C

Guida alla programmazione

INDICE

LINGUAGGIO C

Dotato di tutte le maggiori potenzialità per costruire applicazioni veloci e funzionali. I programmi scritti in C possono essere utilizzati su molti sistemi operativi e sono caratterizzati da velocità e stabilità. La guida di HTML.it (a cura di **Fabrizio Ciacchi** *virusbye@html.it)* vi permetterà di affrontare con chiarezza le sue maggiori funzionalità.

1. Prima di iniziare

In questa guida ci proponiamo come obiettivo quello di insegnare a programmare in uno dei linguaggi più famosi ed utilizzati al mondo, il **C**.

Uno dei motivi per cui, ad oggi, è utile studiare il C è il fatto che esistono già migliaia di righe di codice scritte che permettono di risolvere quasi tutti i problemi legati alla programmazione (algoritmi, strutture, ecc.). Il C è un linguaggio che, come il Pascal ed il Fortran (che sono suoi predecessori), permette di salvare i valori in variabili, di strutturare il codice, di convogliare il flusso del programma utilizzando istruzioni di ciclo, istruzioni condizionali e funzioni, di eseguire operazioni di input/output a video o su file, di salvare dati in array o strutture; ma diversamente da questi linguaggi (e qui i suoi maggiori punti di forza) permette di controllare in modo più preciso le operazioni di input/output, inoltre il C è un linguaggio più sintetico e permette di scrivere programmi piccoli e di facile comprensione.

Per chi si stesse chiedendo q**uali siano le applicazioni reali del linguaggio C**, basti pensare che con il C si possono sviluppare programmi di qualsiasi genere, compresi i videogiochi; inoltre praticamente tutti i sistemi operativi sono sviluppati per la maggior parte in Assembler ed in C (uno su tutti il Kernel Linux), anche se negli ultimi anni viene incluso sempre di più codice scritto in C++ (nda: leggesi C plus plus).

Un **neo-programmatore potrebbe essere impaurito** dal linguaggio C, credendolo poco avanzato e meno utile di linguaggi più moderni come il Visual Basic, il Delphi o il Java; tutto ciò non è vero, il C insegna a programmare, a pensare a come impostare il codice, a risolvere i problemi che ci vengono messi di fronte, e questo bagaglio culturale servirà sicuramente in futuro; infatti una volta imparato il C troverete molto più facile imparare un altro linguaggio di programmazione e, addirittura, le basi insegnate verranno ritrovate in linguaggi come il C++, il Java, il PHP, il JavaScript, ecc.

A coloro che stanno leggendo questa guida e non hanno la minima esperienza di programmazione, si consiglia di leggere la Guida di base alla Programmazione in modo da poter comprendere meglio quello che diremo nelle prossime lezioni.

2. Programma compilato e interpretato

Un programma viene sviluppato **scrivendo il codice sorgente** in un opportuno linguaggio definito, appunto, dalla sintassi del linguaggio stesso. La differenza tra compilazione ed interpretazione è molto importante ed influisce sia sulle prestazioni che sulle possibilità del linguaggio stesso.

Linguaggi come il C, C++, Delphi, Visual Basic sono **linguaggi compilati** e seguono questi passi: si scrive il codice in un editor, al più utilizzando un ambiente di sviluppo IDE che ne facilita la creazione, questo codice viene poi controllato per verificare che non ci siano errori e poi viene compilato, ovvero ogni istruzione viene trasformata nel corrispondente codice in linguaggio macchina che può essere, così, eseguito dal processore; questi sono i linguaggi compilati che vengono detti anche linguaggi imperativi, ed hanno il vantaggio di prestazioni migliori.

I **linguaggi interpretati**, invece, seguono una strada diversa, il codice sorgente viene, appunto, interpretato al volo e vengono, quindi, eseguite le istruzioni così come descritte nel codice sorgente; un esempio su tutti è il PHP il cui codice viene "elaborato" e restituisce una pagina html pura. La potenza di questo genere di linguaggi è, di fatto, l'alta portabilità e l'immediatezza tra quello che scriviamo e quello che viene presentato all'esecuzione del programma, ma rimangono dei problemi come la ricerca di errori nel codice sorgente o il carico di lavoro maggiore per il processore (che ogni volta deve elaborare la pagina).

Un **linguaggio che è a metà strada tra queste metodologie è il Java** che è sia compilato che interpretato; il codice sorgente viene compilato in un formato intermedio (chiamato bytecode), il quale a sua volta viene interpretato dalla Java Virtual Machine (JVM), che ha il compito di interpretare "al volo" le istruzioni bytecode in istruzioni per il processore; la JVM viene sviluppata per ogni Sistema Operativo e permette di astrarre la macchina virtuale creata dal SO ad un livello di standardizzazione superiore (data di fatto dalla creazione della virtual machine sopra un'altra virtual machine) che rende, in pratica, il JAVA altamente portabile.

Questa metodologia implica la possibilità di controllare eventuali errori del codice sorgente (grazie alla compilazione), di creare programmi relativamente leggeri (il bytecode è un formato che crea file di dimensioni ragionevoli), ma ha la pecca di avere delle prestazioni non proprio soddisfacenti, questo perché il codice viene interpretato dalla JVM che a sua volta deve delegare l'esecuzione vera e propria al Sistema Operativo.

3. Le peculiarità del C

Quali sono le peculiarità del C? Sono molte e qui di seguito elenchiamo quelle più importanti cercando di chiarirne il significato.

- **Dimensioni del codice ridotte** - Generalmente il codice sorgente di un programma in C ha un peso (in Kb) relativamente piccolo, in questo modo risulta molto agevole trasportare il codice da un PC ad un altro, anche usando un semplice floppy.
- **Dimensioni dell'eseguibile ridotte** - Anche una volta compilato, un programma in C, risulta molto piccolo e quindi di più facile diffusione; ovviamente un programma in C potrà essere eseguito solamente sul medesimo Sistema Operativo per cui è stato compilato.
- **Efficienza dei programmi** - Un programma scritto in C, proprio per la possibilità messa a disposizione dal linguaggio di gestire a fondo la memoria, e per le sue dimensioni ridotte, risulta particolarmente efficiente.
- **Può essere compilato su una vasta gamma di computer** - Ogni computer può differire dagli altri per almeno due cose, l'architettura ed il sistema operativo; ad esempio un computer con processore x86 e Windows ha delle istruzioni (intese come istruzioni del processore) ed una gestione della memoria diverse da uno Sparc con Linux, però un programma scritto in C può essere compilato su ambedue le macchine, data l'alta disponibilità di compilatori per diverse piattaforme. Certo non è "portabile" come Java, però il fatto di essere sulla scena da molti anni e la sua enorme diffusione ne fanno, di fatto, uno strumento altamente portabile.

- **È un linguaggio di alto livello** - Un linguaggio di programmazione viene definito di alto livello tanto più si avvicina alla terminologia umana, inversamente si dice che un linguaggio è di basso livello se il suo codice si avvicina al linguaggio macchina (quello formato da 0 ed 1); tipico esempio di linguaggio a basso livello è l'Assembler, mentre linguaggi ad alto livello sono, oltre al C, il C++, il Java e molti altri. La particolarità dei linguaggi ad alto livello è quella di avere una semplice sintassi in cui si usano parole della lingua inglese per descrivere comandi corrispondenti a decine di istruzioni in assembler o centinaia di istruzioni in linguaggio macchina.
- **Può maneggiare attività di basso livello** - Il C è considerato il linguaggio di più basso livello tra i linguaggi di alto livello. Questo è dovuto al fatto che ha poche istruzioni, gestisce in maniera efficiente la memoria ed è possibile inserire direttamente all'interno di un file in C del codice Assembler.
- **Implementazione dei puntatori** - Il C fa un largo uso di puntatori per le operazioni riguardanti la memoria, gli array, le strutture e le funzioni.
- **Loose Typing** - In poche parole in C i tipi di dato delle variabili non devono necessariamente essere dichiarati; contrariamente, nel Pascal, nel C++ e nel Java i tipi devono essere dichiarati (strong typing); tale differenza risiede nelle scelte decisionali del progettista del linguaggio, che può volere più o meno flessibilità all'interno dello stesso (compilando un programma Java, la maggior parte delle volte si commettono errori di tipo). C'è chi dice che il metodo loose typing permetta al programmatore di commettere errori di tipo che potrebbero compromettere l'esecuzione del programma, ma è anche vero che con lo strong typing siamo costretti a seguire delle regole rigide che, a volte, non permettono di risolvere "velocemente" un problema.

Nelle lezioni seguenti parleremo della storia del linguaggio C, procedendo poi verso gli strumenti di compilazione disponibili per i vari Sistemi Operativi ed al primo programmino (il famoso Hello World) scritto in C; la parte centrale della guida è tutta incentrata sulla sintassi del linguaggio, in ordine, le variabili, gli operatori, le istruzioni condizionali (come l'IF-ELSE), le istruzioni iterative (FOR e WHILE), gli array, le funzioni, i puntatori, le liste, l'allocazione dinamica della memoria, l'input/output su file e l'utilizzo del pre-processore C; l'ultima parte spiega in dettaglio lo stile per scrivere un programma, l'uso degli header e del Make per la creazione di makefile (sotto Linux), l'utilizzo di funzioni speciali disponibili sotto Linux, le opzioni del compilatore GCC sotto Linux e le funzioni disponibili nelle librerie standard del linguaggio.

Sarà, invece, omessa, volutamente, la trattazione dell'istruzione **goto**, questo perché è un'istruzione poco utilizzata e può portare ad imparare uno stile di programmazione poco chiaro e facilmente soggetto ad errori; tutte le nozioni date in queste lezioni permettono di risolvere tutti i problemi risolvibili con un goto in modo più elegante e più corretto.

4. Storia del linguaggio C

Per capire le origini del C, bisogna parlare della storia del sistema operativo UNIX in quanto è stato sviluppato su questo sistema operativo ed, anzi, lo stesso sistema operativo ed i suoi programmi sono scritti in C.

Nonostante tutto il linguaggio C non è, però, legato solo alla programmazione dei sistemi operativi, ma con esso possono essere scritti (e sono stati scritti) grossi programmi di calcolo, word-processing e database.

Ken Thompson nel 1969 iniziò a lavorare su un computer di nome **PDP-7**, utilizzando il linguaggio assembler; l'idea di base era di scrivere un linguaggio ad alto livello per l'implementazione di sistemi operativi. Thompson prese spunto dal CPL (Combined Programming Language) e dal BCPL (Basic CPL - 1967) per creare il linguaggio B che risultava, però, essere ancora troppo pesante per l'hardware a disposizione.

Nel 1972, un collaboratore di Thompson, **Dennis Ritchie** ottimizzò ulteriormente il B (creando una prima versione chiamata NB), restituendogli alcune funzionalità del BCPL ed inventando quello che ad oggi viene chiamato **Linguaggio C**, che permise di riscrivere quasi totalmente lo UNIX di Thompson, questa volta per un computer più avanzato, il PDP-11.

Inaspettatamente il C permise di riscrivere lo UNIX per il PDP-11 in tempi rapidissimi e ciò spinse Thompson, Ritchie e Kernighan a scrivere il kernel anche per l' "Interdata 8/32" ed il "Dec VAX 11/780", quest'ultimo, all'epoca, particolarmente popolare.

Tra il 1973 ed il 1980 il linguaggio C si diffuse anche su architetture Honeywell 635 e IBM 360/370, grazie al lavoro ad opera del ricercatore Johnson che sviluppò il compilatore **ppc**, alla nascita delle prime librerie (un certo Lesk sviluppò il "portable I/O package", rinominato "standard I/O library") e alla scrittura di un libro di riferimento da parte Kernighan e Ritchie nel 1978, il "C Programming Language", o come viene comunemente chiamato "Libro bianco".

In questi anni c'era un po' di confusione, primo perché il C non era uguale a quello che conosciamo oggi, ma manteneva una compatibilità con i linguaggi da cui derivava (il BCPL ed il B), ed il "Libro bianco" era nato proprio dall'esigenza di scrivere codice portabile che permettesse l'evolversi del linguaggio.

Negli anni '80 i compilatori erano basati principalmente sul pcc di Johnson, ma nacquero molti compilatori indipendenti che non erano conformi al pcc, ma che permisero la diffusione del C praticamente su tutte le macchine più popolari; per questo il comitato ANSI (American Standards Institute) X3J11, nel 1983, iniziò a sviluppare uno standard per il linguaggio C, aggiungendo importanti caratteristiche ed ufficializzando molte caratteristiche presenti nei diversi compilatori.

Nel 1989 si arrivò quindi allo standard **ISO/IEC 9899-1990** (che chiameremo ANSI89 per comodità) che venne usato come base per tutti i compilatori.

Nel medesimo periodo ci fu un'altra importante standardizzazione che mirava a definire l'interfaccia tra linguaggio e sistema operativo. Prendendo spunto proprio dall'ANSI89, fu creata la "Portable Operating System Interface" (**POSIX**) ad opera dell'IEEE (Institute of Electrical and Electronics Engeneers) che definisce, inoltre, i concetti di thread, socket, estensioni realtime e molto altro, in modo unico e indipendente dall'implementazione.

Nonostante negli anni il C sia stato implementato sia con lo standard ANSI, che in maniera proprietaria, grazie a questi strumenti il C è diventato (ed è tutt'oggi) il linguaggio più utilizzato al mondo, almeno fino all'avvento del C++ (chiamato originariamente "C con

classi") che è stato anche la spinta del miglioramento nel tempo del C attraverso lo standard ANSI, di cui è stata rilasciata un'ultima versione denominata ANSI99, che è supportata dalle ultime versioni del **gcc**.

Il C ed il suo UNIX hanno influenzato tutto il mondo dell'informatica, considerando che da esso sono scaturiti colossi come SCO e Sun Microsytem, e prodotti come Linux e BSD. A parer mio, quindi, vale la pena di studiarlo, perché è sicuramente un bagaglio molto utile nella vita di un programmatore.

A chi volesse approfondire la storia dei sistemi operativi (e quindi dell'informatica in generale) proponiamo di leggere i primi capitoli della seguente Guida di Sistemi Operativi, mentre per la storia dei vari linguaggi consigliamo la lezione, presente qui su Html.it, di Storia della programmazione.

5. Software da usare: Windows

Per **sviluppare in C sono necessari solamente due strumenti**, un "sistema di compilazione" (un tool di programmi, come compilatore, linker, debugger, ecc.) che controlla gli errori e traduce il sorgente in C in un eseguibile (composto da istruzioni di codice macchina) ed un buon editor di testi. Anche se è possibile usare qualsiasi editor di testi disponibile sul proprio sistema operativo, qui segnaliamo quelli più usati e diffusi, per venire incontro alle più svariate esigenze, da chi vuole scrivere il codice direttamente dalla shell Linux a chi vuole un sistema integrato sotto Windows che permetta anche di compilare. Abbiamo deciso di fare una lista dei migliori compilatori ed editor disponibili per i più diffusi sistemi operativi.

Microsoft Windows

- DJGPP - Porting del famoso **GCC** (compilatore per macchine unix/linux), permette di compilare programmi scritti in C ed in C++; forse risulta un po' difficile da installare, ma una volta fatto il tutto, si può lavorare tramite l'interfaccia **Rhide**, un editor molto simile all'**Edit** del Dos. Il **Djgpp** è completamente gratuito sia per uso personale che commerciale. E' possibile usare il **Djgpp**, oltre che con sistemi Dos/Windows, anche con Caldera DR-DOS, FreeDOS ed **OS/2**.

CURIOSITA' *- Il nome DJGPP deriva dal fatto che l'organizzatore ed il principale programmatore di questo compilatore è tal DJ Delorie, il quale ricorda come è nata l'idea di un porting del **GCC** per ambienti Dos/Windows: "Il DJGPP è nato all'incirca nel 1989 [...], quando Richard Stallman (il padre del free software) stava parlando ad un meeting del 'Northern New England Unix User Group' (NNEUUG) alla Data General, dove lavoravo. Gli domandai se la Free Software Foundation (FSF) avesse mai preso in considerazione il porting del **GCC** per MS-DOS [...], e lui affernò che ciò non era possibile perché il **GCC** era troppo grande e l'MS-DOS era un sistema operativo a 16-bit. Lanciata la sfida, io cominciai.*

- Lcc-win32 - Questo ambiente di sviluppo ha al suo interno tutto quello di cui c'è bisogno, compilatore (assemblatore, linker, ecc.), interfaccia visuale per scrivere il codice, correggerlo, ecc., il manuale utente e documentazione tecnica. Insomma è un ambiente di sviluppo completo per il linguaggio C. L'uso per scopi personali è gratuito, mentre per scopi commerciali è necessario comprare la licenza.

- DevC++ - Anche questo prodotto è un IDE (Integrated Development Enviroment) di sviluppo e racchiude al suo interno un ottimo editor per windows e un porting del **GCC** come compilatore. Sono disponibili due versioni, la versione stabile, la 4, e la versione beta, la 5.
- Borland C++ - Il compilatore della borland è uno dei migliori disponibili sul mercato, ovviamente l'ambiente di sviluppo completo è a pagamento, ma vale la pena di provare a fare il download della versione 5.5.
- Micorosft Visual C++ - L'ambiente di sviluppo più famoso del mondo è a pagamento, ma non si deve pensare che sia il migliore disponibile, arrivato alla versione .NET il compilatore C/C++ ha raggiunto un buon grado di maturità anche se fino alla versione 6 era possibile fare programmi che seppur non corretti giravano lo stesso o che andavano in conflitto appena eseguiti. Certamente l'interfaccia ha tutte le carte in regola per essere al top (completamento del testo, possibilità di usare wizard per costruire parti di codice, ed ovviamente integrazione tra le librerie per sviluppare in ambiente Windows e lo stesso linguaggio di programmazione). Il mio consiglio è di comprarlo se c'è una reale esigenza nell'ambito professionale; ai principianti consiglio di aspettare per due motivi, il costo (che però può risultare molto inferiore grazie alle licenze studente) e la difficoltà, almeno iniziale, nel suo utilizzo che sarebbe bene fosse accompagnata da un buon manuale.

EDITOR

- 1st Page 2000 - Uno dei migliori editor disponibili per Windows, è abbastanza veloce, permette di aprire più file contemporaneamente, inoltre ha l'highlighting del testo, il conteggio del numero di righe e colonne, e molto altro; pensato soprattutto come editor html, si rivela uno strumento utile anche per scrivere programmi in C, C++ e Java.
- jEdit - Ottimo editor multipiattaforma che servirebbe per scrivere codice in **Java**, ma che può essere usato tranquillamente per scrivere codice in **C**. La sua potenza risiede, oltre che nelle funzionalità avanzate che un editor deve avere (numero di riga, testo con highlighting, ecc.), anche nell'alto livello di configurabilità e nella sicurezza di poter usare JEdit con quasi tutti i sistemi operativi senza perdere in termini di prestazioni (trovandoci sempre lo stesso ambiente di lavoro, non siamo costretti ad imparare l'uso di editor diversi) e di resa del nostro codice (siamo sicuri che un testo scritto con jEdit per Windows è totalmente compatibile con il jEdit per Linux).
- Mini NoteTab (Win 3.x) - Questo è un editor molto versatile disponibile per **Windows 3.x** ed è pensato come sostituto per il notepad. Permette di lavorare su più file ed è leggero, veloce ed altamente configurabile.
- **Edit del DOS** - Anche se un po' datato, l'Edit del DOS si presenta come un buon editor che ha, oltre alle funzioni base, un'ottima velocità e la visualizzazione delle righe e colonne di lavoro, molto utile quando il compilatore ci segnalerà un errore.

6. Software da usare: Linux

Ecco invece una carrellata di editor e compilatori per Linux.

Linux

- **GCC** - GCC è l'acronimo di "GNU Compiler Collection" e permette di compilare codice scritto in **C** (ovviamente), **C++**, **Objective-C**, **Ada**, **Fortran**, **Java**, **Pascal** e qualsiasi altro linguaggio per cui esistano "front-end". E' vero, però, che spesso con il termine GCC si intende "GNU C Compiler", mentre vengono riservati altri nomi quando si intende l'utilizzo per linguaggi come il Fortran (GNAT) o il C++ (G++); sulle specifiche del GCC ci sarebbe da parlare per ore (standard supportati, tipo codice generato, ecc.), basti sapere che il GCC è un compilatore e non un preprocessore, quindi il codice sorgente di un file, sia C sia C++, produce un codice oggetto migliore e permette un'individuazione migliore degli errori, grazie soprattutto al GDB "Gnu Debugger". Imparare ad usarlo è abbastanza facile, ad esempio per compilare un programma in C basta digitare da console

  ```
  # gcc programma.c
  ```

  imparare ad usarlo bene, e quindi averne la totale padronanza è tutt'altra cosa, ma se avete voglia di imparare, basta digitare, sempre da console,

  ```
  # man gcc
  ```

  ed iniziare a leggere. In questa guida, nelle lezioni finali, spiegheremo in dettaglio le maggiori opzioni di compilazione ed anche utilizzi avanzati dell'utility make per creare i makefile.

  EDITOR

- Kdevelop - Giunto alla versione 2.1.5, questo ambiente di sviluppo non ha nulla da invidiare al Visual C++ della Microsoft; riconosce automaticamente i percorsi dove sono installati i programmi necessari (come GCC e Make) e le relative guide di supporto; è possibile creare semplici file in C o in C++, ma anche progetti completi "a finestre" per Linux. Ovviamente uno dei migliori programmi disponibili per Linux, ma, forse, troppo difficile da usare per il principiante che vuole scrivere semplici programmi in C.
- VIM - Insieme ad Emacs è uno dei migliori editor disponibili per Linux; VIM è un editor di testo basato sul famoso **Vi** ed infatti il nome stesso sta per "Vi IMproved" (anche se originariamente stava per "Vi IMitation"). Tra le sue molte caratteristiche menzioniamo l'uso di finestre multiple, aiuto in linea, scrolling orizzontale, formattazione del testo, supporto per il mouse, linguaggio di scripting incluso, uso dei plugin, indentazione automatica, colorazione del testo e molto altro. Forse un po' difficile da usare all'inizio, ma sicuramente uno strumento potente una volta che si è imparato ad usarlo. E' disponibile per praticamente tutti i sistemi operativi, e questo è un altro punto a favore di questo fantastico editor.

- Emacs - Emacs (abbreviazione di Editor MACroS) viene considerato da alcuni molto più di un editor di testi (alcuni lo considerano un mini sistema operativo), per il semplice fatto che il suo cuore è formato da un interprete Lisp (chiamato elisp); le sue potenzialità sono enormi, dalla colorazione del testo, all'indentazione automatica, alla possibilità di interagire in modo trasparente con la documentazione, con il compilatore (ed eventuale debugger) e la totale funzione, volendo, anche di terminale Linux. Oltre ad essere disponibile, come VIM, per quasi tutti i sistemi operativi, ha la sua corrispondente interfaccia per ambienti grafici chiamata XEmacs.
- Quanta Plus - Pensato soprattutto per lo sviluppo di pagine HTML, questo editor è il corrispettivo di **1st Page 2000** del mondo Linux. Proprio la sua semplicità d'uso e il suo supporto per svariati linguaggi ne fanno un editor da scegliere, magari rispetto ad altri più potenti, ma sicuramente più difficili da usare.
- jEdit - Come per Windows, jEdit è disponibile anche per Linux; oltre alle già citate caratteristiche, ricordiamo che sul sito ufficiale è disponibile il download, non solo dei file java multipiattaforma, ma anche dell'.RPM per svariate distribuzioni; gli utenti **Gentoo** invece possono installarlo semplicemente digitando

```
# emerge jedit
```

così come quelli che usano **Debian**, i quali devono seguire queste semplici istruzioni

```
# pico /etc/apt/source.list
> deb http://dl.sourceforge.net/sourceforge/jedit ./
> deb-src http://dl.sourceforge.net/sourceforge/jedit ./
# apt-get update
# apt-get install jedit
```

oppure

```
# apt-get source jedit
```

- GNU Nano - Versione avanzata del più famoso **Pico**, questo semplice ma potente editor è per coloro che vogliono scrivere codice dalla shell, senza entrare nell'ambiente grafico; a mio parere è uno degli editor non grafici migliori disponibili per Linux e vale la pena di usarlo se si vuole semplicità e velocità.

7. Software da usare: Mac OS

Per finire i programmi per Macintosh.

Mac

- **GCC** - In MacOS X, il compilatore GCC è incluso avendo come base un sistema BSD di nome Darwin; quindi non c'è molto da dire, basta aprire una console ed è

possibile iniziare ad usarlo; la compatibilità con quello per Linux dovrebbe essere praticamente assoluta.

- CodeWarrior - Per MacOS Classic (7.x al 9.x) l'unico compilatore decente risulta essere CodeWarrior, che, però, non è gratuito. E' sicuramente un ottimo prodotto, disponibile, non solo per MacOS Classic, ma anche per MacOS X, Windows, Linux, Solaris, PalmOS. Vi consigliamo di comprarlo se ne avete realmente l'esigenza; cercando in rete potrete trovare altri compilatori, forse un po' meno potenti, ma gratuiti, per il vostro MacOS Classic..

EDITOR

- Emacs - Ebbene si, Emacs è installabile anche su sistemi Mac, quindi ve ne consigliamo caldamente l'uso, soprattutto perché in questo modo potrete lavorare con il medesimo editor anche quando cambiate sistema operativo.
- VIM - VIM, come Emacs, può girare tranquillamente su sistemi MAC, ed è quindi possibile sfruttare tutta la potenza di questo fantastico editor anche sui computer "con la mela", senza grossi problemi.
- jEdit - Se avete problemi ad installare i due editor precedenti o se volete un editor, forse un po' meno potente, ma sicuramente più intuitivo e semplice da usare, la vostra scelta potrebbe ricadere sul famoso editor multipiattaforma sviluppato in Java.

Personalmente ritengo che l'accoppiata migliore sia formata dal **GCC** come compilatore e da un editor a scelta tra **jEdit**, **Emacs** e **VIM**, per due semplicissimi motivi, in primis la diffusione di tali editor per praticamente tutti i sistemi operativi, a seguire il fatto che in questo modo si imparano a compilare i programmi da console, con l'enorme vantaggio di esperienza rispetto all'utilizzo di sistemi automatici (IDE).

Ma la mia è una semplice opinione e nulla vi obbliga, soprattutto se siete all'inizio, a non usare gli ambienti di sviluppo come Dev-C++ o Kdevelop, visto che dovete imparare, se non riuscite a compilare con il GCC, non fatevi problemi, questo non vi farà meno programmatori di altri (anch'io per i primi anni di sviluppo, pur sapendo programmare abbastanza bene, utilizzavo Dev-C++ per compilare i programmi).

8. Scriviamo il primo programma in C

***Nota**: Da questa lezione inseriremo, alla fine di ogni lezione, dei pezzi di codice commentati relativi al programma di esempio che ci accompagnerà lungo tutta la guida. Gli esempi saranno pertinenti alla lezione e serviranno al lettore per inquadrare gli argomenti in un programma abbastanza ampio. Sono stati inseriti dei numeri identificativi per ogni riga in modo da poter indentificare facilmente tali parti di codice nel listato completo. Si lascia al lettore la possibilità di approfondire le parti di codice presentate. Lo studio del codice di esempio è facoltativo e non pregiudica la compresione della guida.*

Adesso è giunto il momento di **addentrarsi nella programmazione vera e propria** utilizzando il linguaggio C.

L'impostazione che intendiamo seguire è quella di spiegare i concetti "per esempi", all'inizio alcune cose saranno poco chiare, ma dovete fidarvi e proseguire, perché spesso per spiegare un concetto si deve fare ricorso a costrutti, o istruzioni che verranno spiegate solo nelle lezioni successive.

Il **primo programma in C**, il più semplice in assoluto è il famoso "Hello World!" (Ciao Mondo) che ha soltanto lo scopo di stampare a video la scritta per spiegare la sintassi basilare del linguaggio,

```
#include <stdio.h>

main ()
{
  printf("Hello World!");
}
```

Questo semplice programma ha delle corrispondenze anche in altri linguaggi,

```
// Pascal
program main;
begin
  writeln('Hello World!');
end.

// C++
#include <iostream.h>
main()
{
  cout << "Hello World!";
}

// Java
class Saluto {
  public static void main(String args[]) {
    System.out.println("Hello World!");
  }
}
```

Abbiamo volutamente messo a confronto questi quattro linguaggi per far notare un po' di cose.

Innanzitutto il C è completamente diverso dal **Pascal**, perché ha una logica molto più pratica, ad esempio non dobbiamo dichiarare ogni variabile fuori dal programma prima di usarla; nel **C++**, a differenza del C, si hanno a disposizione comandi alternativi e costrutti di nuova concezione, sia come nomi che come sintassi; a differenza del **Java**, in C, non devo fare una miriade di dichiarazioni prima di poter scrivere il programma e non sono costretto a seguire delle regole rigide per strutturare un programma (in Java c'è un forte controllo sui tipi, lo si può evincere anche dal codice proposto, in cui la funzione main deve essere "esplicitamente" dichiarata pubblica, statica e void, ed inoltre devono essere messi gli argomenti della funzione - args[] - anche se all'interno del codice questi non vengono utilizzati).

9. Elementi fondamentali di un programma in C

Ora che avete chiaro tutto ciò vi starete giustamente chiedendo a cosa serve l'include? cosa è il main? a cosa servono le graffe? ed i punti e virgola? procediamo con ordine riportando il codice scritto nella lezione precedente:

```
#include <stdio.h>

main ()
{
  printf("Hello World!");
}
```

- L'**include** è una direttiva del preprocessore, un comando, che permette di richiamare le librerie standard del C. Senza librerie un programma non avrebbe a disposizione i comandi per eseguire anche le operazioni più semplici, come leggere un file o stampare a video una scritta.
- La funzione principale in un qualsiasi programma in C è la **main** che, in questo caso, non ha parametri, ma può ricevere anche degli input da riga di comando. Il main è indispensabile ed unico e deve esserci sempre;
- Le **parentesi graffe** servono, invece, per delimitare le istruzioni, o come vengono abitualmente chiamate "statement", che devono essere eseguite in ordine, da quella più in alto, giù fino all'ultima;
- Il **punto e virgola**, invece, serve per "chiudere" un'istruzione, per far capire che dopo quel simbolo inizia una nuova istruzione.

Entrando nel dettaglio della sintassi del Linguaggio C, possiamo notare che la prima funzione che incontriamo e che è anche una delle più usate è **printf**, adibita a stampare a video tutto quello che gli viene passato come argomento e che fa parte della libreria **<stdio.h>**, senza la cui inclusione, non avrebbe senso.

All'interno di un programma in C possono essere inseriti dei commenti, basti sapere, per adesso, che esistono due modi:

- // - Tutto quello che sta a destra sulla medesima riga viene considerato commento e viene ignorato (ai fini dell'interpretazione del linguaggio) dal compilatore;
- /* ... */ - Tutto quello che è compreso tra i due asterischi viene considerato commento; questa forma viene utilizzata per commenti su più righe.

10. La compilazione

Come già detto un programma in C può essere scritto anche con un semplice editor di testi come il Blocco note di Windows (o simile). Per poter vedere il risultato del codice dobbiamo salvare necessariamente il file con un'estensione ".c", ad esempio "hello.c". A questo punto non dobbiamo fare altro che **compilare il programma**.

In realtà la compilazione di un programma in C (spesso integrata in ambienti di sviluppo e quindi apparentemente trasparente ed uniforme) segue varie fasi:

- Il **codice sorgente viene controllato dal preprocessore** che ha i seguenti compiti:
  - rimuovere eventuali commenti presenti nel sorgente;
  - interpretare speciali direttive per il preprocessore denotate da "#", come #include o #define.
  - controllare eventuali errori del codice
- Il **risultato del preprocessore** sarà un nuovo codice sorgente "pulito" ed "espanso" che viene tradotto dal compilatore C in codice assembly;
- L'**assembler sarà incaricato di creare il codice oggetto** salvandolo in un file (.o sotto Unix/Linux e .obj in Dos/Windows);
- **Il Link editor ha il compito di collegare tutti i file oggetto** risolvendo eventuali dipendenze e creando il programma (che non è altro che un file eseguibile).

Ricordiamo che tutte queste operazioni, per quanto molto avanzate, possono individuare facilmente errori di sintassi, ma mai e poi mai potranno trovare errori logici (come ad esempio un ciclo che non finisce), anche se spesso vengono segnalati dei **Warning** che non costituiscono errore, ma che segnalano parti di codice strane e quindi sulle quali porre attenzione per eventuali errori logici. Qui ci viene in aiuto il debugger che ha il compito di aiutare a gestire la correzione degli errori con svariati strumenti come i breakpoint ed i watchpoint.

Avete ancora il vostro file "hello.c"? spero di si, perché è giunto il momento di compilarlo; se possedete un ambiente di sviluppo come Dev-C++ o Visual C++ dovete solamente cercare il comando "Compila" od "Esegui", il programma provvederà, non solo ad eseguire tutte le operazioni citate, ma, una volta compilato, manderà in esecuzione il vostro programma presentandovi il risultato a video.

Per chi stesse usando, invece, compilatori come il DJGPP o il GCC, la compilazione (intesa in tutte le sue fasi) deve essere fatta da riga di comando (dalla shell per capirsi), e, successivamente, sempre da riga di comando, si deve lanciare il file eseguibile risultante (in linux con gcc, di default il file eseguibile si chiama **a.out**, mentre in ambiente Dos/Windows generalmente ha lo stesso nome del sorgente ma con estensione **.exe**, invece che ".c").

Per compilare basta posizionarsi nella solita cartella dove risiede il file sorgente e scrivere,

```
# gcc hello.c
```

Adesso avete visto un riassunto di come si possa scrivere un programma e compilarlo, tanto perché non vi troviate disarmati quando procederemo con le spiegazioni; molti dei concetti sopra esposti verranno approfonditi mano a mano che procediamo, come la strutturazione del linguaggio, la compilazione, le funzioni disponibili nelle varie librerie.

11. Cosa sono le variabili

Ripetiamo la spiegazione della Guida di Programmazione Base presente su questo sito:

«Adesso è fondamentale definire il concetto di variabile, visto che nelle prossime lezioni faremo largo uso di questo termine. Pensiamo, ad esempio, a quando salviamo un numero di telefono di un nostro amico sul cellulare; se vogliamo chiamare il nostro amico, basterà inserire il suo nome (nome della variabile) ed il cellulare comporrà automaticamente il numero di telefono (valore della variabile). Si può vedere quindi che una variabile esiste in funzione del nome e del suo valore corrispondente; la comodità risiede (come nel cellulare) nel poter usare un nome per valori, che possono essere numeri o lettere, di grande entità o difficili da ricordare. Un altro vantaggio, non da sottovalutare, è la possibilità di usare il nome della variabile al posto del suo valore per eseguirvi sopra delle operazioni, con la possibilità, in seguito, di modificare il valore come e quante volte vogliamo.»

Detto con un gergo più tecnico, le variabili non sono altro che dei contenitori, identificati da un nome **univoco**, di un qualsiasi valore, sia esso un numero o una stringa. Per poter fare qualcosa di concreto all'interno dei programmi dobbiamo utilizzare le variabili in cui il pregio di avere un corrispondente nome-valore permette di gestire i cambiamenti di valore ad esse associati, fornendo quindi quella dinamicità necessaria ad eseguire operazioni complesse e/o ripetitive con il minimo sforzo.

Le variabili vengono definite da un **tipo** e da un **nome**.

Il nome per identificare una variabile (o una funzione o una costante) viene comunemente riferito come **identificatore**.

Un identificatore è costituito da una o più lettere, cifre o caratteri e deve iniziare con una lettera o il carattere di sottolineatura (underscore "_"); la loro lunghezza massima dipende dal compilatore, ma generalmente non si possono superare i 31 caratteri, ed inoltre il C è case-sensitive, quindi si fa distinzione tra lettere maiuscole e lettere minuscole. Questa piccola parentesi è dovuta, visto che in seguito parleremo appunto delle funzioni e delle costanti per le quali dovranno essere chiare le regole per assegnare un nome "formalmente" corretto.

Il tipo della variabile indica quale tipo di valori può assumere il contenuto della variabile stessa, si può ben capire che un tipo intero conterà soltanto dei numeri, mentre il tipo carattere conterrà solamente lettere dell'alfabeto, numeri e simboli; Va fatto notare che l'intero '7' è estremamente diverso dal carattere '7', infatti l'intero viene trattato come un numero e su di esso si possono svolgere le più svariate operazioni matematiche, mentre il carattere viene gestito come un simbolo (si tratti di una lettera o di un numero).

Tutto ciò ha una spiegazione logica che risiede nella rappresentazione del numero stesso; se l'intero 7 viene rappresentato da un byte (otto bit) come 00000111, il carattere 7 viene rappresentato con un byte ma seguendo quella che è la codifica ASCII, quindi appare, a livello di bit come 00110111 che equivale ad una rappresentazione intera di 55. Le operazioni che permettono di convertire una variabile da un determinato tipo ad un altro (ad esempio da una lettera ad un numero) prendono il nome di **casting**, trattato nel capitolo 12 di questa guida.

Tutte le variabili, prima di essere utilizzate, **devono essere dichiarate**, cioè deve essere detto al compilatore il tipo della variabile ed il suo nome (es. int x), questo per permettergli di allocare la memoria necessaria alla variabile stessa; la dichiarazione generalmente viene fatta all'inizio del programma, ma in programmi di grandi dimensioni può trovarsi anche in altre posizioni (o altri file), ma bisogna ricordare che comunque la dichiarazione di una variabile può essere fatta **una ed una sola volta**.

Successivamente **la variabile deve essere inizializzata**, cioè le deve essere assegnato un valore, operazione che generalmente viene fatta contemporaneamente alla dichiarazione.

```
// solo dichiarazione
int x;
// inizializzazione
x = 10;

// dichiarazione ed inizializzazione
int y = 15;
```

12. Le variabili in C

In C esistono vari tipi di variabili, questo per venire incontro sia all'esigenze di rappresentabilità di grossi numeri sia al maggior risparmio di memoria possibile, utilizzando di volta in volta il tipo più adeguato ad una specifica situazione. Nella tabella seguente sono mostrati i vari tipi, con la parola che in C ne consente l'uso (char per carattere, int per intero e così via), una spiegazione sui dati che possono rappresentare ed il numero di byte necessari, all'interno del C, per la loro rappresentazione:

Tipi di dichiarazione	**Rappresentazione**	N. di byte
Char	Carattere	1 (8 bit)
Int	Numero intero	2 (16 bit)
Short	Numero intero "corto"	2 (16 bit)
Long	Numero intero "lungo"	4 (32 bit)
Float	Numero reale	4 (32 bit)
Double	Numero reale "lungo"	8 (64 bit)

Il tipo **char** è adibito a contenere uno ed un solo carattere; questa imposizione, scomoda quando vogliamo memorizzare una successione di caratteri, è stata risolta in alcuni linguaggi adottando il tipo **string** (stringa), mentre in C questa situazione viene risolta utilizzando un **array di char**, l'array è, a tutti gli effetti, un contenitore di variabili dello stesso tipo, per questo ne parleremo in dettaglio nella lezione 10. Ritornando al nostro char, esso può contenere un qualsiasi carattere definito secondo lo standard ASCII e quindi potrà contenere qualsiasi lettera (maiuscola o minuscola), cifra (da 0 a 9) e simbolo

previsto dalla codifica. Per dichiarare ed inizializzare una variabile char, ad esempio inizializzandola con la lettera 'r', basta scrivere:

```
char a = 'r';
```

Il tipo **int** è, invece, quello che permette di contenere dei numeri; il tipo int ha le sue due varianti che sono short e long, anche se in realtà un tipo int è già di per se' un tipo short, mentre il long permette di estendere (utilizzando due byte in più) il range dei valori che il tipo int può assumere, questo per venire incontro all'esigenza di lavorare con grandi numeri. Il tipo int contiene numeri, appunto, interi, quelli che in matematica vengono chiamati numeri naturali, e cioè senza la virgola e parti frazionate. Qui di seguito presentiamo un pezzo di codice in cui vengono dichiarate ed inizializzate variabili int e, per far capire come il fatto che int rappresenti i numeri naturali comporti una certa attenzione, verrà eseguita l'operazione di divisione tra due interi (utilizzando l'operatore di divisione /):

```
int x = 7;
int y = 3;
int z;

z = x / y;
// z vale 2, cioè la parte intera della divisione tra 7 e 3
```

I tipi **float** e **double** sono chiamati anche numeri in virgola mobile, cioè quelli che in matematica vengono chiamati numeri reali, e quindi possono essere usati per contenere tutti quei numeri che hanno parti frazionarie. La differenza tra i due sta solamente nei bit che sono riservati per la loro rappresentazione, che si va a riflettere, non solo nel range di rappresentazione, ma anche nel numero di cifre dopo la virgola che possono essere rappresentate, in questo caso risulta, quindi, più accurata la rappresentazione utilizzando i double. Qui di seguito presentiamo un semplice pezzo di codice (utilizzando i double) per far vedere come, a differenza degli int, venga rappresentata la divisione tra due numeri in virgola mobile:

```
double x = 7.0
double y = 2.0
double z;

z = x / y
// z vale 3.5
```

Da notare che la notazione usata per rappresentare la virgola è quella inglese, cioè quella in cui si usa un punto (e non una virgola) per dividere la parte intera da quelle frazionaria.

Numeri con segno e senza segno

*In termini di rappresentabilità c'è da far notare che l'uso dei bit per rappresentare un determinato tipo varia a seconda che quel tipo sia **signed** (con segno) o **unsigned** (senza segno); questo è dovuto al fatto che se un tipo è formato, ad esempio, da 8 bit come l'intero 114, che di default è **signed**, si useranno 7 bit per rappresentare il numero e 1 bit (quello più a sinistra, detto anche bit più significativo) per*

rappresentare il segno (che è '+' se il bit vale 0 ed è '-' se il bit vale 1):

```
01110010 // rappresenta, in bit, l'intero 114
11110010 // cambiando il bit più a sinistra si ottiene l'intero -114
```

*Mentre, se avessimo utilizzato un **unsigned int**, i valori sarebbero stati tutti positivi, perché anche il bit più significativo sarebbe stato usato per rappresentare il numero:*

```
01110010 // rappresenta, in bit, l'intero 114
11110010 // rappresenta, in bit, l'intero 242
```

*Ammesso di avere **n** bit per rappresentare un numero, se è **signed**, allora i valori possibili saranno compresi tra $-2^{(n-1)}+1$ e $2^{(n-1)}$, quindi, nel caso di una rappresentazione ad 8 bit si avrebbero valori compresi tra -127 e 128, questo perché tra i numeri entro il range deve essere considerato anche lo zero. Nel caso di un **unsigned**, invece i valori, come già detto, sono tutti positivi e possono assumere valori compresi tra 0 e $2^n - 1$, ritornando al caso degli 8 bit si avrebbero valori compresi tra 0 e 255.*

13. Gli Operatori: introduzione

Abbiamo spiegato come dichiarare, inizializzare e manipolare le variabili, ma non potremo farci niente di concreto se non parliamo degli operatori che permettono di lavorarci sopra.

Gli operatori in programmazione permettono di estrapolare un determinato valore dal frutto dell'operazione che si compie su una o più variabili all'interno del programma; così come l'operatore "+" serve per sommare due numeri in matematica, analogamente serve per compiere la stessa operazione in un programma scritto in C.

Ovviamente ci sono delle dovute differenze, innanzitutto le operazioni del C sono quelle basilari (per funzioni più avanzate dobbiamo usare delle librerie apposite), hanno un risultato "finito", contrariamente a quelle matematiche che possono avere un risultato simbolico o infinito e, infine, contrariamente a quelle matematiche, si possono applicare anche a valori non numerici delle variabili.

Gli operatori che andremo ad analizzare nelle seguenti lezioni si suddividono in:

- **Operatori aritmetici.** Comprendono somma, sottrazione, moltiplicazione, divisione intera, divisione con modulo ecc.
- **Operatori di confronto.** Operatori che permettono di verificare determinate condizioni, come ad esempio l'uguaglianza o la disuguaglianza.

- **Operatori logici.** Da utilizzare con le istruzioni condizionali ed iterative.

Per comprendere meglio gli operatori di confronto e logici è consigliabile e propedeutico leggere l'Introduzione alla logica e diagrammi di flusso.

14. Operatori aritmetici

In realtà abbiamo già visto i semplici operatori di assegnamento (=) e di divisione intera (/), ma non abbiamo spiegato nel dettagli tutte le operazioni che si possono compiere sulle variabili per mezzo dei loro operatori. Ovviamente oltre alla divisione intera, con gli interi, è possibile eseguire la somma (+), la sottrazione (-), la moltiplicazione (*) e la divisione con resto (%), di cui presentiamo una tabella riassuntiva:

Operazioni con gli int	Simbolo	Esempio
Addizione	+	4 + 27 = 31
Sottrazione	-	76 - 23 = 53
Moltiplicazione	*	4 * 7 = 28
Divisione intera	/	10 / 3 = 3 (3 è il n di volte divisibili senza resto)
Divisione con modulo	%	11 / 6 = 5 (5 è il resto della divisione)

Quando lavoriamo con i numeri reali (float o double) l'unica operazione che manca è quella di "divisione con modulo" che ovviamente non ha ragione di esistere, in quanto le divisioni possono contenere le parti frazionarie:

Operazioni con i double	Simbolo	Esempio
Addizione	+	2.5 + 14.3 = 16.8
Sottrazione	-	43.8 - 12.7 = 31.1
Moltiplicazione	*	7.5 * 3.0 = 22.5
Divisione	/	5.0 / 2.0 = 2.5

Esistono poi degli operatori ai quali bisogna porre particolare attenzione, questi sono l'operatore di incremento (++) e quello di decremento (--), che possono essere preposti o posposti alla variabile; se sono preposti il valore è calcolato prima che l'espressione sia valutata, altrimenti viene calcolato dopo; l'incremento ed il decremento avvengono sempre nel valore di una unità sul valore della variabile. Un po' di codice servirà a far capire meglio come funzionano tali operatori:

```
int x = 2;
int y = 5;
int z = 9;
```

```
int k = 6;

// accanto, come commento, il valore stampato a video
printf("%d \n", x++); // 2
printf("%d \n", ++y); // 6
printf("%d \n", z--); // 9
printf("%d \n", --k); // 5
```

Come detto questi operatori incrementano o decrementano il valore di una variabile, quindi

```
x++
```

ad esempio, equivale a scrivere

```
x = x + 1;
```

Anche se la prima forma (quella di incremento) risulta essere più veloce.

Inoltre esiste una forma contratta per espressioni del tipo:

```
espressione₁ = espressione₁ <operatore> espressione₂

// equivalente a
espressione₁ <operatore> = espressione₂
```

Questa forma risulta essere più concisa, e per questo più facile da usare, ma bisogna porre attenzione nel suo uso perché potrebbe indurre in errori dovuti alla poca chiarezza del codice:

```
int y = 4;
y += 2; // i adesso vale 6

int x = 3;
x *= y + 3; // x adesso vale 27
// questo perché equivale a x=x*(y+3) e non x=(x*y)+3
```

15. Operatori di Confronto e Logici

Operatori di confronto

Gli operatori di confronto permettono di verificare determinate condizioni, come ad esempio l'uguaglianza, la disuguaglianza, o semplicemente se un elemento è maggiore di un'altro; la seguente tabella mostra nel dettaglio gli operatori di confronto e la loro funzione:

Simbolo	Significato	Utilizzo
==	uguale a	a == b
!=	diverso da	a != b
<	minore	a < b
>	maggiore	a > b

<=	minore o uguale	a <= b
>	maggiore o uguale	a >= b

Questi operatori servono quando incontreremo le istruzioni condizionali e quelle iterative, perché se la condizione è verificata restituiscono vero, altrimenti restituiscono falso. Gli operatori di confronto sono operatori a due argomenti ed hanno sempre posizione infissa (cioè tra i due argomenti).

Operatori logici

Anche gli operatori logici vengono utilizzati con le istruzioni condizionali ed iterative, e permettono di fare l'AND e l'OR tra due operandi; nella tabella mostriamo i simboli usati e il loro utilizzo:

Simbolo	Significato	Utilizzo
&&	AND logico	a && b
||	OR logico	a || b

Queste operazioni restituiscono necessariamente 1 quando sono vere e 0 quando sono false, quindi facendo "a || b" questa assume valore uno se e solo se "a" o "b" valgono uno, mentre vale zero se entrambi gli operandi valgono zero; analogamente quando si ha "a & b", questa assume valore zero se "a" o "b" valgono zero, uno nel caso entrambi valgano uno. Da far notare che i simboli && e || sono diversi da & e | che, invece, rappresentano il bitwise AND ed il bitwise OR (non trattati in questa guida).

16. Proprietà degli operatori

Gli operatori hanno diverse caratteristiche e regole che permettono a noi di scriverli correttamente, e al compilatore di capire come utilizzarli. Questo accade perché se non fossero specificate delle regole precise, il compilatore che agisce in modo automatico (ed automatizzato) non saprebbe come interpretare quello che è stato scritto.

Visto che non può "fare di testa sua", dobbiamo essere noi a conoscere le regole seguite nella sintassi del linguaggio e presentare, quindi, quello che vogliamo fare in una forma traducibile e sensata per il compilatore.

Le proprietà da conoscere sono poche e non è nemmeno necessario ricordarsele esattamente, basta seguire le convenzioni e, con il tempo, si vedrà che diventerà naturale scrivere programmi "sintatticamente" corretti. Esse sono:

- **Posizione** - Un qualsiasi operatore ha una determinata posizione rispetto ad i suoi operandi. Si dice che un operatore è **prefisso** se compare prima degli operandi, **postfisso** se compare dopo e **infisso** se compare tra gli operandi;

- **Arietà** - Questa rappresenta il numero di argomenti che un operatore può accettare. Ad esempio l'operatore ++ ha arietà uguale a uno (A++), il simbolo + ha arietà uguale a due (A + B), mentre l'unico operatore ad avere arietà uguale a tre è l'operatore ternario " ? : " (A ? B : C), di cui parleremo nella lezione 8;
- **Precedenza (o priorità)** - La precedenza è un valore che identifica gli operatori più importanti e quelli meno importanti; maggiore è la priorità minore è il valore che la identifica: questi valori risultano utili, ad esempio, quando si eseguono delle operazioni matematiche, per stabilire quali operazioni debbano essere eseguite per prime, questo, ovviamente, in mancanza di parentesi tonde che potrebbero modificare l'ordine con cui si eseguono le operazioni sulle variabili (in realtà ciò accade semplicemente perché le parentesi tonde hanno una priorità maggiore);
- **Associatività** - L'associatività stabilisce, a parità di priorità, quale sia l'ordine con cui bisogna eseguire i vari operatori. Se l'operatore è associativo a sinistra, si scorrerà da sinistra verso destra, mentre se è associativo a destra, si farà l'esatto contrario.

Esempi tipici in cui intervengono le proprietà degli operatori, si hanno nell'interpretazione di formule matematiche, come nell'esempio seguente:

```
a + b * c

/* che viene interpretato come
* a + (b * c), e non (a + b) * c
* perché la moltiplicazione ha
* precedenza maggiore
*/

a - b - c

/* che viene interpretato come
* (a - b) - c , perché l'associatività
* della sottrazione è a sinistra
*/
```

17. Prime operazioni di INPUT/OUTPUT

Per adesso abbiamo posto le prime basi del linguaggio C; in questa lezione vogliamo fornire gli strumenti adatti ad operare sull'input/output, inteso come semplici operazione a vedio comandate dalla tastiera. Per fare questo dobbiamo includere il file **<stdio.h>** che mette a disposizione alcune funzioni predefinite per eseguire la lettura da un dispositivo di input (es. tastiera) o scrittura su un dispositivo di output (es. video); le funzioni di cui parleremo sono **getchar**, **getchar**, **printf** e **scanf**.

La prima funzione che incontriamo è la **getchar**, il cui compito è quello di leggere un carattere per volta dalla tastiera; qui di seguito presentiamo un programma che conta il numero di volte che si è digitato un carattere (per interrompere il programma premere CTRL+D, = EOF).

```
#include <stdio.h>

main()
{
  int ch, i = 0;

  while((ch = getchar()) != EOF)
    i ++;

  printf("%d\n", i);
}
```

La funzione immediatamente seguente (e complementare) è la **putchar**, che legge un carattere alla volta e lo stampa a video; viene presentata nel seguente programma che converte ogni carattere inserito nell'equivalente lettera maiuscola (grazie alla funzione **toupper()**.

```
#include <ctype.h> /* Per la definizione di toupper */
#include <stdio.h> /* Per la definizione di getchar, putchar ed
EOF */

main()
{
int ch;

  while((ch = getchar()) != EOF)
    putchar(toupper(ch));
}
```

L'istruzione per stampare a video più usata è la **printf**, che ha il controllo su ciò che viene stampato, nel senso che permette di decidere cosa stampare ed in quale forma. La struttura di printf è la seguente:

```
int printf(char *formato, lista argomenti ...)
```

che stampa sullo **stdout** (il video in questo caso) la lista di argomenti conformemente alla stringa di formato specificata. La funzione ritorna il numero di caratteri stampanti. La stringa di formato ha due tipi di argomenti:

- **caratteri ordinari** - questi vengono copiati nell'output;
- **specificazioni di conversione** - contraddistinte dal carattere percentuale "%" e da un carattere che specifica il formato con il quale stampare le variabili presenti nella lista di argomenti.

La tabella seguente mostra i possibili formati che possono essere usati per formattare le variabili; tali considerazioni si applicano anche al comando **scanf**;

Stringa di controllo	Cosa viene stampato
%d, %i	Intero decimale
%f	Valore in virgola mobile
%c	Un carattere
%s	Una stringa di caratteri
%o	Numero ottale

%x, %X	Numero esadecimale
%u	Intero senza segno
%f	Numero reale (float o double)
%e, %E	Formato scientifico
%%	Stampa il carattere %

È possibile inserire, tra il simbolo % e l'identificativo del formato, una delle seguenti voci:

- Segno meno (-), esegue la giustificazione a sinistra;
- Un numero intero, specifica l'ampiezza del campo;
- m.d, dove m è l'ampiezza del campo e d è la precisione del numero; usato generalmente per le stringhe o per un numero di tipo reale.

Quindi facciamo degli esempi per far capire meglio l'uso del printf:

```
int x = 10;
printf("Il numero è %d", x);
// l'output a video è "Il numero è 10"

print("%-.2.3f \n", 24.392734);
// l'output a video è 24.393

printf("IVA = 20%% \n");
// l'output a video è, "IVA = 20%"
```

Altri linguaggi che usano printf, oltre al C, sono il **C++**, per il quale, però esiste anche il comando proprietario **cout**, ed il **PHP**, nel quale si comporta in modo analogo a quello del C.

All'intero di quello che verrà stampato notiamo spesso il simbolo "\n" che, come avrete intuito serve per andare a capo; tale simbolo viene chiamato **sequenza di escape**. Le sequenze di escape servono per rappresentare quei caratteri "speciali" presenti nella codifica ASCII e che non stampano nulla a video, ma permetto di introdurre all'interno di ciò che verrà stampato eventuali spaziature. Le sequenze di escape iniziano con il carattere backslash (\) e sono interpretate come un singolo carattere.

Tipo di opzione	Descrizione
\n	Ritorno a capo
\t	Tabulazione orizzontale
\b	Tabulazione verticale
\a	Torna indietro di uno spazio
\f	Salto pagina

La funzione **scanf** serve per leggere dallo *stdin* (generalmente la tastiera) una sequenza di caratteri (lettere o cifre) che verranno memorizzate all'interno di opportune variabili. Scanf è, quindi, definita come segue:

```
int scanf(char *formato, lista argomenti ...)
```

Da far notare che la scanf, oltre a poter leggere stringhe di testo, e non un solo carattere alla volta come la getchar, permette di leggere più variabili contemporaneamente, se queste sono specificate nella lista degli argomenti. A differenza della printf, però la variabile deve essere messa preceduta dal simbolo &, perché in realtà tra gli argomenti non dobbiamo passare il nome della variabile, ma il suo indirizzo, cosa che può essere fatta tranquillamente utilizzando un puntatore (ed ecco il perché del &, simbolo che serve ad ottenere l'indirizzo del puntatore; tratteremo i puntatori nel capitolo 13):

```
#include <stdio.h>

int main()
{
   int i;
   scanf("%d \n", &i);
   printf("%d \n", i);
}
```

Questo semplice programma sopra esposto serve solamente per leggere un numero da tastiera e ristamparlo a video; avrete notato che è possibile usare gli specificatori di conversione anche per la scanf, che, quindi, si comporta in modo molto simile a printf. Una piccola particolarità limitatamente all'uso delle stringhe e degli array con la funzione scanf, è la seguente:

```
char stringa[100];

scanf("%s", stringa);
```

In questo caso è possibile omettere il simbolo **&** che richiama l'indirizzo del puntatore, ed usare, invece, il nome della variabile, perché il nome di un array corrisponde all'indirizzo di partenza dell'array stesso.

18. Controlli Condizionali: IF-ELSE

Nella programmazione, come già detto, le istruzioni vengono eseguite dalla prima fino all'ultima. Per deviare il flusso delle scelte (ad esempio per scegliere tra l'opzione A e l'opzione B) basta porre delle condizioni che, se verificate, eseguono un pezzo di codice o altrimenti ne eseguono un altro; queste istruzioni particolari che permettono di incanalare il flusso si chiamano strutture di controllo condizionale o, più genericamente, istruzioni condizionali. Esistono due tipi di strutture di controllo condizionale, l'if-else e lo switch. L'if-else e lo switch hanno, sostanzialmente, comportamenti simili, ed anche se il primo è largamente il più usato, hanno delle differenze per quanto riguarda il tipo di operazione che vogliamo svolgere; ma vediamo nel dettaglio come funzionano.

Nota: in C, a differenza del C++ e del Java, non esiste il tipo booleano (che può assumere valore vero -true- o falso -false-), per questo quando deve essere *verificata una condizione* essa risulta falsa se vale zero e vera altrimenti.

If-Else

L'istruzione if permette di verificare determinate condizioni ed ha la seguente sintassi:

```
[... altre istruzioni ...]
if (espressione)
   istruzione
[... altre istruzioni ...]
```

In questo caso se l'espressione risulta vera, fa eseguire l'istruzione immediatamente successiva, altrimenti (se la condizione è falsa) si salta l'istruzione (od il blocco di istruzioni) facenti parti dell'if e si procede nell'esecuzione delle istruzioni successive, che possono essere la prosecuzione del programma o un semplice **else**, ad indicare la possibile alternativa all'if:

```
if (espressione)
   istruzione1
else
   istruzione2
```

o un **else if** che permette di verificare una o più condizioni:

```
if (espressione)
   istruzione1
else if (espressione)
   istruzione2
else
   istruzione3
```

Così si può comandare il flusso del programma decidendo di eseguire una parte di codice oppure no (nel caso del solo **if**), di fare una scelta tra due parti di codice (nel caso **if** - **else**) o di fare una scelta tra più parti di codice (nel caso **if** - **else if** - **else**). Facciamo un esempio pratico nel codice sottostante, assumendo che sia stato definito, nel codice soprastante all'if, il valore della variabile intera "risultato_esame", che può assumere un valore compreso tra 0 e 30:

```
if (risultato esame >=18)
   printf ("Complimenti hai superato l'esame");
else if (risultato esame >=15)
   printf ("Devi sostenere l'orale per questo esame");
else
   printf ("Non hai superato l'esame");
```

Generalmente dopo un if o un else viene eseguita **solo** la prima istruzione più vicina, regola che può creare dei problemi quando ci sono due o più istruzioni, ma alla quale si può porre rimedio ponendo tutte le istruzioni tra due parentesi graffe, la graffa di apertura ({) andrà sotto all'if, allineata con la prima lettera, mentre la graffa di chiusura (}) andrà posta dopo l'ultima istruzione, su una nuova riga ed allineata con quella di apertura. Per chiarire meglio, ecco un if-else abbastanza semplice che spiega come utilizzare le parentesi graffe per raccogliere istruzioni:

```
if (risultato esame >=18)
{
   printf ("Complimenti hai superato l'esame");
   int passato = 1;
} else {
   printf ("Non hai superato l'esame");
   int passato = 0;
}
```

Dobbiamo fare particolare attenzione all'uso dei blocchi if annidati, come quello dell'esempio:

```
if (risultato esame < 18)
   if (risultato esame < 15) printf ("Non hai superato l'esame");
else printf ("Sei stato ammesso all'orale");
```

In questo caso risulta difficile capire esattamente quale sia il significato reale di questo else, e soprattutto ci si potrebbe domandare se realmente il programmatore abbia voluto porre determinate condizioni o se abbia semplicemente sbagliato a strutturare il costrutto if. Se non capiamo esattamente il codice, conviene ricordarsi la regola secondo la quale *"l'else si riferisce sempre all'if più vicino, a meno che non sia diversamente specificato da eventuali parentesi graffe"*, ma è comunque buona norma usare sempre le parentesi graffe anche quando si tratta di eseguire una sola istruzione, magari indentando (mettendo opportune spaziature) correttamente il codice. L'esempio precedente risulta sicuramente più chiaro se strutturato in questo modo:

```
if (risultato esame < 18)
{
   if (risultato esame < 15)
   {
      printf ("Non hai superato l'esame");
   } else {
      printf ("Sei stato ammesso all'orale");
   }
}
```

19. Controlli Condizionali: Switch e operatori ternari

Operatore Ternario

L'operatore ternario "? :" è una forma sintetica dell'istruzione if-else, e per questo viene usata per ragioni di comodità e sinteticità del codice. L'operatore "? :" è l'unico operatore ternario del C, infatti opera su tre parametri a differenza degli altri operatori che spesso operano su uno o due parametri.
Qui di seguito presentiamo la sintassi dell'operatore e la corrispondente sintassi di un blocco if-else:

```
// Operatore ternario ?
espressione₁ ? espressione₂ : espressione₃

// Corrispondente blocco if-else
if (espressione₁)
{
   espressione₂
} else {
   espressione₃
}
```

Quindi se l'espressione$_1$ risulta vera, si esegue l'espressione$_2$, altrimenti si esegue l'espressione$_3$. Per esempio il seguente codice assegna, nelle due forme equivalenti, alla variabile "max" il massimo tra "alfa" e "beta":

```
if (alfa > beta)
   max = alfa;
else
   max = beta;

// che corrisponde a...
max = (alfa>beta) ? alfa : beta
```

Switch

Come annunciato, l'istruzione switch ha delle differenze dall'if-else, infatti può essere usata solo in alcuni casi dove:

- Viene valutata solamente una variabile, tutte le scelte dipenderanno, infatti, da questa variabile. La variabile deve essere un tipo int, shor, long o char;
- Ogni singolo valore della variabile può controllare solo una scelta. Una scelta finale, chiamata **default** è incaricata di catturare tutti i valori dei casi non specificati;

L'istruzione switch ha la seguente struttura:

```
switch (espressione) {
case elem1:
   istruzione1;
   break; opt
case elem2:
   istruzione2;
   break; opt
...
...
...
case elemn:
   istruzionen;
   break; opt
default:
   istruzione;
   break; opt
}
```

Notiamo che lo switch ha, inoltre, una struttura abbastanza particolare, è a cascata, cioè se eseguo il primo caso (avendo sotto degli altri) e non metto un **break** per uscire, continua ad eseguire anche le istruzioni successive che fanno parte di altre casistiche; questa struttura ricorda molto quella adottata nei file batch (.bat) utilizzati con il DOS e permette di ottenere effetti interessanti sulla selezione del codice, risparmiandone quando è comune a più casistiche. Ma vediamo l'esempio pratico, assumendo di avere la variabile intera "numero" che può assumere valori maggiori od uguali a zero:

```
switch (numero) {
case 0:
   printf("Nessuno");
   break;
case 1:
   printf("Uno");
   break;
case 2:
   printf("Due");
   break;
```

```
case 3:
case 4:
case 5:
   printf("Valore positivo piccolo");
   break;
default:
   printf("Valore positivo grande");
   break;
}
```

20. Controlli Iterativi: WHILE, FOR, BREAK

Presentiamo dei costrutti che permettono di eseguire ciclicamente delle istruzioni fino al verificarsi di alcune condizioni. Questi costrutti prendono il nome di strutture di controllo iterative o istruzioni di ciclo.

Le istruzioni di ciclo sono una delle componenti fondamentali della programmazione e permettono di risparmiare la quantità di codice scritta rendendo quindi il programma più leggero e più facile da comprendere. Le istruzioni di ciclo, come le istruzioni condizionali, hanno bisogno che alcune condizioni vengano verificate affinché il ciclo continui nella sua opera o si interrompa.

Le istruzioni di ciclo del C sono tre:

- il **while**, che continua il suo ciclo fino a quando l'espressione associata non risulta falsa
- il **do-while**, che agisce come il while, ma assicura l'esecuzione delle istruzioni associate almeno una volta
- il **for**, che è il costrutto più usato, versatile e potente tra i tre, ma che proprio per questo è quello a cui bisogna prestare un po' più di attenzione.

Come per le istruzioni condizionali, quando parliamo di "istruzione" ci riferiamo ad una o più istruzioni associate, ricordando che se le istruzioni sono due o più bisogna usare le parentesi graffe per delimitarle, mentre una singola istruzione non ne ha necessariamente bisogno.

While

La struttura del while è la seguente:

```
while (condizione)
   istruzione/i
```

Generalmente l'istruzione o le istruzioni all'interno del while agiscono sulla condizione che il while aspetta essere falsa per poter uscire dal ciclo, questo perché altrimenti il ciclo non terminerebbe. Ad esempio per stampare a video una successione di cifre da 0 a 99, proponiamo il codice seguente:

```
int i = 0;
```

```
while (i != 100)
{
   printf("%d \n", i);
   i++;
}
```

Affinché il while possa verificare la condizione associata, è necessario aver dichiarato la variabile prima del while, questo, come nell'esempio, può essere fatto nella riga soprastante o in un altra parte del programma.

Do - While

Molto simile al **while** è il **do-while** che ha la seguente struttura:

```
do
   istruzione/i
while (condizione)
```

Tenendo valide tutte le considerazioni fatte per il while, va notato che in questo modo si esegue l'istruzione all'interno del do-while almeno una volta, indipendentemente dal fatto che la condizione associata al while risulti vera o falsa. L'esempio seguente mostra come sia possibile utilizzare il do-while, ad esempio, per porre una domanda all'utente e continuare nell'esecuzione solo se l'utente risponde correttamente, oppure ripetere la domanda:

```
do {
   printf("Premere 1 per continuare : ");
   scanf("%d", &valore);
   printf("\n");
} while (valore !=1)
```

For

Prima di spiegare le caratteristiche del for, ne proponiamo la struttura generale:

```
for (inizializzazione ; condizione ; incremento)
   istruzione/i
```

che equivale alla rappresentazione con un while con la seguente struttura:

```
inizializzazione
while (condizione)
   istruzione/i
   incremento
```

Il for, quindi, potrebbe tranquillamente essere sostituito da un while, se non fosse che è più conciso ed è concettualmente usato quando sappiamo a priori il numero di iterazioni che vogliamo fare. I parametri all'interno del for hanno diversi compiti, e sono separati da un punto e virgola:

- Il primo viene eseguito prima di entrare nel ciclo, ed inizializza una variabile, generalmente la variabile è usata come variabile di controllo del ciclo, in poche parole servirà per tenere traccia del numero di iterazioni del ciclo; a differenza del C++, dove la dichiarazione può essere fatta contemporaneamente

all'inizializzazione, nel C, in questo costrutto, la variabile deve essere **dichiarata** fuori, prima del for;

- Il secondo è la condizione (che coinvolge anche la variabile di controllo), che se risulta falsa interrompe l'esecuzione del ciclo;
- Il terzo parametro è l'istruzione di incremento, che viene eseguita dopo ogni ciclo del for; questa istruzione agisce generalmente sulla variabile di controllo incrementandone (o decrementandone) il valore.

Per chiarire meglio l'uso del for presentiamo un semplice codice che conta da zero a cento, stampando a video la variabile di controllo:

```
int i;

for (i=0; i<=100; i++)
   printf("%d \n", i);
```

Ognuna delle tre istruzioni all'interno del for può essere omessa, con il risultato di condizionare il ciclo, o non modificando la variabile, o facendo essere sempre vera la "condizione" del for; in questo ambito bisogna citare un uso particolare del ciclo for, che è quello del cosiddetto "ciclo for infinito". Un ciclo che non termina, abbiamo detto precedentemente, è da considerarsi errore, ma esiste una situazione in cui vogliamo, invece, ciclare all'infinito fino a quando l'utente non decide di interrompere volontariamente il programma, ad esempio avendo un menu generale che ogni volta deve presentarsi sullo schermo. Il for utilizzato in questo caso ha una struttura particolarissima, ovvero non ha nessuno dei tre parametri sopra esposti, ma contiene solamente due caratteri punto e virgola:

```
main ()
{
   for ( ; ; )
   {
      printf ("premi CTRL+Z per interrompere\n");
   }
}
```

È anche possibile utilizzare all'interno delle tre istruzioni, più di una variabile di controllo, in modo da avere un ciclo dinamico; un esempio può essere quello di due variabili, "high" e "low" che convergono:

```
int high;
int low;

for (high=100, low=0; high >= low; high--, low++)
   printf("H=%d - L=%d \n", high, low);
```

Istruzione Break e Continue

Abbiamo già usato l'istruzione break quando stavamo parlando dello switch, senza addentrarci nel dettaglio, adesso, invece, conviene spiegare le caratteristiche del **break** e di un'altra istruzione, **continue**:

- break - esce dal ciclo o dallo switch;
- continue - salta una iterazione del ciclo senza interromperlo;

Per capire la sottile, quanto importante, differenza tra le due istruzioni, presentiamo un semplice codice in cui si legge un numero da tastiera che volgiamo sia compreso tra 0 e 100, se tale valore risulta essere negativo, si esce dal ciclo, mentre se è maggiore di cento si richiede di inserire un valore valido; se il valore è tra 1 e 100, si stampa a video il suo quadrato, se è zero si esce:

```
int valore;

while (scanf("%d", &valore) == 1 && valore != 0)
{
   if (valore < 0)
   {
      printf("Valore non consentito\n");
      break;
      /* esce dal ciclo */
   }

   if (valore > 100)
   {
      printf("Valore non consentito\n");
      continue;
   }

   int quadrato = valore * valore;
   printf("%d \n", quadrato);
}
```

21. Cosa sono gli ARRAY?

Un array può essere definito come una "collezione organizzata di oggetti". Analizziamo la definizione e capiremo molte cose, innanzitutto il concetto di **"collezione"** implica che tali oggetti siano dello stesso tipo, così, prendendo spunto dal mondo reale, potremmo definire un array di mele, che, quindi non può contenere nessun "oggetto pera"; un array in C è una collezione di variabili dello stesso tipo.

"Organizzata" implica che sia possibile identificare univocamente tutti gli oggeti dell'array in modo sistematico; questo in C viene fatto tramite l'uso di indici numerici che, in un array di dimensione N, vanno da 0 ad N-1.

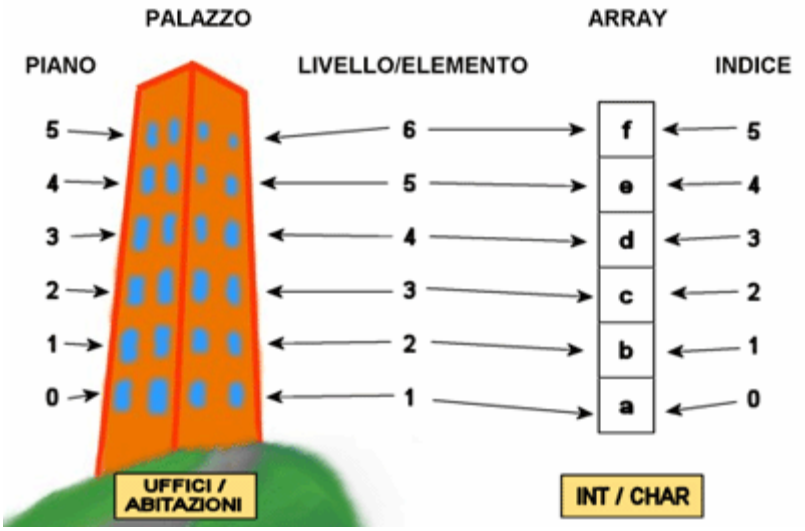
Riprendendo l'esempio della rubrica del cellulare, usato per spiegare le variabili nella lezione 11, si può pensare a quando creiamo un "gruppo suonerie", ad esempio di nome "amici scuola"; tale gruppo può contenere a tutti gli effetti un certo numero di nomi/numeri dei nostri compagni di classe, ecco questo "gruppo" è un array, perchè formato da oggetti dello stesso tipo (nomi/numeri) ed indentificato da un nome (amici scuola) che li accomuna.

Ma vediamo nel dettaglio come è possibile dichiarare un array:

```
int myarray[10];
```

Come si può notare un array viene dichiarato mettendo il nome della variabile (myarray), e ,tra parentesi quadre, la cifra che identifica il numero di elementi dello stesso tipo (int) e quindi la dimensione dell'array.

Nell'esempio, ognuno dei dieci interi viene chiamato **elemento** dell'array e dieci è la dimensione dell'array. In C, come già detto, ogni elemento viene identificato da un numero, contando da 0 (invece che da 1) ed arrivando ad N (la dimensione, nel nostro esempio uguale a 10) - 1 (quindi arriviamo a 9); per far comprendere meglio il concetto abbiamo creato la seguente immagine che ci riporta ad un paragone con il mondo reale;



In questa immagine l'array viene paragonato ad un palazzo. Pensateci bene, quando diciamo che un palazzo ha 5 piani, in realtà ha sei livelli; cioè il piano terra è il primo livello, il primo piano il secondo, e così via; analogamente succede nell'array, se abbiamo un array di dimensione 6, i suoi indici andranno da 0 a 5 e un elemento richiamato, ad esempio, tramite l'indice 3, è il quarto elemento, questo perché si inizia a contare da 0.

L'analogia assume maggiore importanza, anche per far capire che le variabili dell'array sono dello stesso tipo, così come un array di int può contenere solo int (e non char o float), un palazzo che contiene uffici, in questo esempio, può contenere solo uffici (e non abitazioni).

Torniamo al linguaggio C, e vediamo come è possibile dichiarare array di float o di char:

```
float float_array[12];
char char_array[7];
```

Una volta dichiarato un array è possibile assegnare il valore alla posizione corrispondente, richiamandola tramite l'indice, ad esempio se volessi inserire il valore 87.43 nell'array di float alla quinta posizione, basta scrivere:

```
float_array[4] = 87.43
```

Mentre se volessi utilizzare il valore contenuto nella terza posizione dell'array e memorizzarlo in un'altra variabile, dovrei fare:

```
float myvar = float_array[2];
```

22. Gli ARRAY e i cicli FOR

Esiste una forte relazione tra l'uso degli array e di cicli For, per il fatto che un ciclo permette di contare per un certo numero di volte. In questo modo utilizzando una variabile che incrementi (o decrementi) il suo valore ad ogni ciclo, si possono scorrere le posizioni dell'array in maniera semplice, con un notevole risparmio di codice scritto.

Per fare un esempio, assumiamo di avere un array di int di 100 elementi e di voler stampare a video il contenuto di tutte le posizioni dell'array; vista la particolare natura dell'array non inizieremo a contare da 1, ma da 0, fino ad arrivare a novantanove (quindi cento elementi effettivi). Utilizzeremo il comando di stampa sull'array con l'indice, incrementato ogni volta, preso dal ciclo for.

In codice:

```
int int array[100];
int i;
for (i=0; i<100; i++)
{
   printf ("%d", int array[i]);
}
```

In questo caso la variabile di controllo "i" del ciclo for viene usata come indice per l'array; da far notare che, a livello visivo, si capisce subito che si cicla di cento elementi, perché la condizione posta è quella di avere "i<100", iniziando a contare da zero, quindi, si esegue il ciclo fino al novantanovesimo elemento, che è minore di cento; alla successiva iterazione, prima di eseguire il corpo del ciclo, "i" viene incrementata di uno e quindi vale cento, valore che non verifica più la condizione (100 non è minore di 100, semmai è uguale) e che fa terminare il ciclo for; l'intervallo contato (da 0 a 99) è utile per rappresentare tutti e cento gli elementi dell'array, visto che, come spiegato prima, in un array si contano gli elementi partendo dallo zero.

In questo modo potrebbe essere abbastanza semplice anche inizializzare tutti i valori dell'array; supponiamo di voler dare ad ogni elemento dell'array il valore uguale alla sua posizione, con la seguente soluzione si risparmia un sacco di codice:

```
int int array[100];
int i;
for (i=0; i<100; i++)
{
   int array[i] = i;
}
```

Naturalmente un array può essere inizializzato anche passando direttamente i valori, ad esempio, come in questo esempio:

```
int numeri[] = { 7, 23, 4, 94, 120 };
```

in cui si crea un array di 5 elementi, ed è per questo che tra parentesi quadre non si mette la dimensione, in quanto ricavata dal numero di elementi compresi tra le parentesi graffe.

Prendiamo come altro esempio gli array di caratteri, che hanno una particolarità, possono essere inizializzati, senza essere divisi da virgole; ora proponiamo due forme equivalenti di inizializzazione, di cui la prima sicuramente più pratica:

```
char caratteri[] = "Hello World!";

char caratteri[] = { 'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r',
'l', 'd', '!' };
```

Una particolarià cha riguarda il linguaggio C, è che non esiste un vero e proprio "tipo stringa", quello che in altri linguaggi viene chiamato "String" (si pensi al C++ o al Java). In C le stringhe vengono rappresentate tramite array di caratteri, quindi bisogna stare attenti alle operazioni che vi si compiono e ricordarsi che hanno comunque il vantaggio di godere di tutti pregi degli array, tra cui quello di poter scorrere a piacimento nella posizione della stringa stessa. Ogni array di caratteri termina con la sequenza di escape "\0" per far capire che la stringa è terminata.

Le funzioni che possono manipolare il tipo **stringa** possono essere utilizzate solo se si include il file **string.h**; a questo punto ci sono delle precisazioni da fare nell'utlizzo delle stringhe: innanzitutto una stringa viene inizializzata mettendo la frase racchiusa tra doppi apici e non tra un apice singolo,

```
char stringa[] = "Hello World!";
```

Con il printf (ed in maniera analoga con scanf), per stampare una stringa si usa il codice **%s**, come mostrato nell'esempio:

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

int main()
{
   char stringa[] = "Hello World!";
   printf("%s \n", stringa);
}
```

Per sapere la lunghezza di una stringa, si usa la funzione **strlen** che conta, appunto, il numero di caratteri:

```
strlen(stringa);
```

mentre per copiare una stringa in un'altra si usa la funzione **strcpy** che ha la seguente sintassi:

```
strcpy(stringa_destinataria, stringa_sorgente);
```

Se si ricerca un carattere all'interno di una stringa, bisogna usare la **strchr**, come indicato di seguito;

```
strchr(stringa, carattere_da_cercare);
```

La funzione che unisce (concatena) due stringhe è la **strcat** illustrata qui sotto,

```
strcat(stringa_uno, stringa_due);
```

Mentre l'ultima funzione, ed anche la più importante, è la **strcmp** che serve per confrontare la lunghezza di due stringhe.

La funzione restituisce 1 se la prima stringa è più lunga della seconda, 0 se sono lunghe uguali e -1 se la seconda è più lunga della prima.

Adesso che abbiamo visto cosa è un array, come lo si crea e come lo si inizializza, come si accede agli elementi dell'array stesso, possiamo affermare che abbiamo posto le basi per la conoscenza di uno strumento potente ed utile (permette di risparmiare decine di righe di codice) per il programmatore.

23. ARRAY Multidimensionali

Ovviamente la potenza degli array risiede anche nel fatto che si possono usare degli array multidimensionali. Come? in pratica ogni elemento contenuto da un array è a sua volta un array; in questo modo si possono rappresentare facilmente tabelle e matrici, o qualunque altra cosa che richieda un rappresentazione anche superiore, si pensi a programmi di grafica tridimensionale, dove un array cubico può essere usato per disegnare i punti all'interno dello spazio tridimensionale creato, o ad un array a quattro dimensioni, che può servire per registrare anche la variabile tempo.

Con gli array multidimensionali si possono rappresentare insomma cose che hanno più di una dimensione in maniera facile ed intuitiva. L'esempio sotto proposto usa un array bidimensionale per definire una matrice di N righe ed M colonne:

```
int matrix[n][m];
```

i cui elementi possono essere, ad esempio, stampati, utilizzando solo due cicli for, come mostrato qui sotto:

```
int n = 10;
int m = 12;
int matrix[n][m];
int i;
int j;

for (i=0; i<m; i++)
{
   for (j=0; j<n; j++)
   {
      printf("%d", matrix[i][j]);
   }
   printf("\n");
}
```

24. Le funzioni

Le funzioni sono uno degli strumenti più potenti del C; esse infatti permettono un notevole risparmio e riuso del codice, con ovvii vantaggi del programmatore. Le funzioni esistono più o meno in tutti i linguaggi e vengono chiamate anche **procedure** o **subroutine**. Alcuni linguaggi fanno distinzione tra funzioni che ritornano un valore e quelle che, invece, non ritornano valori; il C assume che ogni funzione ritorni un valore, questo accade utilizzando l'istruzione **return** seguita, eventualmente, da un valore; se non si mette l'istruzione return, nessun parametro deve essere passato quando si chiama la funzione. Prendiamo ad esempio, la definizione di una funzione che prende un double ed un int e fa un elevamento a potenza, restituendo il risultato:

```
double elevamento potenza(double valore, int potenza)
{
   double valore ritorno = 1.0;
   int i;

   for(i=0; i<potenza; i++)
   {
      valore ritorno *= valore;
   }
return(valore ritorno);
}
```

Analizzando questa funzione possiamo innanzitutto far notare che il calcolo eseguito, per quanto complesso e poco ortodosso, restituisce il valore corretto anche quando la potenza vale zero; iniziamo, dunque, a studiare il codice nel dettaglio:

```
double elevamento_potenza(double valore, int potenza)
```

Questa è la definizione della funzione, che ci dice il tipo del valore di ritorno (double), il nome della funzione (elevamento_potenza) e la lista di argomenti usati dalla funzione con il tipo (double e int) ed il nome (valore e potenza) corrispondente;

```
return(valore_ritorno);
```

Quando si raggiunge un'istruzione "return", il controllo del programma ritorna a chi ha chiamato la funzione. Il valore ritornato è quello posto dopo la parola return; se si chiude la funzione prima di mettere un'istruzione "return", la funzione ritorna automaticamente, ed il valore ritornato potrebbe non avere un significato valido. Il valore ritornato può essere manipolato a piacimento, infatti se una funzione restituisce un risultato, possiamo assegnarlo, ad esempio, ad una variabile che poi potremmo usare all'interno del programma come qualsiasi altra variabile;

```
double val = 100.0;
int pot = 3;
double risultato = elevamento_potenza(val, pot);
```

Esistono anche funzioni che non ritornano alcun valore, in questo caso si parla di funzioni void, come mostrato di seguito:

```
void stampa errore(int linea)
{
```

```
   fprintf(stderr, "Errore: linea %d\n", linea);
}
```

Una funzione void non deve avere necessariamente un'istruzione "return", anche se può essere usata per uscire dalla funzione in maniera opportuna, un po' come l'uso del "break" all'interno dei cicli. In questo caso abbiamo presentato anche l'uso dell'istruzione "fprintf", una variante della "printf", che spiegheremo in dettaglio nella lezione 16; basti sapere che in questo caso abbiamo stampato il messaggio di errore sullo "stderr" che può essere un file o un messaggio a video (generalmente quest'ultimo). Per richiamare la funzione, basta, ovviamente, scrivere il nome con, tra gli argomenti, il numero di linea in cui si è verificato l'errore;

```
int linea_err = 14;
stampa_errore(linea_err);
```

Va fatta una piccola nota riguardante le funzioni, in merito alla **prototipazione delle funzioni**, ovvero la creazione di prototipi. Un prototipo di una funzione non è altro che la sua dichiarazione, senza specificare il corpo della funzione stessa; si scrive, quindi, solo la dichiarazione iniziale comprendente il nome ed il tipo restituito dalla funzione, e gli argomenti passati come parametro; questo avviene perché ogni funzione è utilizzabile solamente quanto è stata dichiarata, quindi se in un pezzo di codice, prima di dichiarare la funzione, voglio usare tale funzione, non posso farlo, perché "fisicamente" non è ancora stata creata. Creando dei prototipi si ovvia a questo problema e si hanno dei vantaggi anche in termini di funzionalità ed organizzazione del codice stesso. Ecco due esempi, il primo errato, il secondo corretto:

```
// questo non funziona
#include <stdio.h>

int main()
{
   int var = 5;
   stampa doppio(var);
}

void stampa doppio(int variabile)
{
   printf("Il doppio di %d è %d ", variabile, 2*variabile);
}

/* ------------------------------------------------------- */

// questo funziona
#include <stdio.h>

void stampa doppio(int variabile); // prototipo della funzione

int main()
{
   int var = 5;
   stampa doppio(var);
}

void stampa doppio(int variabile)
{
```

```
   printf("Il doppio di %d è %d ", variabile, 2*variabile);
}
```

In C, quando si passano gli argomenti (variabili) alle funzioni, bisogna prestare attenzione al passaggio di array ad una o più dimensioni; la regola dice che la prima dimensione dell'array può non essere specificata, mentre la seconda e le altre devono esserlo. Un esempio chiarirà meglio il concetto:

```
void stampa array uni(int dim, int array uni[])
{
   int i;
   for(i=0; i<dim; i++)
   {
      printf("%d ", array uni[i]);
   }
}

void stampa array bid(ind dimx, int dimy, int array bid[][6]);
{
   int i, j;
   for(i=0; i<dimx; i++)
   {
      for(j=0; j<dimy; j++)
      {
         printf("%d ", array uni[i][j]);
      }
      printf("\n");
   }
}
```

25. Tipi di dato avanzato - 1

Tipi di Dati avanzati
Una volta presa dimestichezza con le principali caratteristiche del C si possono utilizzare dei tipi di Dati avanzati. Qui di seguito presentiamo i tipi di dati avanzati del C, ovvero tipi non semplici nella loro rappresentazione, e per questo spiegati nel dettaglio.

Strutture
Le strutture del C sono simili ai **record** del Pascal e sostanzialmente permettono un'aggregazione di variabili, molto simile a quella degli array, ma a differenza di questi non ordinata e non omogenea (una struttura può contenere variabili di tipo diverso). Per denotare una struttura si usa la parola chiave **struct** seguita dal nome identificativo della struttura, che è opzionale. Nell'esempio sottostante si definisce una struttura "libro" e si crea un'istanza di essa chiamata "biblio":

```
// dichiarazione della struct
struct libro
   {
      char titolo[100];
      char autore[50];
      int anno pubblicazione;
      float prezzo;
   };
```

```
//dichiarazione dell'istanza biblio
struct libro biblio;
```

La variabile "biblio" può essere dichiarata anche mettendo il nome stesso dopo la parentesi graffa:

```
// dichiarazione della struct e della variabile biblio
struct libro
   {
      char titolo[100];
      char autore[50];
      int anno pubblicazione;
      float prezzo;
   } biblio;
```

mentre è possibile pre-inizializzare i valori, alla dichiarazione, mettendo i valori (giusti nel tipo) compresi tra parentesi graffe:

```
struct libro biblio = {"Guida al C", "Fabrizio Ciacchi", 2003,
45.2};
```

Per accedere alle variabili interne della struttura si usa l'operatore "."; una volta che si può accedere ad una variabile interna questa può essere trattata e/o manipolata come qualsiasi altra variabile:

```
// assegna un valore al prezzo del libro
biblio.prezzo = 67.32;

// assegna ad una variabile int l'anno di pubblicazione del libro
int anno = biblio.anno pubblicazione;

// stampa il titolo del libro
printf ("%s \n", biblio.titolo);
```

Nuovi tipi di dato

Per definire nuovi tipi di dato viene utilizzata la funzione **typedef**. Con typedef e l'uso di struct è possibile creare tipi di dato molto complessi, come mostrato nell'esempio seguente:

```
typedef struct libro
   {
      char titolo[100];
      char autore[50];
      int anno pubblicazione;
      float prezzo;
   } t libro;

t_libro guida={"Guida al C", "Fabrizio Ciacchi", 2003, 45.2};
```

In questo modo abbiamo definito un nuovo tipo di nome "t_libro", che non è altro che una struttura; "guida" è la variabile creata di tipo "t_libro"; come per ogni altro tipo di dato, anche con "t_libro" si possono creare degli array:

```
t_libro raccolta[5000];
```

e, per accedervi, o per inizializzare i valori, è sufficiente utilizzare l'indice per identificare l'elemento dell'array ed il punto (.) per accedere alle variabili interne del tipo "t_libro";

```
// assegna un valore al prezzo del 341° libro
raccolta[340].prezzo = 67.32;

// assegna ad una variabile int l'anno di pubblicazione del 659° libro
int anno = raccolta[658].anno pubblicazione;

// stampa il titolo del 23° libro
printf ("%s \n", raccolta[22].titolo);
```

Unioni

Il tipo di dato **union** serve per memorizzare (in istanti diversi) oggetti di differenti dimensioni e tipo, con, in comune, il ruolo all'interno del programma. Si alloca la memoria per la più grande delle variabili, visto che esse non possono mai essere utilizzate contemporaneamente (la scelta di una esclude automaticamente le altre), condividendo il medesimo spazio di memoria. In C una unione viene definita tramite la parola chiave **union**, come mostrato di seguito:

```
union numero
   {
      short numero piccolo;
      long numero lungo;
      double numero grande;
   } numeri;
```

Una unione è molto simile ad una struttura, ed è medesimo il modo di accedervi. Inoltre, usata in combinazione con una struttura, si possono ottenere gradevoli effetti in cui viene selezionato automaticamente il tipo giusto della union:

```
typedef struct
   {
      int max passeggeri;
   } jet;

typedef struct
   {
      int max altezza;
   } elicottero;

typedef struct
   {
      int max carico;
   } aereocargo;

typedef union
   {
      jet un jet;
      elicottero un elicottero;
      aereocargo un aereocargo;
   } velivolo;
```

```
typedef struct
   {
      tipo velivolo tipo;
      int velocita;
      velivolo descrizione;
   } un_velivolo;
```

In questo esempio viene definita un'unione di nome "velivolo" che può essere un jet, un elicottero o un aereocargo. Nella struttura "un_velivolo" c'è un oggetto che identifica il tipo e che permette di selezionare la struttura giusta al momento giusto.

26. Tipi di dato avanzato - 2

Casting
Quando si lavora con tipi di dati diversi tra loro, che siano primitivi (int, char, float, ecc.) od avanzati, può essere necessario convertire valori da un tipo ad un altro. Questa operazione si chiama **casting**, ad assume il nome di **coercizione (coercion)** quando si forza la conversione esplicita di un tipo ad un altro tipo, conversione magari non prevista automaticamente. In C, per convertire esplicitamente un tipo ad un altro tipo, si usa l'operatore **( )**, queste parentesi tonde prendono il nome di operatore di **cast**; all'interno delle parentesi bisogna mettere il nuovo tipo al quale vogliamo passare:

```
// Casting da float ad int
int numero;
float reale;

reale = 47.19;
numero = (int)reale; // vale 47
```

ad esempio è possibile forzare la conversione di tipo dal tipo float al tipo int, o dal tipo char al tipo int;

```
// Casting da int a float
int numero;
float reale;

numero = 18;
reale = (float)numero;

/* ------------------------------------------------------ */

// Casting da char ad int
int numero;
int lettera;

lettera = 'A';
numero = (int)lettera; // vale 65, il valore ASCII di A
```

alcune conversioni di tipo vengono eseguite automaticamente (casting implicito), generalmente quando si utilizzano tipi di altro tipo come int. E' possibile, e consigliabile, utilizzare l'operatore di casting anche quando si compiono operazioni sui valori come, ad esempio, la divisione:

```
int primo;
int secondo;
float ris div;

ris_div = (float)primo / (float)secondo;
```

Comunque la regola dice che se si è nel dubbio bisogna sempre mettere l'operatore di cast, ovvero bisogna sempre convertire esplicitamente i tipi. Il casting è e rimane un'operazione potente che, se ben utilizzata, può apportare notevoli benefici ad un programma.

Tipo enumerazione

Il tipo enumerazione è abbastanza particolare, perché permette di associare a delle costanti letterali, un valore intero; in questo modo possiamo utilizzare tali nomi per identificare il loro valore; facciamo un esempio utilizzando i giorni della settimana:

```
enum giorni { lun, mar, mer, gio, ven, sab, dom } settimana;
```

In questo caso abbiamo definito una nuova variabile di nome "settimana" e di tipo enumerazione "giorni"; l'identificatore "lun" assume il valore 0, "mar" assume il valore 1, e così via; in poche parole si ha un indice iniziale "0" e gli altri assumono una numerazione progressiva. Questo ci può essere molto utile se dobbiamo scrivere un programma che operi sui giorni della settimana (come un calendario); se non esistesse il tipo enumerazione il programma non potrebbe assegnare alcun "valore" ad un determinato giorno e quindi sarebbe molto più difficile (e dispendioso in termini di codice) lavorare in tal senso.
E' possibile, però, assegnare alle costanti anche valori iniziali diversi da 0, o valori non numerici, come spiegato nei due esempi:

```
// valori non numerici
enum seq escape { suono = '\a', cancella = '\b', tab = '\t', invio
= '\r' };

// indice iniziale diverso da 0
enum mesi { gen = 1, feb, mar, apr, mag, giu, lug, ago, set, ott,
nov, dic };
```

che implica "feb" uguale a 2, "mar" uguale a 3, e così via.

Variabili statiche

La variabili hanno la possibilità di essere visibili solo in una funzione (variabili locali) o in tutto il programma (variabili globali). Le variabili locali possono avere un'altra caratteristica, possono essere **statiche**, ovvero, essere definite una volta, ed avere vita propria all'interno di una funzione, con il vantaggio di mantenere il loro valore intatto, cioè il valore che una variabile statica ha all'uscita delle funzione è il medesimo quando si rientra (richiamandola) nella funzione stessa. La parola chiave che identifica le variabili statiche è **static** e deve essere messa prima del tipo della variabile. Come mostrato nell'esempio, vengono definite due variabili in una funzione, una statica ed una non:

```
include <stdio.h>

// prototipo della funzione
void stampa();
```

```
int main()
{
   int i;
   for(i=0; i<5; i++)
   {
      stampa();
   }
}

void stampa()
{
   int alfa = 0;
   static int beta = 0;
   printf("(int) alfa = %d, (static int) beta = %d \n", alfa,
beta);
   ++alfa;
   ++beta;
}
```

L'esecuzione del programma genere il seguente output a video:

```
(int) alfa = 0, (static int) beta = 0
(int) alfa = 0, (static int) beta = 1
(int) alfa = 0, (static int) beta = 2
(int) alfa = 0, (static int) beta = 3
(int) alfa = 0, (static int) beta = 4
```

27. I Puntatori

I puntatori sono la parte più importante della programmazione in C, quella che permette di lavorare "a basso livello" (cioè agendo su singole istruzioni del processore), mantenendo però una praticità unica nel suo genere.

Cosa è un puntatore? **Un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile**. Quando dichiariamo una variabile, a questa verrà riservato un indirizzo di memoria, ad esempio la posizione 1000; un puntatore contiene, appunto, l'indirizzo di tale variabile (quindi il valore 1000). L'importanza risiede nel fatto che si possono manipolare sia il puntatore che la variabile puntata (cioè la variabile memorizzata a quell'indirizzo di memoria).

Per definire un puntatore è necessario seguire la seguente sintassi:

```
// variabile normale
int variabile;

// puntatore
int *puntatore;
```

L'asterisco (*) viene chiamato **operatore di indirezione o deferenziazione** e restituisce il contenuto dell'oggetto puntato dal puntatore; mentre l'operatore **&** restituisce l'indirizzo della variabile e va usato nella seguente forma:

```
// assegno al puntatore l'indirizzo di variabile
puntatore = &variabile;
```

Per fare un esempio pratico, assumiamo di avere due variabili, **alfa** e **beta** ed un puntatore di nome **pointer**; assumiamo anche che alfa risieda alla locazione di memoria "100", beta alla locazione "200" e pointer alla locazione "1000" e vediamo, eseguendo il codice sotto proposto, il risultato ottenuto:

```
#include <stdio.h>

int main()
{
   int alfa = 4;
   int beta = 7;
   int *pointer;

   pointer = &alfa;
   printf("alfa -> %d, beta -> %d, pointer -> %d\n", alfa, beta,
pointer);
   beta = *pointer;
   printf("alfa -> %d, beta -> %d, pointer -> %d\n", alfa, beta,
pointer);
   alfa = pointer;
   printf("alfa -> %d, beta -> %d, pointer -> %d\n", alfa, beta,
pointer);
   *pointer = 5;
   printf("alfa -> %d, beta -> %d, pointer -> %d\n", alfa, beta,
pointer);
}
```

Questo stamperà a video i valori corretti di alfa e beta (4 e 7) e l'indirizzo di alfa memorizzato nel puntatore (100):

```
alfa -> 4, beta -> 7, pointer -> 100
```

poi assegniamo a beta il valore dell'oggetto puntato da pointer (alfa), e quindi il valore 4:

```
alfa -> 4, beta -> 4, pointer -> 100
```

successivamente assegniamo ad alfa il valore memorizzato in pointer e quindi l'indirizzo di memoria di alfa stesso,

```
alfa -> 100, beta -> 4, pointer -> 100
```

continuiamo memorizzando in alfa (l'oggetto puntato da pointer) il valore 5,

```
alfa -> 5, beta -> 4, pointer -> 100
```

Esiste anche un aritmetica base dei puntatori ai quali è possibile aggiungere o togliere dei valori, che generalmente vengono rappresentati come blocchi di memoria; per intendersi, un puntatore esiste sempre in funzione del tipo di oggetto puntato, se creo un puntatore ad int, il blocco di memoria vale 4 byte, un puntatore a char, invece, usa blocchi di 1 byte. In questo modo se utilizzo l'operatore **++**, incremento del blocco a seconda del tipo di variabile puntata.

Come mostreremo di seguito i puntatori possono essere usati con successo in combinazione con **funzioni**, **array** e **strutture**.

28. Puntatori e funzioni

I puntatori risultano molto utili anche usati con le funzioni. Generalmente vengono passati alle funzioni gli argomenti (le variabili) per **valore** (utilizzando return). Ma questo modo di passare gli argomenti, non modifica gli argomenti stessi e quindi potrebbe risultare limitativo quando, invece, vogliamo che vengano modificate le variabili che passiamo alle funzioni.

Pensiamo ad un caso esemplare, uno su tutti, l'uso della funzione **swap (alfa, beta)** che scambia il valore di alfa con beta e viceversa; in questo caso il valore restituito dalla funzione (con return) non va minimamente ad intaccare i valori delle variabili alfa e beta, cosa che vogliamo, invece, accada per poter effettivamente fare lo scambio. In questo caso si passano alla funzione, non i valori delle variabili, ma il loro indirizzo, trovandoci, quindi, ad operare con i puntatori a tali valori. Facciamo un esempio pratico per chiarire:

```
#include <stdio.h>

void swap(int *apt, int *bpt);

int main()
{
   int alfa = 5;
   int beta = 13;

   printf("alfa -> %d, beta -> %d\n", alfa, beta);

   swap(&alfa, &beta);

   printf("alfa -> %d, beta -> %d\n", alfa, beta);
}

void swap(int *apt, int *bpt)
{
   int temp;
   temp = *apt;
   *apt = *bpt;
   *bpt = temp;
}
```

29. Puntatori e Array

Una digressione su array e puntatori potrebbe portare via moltissimo tempo; in questo ambito ci limitiamo ad accennare alla correlazione tra i due strumenti a disposizione nel linguaggio C.

L'aspetto che accomuna i puntatori e gli array è sostanzialmente il fatto che entrambi sono memorizzati in locazioni di memoria sequenziali, ed è quindi possibile agire su un array (se lo si vede come una serie di blocchi ad indirizzi di memoria sequenziali) come se si stesse agendo su un puntatore (e viceversa); ad esempio se dichiariamo un array "alfa" ed un puntatore "pointer":

```
// definisco un array ed una variabile intera
int alfa[20], x;

// creo il puntatore;
int *pointer;

// puntatore all'indirizzo di a[0]
pointer = &a[0];
// x prende il valore di pointer, cioè di a[0];
x = *pointer;
```

Se volessi scorrere, ad esempio, alla posizione i-ma dell'array, basterebbe incrementare il puntatore di i; cioè

```
pointer + i "è equivalente a" a[i]
```

Graficamente questo viene rappresentato come:

```
          0   1   2   3   4 ... n

alfa     |   |   |   |   |   |   |   |

pointer   +1 +2 .. +i
```

Comunque array e puntatori hanno delle sostanziali differenze:

- Un puntatore è una variabile, sulla quale possiamo eseguire le più svariate operazioni (come l'assegnamento).
- Un array non è una variabile convenzionale (è un contenitore di variabili) ed alcune operazioni da array a puntatore potrebbero non essere permesse.

30. Puntatori e Strutture

Nell'uso congiunto di strutture e puntatori, come mostrato nell'esempio seguente:

```
struct PIPPO { int x, y, z; } elemento;
struct PIPPO *puntatore;

puntatore = &elemento;

puntatore->x = 6;
puntatore->y = 8;
puntatore->z = 5;
```

si può notare che abbiamo creato una struttura di tipo PIPPO e di nome "elemento", ed un puntatore ad una struttura di tipo PIPPO. Per accedere ai membri interni della struttura "elemento" abbiamo usato l'operatore **->** sul puntatore alla struttura. Inoltre è possibile utilizzare i puntatori anche per, ad esempio, le liste semplici (tratteremo le liste nella lezione 15), in cui per collegare due elementi, basta scrivere:

```
typedef struct { int inf; ELEMENTO *pun; } ELEMENTO;

ELEMENTO ele1, ele2;

// in questo modo faccio puntare il primo elemento al secondo
ele1.pun = &ele2;
```

31. Allocazione dinamica della Memoria

Il C è, per natura, un linguaggio molto flessibile, e la sua gestione della memoria contribuisce a renderlo ancora più flessibile. A differenza di altri linguaggi (come il C++ o il Java), il C permette di assegnare la giusta quantità di memoria (solo e solamente quella necessaria) alle variabili del programma. In particolare l'uso della memoria allocata dinamicamente risulta utile con gli array. Utilizzare queste caratteristiche del C permette di creare programmi altamente portabili, in quanto utilizzano di volta in volta i valori giusti per la piattaforma.

Una piccola nota su **come è organizzata la memoria** è doverosa: la memoria è divisa sostanzialmente in due parti, una statica, che contiene tutto quello che sappiamo verrà allocato (come una variabile int) e che si chiama **Stack**, ed una dinamica, cioè in cui la dimensione di memoria per rappresentare qualche elemento del programma può cambiare durante l'esecuzione del programma, che si chiama **Heap**.

Le funzioni utilizzate per gestire la memoria dinamica sono principalmente **malloc()** e **calloc()** (calloc() è stata rinominata dall'ANSI), adibite all'allocazione della memoria, **free()** che, come si intuisce, serve per liberare la memoria allocata, e **realloc()** la cui funzione è quella di permettere la modifica di uno spazio di memoria precedentemente allocato. Un comando particolarmente utile risulta essere **sizeof**, che restituisce la dimensione del tipo di dato da allocare.

Per far funzionare il programma dobbiamo includere la libreria **malloc.h**, senza la quale queste funzioni non potrebbero fare il loro dovere. Presentiamo un esempio per chiarire meglio l'uso di queste funzioni, in particolare allocheremo la memoria per un array con malloc(), lavoreremo sull'array stesso ed, infine, libereremo la memoria con free().

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

int main()
{
   int numero, *array, i;
   char buffer[15];
   int allocati;

   numero = 100;
   printf("Numero di elementi dell'array: %d", numero);

   array = (int *)malloc(sizeof(int) * numero);

   if(array == NULL)
   {
      printf("Memoria esaurita\n");
      exit(1);
   }

   allocati = sizeof(int) * numero;

   for(i=0; i<numero; i++)
   {
      array[i] = i;
   }

   printf("\nValori degli elementi\n");
   for(i=0; i<numero; i++)
   {
      printf("%6d%c", array[i], i%10 == 9 ? '\n' : ' ');
   }

   printf("\n\nNumero elementi %d\n", numero);
   printf("Dimensione elemento %d\n", sizeof(int));
   printf("Bytes allocati %d\n", allocati);

   free(array);
   printf("\nMemoria Liberata\n");

   return 0;
}
```

In questo programma possiamo notare l'uso della funzione **malloc()** che ritorna un puntatore a carattere, corrispondente al punto di inizio, in memoria, della porzione riservata della dimensione "intera" passata come argomento; se la memoria richiesta non può essere allocata, ritorna un puntatore nullo.

Nel caso citato si può notare che è stato usata la funzione **sizeof** per specificare il numero esatto di byte, mentre è stata usata la **coercizione** per convertire il tipo di dato "puntatore a carattere" a "puntatore ad int", questo per garantire che i puntatori aritmetici vengano

rappresentati correttamente. Esiste, inoltre, un legame molto forte tra puntatori ed array per trattare la memoria riservata come un array, ed è per questo che in realtà la variabile "array" non è stata definita inizialmente come array, ma come puntatore ad int (vedere lezione 13).

32. Allocazione dinamica della Memoria: funzione realloc()

Adesso proporremo l'uso della funzione **realloc()** per definire un **array flessibile**, ovvero un array cui viene riservata della memoria suddivisa in blocchi di dimensione arbitraria. Una volta "saturato" il primo blocco, utilizziamo **realloc()** per allocare il blocco successivo. La suddivisione in blocchi avviene durante la fase di lettura, che essendo, appunto, dinamica, permette di minimizzare le chiamate a **realloc()** e di avere un dimensionamento abbastanza preciso.

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

int main()
{
   char buffer[20];
   int i=0, n=0, x, *array, nb;

   /* byte allocati */
   int allocati;
   /* byte in un blocco */
   int dimbloc;
   /* byte in un intero */
   int dimint;
   /* byte contenenti interi */
   int usati;

   nb = 1;
   printf("Elementi in un blocco: %d\n", nb);

   dimint = sizeof(int);
   dimbloc = nb * dimint;
   usati = 0;

   array = (int *)malloc(dimbloc);
   if(array == NULL)
   {
      printf("Memoria insufficiente\n");
      exit(1);
   }

   allocati = dimbloc;
   printf("Allocati: %d bytes\n", allocati);
   printf("Input di interi terminati da # :\n");

   while(scanf("%d", &x))
   {
      usati += dimint;
      if(usati>allocati)
```

```
      {
         allocati += dimbloc;
         array = (int *)realloc(array, allocati);
         if(array == NULL)
         {
            printf("Memoria insufficiente\n");
            exit(1);
         }
         i++;
      }
      /* in questo modo vengono letti n interi */
      array[n++] = x;
   }

   printf("\n");
   printf("Allocati: %d bytes\n", allocati);
   printf("Dim. blocchi: %d bytes\n", dimbloc);
   printf("Dim. intero: %d bytes\n", dimint);
   printf("Usati: %d bytes\n", usati);
   printf("Chiamate realloc: %d\n", i);
   printf("Numeri: %d\n", n);
   printf("\nEcco i numeri\n");

   for(i=0; i<n; i++)
   {
      printf("%5d%c", array[i], i%10 == 9 ? '\n' : ' ');
   }

   printf("\n");

   return 0;
}
```

La sintassi della funzione **realloc()** ha due argomenti, il primo riguarda l'indirizzo di memoria, il secondo specifica la nuova dimensione del blocco; il tipo restituito è un tipo puntatore a void, il quale nel programma è stato castato a puntatore ad intero, per verificare che il suo valore fosse NULL.

33. Introduzione alle LISTE

Le liste sono uno degli argomenti più delicati da trattare, per il semplice fatto che sono uno strumento potente e versatile, ma anche molto fragile. Basti pensare che l'uso delle liste permette di impostare programmi di ordinamento in modo molto efficiente (un esempio su tutti l'algoritmo **MergeSort**), ed offre quella dinamicità, tra l'altro tipica del C, di cui si può avere bisogno durante lo sviluppo di un programma.

Una lista non è altro che una collezione di elementi omogenei, ma, **a differenza dell'array**, occupa in memoria una posizione qualsiasi, che tra l'altro può cambiare dinamicamente durante l'utilizzo della lista stessa; inoltre la sua dimensione non è nota a priori e può variare nel tempo (l'opposto dell'array, in cui la dimensione è ben nota e non è modificabile). Una lista può contenere uno o più campi contenenti informazioni, e, necessariamente, deve contenere un puntatore per mezzo del quale è legato all'elemento successivo.

La lista base ha un solo campo informazione ed un puntatore, come mostrato di seguito:

```
Elemento = informazione + puntatore
```

che, tradotto in codice, risulta essere:

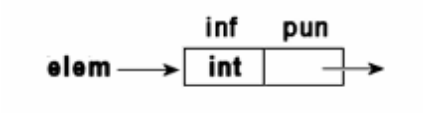
```
struct elemento {
   int inf;
   struct elemento *pun;
}
```

Una particolarità delle liste (a differenza, ad esempio, degli array) è che sono costituite da due funzioni del C, le strutture ed i puntatori, quindi non sono un tipo di dato nuovo, basato su implementazioni particolari, ma sono il risultato di un uso sapiente di tali costrutti.

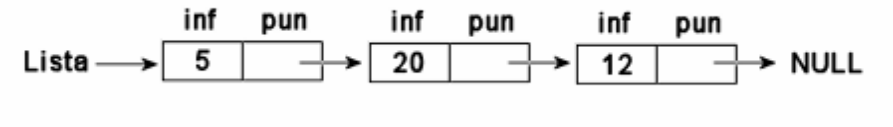
Come si è detto prima, una lista può contenere uno o più campi di informazione, che possono essere di tipo int (come nell'esempio), char, float, ecc.; mentre deve esserci sempre un campo che punta ad un altro elemento della lista, che è NULL se non ci sono altri elementi, mentre risulta essere una struttura elemento nel caso vi sia un successore. Per dichiarare una lista, basta scrivere (riferendosi alla struttura dichiarata precedentemente):

```
struct elemento *lista;
```

che altro non è che un puntatore ad una struttura elemento, e come tale potrebbe essere inizializzata anche ad un valore NULL, che identificherebbe una lista vuota. In questo modo definiamo, quindi, una **lista lineare**, ovvero una lista che deve essere visitata (o scandita) in ordine, cioè dal primo elemento fino all'ultimo, che viene identificato perché punta a NULL. Da ricordare che, anche se nella sua rappresentazione, una lista risulta essere sequenziale, in realtà l'allocazione di memoria relativamente agli elementi è libera, cioè ogni volta che devo aggiungere un elemento alla lista, devo allocare la memoria relativa, connetterlo all'ultimo elemento ed inserirvi l'informazione. La rappresentazione grafica di un elemento della lista è la seguente:



mentre quella di una lista risulta essere come segue:



34. Gestione di una LISTA - 1

Adesso abbiamo intenzione di presentare un programma che permetta di memorizzare e stampare una lista composta da un determinato numero di interi. Sebbene il programma possa sembrare semplice, in realtà la sua stesura, ed il ragionamento posto alla base di essa, è particolarmente accattivante, anche perché non si possono commettere errori, ne sintattici, ne logici.

Innanzitutto dobbiamo partire con l'inclusione delle librerie, in questo caso includiamo solamente **stdio.h**, per le più comuni operazioni di input/output, e **malloc.h** per l'allocazione dinamica della memoria; nessuna libreria particolare deve essere inclusa, in quanto, come già detto prima, una lista è composta da elementi preesistenti nel linguaggio C.

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

Successivamente creiamo il tipo elemento, alla base della lista, che contiene un campo intero di nome "inf" ed un campo puntatore al tipo elemento di nome "pun":

```
/* struttura elementi della lista */
struct elemento {
   int inf;
   struct elemento *pun;
}
```

Si dovranno quindi creare i prototipi delle due funzioni adibite alla soluzione del nostro problema; la prima funzione ha il compito di creare la lista, chiedendo i dati di input all'utente tramite tastiera, che poi verrà restituita dalla funzione stessa; la seconda, invece, restituisce un void e prende in input la lista da stampare.

```
/* prototipi delle funzioni */
struct elemento *crea_lista();
void visualizza_lista(struct elemento *);
```

Il codice prosegue con l'ovvia creazione del main, la dichiarazione della una variabile lista di tipo puntatore ad elemento, l'esecuzione della funzione **crea_lista()**, che ha il compito di "riempire" di valori la lista, e l'esecuzione della funzione **visualizza_lista()** che stamperà a video tutti gli elementi della lista;

```
int main()
{
   struct elemento *lista; // puntatore della lista
   lista = crea lista(); // crea la lista
   visualizza lista(lista); // stampa la lista
}
```

Procediamo con la definizione del corpo della funzione **crea_lista()**, la quale crea due puntatori ad elemento, uno di nome **p** e l'altro di nome **punt**; queste due variabili serviranno per scorrere la lista, infatti p è il puntatore al primo elemento della lista, mentre punt è un puntatore ausiliario che permette di scorrere la lista; la variabile **i** è l'indice del ciclo, mentre **n** serve a memorizzare il numero degli elementi che si intende inserire.

```
struct elemento *crea_lista()
{
   struct elemento *p, *punt;
   int i, n;
```

La variabile n viene inserita tramite tastiera dall'utente,

```
   printf("\n Specificare il numero di elementi... ");
   scanf("%d", &n);
```

Se n vale 0, viene richiesto di creare una lista vuota, quindi si assegna a **p** il valore null,

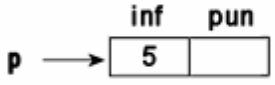
```
   if(n==0)
   {
      p = NULL; // lista vuota
```

altrimenti si costruisce il primo elemento della lista, chiedendo il suo valore da tastiera:

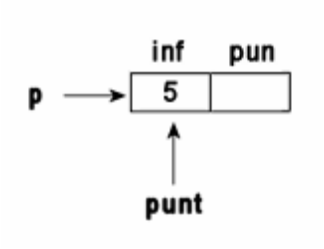
```
   } else {
      /* creazione primo elemento */
      p = (struct elemento *)malloc(sizeof(struct elemento));
      printf("\nInserisci il primo valore: ");
      scanf("%d", &p->inf);
      punt = p;
```

Vediamo che è stata usata la funzione **malloc()** insieme a **sizeof** per allocare dinamicamente la memoria necessaria ad un elemento della lista (vedere lezione 14); il valore ritornato viene castato ad un puntatore allo spazio allocato, che viene assegnato a p.

Assumendo che sia stato inserito un valore di n (il numero di elementi della lista) uguale a 3; nella prima iterazione, viene creato il primo elemento della lista con il codice sopra esposto e viene assegnato (tramite tastiera), ad esempio, il valore 5;



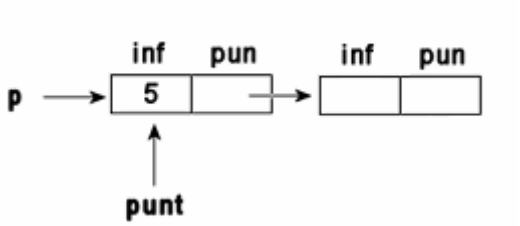
Successivamente viene creato un altro puntatore alla prima posizione, di nome "punt", tale puntatore ausiliario, servirà per scorrere gli elementi della lista e mantenere il collegamento all'ultimo elemento inserito.



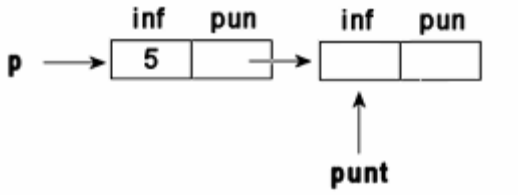
Proseguendo con l'esecuzione del codice, arriviamo ad inserire il secondo ed il terzo elemento (grazie al ciclo for):

```
      /* creazione elementi successivi */
      for(i=2; i<=n; i++)
      {
         punt->pun = (struct elemento *)malloc(sizeof(struct
elemento));
         punt = punt->pun;
         printf("\nInserisci il %d elemento: ", i);
         scanf("%d", &punt->inf);
      } // chiudo il for
      punt->pun = NULL; // marcatore fine lista
   } // chiudo l'if-else
   return(p);
} // chiudo la funzione
```

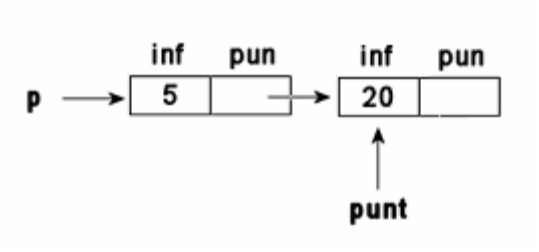
Per prima cosa viene creato un altro oggetto della lista, identificato con punt->pun,



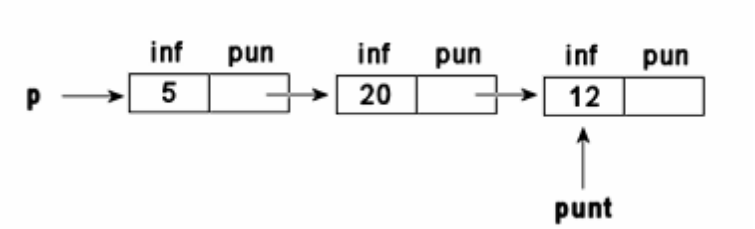
poi "punt", il puntatore ausiliario, viene fatto puntare, non più al primo elemento, bensì al secondo, all'atto pratico "punt" diventa il puntatore dell'oggetto da lui puntato (punt = punt->pun;).



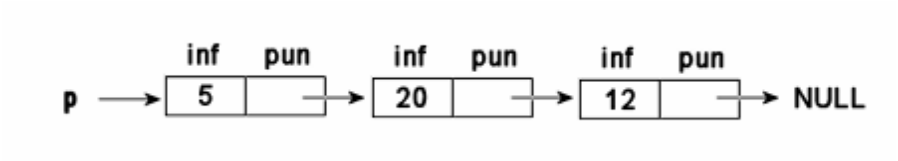
Quindi viene inserito il campo informazione dell'elemento tramite l'input da tastiera dell'utente; in questo caso viene inserito il valore 20;



La procedura, ovviamente, si ripete per il terzo elemento, e, se vi fossero stati altri elementi da aggiungere, si sarebbe ripetuta fino all'inserimento dell'ultimo elemento;



All'ultimo elemento, alla fine del ciclo, viene fatto puntare il valore NULL che serve come marcatore di fine lista.



La funzione **visualizza _lista()** risulta essere molto più semplice di quanto si pensi; essa prende in input la lista da scorrere, viene considerato il primo oggetto puntato da p (che è il primo elemento) e viene stampata la sua informazione; l'iterazione inizia e si assegna a p il suo oggetto puntato, ovvero il secondo elemento, di cui verrà stampata l'informazione relativa; il ciclo continua fino a quando il puntatore p non vale NULL, cioè non coincide con il marcatore di fine lista, momento in cui si esce dal ciclo e la funzione visualizza_lista() termina il suo compito.

```
void visualizza lista(struct elemento *p)
{
   printf("\nlista ---> ");

   /* ciclo di scansione */
   while(p != NULL)
   {
      printf("%d", p->inf); // visualizza l'informazione
      printf(" ---> ");
      p = p->pun; // scorre di un elemento
   }

   printf("NULL\n\n");
}
```

35. Introduzione INPUT ed OUTPUT su File

Scrivere su file è forse la parte più importante di questo corso, perché si pongono le basi per salvare i propri lavori e quindi iniziare a strutturare programmi di una certa complessità. Abbiamo già accennato alle operazioni di input/output limitandoci a quelle che coinvolgevano la tastiera come dispositivo di input e lo schermo come dispositivo di output. Per poter operare correttamente è necessario includere l'header file **<stdio.h>** che contiene tutte funzioni per l'input/output, comprese quelle che operano sui file.

In C le operazioni di I/O vengono semplificate attraverso l'uso degli **stream**, altro non sono che delle astrazioni rappresentative di un file o di un dispositivo fisico, che vengono manipolate attraverso l'uso di puntatori. L'enorme vantaggio di usare gli stream è quello di potersi riferire ad un identificatore senza preoccuparsi di come questo venga implementato; generalmente le operazioni che si compiono su uno stream sono tre, lo si **apre**, vi si **accede** (nel senso di operazioni di lettura e scrittura) e lo si **chiude**. L'altra importante caratteristica è che lo stream è **bufferizzato**, ovvero viene riservato un buffer per evitare ritardi o interruzioni nella fase di lettura e scrittura, infatti il contenuto del buffer non viene mandato al dispositivo (che si tratti di un dispositivo fisico o di un file non ha importanza) fino a quando non viene svuotato o chiuso.

Gli **stream** predefiniti nel linguaggio C (che vengono messi a disposizione includendo la <stdio.h>) sono:

```
stdin, stdout, stderr
```

Questi stream utilizzano principalmente il testo come metodo di comunicazione I/O, anche se i primi due possono essere usati con i file, con i programmi, con la tastiera, con la consolle e lo schermo, lo **stderr** può scrivere soltanto su console o sullo schermo. La console è il dispositivo di default per **stdout** e **stderr**, mentre per **stdin** il dispositivo di default è la tastiera.

Linux e l'I/O

Linux (e Unix) permette una serie di operazioni che non fanno parte del linguaggio C, ma del sistema operativo stesso; queste operazioni permettono di gestire l'input o l'output di un programma e di redirigerlo dove meglio si crede.

- **>**
  redirige lo **stdout**, generalmente stampato a video, in un file; ammettiamo di avere un programma di nome "prog", per redirigere il risultato in un file basta fare, da console:

  ```
  # prog > risultato.txt
  ```

- **<**
  redirige il contenuto del file allo **stdin**; ammettiamo di avere un file di nome "input.txt" e di volerlo passare come input al programma "prog", basta digitare, da console:

  ```
  # prog < input.txt
  ```

- **|** (pipe)
  questo simbolo speciale permette di redirigere lo **stdout** di un programma allo **stdin** di

un altro programma; supponiamo di avere "prog1" che deve mandare il proprio output a "prog2", da console scriveremo:

```
# prog1 | prog2
```

due esempi pratici in linux sono:

```
# prog | lpr
// manda l'output di "prog" alla stampante

# ls -l | more
// visualizza le directory
// una schermata per volta
```

36. La funzione fopen

I file sono la parte più importante degli stream perché, come già detto, sono un elemento essenziale per permettere al programmatore di fare applicazioni interattive. Come menzionato prima, la prima cosa da fare è **aprire** un file; per fare ciò si usa la funzione **fopen**, strutturata nel seguente modo:

```
FILE *fopen(char *nome, char *modo);
```

che prende come parametri di input il nome del file al quale si intende accedere ed il modo in cui si vuole aprirlo, conforme al seguente schema:

"r" - lettura;
"w" - scrittura;
"a" - scrittura in fondo al file (append).

restituendo un puntatore all'oggetto **FILE** che servirà, dopo l'apertura, per poter accedere correttamente allo stream; se non si può accedere al file, viene restituito un puntatore a NULL. Qui di seguito proponiamo un semplice programma per poter leggere un file, ad esempio, di nome **miofile.txt**;

```
#include <stdio.h>

int main()
{

   /* dichiara lo stream e il prototipo della funzione fopen */
   FILE *stream, *fopen();

   /* apre lo stream del file */
   stream = fopen("miofile.txt", "r");

   /* controlla se il file viene aperto */
   if ((stream = fopen("miofile.txt", "r")) == NULL)
   {
      printf("Non posso aprire il file %s\n", "miofile.txt");
      exit(1);
   }

   [...]
```

```
   /* Codice che lavora sul file */
   [...]

}
```

37. Le funzioni fprintf e fscanf

Una volta che si è aperto un file con la funzione **fopen**, possiamo usare due funzioni per accedervi, queste sono la **fprintf** e la **fscanf** che, per quanto simili alla printf ed alla scanf, operano sullo stream del file aperto da fopen(); la forma con la quale si presentano le due funzioni è la seguente:

```
int fprintf(FILE *stream, char *formato, argomenti ...);
int fscanf(FILE *stream, char *formato, argomenti ...);
```

La **fprintf**, come si può intuire, scrive sullo stream, mentre la **fscanf** legge dallo stream; entrambe seguono per i parametri, tranne il primo, quello che è già stato detto nella lezione sette e che riportiamo sinteticamente qui sotto per comodità di consultazione:

formato e argomenti

La stringa **formato** ha due tipi di argomenti, i **caratteri ordinari** che vengono copiati nello stream di output, e le **specifiche di conversione**, contraddistinte dal simbolo percentuale (%) e da un carattere, che illustriamo di seguito e che specifica il formato con il quale stampare le **variabili** presenti nella lista di argomenti:

Stringa di controllo	Cosa viene stampato
%d, %i	Intero decimale
%f	Valore in virgola mobile
%c	Un carattere
%s	Una stringa di caratteri
%o	Numero ottale
%x, %X	Numero esadecimale
%u	Intero senza segno
%f	Numero reale (float o double)
%e, %E	Formato scientifico
%%	Stampa il carattere %

Naturalmente la **fprintf** e la **fscanf** possono scrivere negli stream predefiniti, stdout - stderr e stdin rispettivamente, come mostrato dall'esempio seguente:

```
/* stampa un messaggio di errore */
fprintf(stderr, "Impossibile continuare!\n");
```

```
/* scrive a video un messaggio */
fprintf(stdout, "Operazione completata!\n");

/* riceve da tastiera una stringa e la
* salva nella variabile "miastringa" */
fscanf(stdin, "%s", miastringa);
```

Esistono, inoltre altre quattro funzioni che operano su file scrivendo un carattere per volta, esattamente come fanno la **getchar** e la **putchar** per gli stream predefiniti; queste quattro funzioni sono:

```
int getc(FILE *stream);
int fgetc(FILE *stream);
int putc(char ch, FILE *stream);
int fputc(char ch, FILE *stream);
```

In realtà una curiosità risiede nel fatto che **getc** è una macro del preprocessore, mentre **fgetc** è una funzione di libreria, ma fanno esattamente la stessa cosa.

38. Le funzioni fflush e fclose

Infine gli stream, qualunque uso ne sia stato fatto, devono essere prima "puliti" e poi chiusi, questo si può fare comodamente con le funzioni **fflush** e **fclose**, formalizzate come segue:

```
fflush(FILE *stream);
fclose(FILE *stream);
```

Il seguente programma chiarirà meglio l'uso delle funzioni spiegate per operare su file; infatti simuleremo un input da tastiera che verrà scritto su un file insieme ad un testo predefinito e che verrà appeso al testo già presente nel file:

```
#include <stdio.h>

int main()
{
   char miastringa[40];
   FILE *stream = fopen("miofile.txt","a");
   printf("Inserisci meno di 40 caratteri -> ");
   fscanf(stdin, "%s", miastringa);
   fprintf(stream, "La mia stringa e' : %s\n", miastringa);
   fflush(stream);
   fclose(stream);
}
```

Le ultime funzioni che operano su file servono principalmente per eseguire operazioni di debug, le quali sarebbero difficili da implementare in maniera proprietaria:

```
int feof(FILE *stream);
int ferror(FILE *stream);
void clearerr(FILE *stream);
int fileno(FILE *stream);
```

il cui uso viene chiarito qui sotto:

- **feof()** - Ritorna un valore vero (true) se lo stream raggiunge la fine del file;
- **ferror()** - Riporta uno stato di errore dello stream e ritorna vero (true) se ne incontra uno;
- **clearerr()** - Cancella le indicazioni di errore per un dato stream;
- **fileno()** - Ritorna il descrittore di file intero associato con lo stream.

39. INPUT ed OUTPUT su stringhe

Esistono due funzioni molto simili alla fprintf ed alla fscanf, che però prendono come input una stringa e non uno stream; queste funzioni risultano molto utili perché in C una stringa è un array di caratteri, e potrebbe essere difficile gestirla con gli strumenti presentati fino ad ora. I due prototipi di funzione sono:

```
int sprintf(char *stringa, char *formato, argomenti ...);
int sscanf(char *stringa, char *formato, argomenti ...);
```

Le modalità di utilizzo sono analoghe a quelle spiegate in questa lezione (o nella lezione 7), ma proponiamo comunque un semplice pezzo di codice per far capire meglio come operano queste due funzioni:

```
#include <stdio.h>

int main()
{
   char miastringa[80];
   int distanza;
   int tempo;
   float velocita;
   printf("*** Calcolo della velocita' ***");
   printf("\nInserisci la distanza -> ");
   scanf("%d", &distanza);
   printf("Inserisci il tempo -> ");
   scanf("%d", &tempo);
   velocita = (float)distanza/(float)tempo;
   sprintf(miastringa,"Velocita' -> %2.3f\n", velocita);
   printf("%s", miastringa);
}
```

40. Messaggi di errore ed esempi pratici

Un'altra importante caratteristica del C, è che dispone di una serie di strumenti per riportare, al programmatore, gli errori commessi durante lo sviluppo. Quando prenderete dimestichezza con la programmazione capirete quanto sia importante avere degli strumenti per il debug, visto che buona parte dello sviluppo di un programma risiede proprio nella correzione degli errori; per poter utilizzare queste funzione è bene includere, oltre alla **<stdio.h>**, anche le librerie **<stdlib.h>** e **<errno.h>**. Una delle funzioni maggiormente utilizzate per questo scopo è la `perror()`, usata insieme alla `errno`; la struttura della funzione è la seguente:

```
void perror(const char *messaggio);
```

La **perror()** produce un messaggio a video descrivendo l'ultimo errore riscontrato e ritorna ad **errno** durante una chiamata ad una funzione di sistema o di libreria.

Errno è una variabile di sistema speciale che viene settata se una chiamata di sistema non esegue correttamente i propri compiti; per essere usata in un programma deve essere dichiarata come `extern int errno;`, a quel punto può essere manipolata attraverso gli strumenti messi a disposizione dal C, altrimenti ritorna l'ultimo valore ritornato da una funzione di sistema o di libreria. La funzione **exit()**, invece, si presenta con la seguente struttura:

```
void exit(int stato);
```

e serve essenzialmente per terminare l'esecuzione di un programma e ritornare il valore "stato" al sistema operativo, che può assumere essenzialmente questi due valori:

- EXIT_SUCCESS - nel caso in cui il programma termini senza problemi;
- EXIT_FAILURE - nel caso in cui il programma termini con errori.

41. Il Pre-processore C e le direttive di Inclusione

In questa lezione dobbiamo parlare di un elemento importante nella programmazione in C, il **pre-processore**. Questo strano elemento ha un ruolo apparentemente trasparente, ma allo stesso tempo importantissimo. Il pre-processore è incaricato, fondamentalmente, di trovare delle **direttive** all'interno di un file sorgente e di eseguirle; dove sta l'arcano? che il pre-processore opera su un file sorgente e "restituisce un file sorgente" (che poi, a questo punto, viene passato al compilatore), perché le direttive che lui interpreta sono principalmente di **inclusione**, di **definizione** e **condizionali**.

Una **direttiva** inizia sempre con il carattere cancelletto # ed occupa una sola riga, su tale riga possono essere messi dei commenti (// o /*) o il simbolo di continuazione alla riga successiva (\ ).

Il file sorgente contenente le direttive, viene tradotto dal pre-processore in quella che viene chiamata la **translation unit** (anch'essa formata solamente da codice sorgente), la quale viene poi **compilata** in codice binario, dando origine al corrispondente file oggetto; tutti i file oggetto, infine, vengono collegati dal **linker**, generando un unico programma eseguibile.

Direttive di Inclusione

Le direttive di inclusione sono quelle usate maggiormente, semplicemente perché vengono usate per inserire le librerie standard del linguaggio; la forma sintattica corretta della direttiva, che peraltro conosciamo benissimo, per includere i file è la seguente,

#include <file>

mettendo il nome del file da includere tra il simbolo di minore (<) e quello di maggiore (>); in questa forma il compilatore andrà a cercare i file presenti nelle directory di include prescelta; generalmente questa directory è predefinita, ed è quella che contiene i file delle librerie standard, come ad esempio stdio.h, stdlib.h, math.h, string.h, time.h e così via.

Ma se volessimo includere un file presente, generalmente, nella stessa cartella del nostro programma, dovremmo usare una forma alternativa, come segue,

#include "file"

È possibile nidificare la direttiva **#include**, infatti un file incluso può contenere a sua volta questa direttiva, ma poiché il livello di nidificazione dipende spesso dal sistema, è meglio cercare di non annidare l'include degli stessi file (e per questo proporremo una soluzione più avanti).

42. Il Pre-processore C e le direttive di Definizione

Le direttive di definizione sono sostanzialmente due, **#define** e **#undef**, che, come vedremo sotto, possono essere usate congiuntamente per definire o meno qualcosa in modo condizionale.

Ma passiamo a spiegare l'utilizzo di **#define**; questa direttiva, quando serve per definire una *costante* ha la seguente sintassi,

```
#define identificatore espressione
```

in cui sostanzialmente il pre-processore sostituisce tutte le occorrenze di "identificatore" con l'espressione corrispondente (che può contenere anche spazi o virgolette); generalmente il nome di una costante definita con tale direttiva, oltre ad essere tutto maiuscolo, viene preceduto dal carattere underscore (_). Quando la direttiva serve per definire una macro ha una sintassi leggermente diversa,

```
#define identificatore(argomenti) espressione
```

in questo caso, invece, si suppone che l'espressione contenga gli argomenti come variabili sulle quali operare, il pre-processore non fa altro che sostituire il codice di espressione, modificando gli argomenti definiti nella macro con quelli attuali, ad ogni occorrenza dell'identificatore.

Il vantaggio che si ha ad usare le direttive di definizione è quello di non appesantire il codice con chiamate o allocazioni di memoria, poiché si sostituisce il codice della macro con le

occorrenze dell'identificatore **nel codice sorgente** e solo dopo lo si passa la compilatore. Qualcuno potrebbe controbattere che al posto delle sopraccitate direttive si potrebbero usare le parole chiave del C, **const** per le costanti e **inline** per le macro (che poi sono delle piccole funzioni); è vero, si può fare, anzi è opportuno non eccedere troppo con l'uso delle direttive che, invece, devono essere usate in forma alquanto limitata e solamente in quei casi in cui la sostituzione apporti un beneficio reale al programma ed alla leggibilità del sorgente. Ovviamente **#undef** serve solo per annullare il compito di ciò che abbiamo definito con l'eventuale **#define**.

Per fare un esempio presentiamo un pezzo di codice che fa uso delle direttive di definizione:

```
#include <stdio.h>

#define NUMERO 5
#define QUADRATO(a) (a)*(a)

int main()
{
   int x;

   printf("Numero : %d \n", NUMERO);
   x = QUADRATO(NUMERO);
   printf("Quadrato: %d \n", x);

   #undef NUMERO
   #define NUMERO 7

   printf("Numero : %d \n", NUMERO);
   x = QUADRATO(NUMERO);
   printf("Quadrato: %d \n", x);

   return 0;
}
```

Questo semplice programma stamperà a video,

```
Numero : 5
Quadrato : 25
Numero : 7
Quadrato : 49
```

43. Le direttive condizionali

Le direttive condizionali permettono di definire una serie di istruzioni che verranno compilate in determinate condizioni, questo tipo di direttive viene usato spesso per compilare il medesimo sorgente su diversi sistemi operativi (cambiando dei parametri che in Linux funzionano ed in Windows no, e viceversa); le direttive condizionali sono sei, ma non sono difficili da ricordare:

- `#if` - Se il valore di ciò che sta a destra è zero (quindi FALSO) allora l'if non farà niente; se il valore è diverso da zero (TRUE), allora si eseguirà il codice sotto l'if fino a che non si incontra un elif, un else o un endif.
- `#ifdef` - Riferendosi all'identificatore passato come parametro, se è già stato definito (TRUE) si continua nell'esecuzione del codice sottostante, altrimenti (FALSE) si cerca un elif, un else o un endif.
- `#ifndef` - Complementare al precedente; infatti in questo caso si esegue l'if solo se l'identificatore passato non è stato definito.
- `#elif` - Corrisponde al costrutto del ELSE IF, e quindi verifica la propria condizione ed agisce di conseguenza; può essere usato per definire scelte multiple (tranne l'ultima).
- `#else` - Nell'ultima voce delle scelte condizionali bisogna mettere l'else che copre la casistica non ricoperta dall'if e dagli elif precedenti. Tutto il codice che sta dopo l'else verrà eseguito fino a quando non si incontra un endif.
- `#endif` - Chiude la direttiva #if e, corrisponde, nel linguaggio, alle famose parentesi graffe.

Di seguito facciamo un semplice esempio su come possiamo usare queste direttive per selezionare quale codice eseguire, cosa molto utile, lo ripetiamo, quando vogliamo compilare il medesimo programma su sistemi operativi diversi.

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

#ifndef SISTEMA
#define SISTEMA 1
#endif
// SISTEMA = 1 -> Sistema Linux;
// SISTEMA = 0 -> Sistema Windows;

main ()
{

   #if SISTEMA==1
   printf("Sistema operativo: Linux\n");
   #elif SISTEMA==0
   printf("Sistema operativo: Windows\n");
   #else
   printf("Sistema operativo: sconosciuto\n");
   #endif
```

```
return 0;
}
```

In questo semplice pezzo di codice abbiamo due scenari, o l'identificatore SISTEMA è già stato definito, ad esempio, in miofile.h, con un valore che può assumere 0 (Windows), 1 (Linux), 2 (sconosciuto), oppure viene definito all'interno del codice con valore di default pari ad 1. A questo punto gli strumenti fino ad ora spiegati servono per stampare uno dei tre testi a seconda del valore assunto dall'identificatore SISTEMA. Il codice è volutamente un po' "forzato" per mostrare tutte le dichiarazioni condizionali citate. Si lascia al lettore il compito di fare ulteriori test per scoprire la potenza di questo strumento.

44. Errori comuni e regole di stile in C

- **Assegnazione (=) al posto del confronto (==)** - Bisogna porre attenzione, quando, utilizzando una struttura condizionale come **IF-ELSE**, scriviamo l'operazione di confronto (==), poiché essa può, per un errore di battitura, diventare un assegnamento (=); se ciò dovesse accadere, ad esempio, cercando di confrontare se due numeri sono uguali, potremmo trovarci nella spiacevole situazione in cui, invece di controllare se **a == b**, che restituisce TRUE solamente quando le due variabili hanno lo stesso valore, poniamo **a = b**, che è TRUE praticamente sempre, il suo valore è FALSE solamente quando **b** vale 0.
- **Mancanza di ( ) per una funzione** - Il programmatore inesperto tende a credere che una funzione alla quale non si passano parametri non debba avere le parentesi tonde; ciò è errato, le parentesi tonde, anche senza parametri, devono essere sempre messe.
- **Indici di Array** - Quando si inizializzano o si usano gli array, bisogna porre attenzione agli indici utilizzati (e quindi alla quantità di elementi) poiché se si inizializza un array con N elementi, il suo indice deve avere un intervallo tra 0 (che è il primo elemento) ed N-1 (che è l'n-mo elemento).
- **Il C è Case Sensitive** - Il linguaggio C (come il C++ ed il Java) fa distinzione tra lettere maiuscole e minuscole, interpretandole come due caratteri diversi; bisogna quindi stare attenti, soprattutto quando si utilizzano le variabili.
- **Il ";" chiude ogni istruzione** - E' un errore tanto comune che non può non essere citato, ogni istruzione deve essere chiusa con un punto e virgola; questa facile dimenticanza (Nda: che colpisce anche i più esperti :)), segnalata dal compilatore, può far perdere del tempo prezioso ed appesantisce inutilmente il lavoro del programmatore.
- **I nomi delle variabili, strutture, costanti e funzioni devono essere significativi** - poiché le parole chiave del linguaggio sono in inglese, si consiglia di utilizzare nomi, per le variabili e le funzioni, in lingua italiana, in modo da poter capire, dal nome stesso, se quello che stiamo usando è effettivamente una parola chiave del linguaggio o un costrutto creato all'interno del nostro programma. Inoltre sarebbe buona norma, se il nome è composto da più parole, evidenziare con iniziali maiuscole le parole seguenti la prima; questa regola non vale, invece, per le costanti, che devono essere

scritte tutte in maiuscolo e separate, se formate da più di una parola, dal carattere underscore "_".

- **Uso, funzione e posizione dei commenti** - I commenti devono accompagnare quasi tutte le istruzioni, per spiegare il significato di ciò che stiamo facendo, inoltre devono essere sintetici e devono essere aggiornati appena modifichiamo un'istruzione. I commenti devono essere più estesi, invece, se devono spiegare un determinato algoritmo o se accompagnano una funzione.
- **Espressione ternaria** - L'espressione ternaria <cond> **?** <val> **:** <val> è, generalmente, sostitutiva di un costrutto **IF-ELSE** e restituisce il primo valore se l'espressione è vera (TRUE) o il secondo se, invece, è falsa (FALSE). L'espressione ternaria va messa, se possibile, sulla solita riga e, per evitare problemi, è consigliabile racchiudere tra parentesi tonde sia l'espressione sia i valori.
- Ogni **sorgente** dovrebbe contenere, nell'ordine:
    1. Commento iniziale (con nome del file, nome dell'autore, data, testo del problema ed eventuali algoritmi usati);
    2. Istruzioni **`#include`** per tutti i file da includere (bisogna includere solo i file necessari);
    3. Dichiarazioni di constanti, strutture e tipi enumerativi;
    4. Definizione di variabili;
    5. Prototipi delle funzioni;
    6. Definizione delle funzioni (con la funzione **main** messa come prima funzione).
- **Uso delle parentesi graffe { }** - Nelle dichiarazioni di tipi (STRUCT, UNION, ENUM) o nelle inizializzazioni di variabili all'interno delle definizioni la ***{*** che apre il blocco va posta di seguito al nome del tipo, vicino all'uguale dell'inizializzazione, sulla stessa riga; la graffa di chiusura andrà allineata con l'inizio della dichiarazione, su di una riga a sé stante.

  Nelle funzioni o nei costrutti IF/ELSE, FOR, WHILE, SWITCH la ***{*** di apertura del blocco va allineata sotto la prima lettera del costrutto, mentre nel costrutto DO/WHILE la ***{*** va sulla medesima riga del DO. La ***}*** di chiusura del blocco va allineata esattamente sotto la ***{*** di apertura, tranne nel DO dove la ***}*** di chiusura va allineata sotto la D di DO, seguita dal WHILE.

  Tutte le istruzioni contenute in un blocco vanno indentate rispetto all'allineamento del blocco stesso (che sia la ***{*** di apertura o il DO), tranne per le dichiarazione di variabili locali che vanno allineate sotto la ***{*** (o il DO). Per evitare di avere sorgenti che eccedano la larghezza dello schermo è preferibile indentare di un paio di spazi, invece che di un tab.
- **Uso e commento delle funzioni** - Una funzione deve eseguire un compito e non è semanticamente corretto scrivere funzioni che contengano il codice per fare molte cose assieme. E', piuttosto, conveniente scrivere le funzioni per i singoli compiti ed una

ulteriore funzione che svolga il compito complesso, richiamando le funzioni precedenti. Prima di ogni funzione, eccetto che per il main, è necessario fare un commento che contenga:

1. Scopo della funzione;
2. Significato e valori leciti per ognuno dei parametri;
3. Valori di ritorno nei vari casi;
4. Eventuali parametri passati per riferimento modificati all'interno della funzione;
5. Eventuali effetti collaterali sulle variabili globali.

45. Struttura di grossi programmi

Adesso abbiamo spiegato le basi della programmazione in C, ma sempre limitandoci a programmi di dimensioni medio/piccole; è utile, quindi, affrontare gli aspetti teorici e pratici dello sviluppo di programmi di una certa dimensione, questo perché un progetto con applicazioni reali usualmente raggiunge grandi dimensioni.

In questi casi è consigliabile dividere i programmi in moduli, che dovrebbero stare in file sorgenti separati (quindi ogni file conterrà uno o più funzioni), mentre l'istruzione main dovrà essere in un solo file (generalmente main.c). Alla fine questi moduli interagiranno tra loro semplicemente includendoli in fase di compilazione, con l'ovvio vantaggio anche di riusabilità per altri programmi.

Tutto ciò viene fatto, con gli strumenti adeguati, per ovvie ragioni:

- I moduli verranno divisi in gruppi di funzioni comuni
- È possibile compilare separatamente ogni modulo e poi linkarlo nei moduli compilati
- Usando le utility di Linux, come **make**, si possono mantenere abbastanza facilmente grossi sistemi

Utilizzo dei file header

Utilizzando un approccio modulare, dovremo includere in ogni modulo la definizione delle variabili, i prototipi di funzioni ed i comandi per il preprocessore C, ecc. Questo potrebbe causare problemi di mantenimento del software, così è consigliabile centralizzare queste definizioni all'interno di un file (o più di uno) e condividere, poi, le informazioni con gli altri file. Tali file vengono generalmente definiti "header file" (file di intestazione) e devono avere un suffiso ".h".
Se avete notato, generalmente, includiamo i file header delle librerie standard, come ad esempio **<stdio.h>** o **<stdlib.h>**, mettendoli, appunto tra il simbolo di minore (<) e di maggiore (>), questa è la convenzione usata per includere gli header di sistema (di cui il compilatore conosce il percorso completo);

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

per includere invece, all'interno di un file ".c", un file header che risiede nella medesima directory, dobbiamo usare le virgolette ("), quindi per includere, ad esempio, un file di nome **mia_intestazione.h**, basta scrivere, all'interno del file che vuole richiamarlo,

```
#include "mia_intestazione.h"
```

Per i programmi di moderate dimensioni è meglio avere due o più file header che condividono le definizioni di più di un modulo ciascuno. Il problema sorge quando abbiamo delle variabili globali in un modulo e vogliamo che vengano riconosciute anche negli altri moduli, generalmente, infatti, le variabili esterne e le funzioni devono avere visibilità globale. Quindi per le funzioni vengono usati i prototipi di funzioni, mentre possiamo usare il prefisso "extern" per le variabili globali che, in questo modo, vengono dichiarate ma non definite (cioè non viene allocata memoria per la variabile), saranno definite, poi, una ed una sola volta all'interno di un modulo che comporrà il programma; per questo possiamo avere tantissime dichiarazioni esterne, ma una sola definizione della variabile.

L'utility Make ed i makefile

L'utility Make è un programma che di per se non fa parte del famoso GCC, perché non è altro che un program manager molto usato, soprattutto in Ingegneria del Software, per gestire un gruppo di moduli di un programma, una raccolta di programmi o un sistema completo. Sviluppata originariamente per UNIX, questa utility per la programmazione, ha subito il porting su altre piattaforme, tra cui, appunto, Linux.

Se dovessimo mantenere molti file sorgenti, come ad esempio

```
main.c funzione1.c funzione2.c ... funzionen.c
```

potremmo compilare questi file utilizzando il gcc, ma rimane comunque un problema di fondo anche quando abbiamo già creato qualche file oggetto (.o) che vogliamo linkare durante la compilazione globale del programma:

- È tempo sprecato ricompilare un modulo per il quale abbiamo già il file oggetto, dovremmo compilare solamente quei file modificati dopo una determinata data, e facendo degli errori (cosa che aumenta con l'aumentare delle dimensioni del progetto) potremmo avere il nostro programma finale sostanzialmente errato o non funzionante.
- Per linkare un oggetto, spesso, è necessario scrivere grossi comandi da riga di comando. Scrivere tanti comandi lunghi e diversi per svariati file può indurre facilmente all'errore.

L'utility make viene in aiuto in questi scenari, fornendo automaticamente gli strumenti per fare questi controlli e quindi, garantisce una compilazione esente da errori e solo di quei moduli di cui non esiste già il file oggetto aggiornato.
L'utility make utilizza, a tal fine un semplice file di testo di nome "makefile", con all'interno **regole di dipendenza** e **regole di interpretazione**.

Una regola di dipendenza ha due parti, una sinistra ed una destra, separate dai due punti (:).

```
lato_sinistro : lato_destro
```

La parte sinistra contiene il nome di un target (nome del programma o del file di sistema) che deve essere creato, mentre quella destra definisce i nomi dei file da cui dipende il file target (file sorgenti, header o dati); se il file target (destinatario) risulta non aggiornato rispetto ai file che lo compongono, allora si devono seguire le regole di interpretazione che seguono quelle di dipendenza, infatti quando si esegue un makefile vengono seguite queste regole:

- Viene letto il makefile e si estrapola quali file hanno bisogno di essere solamente linkati e quali, invece, hanno bisogno di essere ricompilati
- Come detto prima, si controlla la data e l'ora di un file oggetto e se esso risulta "antecedente" rispetto ai file che lo compongono, si ricompila il file, altrimenti si linka solamente
- Nella fase finale si controlla data e ora di tutti i file oggetto e se anche uno solo risulta più recente del file eseguibile, allora si ricompilano tutti i file oggetto

Naturalmente possiamo estendere l'utility di make a qualsiasi ambito, perché possiamo usare qualsiasi comando dalla riga di comando, in questo modo risulterà intuitiva qualsiasi operazione di manutenzione, come fare il backup. Ma come si **crea un makefile**?
Ammettiamo di avere i file simili all'esempio precedente,

programma.c funzione1.c funzione2.c header.h

e di voler far si che ci siano delle regole che impongano di ricompilare se cambia qualche file, ecco come, ad esempio, potrebbe essere strutturato un makefile (editabile con un qualsiasi editor di testo):

```
programma : programma.o funzione1.o funzione2.o funzione3.o
  gcc programma.o funzione1.o funzione2.o funzione3.o

programma.o : header.h programma.c
  gcc -c programma.c

funzione1.o : funzione1.c
  gcc -c funzione1.c

funzione2.o : funzione2.c.
  gcc -c funzione2.c

funzione3.o : funzione3.c.
  gcc -c funzione3.c
```

Questo programma può essere interpretato in questo modo:

- "programma" dipende da tre file oggetto, "programma.o", "funzione1.o" e "funzione2.o"; se anche uno solo di questi tre file risulta cambiato, i file devono essere linkati nuovamente

- "programma.o" dipende da due file, "header.h" e "programma.c", se uno di questi due file è stato modificato, allora bisogna ricompilare il file oggeto. Questo vale anche per ultimi due comandi

Gli ultimi tre comandi vengono chiamati **regole esplicite** perché si usano i nomi dei file in modo esplicito, mentre volendo si possono usare regole implicite come la seguente:

```
.c.o : gcc -c $<
```

che può sembrare apparentemente incomprensibile, ma che si traduce facilmente in "prendi tutti i file .c e trasformali in file con estensione .o eseguendo il comando gcc su tutti i file .c (espresso con il simbolo $<).

Per quanto riguarda i commenti, si usa il carattere cancelletto (#), così tutto ciò che è sulla medesima riga viene ignorato; questo tipo di commento è lo stesso disponibile in python ed in perl.

Per eseguire un makefile è sufficiente digitare dalla linea di comando, il comando "make", il quale andrà automaticamente a cercare un file di nome makefile da eseguire. Se però abbiamo utilizzato un nome diverso (ad esempio abbiamo fatto un makefile per fare il backup, di nome backup_all), possiamo dire a make di interpretare tale file come il makefile corrente, digitando,

```
# make -f backup_all
```

Ovviamente esistono altre opzioni per il comando make, che possono essere scoperte digitando da console, il comando,

```
# make --help
# man make
```

Abbiamo cercato di spiegare in modo semplicistico l'uso del Make e dei makefile, questo per farne comprendere le possibilità, senza addentrarci troppo nella programmazione di un makefile complesso, comprensivo di comandi più avanzati o dell'uso delle macro. Tutto ciò spero possa servire come punto di partenza per uno studio più approfondito di tale strumento che, comunque, viene utilizzato quando si inizia a lavorare su progetti di una certa dimensione e che, quindi, non è utile trattare ulteriormente in questo ambito.