anziché preoccuparsi di fare tutto lei, la funzione `main()` delega ad altre funzioni la responsabilità di compiti specifici. Così inizia a *chiamare* la funzione `InserisciUnNumero()`. Come fate a sapere che è una funzione? Beh, innanzitutto perché l'avete elencata tra i prototipi all'inizio del programma. Poi perché il suo nome è seguito da una coppia di parentesi tonde. Infine perché, se seguite un minimo di eleganza formale, avrete cura di identificare le funzioni con nomi che iniziano con una lettera maiuscola (a differenza delle variabili, che farete iniziare con una lettera minuscola). Come da prototipo, la funzione `InserisciUnNumero()` non richiede argomenti (avete messo un `void` nel prototipo), pertanto la *chiamate* senza scrivere nient'altro tra le due parentesi (volendo, potreste scrivere `void`, ma sarebbe ridondante e normalmente non si fa). Sarebbe un errore, segnalato dal compilatore, se cercaste di passare alla funzione `InserisciUnNumero()` uno o più argomenti, dal momento che essa è stata dichiarata priva di argomenti nel suo prototipo. Se andate a vedere la funzione stessa, poco sotto, scoprirete che essa si incarica di chiedere all'utente di inserire un numero intero e, ottenutolo, lo *restituisce* alla funzione chiamante (la `main()` nel nostro caso) mediante l'istruzione `return()` che contiene, tra le parentesi, il valore da restituire (il numero digitato dall'utente). Poiché la

funzione InserisciUnNumero() restituisce un valore di tipo int, esso può essere assegnato ad una variabile di tipo int, come è appunto la variabile numero definita nel ciclo do-while della funzione main(). Come vedete è possibile assegnare valori alle variabili anche attraverso i valori restituiti dalle funzioni.

Subito dopo, tra le parentesi tonde dell'istruzione if, viene chiamata la funzione TestDelNumero(). Qui succede un'altra cosa interessante. La funzione TestDelNumero() ha bisogno di sapere che numero ha inserito l'utente per poter stabilire se esso sia compreso tra 1 e 5 oppure no. Però la variabile numero è una variabile locale del ciclo do-while della fuzione main(), quindi non può assolutamente essere usata dalla funzione TestDelNumero(). Che si fa allora? D'accordo col prototipo, si passa alla funzione TestDelNumero() un argomento di tipo int, la variabile numero per l'appunto, acciocché essa possa conoscerne il contenuto. La funzione TestDelNumero() non lavorerà direttamente sulla variabile numero del ciclo do-while della funzione main(), ma su *una copia* di essa, quella variabile numero presente ad argomento della funzione TestDelNumero() là dove la funzione stessa è definita (cioè dove trovate il codice vero e proprio della funzione). Non lasciatevi confondere: avremmo potuto scrivere int TestDelNumero(int lecitinaDiSoia) e non sarebbe cambiato proprio nulla: la funzione TestDelNumero() lavora su una *sua copia privata personale* della variabile numero, pertanto non c'è il minimo bisogno che il nome di questa variabile coincida col nome di quella che passate ad argomento quando chiamate la funzione (operazione che noi abbiamo fatto nella funzione main()). Naturalmente, il nome che date alla variabile deve essere sempre lo stesso in *tutta* la funzione TestDelNumero(). Questa, di suo, verifica se questo sia compreso tra 1 e 5 oppure no, e restituisce SI oppure NO a seconda del caso.

Qui succede un'altra cosa interessante. La funzione TestDelNumero() viene chiamata *dentro* le parentesi tonde di un'istruzione if. Pertanto essa è usata come se fosse un'espressione logica. E in effetti lo è. Essa restituisce un numero che, se diverso da zero, verrà interpretato come valore logico vero, e se uguale a zero verrà interpretato come valore logico falso. Abbiamo avuto l'accortezza di definire SI pare a 1 (cioè diverso da zero) e NO pari a zero, così che se la funzione restituisce SI l'espressione logica risulta vera e il programma mostrerà i complimenti, se no ci bacchetterà rimproverandoci. Anche questo è un esempio di quella roba che avevamo chiamato *assegnazione implicita a target nullo*.

Alla fine, se avremo indovinato o se avremo esaurito i tentativi a nostra disposizione, la funzione MostraRisultato() verrà chiamata, passando ad argomento in numero di tentativi che abbiamo usato, cosicché possa mostrarlo a schermo. Siccome la funzione MostraRisultato() non restituisce nessun valore (nel prototipo è stata dichiarata come void), il suo risultato (che non esiste) non verrà assegnato a nessuna variabile (ovviamente). E poi, la funzione main() esce con la sua istruzione return() e il programma termina lì.

Finora, abbiamo lavorato solo con funzioni che accettano un argomento solo. In realtà, se il prototipo ce lo consente, le funzioni possono accettare un numero arbitrario di argomenti. Per fare un esempio, creeremo forse l'unico programma un pelino utile di questa *Introduzione al linguaggio C*. Come ricorderete, infatti, abbiamo detto che non esiste un operatore per effettuare l'elevamento a potenza. Questo in realtà non è un grosso problema, perché una delle librerie standard del C, il cui file di header si chiama math.h, fornisce una funzione, di nome pow(), che fa proprio l'elevamento a potenza. Qui, comunque, noi ne vedremo una versione semplificata, che ci faremo noi a titolo di esempio. Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

double Potenza(double base,int esponente);

int main(void)
{
    double        base;
```

```
     int           esponente;

     printf("Inserisci la base: ");
     scanf("%lg",&base);
     printf("Inserisci l'esponente (intero): ");
     scanf("%d",&esponente);
     if(esponente<0)
          printf("Solo esponenti positivi sono permessi!\n");
     else if(esponente==0)
     {
          if(base==0)
               printf("Risultato indeterminato.\n");
          else
               printf("Risultato = 1\n");
     }
     else
          printf("Risultato = %g\n",Potenza(base,esponente));

     return(0);
}

double Potenza(double base,int esponente)
{
     int       i;
     double    risultato;

     risultato=1.0;

     for(i=1;i<=esponente;i++)
          risultato*=base;

     return(risultato);
}
```

Salvatelo come Esempio16.c, compilatelo ed eseguitelo a piacimento. Verificate che i conti siano corretti. Come vedete ci sono due novità di rilievo in questo programma. La prima è che abbiamo definito una funzione, `Potenza()`, che richiede due argomenti, il primo di tipo `double` e il secondo di tipo `int` e che restituisce un valore di tipo `double`. Come già ricordato prima, i due argomenti della funzione, `base` ed `esponente` nel nostro caso, sono delle *copie* delle variabili locali della funzione `main()`. Hanno lo stesso nome solo per comodità, ma sono variabili completamente distinte che non si conoscono l'una con l'altra. Se la funzione `Potenza()` modificasse il valore di una di esse, la funzione `main()` non se ne accorgerebbe nemmeno perché, giova ricordarlo, le due funzioni lavorano su copie diverse delle variabili, che potrebbero tranquillamente avere nomi diversi. Nel prossimo capitolo, parlando di puntatori, approfondiremo ancora questo aspetto.

La seconda novità è che la funzione `Potenza()` è chiamata direttamente dall'interno della funzione `printf()`, come argomento della stessa, risparmiandoci la fatica di definire una variabile locale di tipo `double` in cui memorizzare temporaneamente il valore restituito dalla funzione `Potenza()`.

Le chiamate a funzioni standard e del sistema operativo come funzioni

Nel C, quasi tutto è una funzione. Infatti, funzione è il `main()`, funzioni sono quelle che definite voi nei vostri programmi, funzioni sono quelle che vi mettono a disposizione le librerie

che includete nei vostri programmi, come le `printf()` e `scanf()` disponibili nella libreria il cui file di header è il famoso `stdio.h`.

Il meccanismo delle funzioni è anche quello che utilizzate nel momento in cui volete realizzare un programma che sfrutta gli strumenti che vi mette a disposizione il sistema operativo stesso. Ce ne occuperemo nella terza puntata di questa saga, quando parleremo dell'uso di ObjectiveC per programmare Cocoa, l'ambiente nativo delle applicazioni per MacOS X. Anche in questo caso, includerete nel programma delle librerie (frameworks si chiamano in MacOS X) che vi permetteranno di accedere a centinaia di funzioni del sistema operativo che vi permetteranno di aprire finestre, gestire pulsanti e menu, connessioni di rete, timer, dischi, schermo e decine di altre cose. Sarà uno spettacolo. E tutto questo sarà possibile graze a delle semplici funzioni.

Variabili statiche

Abbiamo detto più volte che una variabile locale *vive* fin tanto che l'esecuzione del programma è all'interno del blocco in cui è definita. Questo, ovviamente, vale anche per le variabili locali delle funzioni. Una conseguenza di questo fatto è che quando entrate in una funzione, le sue variabili locali vengono create e voi potete pacioccarle quanto volete, ma quando il programma esce dalla funzione, le sue variabili locali vengono distrutte, e quindi il loro contenuto va perso. Tuttavia, può darsi che vi possa tornare comodo fare sì che il contenuto di una o più variabili locali all'interno di una funzione venga preservato anche quando l'esecuzione del programma non è all'interno di quella funzioni lì. Come fate? Beh, adesso lo vedremo!

Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

int Contatore(void);

int main(void)
{
     int       numero;

     printf("Questo programma conta quante volte\n");
     printf("digitate un numero intero maggiore di 5\n");
     do
     {
          printf("Inserisci un numero intero (zero per uscire): ");
          scanf("%d",&numero);
          if(numero>5)
               printf("Hai  digitato  un  numero  maggiore  di  5  per  %d
volte.\n",Contatore());
     } while(numero!=0);
}

int Contatore(void)
{
     int       contatore=0;

     contatore++;
     return(contatore);
}
```

Salvatelo come Esempio17.c, compilatelo ed eseguitelo. Come? Non funziona? Già; vediamo perché. Quando inserite un numero maggiore di 5, il programma chiama la funzione `Contatore()`. Qui c'è già una novità: la variabile locale `contatore` viene dichiarata come tipo

`int` e, nella stessa riga, viene *inizializzata* al valore zero, ovvero zero è il valore che viene memorizzato nella variabile `contatore` appena essa viene creata. Inizializzare una variabile è una procedura necessaria, perché quando la dichiarate il compilatore C non dà nessuna garanzia su quale sia il suo valore numerico (in linea di massima è qualche cosa di molto diverso da quello che servirebbe a voi). È quindi una cosa saggia assegnare un valore a voi noto e gradito ad ogni variabile che dichiarate, sia a quelle locali che a quelle globali. Per fare ciò potreste semplicemente scrivere:

```
int  contatore;

contatore=0;
```

e sareste soddisfatti. Non ci sarebbe nessuna differenza rispetto all'Esempio17. Se non fosse che l'Esempio17 non funziona! Infatti, la variabile `contatore` è una variabile locale. Quando chiamate la funzione `Contatore()`, la variabile contatore viene creata, inizializzata a zero, incrementata ad 1 e restituita alla funzione `main()`. A questo punto la variabile `contatore` viene distrutta, e il processo si ripeterà quando richiamerete la funzione `Contatore()` la prossima volta. È chiaro che non potrete mai realizzare un contatore funzionante in questa maniera! Potete risolvere il problema rendendo la variabile `contatore` una variabile globale, oppure rendendola una variabile s*tatica*. Modificate allora l'Esempio17 nel seguente Esempio17bis.c:

```
#include <stdio.h>

int Contatore(void);

int main(void)
{
     int       numero;

     printf("Questo programma conta quante volte\n");
     printf("digitate un numero intero maggiore di 5\n");
     do
     {
          printf("Inserisci un numero intero (zero per uscire): ");
          scanf("%d",&numero);
          if(numero>5)
               printf("Hai  digitato  un  numero  maggiore  di  5  per  %d
volte.\n",Contatore());
     } while(numero!=0);
}

int Contatore(void)
{
     static int         contatore=0;

     contatore++;
     return(contatore);
}
```

Compilate, eseguite e sbalordite: funziona!!! Quando dichiarate la variabile `contatore` come *statica* (per mezzo della parolina magica `static`), state dicendo al compilatore che la variabile *non deve essere distrutta* all'uscita della funzione `Contatore()`. Essa deve rimanere lì,

pronta per la prossima volta che entrerete nella funzione `Contatore()`; il suo contenuto sarà preservato immutato per voi.

Va notato che questo trucchetto funziona perché abbiamo deciso di inizializzare la variabile nella stessa riga in cui l'abbiamo dichiarata, con l'istruzione `static int contatore=0`; se avessimo inizializzato la variabile scrivendo l'istruzione `contatore=0;` in una riga della funzione `Contatore()`, il trucchetto non avrebbe funzionato, perché la variabile `contatore` sarebbe sì stata preservata, ma l'istruzione `contatore=0` sarebbe stata eseguita *tutte le volte* che entrate nella funzione `Contatore()`.

Ricorsività

Trattiamo ora molto brevemente un ultimo aspetto riguardante le funzioni, cioè quello della *ricorsività*. È un argomento piuttosto delicato, pertanto prendetelo più che altro come un approfondimento culturale. L'uso della ricorsività è spesso dibattuto, perché introduce un'indubbia difficoltà concettuale, notevoli problemi di uso della memoria (se la profondità della ricorsività è grande) e può portare ad errori del programma estremamente difficili da individuare. Ma può anche essere uno strumento estremamente comodo ed efficace.

Prendete il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

double Fattoriale(int n);

int main(void)
{
     int       numero;

     printf("Questo programma calcola il fattoriale di un numero
intero.\n");
     do
     {
          printf("Inserisci un numero intero (zero per uscire): ");
          scanf("%d",&numero);
          if(numero>0)
               printf("Risultato = %g\n",Fattoriale(numero));
          else if(numero<0)
               printf("Solo numeri positivi sono ammessi!\n");
     } while(numero!=0);
}

double Fattoriale(int n)
{
     int            i;
     double         risultato;

     risultato=(double)n;

     for(i=n-1;i>1;i--)
          risultato*=i;

     return(risultato);
}
```

Salvatelo come Esempio18.c, compilatelo ed eseguitelo (fate attenzione ad inserire solo

numero interi!). Che c'entra questo con la ricorsività, direte voi? Niente! Questo esempio non fa nulla per usare la ricorsività. Si limita a calcolare il fattoriale di un numero n con la ben nota formula: n! = n*(n-1)*(n-2)+…*2*1 iterando con un ciclo `for`. Però possiamo modificare l'Esempio18 in quello che segue:

```
#include <stdio.h>

double Fattoriale(int n);

int main(void)
{
     int      numero;

     printf("Questo programma calcola il fattoriale di un numero
intero.\n");
     do
     {
          printf("Inserisci un numero intero (zero per uscire): ");
          scanf("%d",&numero);
          if(numero>0)
               printf("Risultato = %g\n",Fattoriale(numero));
          else if(numero<0)
               printf("Solo numeri positivi sono ammessi!\n");
     } while(numero!=0);
}

double Fattoriale(int n)
{
     double        risultato;

     risultato=(double)n;

     if(n>1)
          return(risultato*Fattoriale(n-1));
     else
          return(1);
}
```

Salvatelo come Esempio18bis.c, compilatelo ed eseguitelo. Apparentemente funziona alla stessa maniera. E invece, internamente, la funzione `Fattoriale()` è profondamente diversa. In questa versione si fa utilizzo della ricorsività. Che vuol dire? La funzione `main()` chiama la funzione `Fattoriale()` passando, come argomento, il numero intero di cui volete calcolare il fattoriale. La funzione `Fattoriale()`, a sua volta, prende una decisione cruciale: se il numero `n` che le viene passato è uguale ad 1 (non può mai essere uguale a zero o minore di zero perché queste eventualità vengono già filtrate a monte dalla funzione `main()`), sfrutta il fatto che 1! = 1, e restituisce il risultato. Se invece `n` è maggiore di 1, sfrutta la ben nota formula che n! = n*(n-1)! che è a sua volta una formula *ricorsiva*, nel senso che il fattoriale di una certa quantità (n) è definito mediante il fattoriale di un'altra quantità (n-1). Ed ecco che succede la cosa straordinaria. `Fattoriale()` restituisce come risultato il prodotto di `n` con `Fattoriale(n-1)`. Di fatto, `Fattoriale()` chama sé stessa, solo con un altro argometo. La faccenda è ancora più interessante perché non è sempre la stessa funzione `Fattoriale()` ad essere eseguita, ma una sua *copia*, in tutto e per tutto identica all'originale, che però vive una vita sua, indipendente: l'argomento `n` non è più quello che avevate passato dalla funzione `main()`, ma è quello *meno*

uno; la variabile locale `risultato` è un'altra, e se necessario quest'altra copia della funzione `Fattoriale()` richiamerà ricorsivamente sé stessa diminuendo ancora di uno il valore dell'argomento, finché, un bel giorno, l'argomento sarà diventato pari ad uno. A quel punto, l'ultima copia della funzione `Fattoriale()` ad essere stata chiamata saprà che 1! = 1, restituirà il risultato, e la copia precedente potrà prendere il valore del suo argomento (ovvero 2) e moltiplicarlo per il risultato appena ricevuto (ovvero 1). Il totale 2*1=2 verrà restituito alla copia di `Fattoriale()` ancora a monte, che prenderà questo risultato e lo moltiplicherà per il valore del suo argomento (ovvero 3), restituendo il valore 6, e così via, finché si giungerà alla copia di `Fattoriale()` chiamata dalla funzione `main()` che restituirà a quest'ultima il risultato finale.

È un po' complesso, non c'è che dire, e si perde facilmente il conto. Ma è un metodo portentoso. L'uso della ricorsività fornisce ad esempio una soluzione elegante ed efficace per il calcolo dei determinanti.

6. Puntatori ed Array

Puntatori ed Array sono, apparentemente, argomenti molto diversi. E invece, in C (ma non solo in C), sono strettamente legati. Per questa ragione si tende sempre a trattarli insieme. Tuttavia sono anche argomenti estremamente delicati, che portano necessariamente a dover approfondire temi molto rognosi, come la gestione della memoria. Tuttavia, questa *Introduzione al linguaccio C* è, come dice il titolo, un'introduzione, non un trattato; pertanto ci daremo un taglio, e in questo capitolo affronteremo l'argomento dei puntatori e delle array in maniera molto incompleta, limitandoci a parlare di *variabili allocate staticamente e del loro indirizzo*, senza toccare il ben più complesso tema delle *variabili allocate dinamicamente* e della gestione della memoria.

Chiaro? No? Per forza! Se lo fosse, non vi servirebbe leggere questa sbrodolata giunta quasi a pagina 50!

Variabili allocate staticamente e dinamicamente

Ci è successo tante volte, negli esempi precedenti, di dichiarare una variabile, ad esempio `int numero;` e successivamente di usare questa variabile in qualche tipo di espressione, ad esempio assegnandole un valore. Ciò che abbiamo fatto è stato *allocare staticamente* la variabile `numero`. Il programma, una volta compilato, sa che ad un certo punto vi serve la variabile `numero`; sa che essa è di tipo `int`, per cui sa anche quanto spazio bisogna riservare (*allocare*, in gergo) nella memoria del computer affinché nella variabile possa essere memorizzato un numero di tipo `int` (vedi l'Esempio1 per sapere quanta memoria occupano i vari tipi di variabili). Tutta questa procedura è fatta automaticamente: il programma compilato chiede al sistema operativo del computer di poter avere a disposizione un certo numero di byte per la variabile `numero`, il sistema operativo li cerca e li *alloca*, ovvero li riserva perché possano essere usati solo ed esclusivamente dalla variabile `numero`. Voi non sapete esattamente *dove* sarà memorizzata la variabile`numero` nella memoria del computer, né vi interessa saperlo: infatti, il concetto di variabile è un portento proprio per questo: una certa locazione di memoria la chiamate per nome, anziché per indirizzo!

In realtà si può fare anche diversamente: voi potete dire al sistema operativo (mediante opportune istruzioni del C che non vedremo) che vi serve un certo quantitativo di memoria, per farne quello che volete voi. Il sistema operativo ve lo procura, ma ora sta a voi gestirvelo; quello che il sistema operativo vi dà è *l'indirizzo di memoria* che vi è stato assegnato; se volete usarlo, dovete prendere il numero o la lettera o quello che volete e scriverlo *non* dentro una variabile, ma dentro l'indirizzo che vi è stato assegnato. Siccome questo è scomodo, il C vi offre l'alternativa di usare un *puntatore*, ovvero una variabile che, anziché contenere un numero, contiene l'indirizzo al quale vi dovete rivolgere per gestire quella porzione di memoria. È un po' contorto, è vero. Ma qui ne stiamo parlando solo superficialmente. Questa si chiama *allocazione dinamica* di una variabile, ovvero *creazione di puntatori*.

Bene, adesso che ne sappiamo quanto prima, vediamo un po' meglio che cosa sono i puntatori, limitandoci al caso dell'allocazione statica.

Puntatori

L'Esempio16, se ricordate, calcolava la potenza di una certa base con un certo esponente. Non era un esempio molto raffinato, e soffriva, tra le varie cose, di un difetto: la funzione `main()` era responsabile di verificare che l'esponente non fosse negativo. Perché questo sarebbe un difetto? Perché immaginate di avere un programma molto più complesso, dove la funzione `Potenza()` viene chiamata non una, ma molte e molte volte, nei posti più disparati. Ogni volta che la chiamate, dovete preoccuparvi di verificare che l'esponente non sia negativo. È scomodo e soggetto ad errori. Sarebbe molto più semplice se il controllo dell'esponente fosse fatto in un posto solo, ovvero all'interno della funzione `Potenza()` stessa. Così com'è strutturata, però, la funzione `Potenza()` non permette di fare una cosa del genere, perché delle due l'una: o scegliete di far restituire a `Potenza()` il valore del calcolo *base elevato esponente*, e

allora non fate il controllo del segno dell'esponente, oppure fate restituire a `Potenza()` un codice d'errore (ad esempio 0 se va bene e -1 se il segno dell'esponente è negativo), ma allora rinunciate a restituire il risultato del calcolo, che va messo in una variabile globale. Brutta cosa. Ma c'è una soluzione; indovinate un po'? I *puntatori*!

Prendete il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

#define   kNessunErrore          0
#define   kIndeterminato         -1
#define   kEsponenteNegativo     -2

int Potenza(double base,int esponente,double *risultato);

int main(void)
{
     double        base;
     int           esponente;
     double        risultato;

     printf("Inserisci la base: ");
     scanf("%lg",&base);
     printf("Inserisci l'esponente (intero): ");
     scanf("%d",&esponente);
     if(Potenza(base,esponente,&risultato)==kNessunErrore)
          printf("Risultato = %g\n",risultato);
     else
          printf("Dati inseriti sbagliati!\n");

     return(0);
}

int Potenza(double base,int esponente,double *risultato)
{
     int      i;

     *risultato=1.0;

     if(esponente==0)
     {
          if(base==0)
               return(kIndeterminato);
          else
               *risultato=1;
     }
     else if(esponente<0)
          return(kEsponenteNegativo);
     else
     {
          for(i=1;i<=esponente;i++)
               *risultato *= base;
     }
```

```
    return(kNessunErrore);
}
```

Salvatelo come Esempio19.c, compilatelo e divertitevi ad eseguirlo. Quando avete finito, restate con me che vediamo come funziona.

Qui ci sono molte novità. La prima la vedete già nel prototipo della funzione `Potenza()`, ma ce ne occuperemo un po' più tardi (è quell'asterisco che precede l'argomento `risultato`). Partiamo invece dal corpo della funzione `main()`. Questa volta abbiamo deciso di organizzare le cose un po' diversamente: la funzione `Potenza()` restituisce un valore intero, ovvero un messaggio d'errore; se restituisce `kNessunErrore` vuol dire che ha potuto eseguire il conto senza problemi, se no restituisce un valore diverso. Il nostro programma non fa, al momento, una significativa analisi delle situazioni di errore, ma nessuno ci vieterebbe di prendere il valore restituito dalla funzione `Potenza()` (se diverso da `kNessunErrore`) e, all'interno di un blocco `if` o di una funzione apposita, analizzarlo per prendere decisioni opportune e richiedere all'utente le necessarie modifiche ai dati.

Siccome la funzione `Potenza()` non restituisce più il risultato del calcolo e non esiste una variabile globale per memorizzare lo stesso, bisogna trovare un'altra strada. L'altra strada è passare alla funzione `Potenza()` un puntatore. La funzione, infatti, viene chiamata con un argomento in più rispetto all'Esempio16, argomento che, nella funzione `main()`, è individuato dalla variabile `risultato`. Tuttavia, se esso fosse passato così com'è (come ad esempio gli altri due argomenti, `base` ed `esponente`), non saremmo andati molto lontano: la funzione `Potenza()`, lavorando sulla sua copia privata personale dell'argomento `risultato`, non potrebbe assolutamente modificarlo *all'interno della funzione* `main()`, che quindi resterebbe all'oscuro del risultato dell'operazione di elevamento a potenza. Facciamo allra precedere il nome della variabile `risultato` da un segno di e-commerciale `&` (*ampersand*, in inglese), che ha il seguente significato: "cara funzione `Potenza()`, ti passo non già il valore numerico della variabile `risultato`, ma il suo *indirizzo* in memoria; in questa maniera, tu potrai modificare il contenuto di questa variabile perché sai esattamente dove andare a scrivere il nuovo valore che questa variabile dovrà assumere".

Cerchiamo di capire meglio: quando passo ad una funzione un argomento qualunque, il computer mette a disposizione della funzione *una copia* di quell'argomento; la funzione lo può pacioccare quanto vuole, ma si tratta della sua copia personale; l'originale, la variabile che abbiamo passato alla funzione, *non* viene modificata. Quando passiamo ad una funzione l'*indirizzo* in memoria di una variabile mediante il segno *ampersand*, il computer mette a disposizione della funzione *una copia* dell'indirizzo della variabile. E qui avviene il miracolo: la copia dell'indirizzo è, ovviamente, identica all'originale. Se la funzione modificasse questa copia non avremmo risolto niente; ma il trucco è che la funzione non ha bisogno di modificare l'indirizzo, ma ciò che è *contenuto* nell'indirizzo di memoria. La funzione ora *sa* dove andare a trovare, in memoria, il valore della variabile il cui indirizzo le è stato passato per argomento; la funzione, pertanto, può andare a modificare il valore di questa variabile (l'unica copia che esiste, quella all'interno dello *scope* della funzione chiamante, il `main()` nel nostro caso) e rendere disponibile tale modifica alla funzione chiamante.

Nel nostro caso concreto: la variabile `risultato` è locale nella funzione `main()`; passare `&risultato` come argomento alla funzione `Potenza()` fa sì che questa venga a conoscenza dell'*indirizzo* in memoria della variabile `risultato`; benché `risultato` *non sia* una variabile a cui la funzione `Potenza()` può accedere (è al di fuori del suo *scope*), `Potenza()` sa dove essa risiede in memoria, pertanto può andare a scrivere un valore in quell'indirizzo di memoria lì. La funzione `main()`, allora, leggerà il valore della variabile `risultato` che sarà stato modificato dalla funzione `Potenza()` poiché questa ne conosceva l'indirizzo.

Naturalmente, bisogna che `Potenza()`, nel suo prototipo, sia stata dichiarata *in grado di ricevere l'indirizzo di una variabile* `double` *come terzo argomento*. È proprio per fare questo che il terzo argomento della funzione `Potenza()`, nel prototipo, è stato dichiarato come `double *risultato`. Lo scopo di quell'asterisco è proprio questo.

All'interno della funzione `Potenza()`, l'argomento `risultato` (senza asterisco!)

rappresenta *l'indirizzo* della variabile `risultato` della funzione `main()`. Da questo punto di vista, benché i nomi siano uguali, le due variabili solo molto diverse: `risultato` nel `main()` *è una variabile locale di tipo* `double`; `risultato` in `Potenza()` *è una variabile locale che contiene l'indirizzo della variaible* `risultato` *del* `main()`, ovvero *è un puntatore* alla variabile `risultato` del `main()`. Scrivere `*risultato` all'interno della funzione `Potenza()` significa dire che si vuole accedere *al contenuto* della locazione di memoria *il cui indirizzo è memorizzato* nel puntatore `risultato`, variabile locale nella funzione `Potenza()`.

Scusate se sono stato un po' pedante, ma sono concetti un po' ostici e ci si perde facilmente. Quindi: quando passate ad una funzione un argomento che volete che venga modificato, la funzione chiamante lo deve passare *per indirizzo* mediante la preposizione dell'*ampersand*; la funzione ricevente lo deve accettare come *puntatore* mediante la preposizione dell'asterisco, e modifica il contenuto della variabile sempre anteponendo l'asterisco al nome del puntatore.

Tutto questo discorso non è rigorosissimo, ma, ribadisco, questa è un'introduzione, non un trattato, al linguaggio C.

Se avete ancora due minuti di pazienza vi propino un terzo *box di approfondimento*:

Il Sistema Operativo e i puntatori

Anche se voi, spaventati a morte da questi benedetti puntatori, doveste decidere che mai e poi mai li userete, vi dovreste ricredere molto in fretta. I puntatori, infatti, sono il metodo preferito dalle librerie e dai sistemi operativi per dialogare con il vostro programma. Senza andare a fare una lunga trattazione che esulerebbe da questa *Introduzione*, mi limito a farvi notare quanto più volte abbiamo scritto nei nostri esempi, ovvero (ad esempio): `scanf("%d",&numero)` avendo dichiarato `numero` come una variabile di tipo `int`. Avete notato l'*ampersand*? State passando alla funzione `scanf()`, definita nella solita libreria `stdio`, una variabile intera *per indirizzo*. Questo perché voi avete*bisogno* che la funzione `scanf()` la modifichi, dal momento che intendente memorizzare nella variabile `numero` il numero intero che l'utente digiterà da tastiera. Se non passaste la variabile `numero` per indirizzo alla funzione `scanf()`, questa non avrebbe modo di farvi sapere quale numero l'utente abbia effettivamente digiato. Siano benedetti i puntatori!

Array

Che cosa c'entrano, in tutto questo, le array? E poi, che cosa sono le array? La risposta alla prima domanda è difficile, e, se permettete, la rinvieremo a un po' più avanti. Per ora tenteremo di rispondere alla seconda domanda, limitandoci a parlare di array allocate staticamente.

Allora, il Servizio Meteorologico Nazionale vi ha commissionato la creazione di un programma il cui scopo è calcolare la temperatura media registrata in una certa località nell'arco di una settimana a mezzogiorno in punto. Lusingati da cotanto incarico, prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

#define   kGiorniInUnaSettimana   7

// variabili globali
double    temperature[kGiorniInUnaSettimana];

// prototipi
void InserisciDati(void);
```

```
double CalcolaMedia(void);
void MostraRisultato(double media);

int main(void)
{
     InserisciDati();
     MostraRisultato(CalcolaMedia());

     return (0);
}

void InserisciDati(void)
{
     int           i;

     for(i=0;i<kGiorniInUnaSettimana;i++)
     {
          printf("Inserisci la temperatura di mezzogiorno\n");
          printf("per il %do giorno della settimana: ",i+1);
          scanf("%lg",&temperature[i]);
     }
}

double CalcolaMedia(void)
{
     double        media;
     int           i;

     media=0.0;
     for(i=0;i<kGiorniInUnaSettimana;i++)
          media+=temperature[i];
     media/=kGiorniInUnaSettimana;
     return(media);
}

void MostraRisultato(double media)
{
     printf("La temperatura media e' stata di %g gradi\n",media);
}
```

Salvatelo come Esempio20.c, compilatelo ed eseguitelo a piacimento. Poi rileggetelo per bene con me. La prima novità sta nel come è definita la variabile globale `temperature`; il suo nome è infatti seguito da una coppia di parentesi quadre, all'interno delle quali è contenuto un numero *intero* (rappresentato da una costante, nel nostro caso). Questo è il modo con cui si dichiarano le array. Sostanzialmente si tratta di una variabile che, come al solito, è identificata da un nome; tuttavia, essa può memorizzare *più valori* tutti dello stesso tpo (`double`, nel nostro esempio), anziché uno solo come per le variabili normali. Per fare questo, al nome della variabile si associa un indice, che viene racchiuso tra parentesi quadre, il cui valore minimo è sempre zero e il cui valore massimo è pari al numero racchiuso tra parentesi quadre quando dichiarate l'array *meno uno* (`kGiorniInUnaSettimana` -1, ovvero 6 nel nostro esempio). Assegnate valori ad un'array e leggete valori da questa come fareste con tutte le altre variabili, ricordatevi solo di specificare tra partentesi quadre a quale indice vi state riferendo.

Siccome i giorni della settimana sono 7, abbiamo scelto di dichiarare la variabile

`temperature` come avente un indice compreso tra 0 e 6 inclusi (ovvero 7 valori possibili), così da poter memorizzare in corrispondenza di ciascun indice la temperatura raggiunta a mezzogiorno nella località di interesse in ciascuno dei giorni della settimana. È quanto viene fatto dalla funzione `InserisciDati()`: si passa alla `scanf()` l'indirizzo dell'elemento `i`-esimo dell'array `temperature` affinché possa essere memorizzato in esso il valore della temperatura registrata nel giorno `i`-esimo. Vorrei farvi notare la comodità di questo approccio: senza array, saremmo stati costretti a definire 7 variabili globali di tipo `double` in cui memorizzare singolarmente le temperature di ognuno dei 7 giorni, ripetendo per ben 7 volte nella funzione `InserisciDati()` la richiesta di inserire il numero e l'assegnazione del numero inserito nella variabile corrispondente (senza array non sarebbe possibile usare un indice e le variabili andrebbero tutte scritte singolarmente). Questo, per 7 volte forse si può anche fare. Ma che succede se volete calcolare la temperatura media di un anno intero leggendo magari i dati da un file? Scrivete 365 volte l'istruzione di assegnazione di un valore ad una delle 365 variabili distinte che vi servono per memorizzare le temperature? (per non parlare degli anni bisestili!) Con un'array, vi basta modificare il valore della costante che definisce la *dimensione* dell'array (`kGiorniInUnaSettimana` nel nostro caso) e non avete bisogno di altre modifiche! Provare per credere: modificate `kGiorniInUnaSettimana` facendolo diventare pari a 10 e vedrete che il programma continuerà a funzionare benissimo, solo che la settimana sarà di 10 giorni. Ricordate comunque sempre che *è un errore ed è comunque potenzialmente pericoloso* (nel senso che potreste inchiodare il programma o il sistema) tentare di leggere o scrivere un'array con un valore dell'indice minore di zero o maggiore del massimo valore permesso, che è quello con cui avete dichiarato l'array *meno uno*.

Tutto questo, con i puntatori sembra azzeccarci poco. E invece…

Array come puntatori

Un'array è dichiarata come una roba del genere: `int pippo[5];` questa è un'array che può memorizzare 5 numeri interi di tipo `int` identificati da un indice che va da 0 a 4 inclusi. Se volete scrivere il numero 25 nell'array in corrispondenza dell'indice che vale 3 scrivete semplicemente `pippo[3]=25;` se volete leggere il valore contenuto in corrispondenza dell'indice pari a 0 scrivete semplicemente `pippo[0];` è tutto qui? No!

Se prendete il *nome* dell'array *senza* farlo seguire dalla coppia di parentesi quadre, esso è in realtà un puntatore! Scrivere `pippo`, ad esempio, rappresenterebbe *l'indirizzo di memoria* in cui è memorizzato *il primo elemento dell'array* (quello con indice pari a zero). Scrivere`*pippo` vuol dire leggere *il contenuto* dell'indirizzo di memoria in cui c'è il *primo elemento* dell'array, ovvero è equivalente a scrivere `pippo[0]. *(pippo+1)` è equivalente a `pippo[1]` e via discorrendo. Carino, vero?

L'equivalenza tra array e puntatori, che qui abbiamo appena accennato e che in realtà è una cosa molto più complessa di quando detto fin qui, ci permette di trattare l'ultimo punto importante di questo capitolo, ovvero *le stringhe*. Con questo termine si intendono *sequenze di caratteri*, variabili il cui contenuto non è *un solo* carattere, come quelle di tipo `char`, ma sequenze di caratteri.

Beh, se anziché dire *sequenza* di caratteri dicessi *array* di caratteri, non credo che gridereste allo scandalo. E infatti una stringa è proprio questo: una sequenza ordinata di caratteri che, volendo, sono identificabili da un indice (la loro posizione). Sì, è decisamente un'array.

Solo che, convenzionalmente, le stringe si trattano come puntatori (tanto sono la stessa cosa delle array!), per una ragione molto semplice: trattarle come puntatori veri e propri permette di poter gestire stringhe di lunghezza arbitraria. In questo capitolo, invece, non ci siamo occupati di allocazione dinamica delle variabili, e anche nel prossimo esempio ci limiteremo all'allocazione statica. Questo comporterà il fatto che le nostre stringhe avranno sempre una lunghezza massima che non potremo superare. Poco male: cercheremo di rendere la lunghezza massima sufficientemente grande da farci stare tutto quello che ci serve.

Prendete il vostro editor di testo preferito e digitate il seguente codice:

```
#include <stdio.h>

#define   kGiorniInUnaSettimana   7

// variabili globali
double    temperature[kGiorniInUnaSettimana];
char
     *nomi[kGiorniInUnaSettimana]={"lunedi'","martedi'","mercoledi'","giove
di'","venerdi'","sabato","domenica"};
char luogo[30];

// prototipi
void InserisciDati(void);
double CalcolaMedia(void);
void MostraRisultato(double media);

int main(void)
{
     InserisciDati();
     MostraRisultato(CalcolaMedia());

     return (0);
}

void InserisciDati(void)
{
     int           i;

     printf("Come si chiama la localita'?\n");
     scanf("%s",luogo);
     for(i=0;i<kGiorniInUnaSettimana;i++)
     {
          printf("Inserisci la temperatura di mezzogiorno\n");
          printf("per %s: ",nomi[i]);
          scanf("%lg",&temperature[i]);
     }
}

double CalcolaMedia(void)
{
     double        media;
     int           i;

     media=0.0;
     for(i=0;i<kGiorniInUnaSettimana;i++)
          media+=temperature[i];
     media/=kGiorniInUnaSettimana;
     return(media);
}

void MostraRisultato(double media)
```

```
{
    printf("La temperatura media per la localita' di\n");
    printf("%s e' stata di %g gradi\n",luogo,media);
}
```

Salvatelo come Esempio21.c, compilatelo ed eseguitelo. Avete notato i miglioramenti? Commentiamoli insieme! Abbiamo aggiunto due nuove variabili globali, `nomi` e `luogo`. La seconda (`luogo`) è un'array di caratteri nella quale vogliamo memorizzare il nome della località di cui stiamo calcolando la temperatura media di mezzogiorno nell'ultima settimana. Dal momento che non siamo capaci di allocare dinamicamente la memoria, abbiamo deciso di limitare la stringa `luogo` a 30 caratteri al massimo, confidando che qualunque località, città, borgata e frazione abbia un nome non più lungo di 30 caratteri. Se dovesse risultare insufficiente, potremo sempre aumentare questo valore e ricompilare il programma.

La variabile `nomi`, invece, è un po' più complicata: essa è un'array di puntatori a `char`, ovvero un'array di stringhe, se volete un'array di array di caratteri. Anche qui, gestire per benino questa roba non è facile e richiederebbe allocazione dinamica della memoria, che noi non facciamo. Allora usiamo un trucco: sfruttiamo il fatto che i nomi della settimana già li conosciamo, e li scriviamo direttamente dove dichiariamo la variabile `nomi`. Questo è un altro modo per assegnare valori ad un'array: gli elementi sono elencati tra parentesi graffe e separati da una virgola; se gli elementi sono stringhe, vanno scritti tra virgolette. Se dovessimo mai modificare il valore della costante `kGiorniInUnaSettimana`, ricordiamoci solo di modificare di conseguenza anche i nomi dei giorni della settimana! Ogni elemento dell'array `nomi` è un'array di caratteri la cui dimensione (il valore massimo che può assumere l'indice, ovvero il numero massimo di caratteri che possono essere memorizzati nell'array) è calcolato automaticamente a partire dai nomi che abbiamo scritto esplicitamente tra parentesi graffe. Se non avessimo fatto così, gestire i nomi dei giorni della settimana sarebbe stato più complicato e avrebbe probabilmente richiesto allocazione dinamica della memoria. Ma nel prossimo capitolo troveremo un'elegantissima soluzione al problema definendo una *struttura*!

7. Strutture

Soddisfatto dal vostro programma per calcolare la temperatura media di una località nell'arco di una settimana (vedi Esempio21), il Servizio Meteorologico Nazionale vi affida un altro importante incarico: migliorare il vostro programma affinché possa essere calcolata anche l'umidità media. Beh, direte voi, è facile: creo un'altra array di valori `double` in cui memorizzare l'umidità, chiedo all'utente di inserire i dati così come ho già fatto per le temperature, calcolo le medie anche sull'array dell'umidità e il gioco è fatto. Vero; ma non è elegante.

Il C vi mette a disposizione un metodo molto più elegante per fare questa cosa; talmente elegante che, nella prossima puntata in cui ci occuperemo di ObjectiveC, scopriremo che è stato preso a modello per creare gli *oggetti* di cui sono costituiti, per l'appunto, tutti i programmi scritti con linguaggi *ad oggetti*: un esempio? C++ e ObjectiveC. Questo metodo elegante è costituito dalle *strutture*.

Prendete il vostro editor di testo preferito e digitate il seguente programma:

```
#include <stdio.h>

#define   kGiorniInUnaSettimana   7

// variabili globali
struct
{
     double    temperatura;
     double    umidita;
     char      *nome;
} datiGiornalieri[kGiorniInUnaSettimana];

char luogo[30];

// prototipi
void InizializzaDati(void);
void InserisciDati(void);
void CalcolaMedie(double *temperatura,double *umidita);
void MostraRisultato(double tMedia,double uMedia);

int main(void)
{
     double        t,u;

     InizializzaDati();
     InserisciDati();
     CalcolaMedie(&t,&u);
     MostraRisultato(t,u);

     return (0);
}

void InizializzaDati(void)
{
     int       i;

     for(i=0;i<kGiorniInUnaSettimana;i++)
```

```
     {
          switch(i)
          {
               case 0:
                    datiGiornalieri[i].nome="lunedi'";
                    break;
               case 1:
                    datiGiornalieri[i].nome="martedi'";
                    break;
               case 2:
                    datiGiornalieri[i].nome="mercoledi'";
                    break;
               case 3:
                    datiGiornalieri[i].nome="giovedi'";
                    break;
               case 4:
                    datiGiornalieri[i].nome="venerdi'";
                    break;
               case 5:
                    datiGiornalieri[i].nome="sabato";
                    break;
               case 6:
                    datiGiornalieri[i].nome="domenica";
                    break;
          }
     }
}

void InserisciDati(void)
{
     int           i;

     printf("Come si chiama la localita'?\n");
     scanf("%s",luogo);
     for(i=0;i<kGiorniInUnaSettimana;i++)
     {
          printf("Inserisci la temperatura di mezzogiorno\n");
          printf("per %s: ",datiGiornalieri[i].nome);
          scanf("%lg",&datiGiornalieri[i].temperatura);

          printf("Inserisci l'umidita' di mezzogiorno\n");
          printf("per %s: ",datiGiornalieri[i].nome);
          scanf("%lg",&datiGiornalieri[i].umidita);
     }
}

void CalcolaMedie(double *temperatura,double *umidita)
{
     double        t,u;
     int           i;

     t=0.0;
```

```
     u=0.0;
     for(i=0;i<kGiorniInUnaSettimana;i++)
     {
          t+=datiGiornalieri[i].temperatura;
          u+=datiGiornalieri[i].umidita;
     }
     t/=kGiorniInUnaSettimana;
     u/=kGiorniInUnaSettimana;
     *temperatura=t;
     *umidita=u;
}

void MostraRisultato(double tMedia,double uMedia)
{
     printf("\n");
     printf("La temperatura media per la localita' di\n");
     printf("%s e' stata di %g gradi.\n",luogo,tMedia);
     printf("L'umidita' media per la localita' di\n");
     printf("%s e' stata del %g per cento.\n",luogo,uMedia);
}
```

Salvatelo come Esempio22.c, compilatelo e giocateci un po'. Adesso vediamo come funziona: come vedete, tra le variabili globali abbiamo fatto dei cambiamenti. In particolare, abbiamo definito una nuova array globale, chiamata `datiGiornalieri`, il cui tipo è una struttura (`struct{}`). Una struttura è un modo per raggruppare variabili che può avere senso mettere insieme. Nel nostro caso, ad esempio, abbiamo fatto così: siccome, per ogni giorno, dobbiamo sapere tre cose (nome del giorno, temperatura a mezzogiorno e umidità a mezzogiorno), perché creare tre array diverse? Ne creiamo una sola, `datiGiornalieri` per l'appunto, che, essendo definita come una struttura, può contenere per ogni suo elemento (per ogni giorno, quindi), tutte le informazioni che ci interessa memorizzare. Abbiamo sostanzialmente aggiunto un livello gerarchico: ora la variabile globale è l'array `datiGiornalieri`, i cui elementi sono *tutto ciò che ci interessa sapere per ogni giorno della settimana*. Ognuno di questi elementi, a sua volta, contiene una collezione di dati, nel nostro caso, nome del giorno, temperatura ed umidità a mezzogiorno.

Ci serve una funzione `InizializzaDati()` solo per scrivere il corretto nome del giorno della settimana in corrispondenza del corrispondente elemento dell'array `datiGiornalieri`.

Poi si procede grosso modo come al solito: l'utente inserisce i dati relativi a temperatura e umidità, ma questa volta dobbiamo accedere ad un *elemento* della struttura corrispondente al giorno in questione. Ecco allora che la scrittura `datiGiornalieri[i].temperatura` indica che vogliamo leggere o scrivere (a seconda che mettiamo l'uguale a sinistra o a destra) la variabile `temperatura` della struttura corrispondente al giorno `i`-esimo rappresentato dall'elemento `i` dell'array `datiGiornalieri`. Idem dicasi per le variabili rappresentanti il nome del giorno e l'umidità.

Il resto dei cambiamenti nelle funzioni `CalcolaMedie()` e `MostraRisultato()` è dovuto sostanzialmente al fatto che ora dobbiamo calcolare due valori, e quindi siamo costretti ad usare dei puntatori dal momento che una funzione non può ritornare più di un valore (se no bisognava definire due funzioni distinte, una per calcolare la media della temperatura e un'altra per calcolare la media dell'umidità).

Una struttura può avere un numero arbitrario di elementi, ogni elemento può essere di qualunque tipo e potrebbe essere persino una struttura.

Naturalmente potete definire puntatori a strutture, strutture auto-referenziali e ogni genere di amenità. Ma qui non ce ne occuperemo.

8. printf() e scanf(). rudimenti

Le funzioni `printf()` e `scanf()` che abbiamo usato per tutta questa *Introduzione al linguaggio C* sono molto complesse e padroneggiarle completamente richiede parecchia fantasia e spirito masochistico. Qui *non* le tratteremo in maniera esaustiva. Ci limiteremo a dare i commenti necessari per spiegare tutto e solo quello che avete visto in questa prima puntata della trilogia per quanto riguarda queste due funzioni.

Innanzitutto una cosa che ho già detto più volte: se volete usare una o entrambe queste funzioni,*dovete* includere nel vostro programma la libreria `stdio` mediante l'istruzione `#include <stdio.h>` posta all'inizio del vostro programma.

printf()

Il prototipo della funzione `printf()` è:

```
int printf(char *formato,arg1,arg2,...);
```

La stringa `formato` è una stringa vera e propria, ovvero una sequenza di caratteri racchiusi tra virgolette doppie. Tutti i caratteri che sono racchiusi tra le virgolette verranno scritti a schermo sulla finestra del terminale. Caratteri speciali come `\"` e `\n` indicano rispettivamente che volete scrivere una virgoletta doppia (se non fosse preceduta dalla barra inversa verrebbe interpretata come la virgoletta di chiusura della stringa) oppure un fine-riga (a-capo).

Opzionalmente, la stringa `formato` può contenere dei segni percento `%` che, seguiti da un carattere, specificano delle cose carine:

`%d` indica che volete scrivere il valore di una variabile di tipo `int`;

`%c` indica che volete scrivere il valore di una variabile di tipo `char`;

`%s` indica che volete scrivere il valore di una variabile stringa (tipo `char *`);

`%g` indica che volete scrivere il valore di una variabile di tipo `double`.

In realtà la faccenda è molto più complicata, ma qui ho riportato solo le cose che abbiamo usato noi nei nostri esempi. Le variabili di cui volete scrivere il valore sono elencate, *nell'ordine con cui compaiono gli indicatori di formato* (quelle sequenze `%` più `d`, `c`, `s` o `g`), dopo la stringa di formato, separate tra di loro da una virgola. Vi rimando a tutti gli esempi, a partire dall'Esempio1.

La funzione `printf()` restituisce il numero che caratteri scritti sullo schermo.

scanf()

La funzione `scanf()` è assolutamente identica a `printf()` ed ha lo stesso prototipo (non è vero, ma noi l'abbiamo usata così). Le uniche differenze sono:

1. viene usata per consentire all'utente di *inserire dati* e *memorizzarli in variabili*;
2. le variabili di tipo `double` sono indicate nella stringa `formato` dalla sequenza `%lg`;
3. *tutte* le variabili passate come argomento dopo la stringa formato *devono* essere puntatori (fate precedere il nome della variabile dal simbolo *ampersand* `&`).

Ce ne sarebbe molto di più da dire, ma ci vorrebbe un tutorial solo per queste due funzioni!

Appendice. Usiamo uno straccio di file di header

"Ma come?", direte voi, "Ci hai rotto tanto per questi dannati file di header, e poi, nei tuoi esempi, non ne hai usato nemmeno uno!" (a parte l'header stdio.h, naturalmente, ma non l'abbiamo fatto noi!).

Vero. E allora, per accontentarvi, ecco qui, il famosissimo e introvabile Esempio23. Prendete il vostro editor di testo preferito e digitate il seguente file di header:

```
#include <stdio.h>

#define   kGiorniInUnaSettimana   7

// variabili globali
struct
{
     double    temperatura;
     double    umidita;
     char *nome;
} datiGiornalieri[kGiorniInUnaSettimana];

char luogo[30];

// prototipi
void InizializzaDati(void);
void InserisciDati(void);
void CalcolaMedie(double *temperatura,double *umidita);
void MostraRisultato(double tMedia,double uMedia);
```

Salvatelo come Esempio23.h. Poi, usate ancora il vostro editor di testo preferito e digitate il seguente codice:

```
#include <Esempio23.h>

int main(void)
{
     double        t,u;

     InizializzaDati();
     InserisciDati();
     CalcolaMedie(&t,&u);
     MostraRisultato(t,u);

     return (0);
}

void InizializzaDati(void)
{
     int      i;

     for(i=0;i<kGiorniInUnaSettimana;i++)
     {
          switch(i)
```

```
          {
               case 0:
                    datiGiornalieri[i].nome="lunedi'";
                    break;
               case 1:
                    datiGiornalieri[i].nome="martedi'";
                    break;
               case 2:
                    datiGiornalieri[i].nome="mercoledi'";
                    break;
               case 3:
                    datiGiornalieri[i].nome="giovedi'";
                    break;
               case 4:
                    datiGiornalieri[i].nome="venerdi'";
                    break;
               case 5:
                    datiGiornalieri[i].nome="sabato";
                    break;
               case 6:
                    datiGiornalieri[i].nome="domenica";
                    break;
          }
     }
}

void InserisciDati(void)
{
     int           i;

     printf("Come si chiama la localita'?\n");
     scanf("%s",luogo);
     for(i=0;i<kGiorniInUnaSettimana;i++)
     {
          printf("Inserisci la temperatura di mezzogiorno\n");
          printf("per %s: ",datiGiornalieri[i].nome);
          scanf("%lg",&datiGiornalieri[i].temperatura);

          printf("Inserisci l'umidita' di mezzogiorno\n");
          printf("per %s: ",datiGiornalieri[i].nome);
          scanf("%lg",&datiGiornalieri[i].umidita);
     }
}

void CalcolaMedie(double *temperatura,double *umidita)
{
     double        t,u;
     int           i;

     t=0.0;
     u=0.0;
     for(i=0;i<kGiorniInUnaSettimana;i++)
```

```
    {
        t+=datiGiornalieri[i].temperatura;
        u+=datiGiornalieri[i].umidita;
    }
    t/=kGiorniInUnaSettimana;
    u/=kGiorniInUnaSettimana;
    *temperatura=t;
    *umidita=u;
}

void MostraRisultato(double tMedia,double uMedia)
{
    printf("\n");
    printf("La temperatura media per la localita' di\n");
    printf("%s e' stata di %g gradi.\n",luogo,tMedia);
    printf("L'umidita' media per la localita' di\n");
    printf("%s e' stata del %g per cento.\n",luogo,uMedia);
}
```

Salvatelo come Esempio23.c, compilate ed eseguite. Come? È identico all'Esempio22? Infatti! Solo che qui abbiamo fatto uso di una tecnica diffusa e, direi, *essenziale* quando le dimensioni dei programmi iniziano a farsi ragguardevoli: mettere in un file di header (estensione .h) le direttive `#include` delle librerie da usare, le costanti, le variabili globali e i prototipi. Poi nel (o nei!) file contenenti il codice, limitarsi ad includere il file di header necessario (`#include <Esempio23.h>`, nel nostro caso). Se lavorate in squadra, o se il vostro programma è fatto da più file di codice (sarà molto presto così, credetemi), creare file di header e tenerli in ordine sarà fondamentale per non perdere la salute mentale.

Bibliografia

Ce n'è un mucchio, in tutte le lingue. Ma il testo fondamentale per chi vuole imparare il C resta comunque e sempre (secondo me, ovviamente), il

B.W. Kernighan, D.M. Ritchie, *Linguaggio C*, seconda edizione, Gruppo Editoriale Jackson

Su questo libro trovate anche una descrizione sommaria di tutte le funzioni delle librerie standard del C.