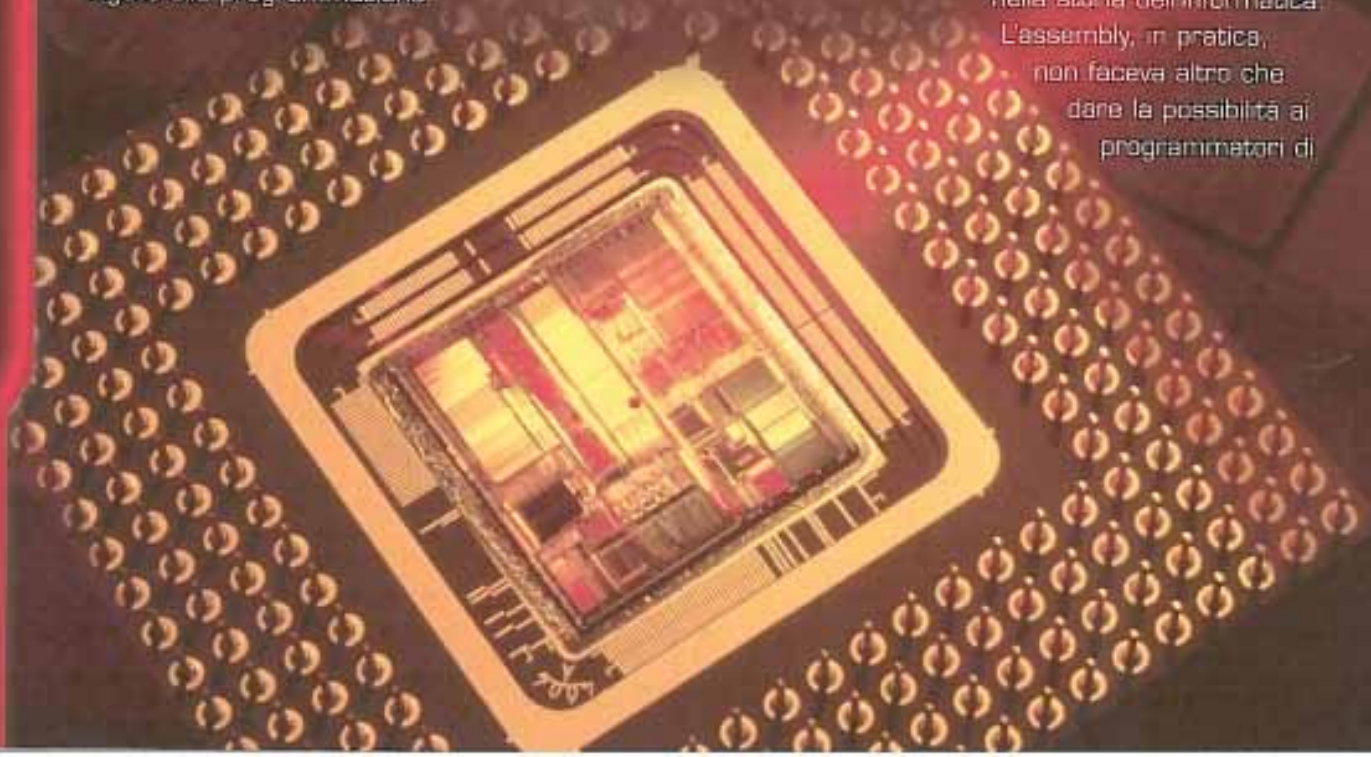
INIZIAMO A PROGRAMMARE

In questo Workshop iniziamo a conoscere il mondo dei linguaggi di programmazione e introduciamo, con un esempio pratico, un linguaggio che si rivelerà molto utile in futuro, quando utilizzeremo i microcontrollori.

Come abbiamo più volte ricordato, la maggior parte dei robot è composta da una grande quantità di sottosistemi meccanici ed elettronici. Specialmente nel campo della ricerca, però, i progettisti preferiscono non ricorrere a elettroniche e hardware progettati ad hoc, ma piuttosto a **dispositivi programmabili** che possono essere sfruttati e personalizzati nei più svariati modi. Ecco perché anche noi, in questo Workshop, iniziamo ad affrontare le problematiche legate alla programmazione.

PERCHÉ UN LINGUAGGIO DI PROGRAMMAZIONE? ›››

I computer e i dispositivi elettronici analoghi rispondono all'esigenza di avere a disposizione macchine che possano essere adattate per risolvere una quantità quasi infinita di problemi. Programmare significa questo: **produrre una sequenza di istruzioni che permettano ai calcolatori di soddisfare le nostre necessità operative**, ma proprio questo è lo scoglio maggiore da superare. I computer, come è noto, sono realizzati su un'architettura di tipo binario di difficile manipolazione per l'uomo. Inoltre, ogni sistema programmabile è in grado di svolgere solo un numero molto limitato di istruzioni che cambiano da hardware ad hardware: questo, a causa delle differenti architetture fisiche e delle diverse 'filosofie' di progettazione dei vari dispositivi. Le sequenze binarie ed esadecimali hanno, così, molto rapidamente lasciato spazio all'**assembly**, uno dei primi linguaggi simbolici adottati nella storia dell'informatica. L'assembly, in pratica, non faceva altro che dare la possibilità ai programmatori di



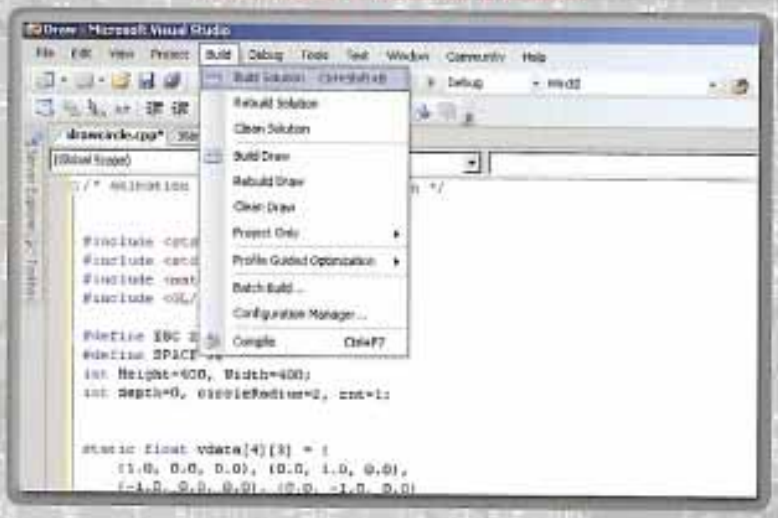
sostituire i poco pratici codici numerici delle istruzioni con abbreviazioni letterali più facilmente comprensibili. Per fare un esempio pratico, l'assembly delle attuali architetture **x86** (lo standard hardware derivato dai chip Intel usati nei personal computer) include alcune funzioni matematiche che consentono ai microprocessori (o CPU) di eseguire le operazioni matematiche di base. Esiste, ad esempio, l'istruzione '**ADD AX**' che permette di eseguire la somma binaria tra un numero utilizzato come parametro e il contenuto del registro AX (una piccola cella di memoria interna alla CPU). Chi ha un minimo di conoscenza dell'inglese, alla lettura di questo comando avrà subito associato a esso il concetto di 'somma'. Per un computer, tuttavia, l'istruzione appena incontrata è incomprensibile così come la abbiamo presentata, in quanto ogni comando è immagazzinato ed elaborato sotto forma numerica. Se avessimo voluto esprimere la somma in **linguaggio macchina** avremmo dovuto utilizzare il codice numerico (detto ***opcode***) '**05**'. Il compito di 'convertire' le istruzioni simboliche in opcode non spetta, comunque, al programmatore, ma a particolari strumenti software chiamati **compilatori**, che si occupano di 'tradurre' il **codice sorgente** (scritto dall'uomo) in sequenze di istruzioni e dati comprensibili agli elaboratori (**codice macchina binario**). L'assembly facilitava notevolmente il lavoro degli sviluppatori, ma rimaneva caratterizzato da una serie di difetti non trascurabili, legati alla sua natura molto prossima al funzionamento fisico degli elaboratori. Infatti, poiché l'assembly è un linguaggio dipendente dal set di istruzioni delle CPU, la sostituzione di un computer con un altro avente un differente set di istruzioni rendeva necessario apprendere un assembler differente e, quindi, un nuovo linguaggio di programmazione. Un secondo difetto dell'assembly è legato alla sua 'difficoltà' di comprensione sotto il profilo 'algoritmico'. Immaginiamo di voler implementare un programma che risolva un semplice problema: 'calcolare la somma dei primi dieci numeri interi' (da 0 a 9). Anche chi non ha basi di programmazione risolverà questo problema in un modo molto elementare, sommerà ripetutamente i numeri da 0 a 9, tenendo a mente le somme parziali in quello che possiamo definire un '**accumulatore**'. Inconsciamente, inoltre, dopo ogni somma (o prima) si dovrà svolgere un **confronto** tra il numero che stiamo attualmente sommando e l'ultimo che vogliamo sommare (ad esempio 9 se la somma è iniziata da 0). Abbiamo, in sostanza, fatto ricorso a un '**ciclo**', a una '**struttura condizionale**' e a una '**variabile di accumulazione**'. Se è vero che risolvere questo problema è stato molto semplice, la cosa inizierebbe a complicarsi se volessimo implementare tale algoritmo con un programma assembly, per il quale dobbiamo ricorrere a operazioni di **salto condizionato**, ossia particolari comandi che impongono all'elaboratore di muoversi da un punto all'altro della sequenza di istruzioni del programma in esecuzione. Per un algoritmo molto semplice, come questo, basterebbero poche righe di codice e un solo salto, ma immaginiamo di dover 'comprendere' un programma, anche scritto in

Sullo sfondo, un sistema elettronico dei nostri giorni. Nei moderni circuiti è sempre più frequente la presenza di microprocessori programmabili.

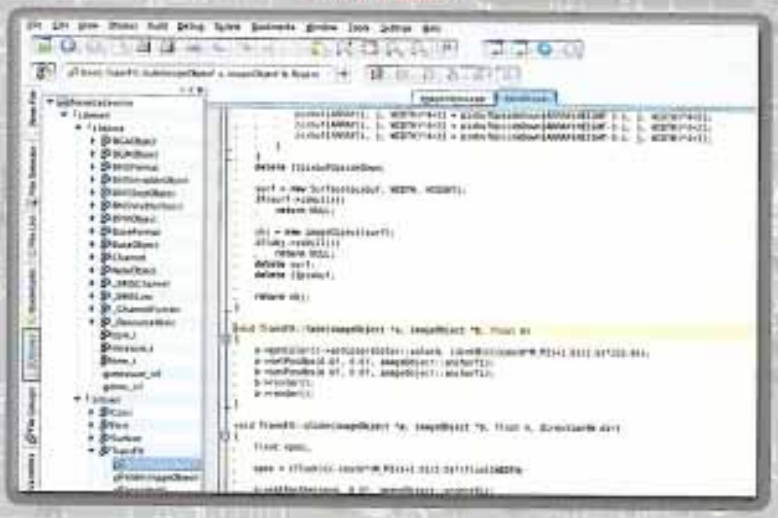
GLI AMBIENTI DI SVILUPPO

Con l'aumentare della complessità dei linguaggi di programmazione, anche gli strumenti di sviluppo si sono evoluti offrendo sempre più funzioni. Sono nati, così, gli **ambienti di sviluppo integrato** (brevemente detti **IDE**, tre esempi dei quali sono presentati nelle immagini a lato), speciali editor che offrono, oltre alla possibilità di scrivere fisicamente il codice sorgente, tutta una serie di ausili per la programmazione. Le differenze tra un IDE e un editor di testo generico sono numerose: si va dalla semplice **enfatizzazione cromatica** delle parole chiave del linguaggio di programmazione (funzionalità presente anche in molti editor evoluti come 'VI') alla **scrittura assistita**, con funzioni di **autocompletamento** e **verifica in real-time della correttezza sintattica** del codice. Inoltre, questi complessi software sono in grado di automatizzare molte delle procedure di generazione del codice eseguibile (compilazione e linking) e di fornire un importantissimo contributo alle operazioni di **identificazione e correzione degli errori** (*debug*). Con essi, infatti, è possibile monitorare l'esecuzione dei programmi, visualizzando lo stato delle variabili e potendo seguire l'avanzamento del programma 'passo passo'. Tra gli IDE commerciali più diffusi spicca **Visual Studio** di Microsoft, particolarmente usato per lo sviluppo di software per Windows basati su interfacce grafiche. In ambito 'freeware', invece, troviamo una gran varietà di applicativi, come **KDevelop** (creato per i sistemi Linux KDE) o **Eclipse**, un famoso ambiente disponibile per i principali sistemi operativi. Anche se non appartiene alla categoria degli IDE, non possiamo non citare uno degli editor storicamente più amati dai 'puristi' della programmazione: **EMACS**, diventato oramai parte di tutte le distribuzioni Linux più importanti.

‹MICROSOFT VISUAL STUDIO›



‹KDEVELOP›



‹EMACS›

Alcuni dei linguaggi di programmazione più diffusi. Java è un caso particolare, in quanto la compilazione genera un codice intermedio (bytecode) che viene eseguito da una sorta di 'interprete' (JVM).

linguaggio simbolico, formato da milioni di istruzioni e migliaia di salti: praticamente impossibile.

LINGUAGGI DI ALTO LIVELLO

La difficoltà di programmazione e la ridotta portabilità (ossia nutilizzabilità su sistemi diversi da quelli per cui sono stati scritti) dei codici sorgente assembly hanno portato alla nascita di linguaggi con un livello sintattico più vicino a quello dell'uomo. Una rivoluzione importantissima, che ha modificato sia il modo di 'pensare' il software, sia le tecnologie di supporto alla programmazione. La storia dell'informatica ha, così, visto affiancarsi una moltitudine di linguaggi, molti dei quali con caratteristiche comuni e molti altri con peculiarità uniche. Praticamente tutti i linguaggi moderni, però, sono basati su alcuni principi elementari come le **istruzioni**, le **variabili** (come quella usata in precedenza per memorizzare le somme dei numeri da 0 a 9), le **strutture di controllo** e i **sottoprogrammi**.

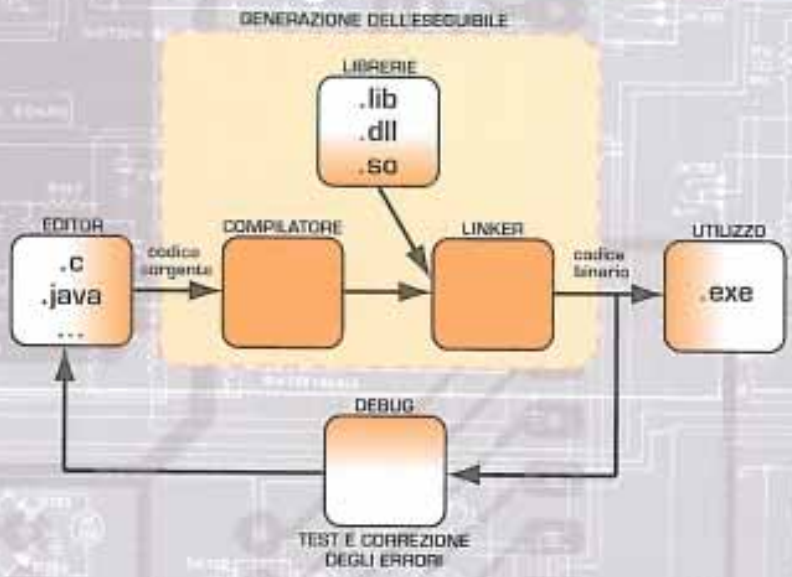
UNA MAGGIORE ASTRAZIONE

Un primo forte cambiamento tra l'assembly e i linguaggi di alto livello è legato alle **istruzioni**: su un piano 'concettuale' il loro significato è lo stesso; tuttavia le istruzioni di alto livello permettono di 'svincolarsi' maggiormente dall'hardware offrendo un livello di astrazione maggiore. Tali istruzioni, infatti, non sono più, in genere, legate in modo diretto al set di istruzioni del computer, ma sono associate a una ben precisa semantica che può essere equivalente a sequenze di decine, centinaia o anche migliaia di righe di codice macchina. Le **strutture di controllo**, invece, offrono al programmatore la possibilità di controllare 'i flussi' del proprio programma, stabilendo **casistiche di esecuzione** basate su test o creando **sequenze di istruzioni ripetute in maniera ciclica**. Infatti, a differenza dei salti offerti dall'assembly, i cicli strutturati dei linguaggi di alto livello facilitano enormemente la lettura poiché mantengono il codice più 'compatto'.
Il concetto di **sottoprogramma**, invece, costituisce la possibilità di creare 'blocchi di codice' richiamabili a piacimento (come se definissimo nuove istruzioni).

LINGUAGGI COMPILATI

Abbiamo precedentemente presentato il compilatore come lo strumento software in grado di 'tradurre' il codice simbolico assembly in codice macchina. Abbiamo anche detto, però, che una delle caratteristiche peculiari dei linguaggi di alto livello è la presenza di istruzioni 'complesse', corrispondenti a centinaia o migliaia di linee di codice assembly. In che modo, allora, avviene l'associazione tra un'istruzione complessa e la sua implementazione 'in linguaggio macchina'? La procedura di compilazione, in pratica, traduce il codice sorgente in linguaggio macchina che viene, poi, completato aggiungendo le 'implementazioni' delle istruzioni usate attraverso il collegamento a **librerie** (*linking*). Le librerie non sono altro che moduli software contenenti raccolte di istruzioni e funzioni più o meno complesse sviluppate in modo da essere compatibili con una ben precise architettura informatica ed essere integrate nei propri programmi. Quindi, in presenza di due sistemi hardware con assembly diversi, ma che supportano lo stesso linguaggio di alto livello e sono dotati di librerie equivalenti,

IL CICLO DI VITA DEI SOFTWARE



Lo sviluppo del software è un processo molto complesso che può essere diviso in 'stadi' sequenziali. Se tralasciamo quelli 'iniziali' (ossia l'analisi del problema e lo studio dei requisiti), la realizzazione di un programma parte, ovviamente, dalla scrittura del codice (svolta tramite un editor o un IDE) seguita, successivamente, dalla procedura di generazione dell'eseguibile (compilazione e linking sono presenti nel caso dei linguaggi compilati). A questo punto ha inizio una fase critica, ossia quella della ricerca e della correzione degli errori, operazione nota con il nome *debug*. In questa fase, il software sviluppato viene testato in condizioni di utilizzo critiche (ad esempio con grandi volumi di dati o eseguendo volutamente operazioni non consentite ecc.) per verificare che non vi siano anomalie nelle risposte e nei comportamenti che, se riscontrate, devono essere corrette intervenendo sul codice sorgente.

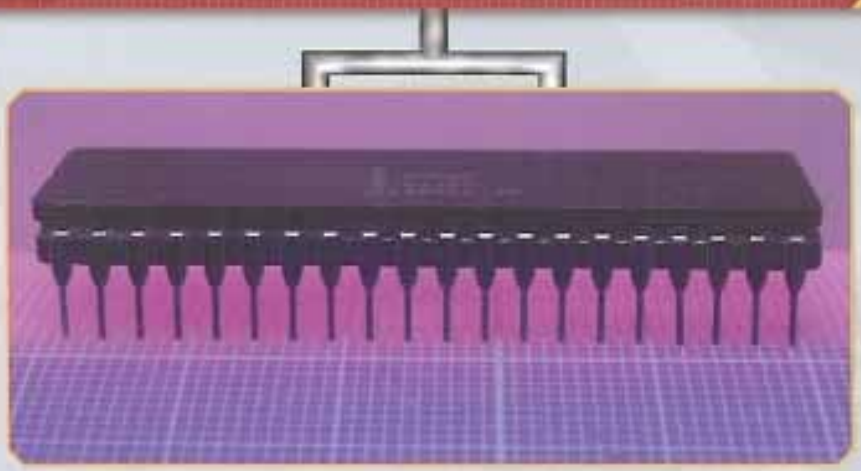
possiamo compilare lo stesso codice sorgente, senza ricorrere a modifiche. Le librerie vengono classificate in **statiche** e **dinamiche**. La differenza risiede nel modo in cui vengono gestite in fase di linking: nel caso di **moduli software statici**, il *linker* (ossia il software che effettua il linking) **ingloba all'interno del file eseguibile** (ossia del programma tradotto in linguaggio macchina) **il codice presente in essi**. Ciò significa che se compiliamo due programmi differenti che usano le stesse funzioni di libreria e li colleghiamo in modo statico, otteniamo file binari contenenti parti di codice in comune (l'implementazione delle funzioni delle librerie incluse). **Le librerie dinamiche** (file .**DLL** nei sistemi Windows, .**SO** in quelli Linux), al contrario, **vengono caricate dal sistema operativo nel momento in cui i programmi hanno necessità di utilizzarle**. Restano, quindi, fisicamente 'scorporate' dagli eseguibili prodotti. Questa tecnica ha il **vantaggio** di poter produrre **file eseguibili di piccole dimensioni** e di **ridurre il codice 'duplicato'**; tuttavia presenta lo **svantaggio** di dover **distribuire, oltre al programma realizzato, anche tutto il pacchetto di librerie dinamiche necessarie**. Un altro vantaggio è la più **efficiente gestione degli errori**: in caso di ***bug*** (ossia di errore) di un singolo modulo, è sufficiente **sostituire il file di libreria** contenente la 'falla' per correggere il problema su tutti i programmi che utilizzano la funzione anomala. Nel caso di programmi con link statici, invece, sarebbe necessario reinglobare le librerie in ogni singola applicazione anomala.

LINGUAGGI INTERPRETATI

L'alternativa alla compilazione è l'**interpretazione del codice**. Questo tipo di tecnica differisce da quanto abbiamo appena visto poiché il codice sorgente non viene 'tradotto' in linguaggio macchina ma, al contrario, viene analizzato ed elaborato a ogni esecuzione da un software intermedio chiamato '**interprete**'. Pur essendo una tecnica di programmazione poco efficiente (in quanto **consuma molte risorse**), l'uso dei linguaggi interpretati è sempre più di moda in tutte quelle applicazioni dove ci si vuole 'svincolare' da una precisa piattaforma hardware e/o software e si vuole **garantire l'eseguibilità del proprio codice su sistemi di natura eterogenea** (come nel caso delle applicazioni legate al web, ai linguaggi di scripting per Internet ecc.).

A, B... C!

Ora che abbiamo una base, seppur molto elementare, di alcuni dei concetti legati ai linguaggi di programmazione, iniziamo a immergerci nel mondo del software partendo proprio da uno dei linguaggi storici tuttora usato: il **C**. Per il momento inizieremo a studiarne la sintassi e le principali proprietà e strutture di controllo, realizzando semplici programmi per PC, ma quando sarà il momento lo riutilizzeremo per programmare i microcontrollori che potranno, così, essere utilizzati per realizzare i 'cervelli' dei nostri robot. Cambierà la piattaforma hardware (da un PC a un microprocessore programmabile) e l'ambiente di sviluppo, ma, come vedremo, il linguaggio rimarrà lo stesso.



Il linguaggio C può essere impiegato sia nello sviluppo di software per PC, sia per il software dei microcontrollori.

LA STRUTTURA DI UN PROGRAMMA C

Iniziamo a conoscere il C in modo pratico attraverso il più classico degli esempi: un semplicissimo programma '**Ciao RoboZak**'. Come vedremo non sarà nulla di eclatante in termini di 'funzioni', ma ci permetterà di familiarizzare con la sintassi del linguaggio C. Analizziamo le righe di codice presenti nel box a fondo pagina. Quella **colorata in grigio** e contenuta tra i delimitatori `/*` e `*/` è un semplice **commento al codice**. L'unica sua funzione è quella di permettere al programmatore di aggiungere note, appunti e, ovviamente, commenti al proprio lavoro. **Un codice ben commentato permette, infatti, una migliore gestione dell'intero sorgente**, specialmente se di grandi dimensioni, in quanto ne facilita la leggibilità. I commenti non vengono elaborati dal compilatore e quindi possono contenere qualsiasi tipo di testo ed essere posizionati accanto a ogni istruzione desiderata. La seconda riga, invece, è una **inclusione**. Il suo scopo è quello di permettere l'**utilizzo di librerie** di funzioni e istruzioni ausiliarie implementate in moduli esterni al proprio codice. In questo caso stiamo 'avvisando' il compilatore che siamo intenzionati a usare la libreria '**stdio**', che mette a disposizione numerose funzioni di I/O, ossia di input/output. L'**estensione '.h'** (file *header*), invece, identifica i file che contengono i '**prototipi**' delle funzioni messe a disposizione dalle librerie (un **prototipo non è altro che la 'definizione' di una funzione, priva della sua implementazione**). Ogni volta che vorremo usare particolari librerie, quindi, dovremo

Ciao_Robozak.c

```
/* programma 'CIAO ROBOZAK!' */
#include <stdio.h>

int main()
{
   printf("\nCIAO ROBOZAK!\n");
   return 1;
}
```

iniziare il nostro codice sorgente dichiarandole con la direttiva **#include**. Successivamente incontriamo la riga **int main()**. Essa rappresenta la **definizione della funzione main**, una speciale procedura che deve essere presente in tutti i nostri sorgenti C. Essa, infatti, **contiene il codice che viene eseguito all'avvio dell'eseguibile**. Notiamo la presenza delle due **parentesi tonde** e della dicitura **'int'**: le **parentesi tonde**, in particolare, contengono i **parametri** delle funzioni, mentre il termine **int** identifica il **tipo di dato** restituito (questi due concetti verranno affrontati in futuro). Nell'attuale esempio la funzione main non richiede parametri (tra le parentesi non è indicato nulla), mentre la presenza di 'int' indica che 'main' restituirà un dato numerico di tipo 'intero'. Osserviamo, poi, che compaiono **due parentesi graffe**: esse **delimitano un blocco di codice** e dovremo utilizzarle ogni volta che vorremo definire una funzione o associare una serie di istruzioni a una struttura di controllo. In ultimo arriviamo all'istruzione **printf**, una particolare funzione offerta dalla libreria 'stdio' che consente di scrivere a video una **stringa**, ossia una sequenza di caratteri, nel nostro caso rappresentati da **"\nCIAO ROBOZAK!\n"**. Una nota particolare va fatta a riguardo dei simboli '\n'. In C, il **simbolo '\' seguito da un simbolo** è utilizzato per rappresentare caratteri 'non visualizzabili graficamente'. Ne sono un esempio la **tabulazione (\t)**, il *carriage return* (\r) e il carattere *new line* (\n), che ci permette di mandare a capo il testo. Nel nostro caso, quindi, l'effetto della funzione sarà quello di saltare una riga del video, scrivere 'CIAO ROBOZAK' e andare nuovamente 'a capo'. L'istruzione **return** seguita da '1', infine, segnala al compilatore che la funzione main è terminata e impone la 'restituzione' del numero intero 1. Ultimo elemento fondamentale è la presenza del carattere ';', che dovremo usare per indicare al compilatore la fine di ogni istruzione. Nel prossimo StepbyStep installeremo un ambiente di sviluppo e testeremo l'esempio appena presentato.

QUICK REFERENCE

Nei Workshop che trattano di programmazione, utilizzeremo i box 'Quick Reference' per riassumere quanto visto nella sezione appena terminata. Ecco cosa abbiamo presentato in questo numero:

/* e */	**Delimitano un commento. Il testo incluso tra questi due simboli viene considerato un commento e, quindi, viene trascurato dal compilatore.**
#include <>	**Permette di comunicare al compilatore che si vuole utilizzare una particolare libreria software.**
main ()	**La funzione main rappresenta la funzione principale del programma, ossia quella che viene richiamata con l'avvio dell'eseguibile. Dovrà essere presente all'interno di ogni programma.**
printf ()	**La funzione printf può essere usata per 'stampare a video' una stringa passata per parametro. Viene messa a disposizione dalla libreria stdio.h. Nei prossimi fascicoli vedremo il suo utilizzo per la stampa di stringhe formattate, contenenti numeri ecc.**
{ e }	**Le parentesi graffe delimitano rispettivamente l'inizio e la fine di un blocco di codice.**
****	**Il carattere 'backslash' è utilizzato per indicare caratteri speciali, come la tabulazione (\t), il carriage return (\r) e il new line (\n).**
return	**La parola chiave return è utilizzata per terminare le funzioni 'restituendo' un risultato.**
;	**Il punto e virgola serve per indicare al compilatore la fine di un'istruzione.**