1 Software

Il software è l'insieme di tutti i programmi di tutti i computer.
Può essere usato per far funzionare un computer, per risolvere problemi particolari ed anche per produrre altro software.
Nella prima parte di questo capitolo faremo una classificazione del software secondo l'uso che ne viene fatto. Daremo anche dettagli riguardo ai vari tipi di software definiti. Nell'ultima parte del capitolo opereremo un'altra classificazione del software, secondo il suo modo di distribuzione.

1.1 Algoritmi

I programmi per computer eseguono sequenze di istruzioni di macchina che realizzano **algoritmi**.
Si possono trovare molte definizioni di algoritmo, quella più completa, a parere dell'Autore, è attribuita a Richard Dorf:

"Un algoritmo è un procedimento completo e non ambiguo per risolvere uno specifico problema in un numero finito di passi".

La definizione evidenzia il fatto che un algoritmo serve per risolvere un problema, che la soluzione deve essere sempre raggiunta, seguendo una serie non infinita di piccoli passi, che non si può omettere nulla dalla descrizione del procedimento (completa), che quella descrizione deve essere precisa (non ambigua).
Nel caso della realizzazione di un algoritmo con un computer i passi elementari di cui si parla sono, in ultima analisi, le istruzioni di macchina.

1.2 Software di basso o alto livello

Il software si può anche classificare come di alto o basso "livello". La definizione di software di basso o alto livello non ha nessuna connotazione "morale" o di valore.
Per software di basso livello non si intende un programma "cattivo", ma un software che utilizza un linguaggio molto vicino a quello comprensibile direttamente dall'hardware.
Dato che un'istruzione di macchina è il comando "più piccolo" che si può affidare ad un computer ed è comprensibile senza intermediari da parte della sua CPU, un programma scritto in linguaggio macchina è al più basso livello possibile. Peraltro esso è impossibile da decifrare per gli umani, dato che è costituito da una sequenza del tutto astrusa di numeri. Ciò fa capire che il "livello" di cui si parla in questo contesto è il cosiddetto "livello di astrazione" del software.
I programmi di maggiore **livello d'astrazione** hanno comandi sintetici e potenti, che esprimono con pochi simboli istruzioni complesse. I programmi a basso livello sono invece espressi con comandi che richiedono molte istruzioni per realizzare funzioni complesse.
Il "livello" del software ha quindi a che fare con il grado di astrazione dei comandi che si possono impartire. Un alto livello di astrazione può essere ottenuto sia direttamente con l'uso di un linguaggio di programmazione di alto livello sia con una sapiente strutturazione dei programmi scritti con linguaggi di livello più basso.

abstraction 1. Generalisation; ignoring or hiding details to capture some kind of commonality between different instances. (..)

Da "The Free On-line Dictionary of Computing", http://foldoc.doc.ic.ac.uk/, Editor Denis Howe

Fra i linguaggi di alto livello si possono elencare C++, BASIC, Pascal, Java; Prolog, LISP.
I linguaggi di basso livello tipici sono i linguaggi macchina e gli Assembly, almeno uno per ogni famiglia di CPU. Alcuni classificano anche il C fra i linguaggi di basso livello. Esso è senz'altro un po' "al confine" fra le due categorie, dato che con esso si può ottenere un grande controllo del computer, sia pur con istruzioni che non dipendono dalla CPU usata. Alcuni parlano del C come linguaggio di "medio livello" o come "Assembly strutturato" o "portabile". Si può senz'altro dire che il livello di astrazione di un programma scritto in C dipende più dal programma che dal linguaggio!

Vediamo ora, come esempio, due brani di programma, scritti in C++ e in Assembly, che risolvono lo stesso problema:

Ecco la versione in C++:

```
for (i=0; i<10; i++)
    cout << i;
```

E la versione in Assembly:

```
        mov     word ptr DGROUP:_i,0
        jmp     short @1@114
@1@58:
        mov     ax, word ptr DGROUP:_i
        mov     word ptr [bp-2], ax
        mov     ax,word ptr [bp-2]
        cwd
        push    dx
        push    ax
        mov     ax, offset DGROUP:_cout
        push    ax
        call    near ptr @ostream@$blsh$ql
        add     sp,6
        inc     word ptr DGROUP:_i
@1@114:
        cmp     word ptr DGROUP:_i,10
        jl      short @1@58
```

Figura 1: un programma in C++ e in Assembly 8086

I due brani di programma visualizzano i numeri da uno a dieci. Quello in Assembly è la "traduzione" di quello in C++, effettuata da un programma traduttore (vedi oltre in "Traduttori").
Come si può vedere la versione in Assembly è un po' più prolissa :-(.
Sicuramente la si potrebbe migliorare, dato che è il prodotto di un programma e non dell'intelligenza di una persona, ma il codice in Assembly resterebbe sempre più ingombrante e meno comprensibile, "meno astratto", in sintesi più "a basso livello".

Portabilità

In gergo informatico si parla di "**portabilità**" intendendo la capacità dei programmi di essere convertiti facilmente per lavorare su piattaforme diverse o con diversi compilatori dello stesso linguaggio.
Se il linguaggio di programmazione è disponibile su molte piattaforme è possibile realizzare con esso applicazioni con maggiore portabilità.
Esempio: esistono compilatori per il linguaggio C praticamente su tutti i microprocessori ed i microcontrollori esistenti, C++ o Java esistono per moltissime piattaforme, invece Visual BASIC funziona solo con CPU X86, su computer di architettura PC e con Sistema Operativo Windows.
L'operazione di trasformare un programma, generandone una nuova versione che funzioni in un'altra piattaforma è detto "**port**" o "porting". In genere il porting è facilitato dalla scrittura dei programmi in linguaggi di alto livello.

1.3 Classificazione del software secondo l'uso

Focalizzando l'attenzione sull'uso del software possiamo classificarlo in tre ampi gruppi: software **di base** (system software), che serve per "completare" ogni computer e ne rende possibile il funzionamento, software per lo **sviluppo** (development software), che serve a realizzare nuovi programmi, e software **applicativo** (application software) o "**applicazione**", che serve a risolvere problemi specifici non di tipo informatico, come scrivere un libro, disegnare una barca o suonare musica.
Nella trattazione successiva fra gli strumenti di sviluppo non includeremo solo software; essi verranno trattati un po' più in dettaglio, dato che fra gli obbiettivi di questo libro c'è quello di insegnare a programmare. Di software applicativo non tratteremo.

1.3.1 Software di base

Senza il software di base ogni computer sarebbe solo un mucchietto inanimato di ferro e silicio. Il software di base è la dotazione minima di programmi con cui il computer viene venduto e che lo trasforma in uno strumento utile e funzionante. E' per la presenza del software di base che un computer può eseguire i programmi applicativi.

Firmware
Il software che risiede per sempre in un computer, e che quindi non è rimovibile viene detto "firmware".

Almeno parte del software di base deve essere sempre presente in memoria di lavoro, su un supporto non volatile (ROM). Infatti, come abbiamo già detto, la prima istruzione che una CPU esegue dopo il "reset" deve essere presa da una locazione di memoria di lavoro.

Ogni computer deve perciò avere almeno una piccola parte del suo software di base in ROM, all'indirizzo fisso al quale la CPU fa il suo primo fetch dopo il reset. Questo software fa le prime verifiche di funzionamento all'accensione e carica dall'hard disk le parti più importanti del Sistema Operativo, che devono stare in memoria tutto il tempo.

Alcuni computer hanno solo firmware. Si pensi ad esempio ad un computer che controlla una lavatrice; non è il caso di equipaggiarlo con un hard disc per la memorizzazione dei programmi, inoltre i programmi non devono mai cambiare nella vita dell'elettrodomestico, per cui possono rimanere per sempre in memoria di lavoro, su un supporto di tipo ROM.

Per completezza si accenna ad un significato inconsueto che alcuni danno alla parola "firmware", intendendo con essa i microprogrammi che determinano le sequenze della Control Unit delle CPU microprogrammate.

Sistema Operativo

Il **Sistema Operativo** (**O**perating **S**ystem o OS) è il software che gestisce tutto il funzionamento del computer.
Gestisce la memoria, i dispositivi di I/O, i programmi applicativi e permette all'utente di decidere quali programmi applicativi fare eseguire.
Alcuni S.O. sono progettati e realizzati per poter eseguire applicazioni di ogni tipo. Essi vengono detti S.O. "general purpose". Altri invece sono pensati per lavorare con applicazioni particolari. Come esempio si possono portare in S.O. "real time", che servono per i controlli automatici, o i S.O. per i server delle reti di computer.
Alcune parti del S.O. devono stare permanentemente in memoria di lavoro, mentre altri moduli del S.O. possono essere caricati e scaricati dall'hard disk all'occorrenza.

"Famiglia"	Nome del S.O.	Piattaforme	Descrizione	GUI	CUI
Windows	Windows 95 – 98 -Me - XP	PC X86	S.O. dei PC ordinari	X	X
	Windows CE – Pocket PC		Per portatili e embedded	X	X
	Windows NT – 2000 – XP server		Per server e PC workstation	X	X
Unix	Linux	PC X86, Alpha, PowerPC, ARM e moltissime altre	Unix "libero"	X	X
	Solaris	Sun SPARC	Unix Sun	X	X
	HP-Ux	HP PA	Unix HP	X	X
	Digital Unix	Alpha, VAX		X	X
	Sco Unix	PC X86		X	X
	IRIX	Silicon Graphics		X	X
Apple	Apple OS – Apple System X	PowerPC (prima 68000)	S.O. dei Macintosh	X	
Novell	Netware	PC X86	S.O. per reti locali di computer		X
IBM	MVS	Mainframe IBM	S.O. per computer mainframe		X
	OS/400	AS/400	S.O. per minicomputer IBM AS/400		X
	VMS	VAX	S.O. dei VAX Digital		X
	Microsoft MS-DOS	PC 8086	S.O. dei primi PC		X

Tabella 1: alcuni Sistemi Operativi general purpose

Funzioni di un S.O.

Interfaccia utente

Il S.O. è il programma che si presenta all'utente quando nessun applicativo è in esecuzione. In questa occasione il S.O. è l'"interfaccia" che permette all'utente di comunicare con il suo computer. La funzione più appariscente di un S.O., anche se tecnicamente non è la più importante, è proprio quella di presentare una "**interfaccia utente**" tramite la quale è possibile usare il computer.
Alcuni S.O. hanno una interfaccia utente composta solo con caratteri alfanumerici o semigrafici (CUI: **C**haracter **U**ser **I**nterface, user significa "utente"). L'esempio più tipico di un S.O. con interfaccia CUI è MS-DOS.
Molti altri S.O. hanno invece una interfaccia grafica (GUI: **G**raphical **U**ser **I**nterface), in grado di gestire le immagini "punto per punto"[1].
Con questi S.O. l'utente lavora in un ambiente a finestre dal quale è possibile lanciare le applicazioni, vedere i file contenuti nell'hard disk e tutte le altre operazioni che conosce chi ha usato qualche volta un computer con S.O. Windows.

1 Un elemento di una immagine grafica digitalizzata, ovverosia un "punto" sul monitor quando quella immagine viene utilizzata, viene detto "**pixel**" (contrazione, da "Picture element")

In questi casi anche il software applicativo tende ad adeguarsi e ad assumere l'interfaccia utente del S.O., in modo da semplificare la vita all'utente, facendogli usare l'applicativo in un modo cui è già abituato.

Figura 2: interfacce utente CUI e GUI

Gestione dei file

Un altro compito del sistema operativo è "realizzare i file".

Un **file** è un insieme di informazioni omogenee racchiuse in una memoria di massa.

I file hanno un nome simbolico, che li distingue uno dall'altro, ed un'organizzazione gerarchica, che li divide in "directory", che a loro volta possono contenere altre directory.
Come abbiamo visto le memorie di massa, come per esempio un hard disk, sono divise in tracce e settori, non in file o directory. Il file quindi è costituito fisicamente da un insieme apparentemente disordinato di settori di hard disk.
Come oggetto unico il file non esiste se non esiste un Sistema Operativo.
E' il Sistema Operativo che tiene traccia del nome di tutti i file e "sa" quali sono i settori dell'hard disk nei quali si trova ogni pezzetto di ogni file. Inoltre il S.O. dispone di comandi che possono creare un file, leggerlo, modificarlo o cancellarlo.
L'accesso ai file è quindi completamente regolato dal Sistema Operativo, che mette a disposizione dell'utente un vero e proprio "gestore di file".
Il programma del S.O. che fa la gestione dei file viene detto "**file system**". Il file system, "realizza" fisicamente i file, nel senso che tiene traccia dei loro nomi, di dove sono sul disco, delle loro dimensioni, dei permessi di accesso e ne permette la gestione corrente.
Alcuni Sistemi Operativi ammettono l'utilizzazione di file system diversi, per cui sono in grado di operare con file creati da S.O. di altro tipo (es. Linux).

File di testo o binari

Si dice che un file è un "file di testo" ("text file") se contiene solo codici alfanumerici (es. file di testo "puri", che non danno alcun attributo ai caratteri, file sorgenti di programmi (.C, .BAS), HTML).
Si dice che è un "file binario" ("binary file") se contiene informazioni codificate in modo diverso dai classici codici alfanumerici (ASCII, ANSI, Unicode ..). Esempi di file binari sono i file che contengono programmi (.exe in DOS/Windows) e i file che contengono dati di database.
Alcuni tipi di file che contengono testo non possono essere classificati come file di testo, perché contengono parti codificate non in ASCII "stretto" (es. file .DOC e .WRI in Windows).

Gestione delle applicazioni

Una delle funzioni fondamentali di un S.O. è il "lancio" dei programmi.
Quando non c'è bisogno di un programma esso risiede sull'hard disk, sotto forma di file.
Cosa succede quando vogliamo utilizzare quel programma? Senz'altro esso non può rimanere sull'hard disk quando viene eseguito, perché la CPU effettua il fetch delle istruzioni leggendone i codici operativi da locazioni della memoria di lavoro.
Perciò quando l'utente vuole eseguire un programma, e lo richiede al Sistema Operativo attraverso un apposito comando dell'interfaccia utente, il S.O. deve caricare il file contenente il programma in memoria di lavoro, poi lo deve far eseguire.
La parte del S.O. che svolge questo compito si chiama "**loader**" (caricatore).
Il loader cerca sull'hard disk il nome del programma che deve essere eseguito, carica in memoria il contenuto del relativo file, poi inizia l'esecuzione del programma "saltando" alla sua prima istruzione. A quel punto il programma applicativo ha il controllo della CPU e viene regolarmente eseguito.
Quando il programma finisce, esso restituisce il controllo al S.O. che libererà la memoria di lavoro che occupava. Successivamente quella memoria verrà utilizzata da un altro programma.

Gestione delle periferiche

Oltre alla memoria di massa in un computer esistono anche altre periferiche, come per esempio stampanti e tastiera. Fra i compiti del Sistema Operativo c'è anche la gestione delle periferiche, con software di basso livello che controlla direttamente i dispositivi.

Usando i programmi messi a disposizione dal Sistema Operativo gli utenti ed i programmatori potranno dare alle periferiche comandi potenti ed astratti, senza occuparsi della loro programmazione a basso livello.
Per esempio sarà possibile stampare un file di testo senza neppure sapere in che modo la stampante è collegata al computer, dato che di questi dettagli si occupa il S.O.
Si può quindi senz'altro dire che il S.O. realizza un'astrazione delle periferiche, dato che è possibile utilizzarle con comandi molto più potenti di quelli che bisognerebbe emettere se non ci fosse il S.O.

In generale si può dire che il Sistema Operativo ha il ruolo di "**gestore delle risorse**" del computer; ovverosia è il soggetto che ha la responsabilità di coordinare il funzionamento di tutto ciò che "serve" in un computer.

Quelle che abbiamo delineato sono le principali funzioni del S.O. visibili dall'esterno, da parte di un utente che non si interessi del dettaglio tecnico. Come avremo modo di vedere nel prossimo volume il S.O. svolge molte altre funzioni "interne", anche più importanti di quelle che abbiamo visto ora, ma che non sono visibili all'utente normale.

Programmi di utilità (utility)

Le **utility** sono programmi applicativi che aiutano ad effettuare operazioni di manutenzione o ripristino del computer, quali salvataggio di sicurezza di file, dall'hard disk ad altri supporti (**backup**), diagnostica di errori e malfunzionamenti, verifica e ottimizzazione delle prestazioni del sistema.
Insieme al S.O. vengono vendute anche alcune utility, altre possono essere acquistate a parte.

1.4 Gli strumenti di sviluppo

Classifichiamo come "strumenti di sviluppo" (developing **tools**) oggetti hardware o software che servono a realizzare programmi per computer.
La maggior parte degli strumenti di sviluppo sono programmi software ma esistono anche dispositivi hardware che aiutano nella progettazione e realizzazione dei programmi.

1.4.1 Linguaggi di programmazione

Ogni programma viene espresso in un linguaggio di programmazione.
Un **linguaggio di programmazione** è un linguaggio artificiale, cioè un linguaggio che non è stato definito dall'uso corrente da parte delle persone, ma che è stato inventato di sana pianta per scopi specifici.
Le regole di un linguaggio artificiale definiscono la sua grammatica (anche in un linguaggio non artificiale, peraltro!).
La **sintassi** stabilisce come deve essere organizzata una frase corretta in quel linguaggio, la semantica stabilisce il significato di ciascuna delle frasi.
Per quanto quest'ultima distinzione sia giusta ed applicabile, nell'uso informatico corrente si parla quasi sempre di sintassi, raramente di grammatica e non si parla quasi mai di semantica. Così la parola "sintassi" viene di fatto ad assumere tutti e tre i significati elencati prima.
Le frasi corrette di un linguaggio di programmazione vengono dette **istruzioni** (statement).
Nei linguaggi di programmazione **imperativi** (o procedurali) le istruzioni specificano completamente, in modo non ambiguo, i passi che devono essere compiuti per risolvere un problema. Perciò i programmi espressi in linguaggi imperativi realizzano algoritmi.
Alcuni fra i più diffusi delle diverse centinaia di linguaggi imperativi sono: C, BASIC, COBOL, Perl, Pascal, FORTRAN.
Nei linguaggi **dichiarativi** le istruzioni si limitano a descrivere in termini di funzioni o di regole le relazioni fra le variabili, lasciando al traduttore del linguaggio il compito di ottenere il risultato applicando un procedimento automatico.
Nei linguaggi dichiarativi il programmatore non deve esprimere l'algoritmo risolutivo, che viene invece applicato dal traduttore del linguaggio.
Un esempio di linguaggio dichiarativo è il Prolog.
I linguaggi **a oggetti**, come Smalltalk, Java o C++, si possono vedere come una via di mezzo fra i linguaggi imperativi e quelli funzionali.
Essi sono senz'altro linguaggi "algoritmici", quindi richiedono di specificare nel completo dettaglio la soluzione del problema, come succede nei linguaggi imperativi.
Peraltro, se il programma è progettato e realizzato bene, è possibile programmare in modo da "nascondere" il dettaglio algoritmico fino al punto di programmare quasi solo per funzioni.

1.4.2 Traduttori

Un'istruzione di macchina è l'unico tipo di comando che la CPU di un computer può capire ed eseguire. Dunque tutti i programmi, anche quelli scritti in linguaggi di programmazione molto più simili all'Inglese che al linguaggio macchina, per poter essere eseguiti su un computer devono essere in qualche modo tradotti in istruzioni di macchina. La traduzione viene affidata a speciali programmi "traduttori".
Un traduttore ha il compito di convertire codice espresso in un linguaggio di programmazione in un altro linguaggio.
Come si può vedere nella Figura 3: traduttore il linguaggio in ingresso viene detto linguaggio "**sorgente**" (**source** language), mentre l'uscita del traduttore è espressa nel linguaggio "obbiettivo". Quando il linguaggio obbiettivo è il lin-

guaggio macchina della CPU prescelta il risultato della traduzione viene anche detto "codice **oggetto**". Il file che contiene il codice oggetto è detto "file oggetto" o OBJ.

Figura 3: traduttore

Programma in forma sorgente

Il "sorgente" (source code) di un programma è l'insieme dei comandi che specificano il suo funzionamento, espresso secondo le regole di un ben determinato linguaggio di programmazione.
I comandi di un linguaggio di programmazione si chiamano "istruzioni" (instructions), più raramente "clausola" ("statement").
Se memorizzati su un hard disk i programmi in forma sorgente sono **file di** tipo **testo**, contenenti i codici ASCII delle lettere che il programmatore ha digitato sulla tastiera. Se si prova a leggere un file sorgente con un normale programma correttore di testi (text editor), come p.es. "blocco note" di Windows, il contenuto del file sarà immediatamente comprensibile.
Il contenuto di un file sorgente per essere eseguito deve essere tradotto in linguaggio macchina.
Nella Figura 1 entrambi i programmi sono espressi in forma sorgente, il primo in linguaggio C, il secondo in linguaggio Assembly.

Programma in forma eseguibile

I programmi in forma **eseguibile** (executable) sono in linguaggio macchina. Sono costituiti dai codici operativi e dagli operandi del programma, rappresentati in forma binaria.
Se memorizzati su un hard disk gli eseguibili sono **file** di tipo "**binario**", il cui contenuto è comprensibile solo alla macchina.
Se proviamo a guardare un file eseguibile con un editor di testi otteniamo una sequenza apparentemente senza senso di caratteri strani e di lettere. Si tratta dei codici ASCII che corrispondono ai codici operativi delle istruzioni, che naturalmente non hanno senso perché usano una codifica del tutto diversa (per provare in Windows si può aprire con "blocco note" un qualsiasi file con estensione .EXE).
Il contenuto di un file eseguibile, una volta copiato dall'hard disk alla memoria, può essere eseguito direttamente dalla CPU.
I programmi in forma eseguibile vengono anche detti "codice binario".

Interpreti e compilatori

I traduttori possono essere distinti in due grandi classi: interpreti e compilatori.

Un **interprete** è un programma che esegue altri programmi. Traduce il codice sorgente in codice oggetto una istruzione alla volta, facendo subito in modo che le istruzioni tradotte vengano eseguite.

Un interprete prende un'istruzione in linguaggio sorgente, la traduce in linguaggio macchina, la fa eseguire, prende l'istruzione successiva, la traduce, la esegue, poi continua con la stessa sequenza con tutte le istruzioni successive.

Un **compilatore** traduce in una sola operazione tutto il codice sorgente che riceve, produce un file con il programma tradotto e con questo finisce il suo compito, non facendo mai eseguire il programma.

Quando un compilatore ha finito il suo lavoro esiste un file oggetto, che può essere usato successivamente per far eseguire il programma senza aver più bisogno del compilatore. Ciò significa che, a differenza di un interprete, un compilatore non esegue mai il programma che traduce.
Il codice compilato è intrinsecamente più veloce del codice interpretato, dato che il compilatore risolve una volta per tutte il problema della traduzione, prima che il programma venga eseguito. Al contrario un interprete potrebbe dover tradurre molte volte la stessa istruzione, visto che traduce mentre il programma esegue.

"Storia" di un programma

La prima fase della "vita" di tutti i programmi è quella dello sviluppo. Alla fine della fase di sviluppo il programma viene compilato e diventa un file, cioè un insieme di numeri che, quando il programma non serve, viene memorizzato su un supporto di memoria non volatile come un hard disk.

Il programma compilato è "chiuso", non si può più modificare e farà le cose sempre nello stesso modo per tutto il resto della sua vita.

Quando il programma deve essere eseguito viene caricato in memoria principale dal loader e poi eseguito.
Si individuano perciò tre "fasi" nella vita di un programma, i periodi di tempo che corrispondono a quelle fasi sono:
1 - "**tempo di compilazione**" (compile time) il momento in cui il compilatore effettua la generazione del codice macchina del programma;
2 - "**tempo di caricamento**" (load time) il momento in cui il loader carica il programma dall'hard disk alla memoria, prima dell'esecuzione;
3 - "**tempo di esecuzione**" (**runtime**) il periodo di tempo durante il quale il programma viene eseguito.
Queste espressioni sono piuttosto diffuse nell'Informatica e le ritroveremo in futuro in diversi contesti:
Si può dire che un interprete traduce da linguaggio ad alto livello in linguaggio macchina a tempo di esecuzione, mentre un compilatore traduce a tempo di compilazione :- |.

Confronto fra interpreti e compilatori

Gli interpreti sono stati molto utilizzati nei primi personal computer. Quei computer infatti avevano una grande scarsità di risorse: avevano cioè poca memoria ed erano lenti.
Il vantaggio dell'interprete in condizioni di scarsità risorse risiede nel fatto che il programma viene memorizzato nel linguaggio ad alto livello, in forma sorgente, quindi occupa meno memoria, sia nel supporto di memoria di massa, sia nella memoria principale.
Inoltre con un interprete è molto più semplice modificare i programmi durante la loro fase di prova. Infatti non è necessario ricompilare ogni volta che si fanno modifiche al sorgente. I programmi interpretati "partono subito", senza bisogno di essere prima compilati. Per questo il processo di sviluppo, durante il quale è necessario modificare e far partire il programma molto spesso, è facilitato.

Quando i personal computer hanno cominciato ad avere più memoria e più velocità i vantaggi degli interpreti sono scemati, sono nati ambienti di programmazione in cui, dato che il computer era veloce, il programma veniva compilato ed eseguito automaticamente ogni volta che il programmatore voleva provarlo. Per questo anche i linguaggi compilati hanno assunto la stessa facilità di verifica e modifica che era caratteristica dei linguaggi interpretati.
L'esempio più tipico di questa evoluzione fu il Turbo Pascal, primo ambiente di sviluppo compilato che ottenne una semplicità di debugging paragonabile a quella del BASIC. A partire da quel momento gli interpreti furono meno usati ed anche il più classico dei linguaggi interpretati, il BASIC, offrì l'opzione di compilare il suo codice.

Caratteristica	Meglio compilatore	Meglio interprete
Velocità del programma realizzato	+	-
Velocità dell'esecuzione in fase di test del programma	-	+
Ambiente di programmazione	Oggi =	+
Occupazione di memoria del codice	-	+
Portabilità del codice	Oggi =	+

Tabella 2: confronto fra linguaggi interpretati e compilati

La tabella non rende onore ai compilatori, che hanno una minore quantità di vantaggi, ma una maggiore "qualità". Infatti la velocità di esecuzione del programma realizzato è quasi sempre di gran lunga la cosa più importante che si chiede ad un traduttore di linguaggi di programmazione.
Non per nulla al giorno d'oggi la grandissima maggioranza dei programmi applicativi ordinari è compilata.
Recentemente si è assistito ad rilancio degli interpreti, dovuto all'ascesa della programmazione per Internet. Una caratteristica decisiva di Internet è che la velocità di trasmissione delle informazioni è molto bassa.
In queste condizioni diventa importante la maggiore compattezza del linguaggio in forma sorgente (vedi Figura 1).
Se si spedisce il programma in forma sorgente e sul computer che lo riceve c'è il relativo interprete si possono eseguire elaborazioni in cui il programma è fornito dal computer "remoto" ed il computer locale mette a disposizione la sua CPU.
JavaScript, VBscript, Perl, PHP, Python sono esempi di linguaggi interpreti sviluppati recentemente, spesso per l'accesso a Internet.
Un altro vantaggio importante degli interpreti è la "portabilità". Lo stesso programma sorgente può essere eseguito su CPU diverse, semplicemente cambiando l'interprete.

Emulatori di CPU

Simile ad un interprete è il funzionamento di un emulatore di CPU. Un **emulatore** di CPU è un programma che traduce a tempo d'esecuzione il contenuto di un programma compilato per una CPU nelle istruzioni del linguaggio macchina di una CPU diversa.

Un emulatore usa direttamente un file che sarebbe eseguibile sulla CPU emulata e lo traduce "al volo" in codice della CPU "ospite" (host CPU).
Date le premesse si capisce come il codice emulato avrà delle prestazioni in velocità molto inferiori del codice "nativo", cioè del codice scritto proprio per quella CPU che lo deve eseguire.
Come esempio si può portare un emulatore di Commodore 64 su PC, che esegue il codice binario scritto per il 6502 del C64 traducendolo in codice 8086. Naturalmente oltre al codice macchina del 6502 un emulatore per C64 dovrà "impersonare" anche il resto del software e dell'hardware del C64. Dato che i computer odierni sono molto più veloci del C64 tutto questo processo di traduzione "sottobanco" potrà avvenire senza che l'utente se ne accorga.
Le CPU moderne, come per esempio PowerPC, hanno molte caratteristiche pensate appositamente per facilitare l'emulazione del set d'istruzioni di CPU "straniere".

Un emulatore deve tradurre anche tutte le chiamate al Sistema Operativo originario alle corrispondenti chiamate del S.O. ospite. Ciò avviene perché il codice compilato fa uso di chiamate al Sistema Operativo del computer che usava originariamente e l'uso dell'emulatore non deve prevedere la ricompilazione del codice sorgente originale.

Macchine virtuali

Internet ha rilanciato, oltre che gli interpreti, anche un modo di tradurre i programmi che sta un po' a metà strada fra l'interpretato ed il compilato.
Si tratta della compilazione con codice intermedio. In questo caso esistono sia un compilatore che un interprete.
Come si vede anche nella Figura 4: compilazione con codice intermedio, durante la fase di sviluppo del programma viene utilizzato un compilatore, che però non produce un programma eseguibile, ma un codice intermedio, detto "**bytecode**", o p-code, che non può essere eseguito direttamente da nessuna CPU e che è fatto apposta per essere facilmente interpretato.
Quando il programma viene fatto eseguire il bytecode viene dato in pasto ad un interprete, che lo traduce a tempo d'esecuzione nel codice della macchina in cui il programma sta eseguendo in quel momento.
Questo approccio non dà luogo ai programmi più veloci possibili, come invece succede con i compilatori "nativi" (native compilers), ma ha il grande vantaggio che è molto più facile rendere portabili i programmi.
Infatti lo stesso codice compilato in bytecode può funzionare su CPU diverse semplicemente usando interpreti di bytecode diversi.

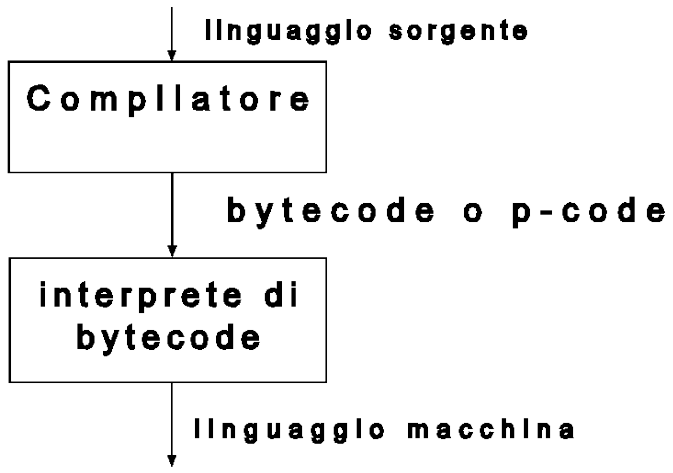

Figura 4: compilazione con codice intermedio

La compilazione con codice intermedio è stata usata da particolari versioni del Pascal e da linguaggi come il Lisp ed è portata al suo estremo nel linguaggio Java.

Java
Java, è un linguaggio di uso generale ed è stato progettato all'insegna dello slogan "write once, run everywhere".
Il linguaggio è compilato in bytecode. Il bytecode di Java è il codice di macchina di una vera e propria CPU virtuale.
Le specifiche di Java comprendono infatti la definizione di una "**macchina virtuale Java**" (Java Virtual Machine, JVM").
La macchina virtuale Java è costituita da una CPU, di architettura piuttosto semplice che non esiste in hardware, e da diverse aree di memoria. La CPU della JVM ha il suo set di istruzioni di macchina i relativi codici operativi.

La JVM fa uso di dati tipizzati, anche se non fa controlli di tipo che sono lasciati al compilatore, ha istruzioni per numeri a virgola mobile, per gli array, ha meccanismi potenti per la sicurezza e l'allocazione della memoria, gestisce gli errori in modo potente ed originale.
In definitiva è una macchina che accetta comandi di un livello un po' più alto delle CPU tradizionali. Questo permette al bytecode JVM di essere più compatto del codice macchina delle normali CPU.
Per poter eseguire il codice macchina di una CPU che non esiste fisicamente, su ogni computer che deve eseguire un programma Java deve essere presente un emulatore di macchina virtuale, cioè un programma che, a tempo d'esecuzione, traduce il bytecode nel linguaggio macchina della CPU ospite.
Questo approccio, molto utile sull'Internet ove sono presenti computer di piattaforme diverse, permette di eseguire su CPU del tutto diverse lo stesso codice binario in bytecode.
Ciò viene messo in evidenza nella Figura 5, nella quale si vede come il codice prodotto e compilato sul computer dello sviluppatore viene fatto girare sui due computer dell'utente 1 e dell'utente 2, che sono di piattaforme diverse, distribuendo ad entrambi lo stesso bytecode.
La macchina virtuale Java accetta bytecode che potrebbe anche essere prodotto da un compilatore di sorgenti non Java.

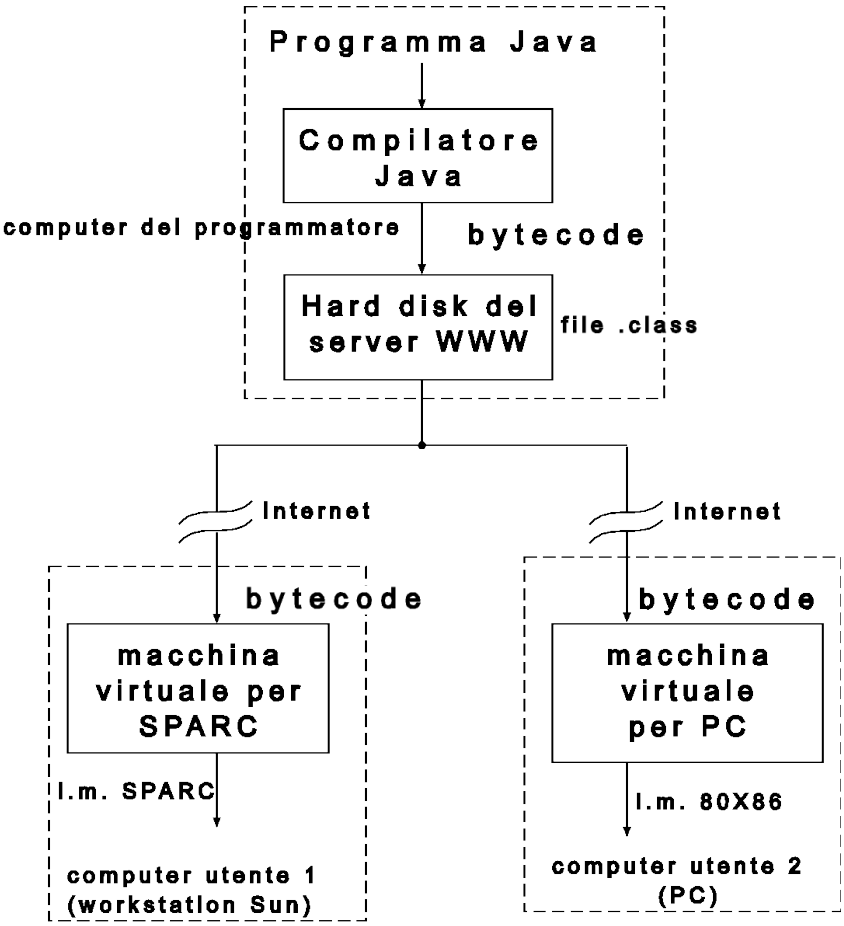

Figura 5: modello di distribuzione del codice Java

Compilatori "just in time"

I compilatori "just in time" ("j.i.t.") sono fatti per ricevere le istruzioni in bytecode, tipicamente attraverso una rete di computer, e per tradurle immediatamente in linguaggio macchina, generando un codice eseguibile.
La compilazione just in time è usata per rendere più veloci i programmi Java, sostituendo la macchina virtuale interpretata con un secondo compilatore, che genera codice nativo e perciò più veloce.

C#

C# (leggi C sharp) è un linguaggio di programmazione recente, nato insieme al ".NET framework" d Microsoft.
In molte cose è molto analogo a Java e ne condivide l'approccio con compilazione in codice intermedio, interpretato successivamente da una macchina virtuale.
Il linguaggio con cui è espresso il codice intermedio si chiama CIL (Common Intermediate Language), ed è pensato per essere fortemente indipendente dal linguaggio ad alto livello usato per lo sviluppo delle applicazioni. In questo modo un'applicazione può essere scritta in molti linguaggi diversi (es. C#, Visual BASIC .NET, Java e molti altri non

sviluppati direttamente da Microsoft). Data la presenza dell'interfaccia CIL è anche possibile, come con Java, avere una maggiore possibilità di far girare il software su piattaforme diverse.

Cross compilatori
Se un compilatore per una certa CPU esegue su un tipo di CPU diversa viene detto **cross** compilatore (**X**compiler). Data la ubiquità ed il basso costo del PC quasi tutti i produttori lo usano come piattaforma di sviluppo per il codice dei loro microcontrollori.
Perciò molto spesso il compilatore C o Assembly dei microcontrollori è un programma per PC, che gira su un PC, con codice macchina 80X86. Il risultato del cross compilatore è invece un codice eseguibile per la CPU "target" (obiettivo), quindi è costituito di codice macchina eseguibile dal microcontrollore, non dal PC.
Un Assembler che compila codice per una CPU diversa da quella sulla quale gira viene detto **cross assembler** (XASM).

1.4.3 Realizzazione dell'eseguibile

Normalmente il processo di realizzazione di un eseguibile è diviso in più fasi, come evidenziato nella Figura 6.
La prima fase è la progettazione del programma, durante la quale si stendono le specifiche e si analizza il problema.
La seconda fase è la scrittura del codice (punto (2) di Figura 6), che può essere fatta attraverso un normale programma per la redazione dei testi ASCII (**text editor**), come per esempio EDIT di MS-DOS o "Blocco note" di Windows.
Durante questa fase si produce un file di testo, che verrà messo in ingresso al compilatore.

<FILE>
realizz.FH5
</FILE>
Figura 6: realizzazione e collaudo di un programma eseguibile

La terza fase è la compilazione del programma, che dal programma in forma sorgente produce il programma in forma **oggetto** (punto (3) di Figura 6).
Il file oggetto è un file binario che contiene le istruzioni in linguaggio macchina.
Il nome del file oggetto può terminare in modi diversi, in base al tipo di computer che si usa. Per esempio nel caso di DOS – Windows l'estensione è .OBJ, in Linux è .o.
Il formato del file oggetto permette di collegare insieme parti di programma compilate in tempi diversi ed eventualmente anche scritte in linguaggi sorgente diversi.
Se il programma sorgente non rispetta tutte le regole del linguaggio sorgente la fase di compilazione non si può concludere e il compilatore non crea il file .OBJ.
In questo caso si deve tornare all'editor per correggere gli errori, ricompilare e continuare così fino a che il programma non rispetta perfettamente la sintassi del linguaggio sorgente. Quando ciò accade il compilatore produce il suo risultato, cioè il file .OBJ (**Obj**ect file).
Quando il programma compila regolarmente si può passare alla fase successiva (punto (4) di Figura 6).
I file oggetto prodotti dai compilatori vengono messi insieme dal **linker** (vedi oltre). Utilizzando il **linker** si produce il programma eseguibile.
Anche nella fase di "linkage" potrebbero esserci errori, che vanno eventualmente corretti utilizzando il text editor.
Quando sono rispettate tutte le regole per il collegamento dei vari file oggetto l'esecuzione del linker produce un file eseguibile[2], in linguaggio macchina, che è in grado di essere eseguito dalla CPU senza ulteriori traduzioni.

Naturalmente non è detto che il programma che si è ottenuto esegua perfettamente le funzioni per le quali è stato scritto. Per verificare il suo funzionamento e trovare le cause di eventuali errori logici esistono i "**debugger**".

Un debugger è un programma che aiuta nella verifica passo passo del funzionamento di un altro programma.

Utilizzando il debugger si controlla il funzionamento del programma, fino a che non c'è una ragionevole certezza che esso funzioni correttamente in tutti i casi che si presenteranno durante la sua vita operativa.
A questo punto il programma è finito: lo si può consegnare al committente.
Al processo di sviluppo di un programma si può aggiungere un'ulteriore fase (punto (6) di Figura 6). Una volta che il programma funziona, potrebbe essere necessario migliorarne la velocità di esecuzione, oppure diminuirne l'occupazione di memoria. Per aiutare nell'ottimizzazione del software si usano programmi detti "**profiler**" (pronuncia "profailer").

Ambienti di sviluppo

I linguaggi di programmazione moderni dispongono di veri e propri "**ambienti di sviluppo**" (IDE **I**ntegrated **D**evelopment **E**nvironment) che comprendono un text editor per la modifica dei sorgenti, un programma che è in grado di

2 L'estensione dei file eseguibili in DOS e Windows è .EXE, mentre in Unix – Linux i file eseguibili non devono avere alcuna estensione particolare.

lanciare il compilatore per produrre il file eseguibile, un debugger. Con questi strumenti si possono seguire tutte le fasi dello sviluppo senza uscire dall'ambiente.
Si fa notare che in questi casi è il software stesso dell'ambiente di sviluppo che tiene traccia di tutti i file sorgente usati, provvede automaticamente a lanciare il compilatore per tutti i file sorgente e ad unire gli OBJ chiamando il linker. Dunque il lavoro di compilazione e linking ha luogo comunque, anche se il programmatore se la cava con una semplice scelta di menù o con il click del mouse su un bottone.

1.4.4 Funzionamento di un compilatore

Un compilatore deve riconoscere tutte le istruzioni del linguaggio sorgente e tradurle in istruzioni del linguaggio obiettivo. Per effettuare questo riconoscimento il linguaggio sorgente deve essere fatto in modo da non essere ambiguo, dato che il compilatore deve sapere esattamente cosa fare in ogni caso.
Delineiamo formalmente le funzioni che devono essere svolte da un compilatore ed anche, sia pur con modalità diverse, da un interprete.
Queste distinzioni formali, interessanti ed utili per capire, spesso sono disattese. Infatti le funzioni che illustreremo potrebbero essere svolte da un compilatore nell'ordine indicato oppure tutte contemporaneamente, ciò in base alle scelte di progetto effettuate dai progettisti del compilatore.

Scanner

Le istruzioni dei linguaggi di programmazione sono costituite da simboli, organizzati secondo le regole del linguaggio. Per prima cosa il compilatore dovrà andare alla ricerca di questi simboli, che costituiscono le "parole" del linguaggio.
La prima fase di un processo di compilazione è detta "analisi lessicale", o "scanning", ed è effettuata da un componente del compilatore detto "analizzatore lessicale" (lexical analyser) o scanner. Lo scanner analizza il file sorgente una lettera dopo l'altra, identificando tutti i componenti fondamentali che lo costituiscono.
I componenti individuati dallo scanner sono le "parole" del linguaggio e vengono detti "**token**" (gettone).
Si chiama "**separatore**" un carattere che non può far parte di nessuna parola. Il carattere separatore tipico è lo spazio (**blank**)[3]

Un token è una stringa compresa fra due separatori. Inizia dopo un separatore e finisce prima del separatore successivo.

Ogni linguaggio definisce i suoi tipi di token, per esempio il C++ ne ha sei: parola riservata (keyword), identificatore (identifier), costante (constant), stringa alfanumerica (literal string), operatore (operator), separatore (punctutation).
Alcuni token sono comandi del linguaggio, essi vengono detti "parole riservate" (**keyword**). Per esempio "begin" è una parola riservata del Pascal, "MOV" dell'Assembly 8086, "for" di Pascal, BASIC e C.
Altri tipi di token sono i simboli definiti dall'utente, che chiamiamo identificatori. Un **identificatore** (identifier) è una successione di caratteri che inizia con una lettera e che termina con un separatore. Esso è quindi un nome arbitrario, con il quale il programmatore vuole indicare "qualcosa".
Nei linguaggi d'alto livello gli identificatori più tipici sono i nomi delle **variabili** o delle procedure.

In Assembly i simboli definiti dall'utente sono nomi assegnati dal programmatore a locazioni di memoria. Il compilatore deve associare automaticamente indirizzi agli identificatori che trova nel programma.

I simboli definiti dall'utente (identificatori) non possono essere parole chiave (keyword).

Un altro tipo di token è la costante. Una costante è un valore fisso che rappresenta un numero o una stringa.
Un **numero** inizia con una cifra da 0 a 9, anche se è esadecimale.

```
Per esempio, il token "FFFh" è interpretato dal compilatore Assembly come un identifica-
tore; per far capire al compilatore che si sta operando con un numero bisogna farlo co-
minciare con una cifra decimale, cioè bisogna scriverlo così:  "0FFFh".
```

Un **operatore** è un simbolo che indica una operazione, come per esempio il "+".
La Figura 7 riassume le definizioni date in questo paragrafo, identificate in un programma Assembly:
<FILE>
!!!! DA FARE !!!!
</FILE>

Figura 7: elementi lessicali identificati dallo scanner

Parser

Una volta che il programma è stato analizzato dallo scanner tutti i token sono stati individuati, ora è il momento di analizzare come i token sono messi assieme, per vedere se ci sono errori nelle "frasi" del programma.

3 gli altri caratteri separatori dipendono dal linguaggio; in Assembly in alcuni casi si usa il due punti ":", altri separatori sono il punto e virgola e la virgola.

Questo è il compito dell'analizzatore sintattico o "parser".
Il parser prende ciascun token e gli assegna un significato, verifica la congruenza delle istruzioni del programma con la sintassi del linguaggio sorgente, genera messaggi di errore o avvertimenti se trova cose che non funzionano.
Se il programma è corretto il parser produce una forma intermedia di codice che viene usata dal generatore di codice per compilare.
Esistono parser "generici" che accettano in ingresso la descrizione formale di una qualsiasi grammatica arbitraria e fanno l'analisi sintattica di qualsiasi testo secondo quella grammatica (es. yacc di Unix).

Generatore del codice

Qualora il programma sia sintatticamente corretto si procede alla generazione del codice.
Il generatore di codice produce:
- una "tabella dei simboli", che associa indirizzi di memoria agli identificatori definiti dall'utente
- un programma in linguaggio target che è equivalente al programma originario in codice sorgente
Il secondo di questi elementi viene salvato come file oggetto (.OBJ in DOS-Windows).

Errori di sintassi ed errori logici

Se il compilatore trova istruzioni espresse secondo una sintassi non esatta per il linguaggio sorgente non può produrre il file oggetto. In questo caso si dice che c'è stato un "**abort**" della compilazione, causato da un "fatal error" o **errore di sintassi**. Il compilatore prova a proseguire, per individuare eventuali altri errori di sintassi, ma alla fine della sua esecuzione non produce il file oggetto e conclude una lista degli errori che ha trovato.
Se invece incontra istruzioni ambigue che però può risolvere senza interrompere la compilazione, prosegue e genera comunque un file oggetto, ma emette un "**warning**" (avvertimento). Il warning è una scritta che avverte il programmatore che c'è qualcosa di strano in ciò che il compilatore ha generato. Il programmatore potrà così essere al corrente di possibili problemi, che rischiano di dar luogo ad errori logici.

Dopo aver corretto tutti gli errori di sintassi, il compilatore ed il linker producono un file eseguibile.
Purtroppo non è detto che il programma eseguibile sia un programma funzionante, perché un programma funzionante è quello che fa sempre ciò che deve fare.
In un programma che compila felicemente potrebbero essere nascosti **errori logici**. Un errore logico è una parte di una programma corretta dal punto di vista del linguaggio di programmazione che però non fa ciò che dovrebbe fare.

Funzionamento del compilatore Assembly

Sarebbe interessante, a questo punto, entrare nel dettaglio del funzionamento del compilatore Assembly, che, avendo una corrispondenza così stretta con il linguaggio macchina, ha un funzionamento piuttosto semplice.
Però per far questo bisognerebbe introdurre troppe istruzioni Assembly non ancora viste, per cui si rimanda questa trattazione al capitolo , "", paragrafo "".

1.4.5 Linker e librerie

Ci si potrebbe chiedere come mai il processo di realizzazione di un eseguibile comprende l'uso del linker e perché il compilatore non produce direttamente un file eseguibile.
Una delle ragioni più importanti è già stata accennata: è possibile collegare insieme, nello stesso programma, parti scritte in linguaggi diversi.

<FILE>

MultLang.FH5
<FILE>

Figura 8: programmi scritti in diversi linguaggi

La Figura 8 illustra il collegamento di un programma costituito di parti in C in Pascal ed in Assembly. Il programma principale è contenuto nel file MAINPROG:C, che utilizza, in tutto o in parte, parti di programma il cui sorgente è in PROCEDURE.PAS e ProcASM.ASM. I compilatori dei tre linguaggi producono ciascuno un file .OBJ, poi il linker unisce i tre .OBJ, creando un unico file MAINPROG.EXE, che contiene tutto il programma.

Il linker più usato in Linux è ld. Il procedimento da seguire per produrre un file per produrre un file eseguibile in Assembly può essere il seguente:

```
# nasm -f elf programma.asm
```

Questo comando, dato al prompt del Sistema Operativo, fa partire uno dei diversi Assembler che si possono usare in Linux, in questo caso si tratta di "nasm". L'Assembler produce il file oggetto di nome programma.o, che può essere collegato con

```
# ld -s -o progr programma.o
```

Questa linea di comando invoca l'uso del linker (ld), che produce il file eseguibile progr, a partire dal file oggetto programma.o.
Per lanciare il programma così prodotto si potrà fare così:

```
# ./progr
```

Si può notare che in questo caso il file eseguibile è senza estensione; in tutti gli Unix i file eseguibili non hanno un'estensione particolare, ma sono distinti dai file normali attraverso uno dei loro "attributi").

La seconda ragione per cui esiste una fase di linkage è che con compilatore e linker separati è possibile collegare insieme parti di programma compilate in tempi diversi.
Ciò significa che programmatori o aziende possono accumulare nel corso del tempo le parti più significative del software da loro prodotto e riutilizzarlo in tempi diversi senza neppure doverlo compilare, semplicemente collegandolo con il linker. Un'altra conseguenza è che le aziende possono acquistare **librerie** di software già complete, verificate e compilate, e collegarle con il linker, invece di scriverle daccapo.
Per chi scrive le librerie il vantaggio della presenza del linker è che potranno distribuire il solo file oggetto, evitando di divulgare il sorgente del programma.
Come esempio tipico di librerie si possono considerare quelle che eseguono le operazioni in virgola mobile. Molte CPU non hanno istruzioni per eseguire direttamente operazioni in virgola mobile, per cui esse devono essere eseguite in software, realizzando programmi complicati e difficili da verificare. Al giorno d'oggi solo poche aziende realizzano questi programmi "in casa". La maggior parte dei produttori di software acquista librerie commerciali che comprendono tutte le operazioni fra numeri in virgola mobile.

1.4.6 Debugger

Il nome "de-bugger" implica la disinfestazione dai "bug". Il termine bug (scarafaggio) nel gergo tecnico americano indica un errore od un malfunzionamento.
Il debugger non "trova" nessun errore per conto del programmatore. Infatti esso non ha nessun ruolo attivo nella scoperta degli errori, ma è solo un modo per vedere "cosa succede", con in massimo dettaglio possibile
Il debugger è un programma non intelligente e non può certo trovare gli errori logici nascosti in un programma, che sono difficili da trovare anche per un umano. Sarà compito di chi lo usa capire dove si trova l'errore, analizzando i sintomi che gli si presentano durante il collaudo.
Le principali funzioni di un debugger sono:
- esecuzione passo-passo dei programmi, un'istruzione di macchina alla volta
- visualizzazione e possibilità di modificare il contenuto dei registri e di qualsiasi locazione di memoria in ogni istante
- interruzione dell'esecuzione al momento in cui si verificano certe condizioni

Con un debugger si può vedere e modificare ogni registro, flag e locazione di memoria, in ogni punto dell'esecuzione del programma.

Nel prosieguo di questo paragrafo si faranno inevitabilmente diversi salti in avanti. La comprensione minimale delle caratteristiche di un debugger non è preclusa a chi non ha ancora appreso i concetti cui si fa riferimento, ma senz'altro questo è un paragrafo sul quale è necessario tornare nuovamente dopo aver "quasi finito" il libro.

Disassembler

La "minima" funzione che un debugger contiene, oltre l'esecuzione al passo, è il "disassembler".
Questa funzione permette di trasformare il linguaggio macchina della CPU in un formato più leggibile, utilizzando i codici mnemonici (vedi indice analitico) dell'Assembly.
Come suggerisce il nome il Disassembler è un programma che ha la funzione inversa dell'Assembler. Mentre l'Assembler prende un programma in Assembly e lo traduce in linguaggio macchina, il Disassembler prende un programma in linguaggio macchina e "fa il possibile" per visualizzarlo in Assembly.
Nella nell'"area programma" compare, accanto al codice in linguaggio macchina rappresentato con numeri esadecimali, la vista disassembler, che mostra l'istruzione di macchina in forma più "umana".
Quando un Assembler traduce dal file sorgente al file oggetto evita di inserire nel file oggetto molte informazioni importanti, che occuperebbero molto spazio di memoria e non sarebbero utili alla gran parte degli utenti. Un'altra ragione per cui le informazioni simboliche non sono comprese nel file oggetto è la protezione del sorgente dalla copia illecita.
Dunque molte informazioni del sorgente non sono presenti nel file eseguibile.
Il disassembler, che lavora sul file eseguibile, non potrà ricostruire il file sorgente in Assembly, perché mancano informazioni molto importanti come per esempio:
- tutti i simboli definiti dal programmatore, che il compilatore sostituisce con numeri (indirizzi)
- tutte le direttive (pseudoistruzioni, vedi oltre) date al compilatore, che non hanno conseguenze sul codice macchina
- tutti i commenti messi dal programmatore nel codice sorgente.

Se queste informazioni non sono disponibili il disassembler non le può mostrare e la correzione dei programmi diviene più difficile.

Debugger non simbolico

I debugger che non sono in grado di mostrare i simboli definiti dall'utente e di riferirsi al codice sorgente durante il loro uso vengono detti "debugger **non simbolici**".
Un esempio di debugger non simbolico è il programma DEBUG, che viene copiato in ogni "disco di ripristino" di Windows 9X". Le sue funzioni sono piuttosto limitate e l'interfaccia utente è del tutto spartana, ma possiede tutte le funzioni che sono presenti in (che mostra la "finestra CPU" del Turbo Debugger (TD)).
Esaminiamo puntualmente ciò che un debugger ci mette a disposizione.

Area programma

Un debugger può visualizzare un'area di memoria "come se" contenesse un programma ("area programma" in).
Visualizzando il programma ci vengono proposte tre colonne:
1) indirizzo delle locazioni di memoria ove il programma risiede (in forma segmento:offset (vedi segmentazione))
2) codice in linguaggio macchina, così come lo vede la CPU, in rappresentazione esadecimale per occupare meno spazio
3) codice in linguaggio Assembly, così come lo traduce il disassembler, quindi senza le informazioni simboliche
Durante l'esecuzione passo-passo il "program counter" si sposta; anche la visualizzazione dell'area di programma si sposta di conseguenza.
E' possibile modificare le istruzioni presenti in memoria, perché è il debugger include un Assembler minimale che traduce una singola istruzione e la scrive in memoria nel punto indicato dall'utente.

Breakpoint

Il funzionamento di un programma può richiedere l'esecuzione di miliardi di istruzioni. Sarebbe improponibile il controllo manuale passo-passo di un programma di tali dimensioni.
Il debugger deve fare in modo che il programma possa essere lanciato alla sua normale velocità, ma anche che sia mantenuta la possibilità di interromperlo, per controllare se lavora correttamente.
Questo è possibile in quasi tutti i debugger (per esempio con TD il programma si "lancia" con F9 e si interrompe con Ctrl-BlocScorr (Ctrl-ScrollLock)).
Ma l'interruzione "da tastiera" non è sufficiente, perché il programma si ferma in un punto qualsiasi, sul quale l'utente del debugger non ha controllo.
Serve un meccanismo che interrompa il programma in un punto specifico. Questo meccanismo esiste in tutti i debugger ed è detto "breakpoint" (il termine non viene tradotto in Italiano).

Come dice il nome (to break significa "rompere"), il breakpoint è un punto in cui il programma viene interrotto dal debugger.

Quando il programma, che sta girando a "tutta velocità", incontra un breakpoint, si ferma. A quel punto l'utente può riprendere il controllo del debugger e verificare se il programma ha funzionato bene, mentre girava a "tutta velocità".
Con TD per mettere o togliere un breakpoint basta premere F2 mentre di è "sopra" all'istruzione interessata.

Area dati

Un debugger può visualizzare un'area di memoria come se contenesse dati ("area dati" in).
Se un'area di memoria contiene dati, il programmatore non è interessato ad usare il disassembler, come succedeva nell'"area programmi", ma piuttosto a verificare quali sono i numeri contenuti in memoria.
I numeri contenuti in memoria sono visualizzati in due modi: come numeri veri e propri o come caratteri alfanumerici.
Visualizzando un'area dati ci vengono proposte tre colonne:
1) indirizzo delle locazioni di memoria il cui contenuto è visualizzato a fianco
2) numero contenuto nelle locazioni, in rappresentazione esadecimale per occupare meno spazio
3) codice ASCII corrispondente al numero contenuto nelle locazioni
In questo modo se si è interessati al numero contenuto in memoria si guarda nella "vista numeri", se si sta esaminando una stringa di codici ASCII (vedi "stringa" e "ASCII" nell'indice analitico), si guarda nella "vista ASCII".
Con tutti i debugger è possibile modificare tutti i valori contenuti in memoria.

Area registri

Ogni debugger è in grado di visualizzare il valore corrente di tutti i registri della CPU, e di modificarlo se l'utente chiede di farlo (in TD basta "scriverci sopra").

Area flag

Ogni debugger è in grado di visualizzare e di modificare il valore corrente di tutti i flag della CPU (in TD basta andarci sopra e premere Enter).

Area stack

Lo stack è un'importante struttura dati che viene usata molto estesamente dalla CPU e dal programmatore Assembly. La maggior parte dei debugger ha un modo agevole per visualizzare il contenuto di questa importante area di memoria.
Siccome lo stack si trova in memoria nell'"area stack" saranno visualizzate due colonne: una per gli indirizzi di memoria, l'altra per i numeri contenuti a quegli indirizzi dello stack (visualizzati in esadecimale).

Le funzioni illustrate finora sono tutto ciò che può fare un debugger non simbolico, oppure un debugger simbolico che verifichi un programma eseguibile del quale non si ha il codice sorgente.

Debugger simbolico

Se si dispone del codice sorgente di un programma si può ordinare all'Assembler ed al linker di inserire nel codice eseguibile delle informazioni in più, che servono al debugger per reperire informazioni molto importanti sul programma. [4]

Un debugger è "**simbolico**" quando può visualizzare e modificare la memoria usando i nomi simbolici stabiliti nel programma sorgente.
Quando nell'eseguibile sono "pubblicati" i dati simbolici per il debugger esso può agevolare il compito del programmatore mettendo a disposizione le seguenti funzioni:
- visualizzazione direttamente sul sorgente del flusso del programma, in esecuzione passo-passo
- visualizzazione e modifica della memoria usando i simboli definiti nel sorgente e non gli indirizzi assoluti
- interruzione dell'esecuzione del programma al verificarsi di condizioni espresse usando i simboli (watchpoint)

Il programma è lo stesso di .
La mostra una finestra di un debugger che lavora in modo simbolico.
Si può notare che nell'"area sorgente" è visibile il sorgente in Assembly, con un triangolo che identifica il punto in cui l'esecuzione è ferma. Se eseguiamo il programma passo-passo, ogni volta il triangolo si sposterà all'istruzione successiva.
La vita di chi deve collaudare il programma diventa molto più facile, perché ora si può avere una corrispondenza diretta fra ciò che si fa sul codice sorgente e ciò che si verifica con il debugger.
Nell'"area watches" di l'utente può chiedere di visualizzare il contenuto della memoria, utilizzando i simboli presenti nel programma sorgente.
Verrà visualizzato il nome del simbolo ed contenuto corrente dell'area di memoria che gli corrisponde.
Questa è una grande facilitazione, perché non ci sarà più bisogno di sapere quale indirizzo il compilatore ha assegnato ad ogni simbolo.
Ogni volta che il programma viene fermato, per esempio perché si incontra un breakpoint o perché si esegue al passo, il debugger accede all'area di memoria che corrisponde a quel simbolo e ne visualizza automaticamente il valore.

Watchpoint

Un debugger simbolico permette anche di fermare l'esecuzione del programma al verificarsi di condizioni che riguardano i simboli definiti dall'utente.
Supponiamo per esempio che il sintomo del malfunzionamento di un programma sia il fatto che la "variabile" Somma ha il valore 6. Supponiamo anche di non sapere quando quella "variabile" diventa 6.
Allora possiamo introdurre un watchpoint che controlli l'espressione: Somma = 6 e lanciare il debugger a "tutta velocità".
Ad ogni passo del programma il debugger andrà in memoria alla locazione Somma e controllerà se vale 6. Se non vale 6 proseguirà con la prossima istruzione, se vale 6 invece si ferma.
E' quindi chiaro che con i watchpoint è possibile arrivare direttamente al cuore del problema, al punto dove, dato che Somma vale 6, ci deve essere l'errore.

Abbiamo perciò visto come i debugger simbolici estendano notevolmente le funzionalità di quelli non simbolici. Naturalmente essi mettono anche a disposizione viste analoga a quella di , con le quali esaminare i registri, i flag e la memoria, più a "basso livello".

1.4.7 Profiler (misuratore di prestazioni)

Nei programmi il 10% del codice gira per il 90% del tempo. Per migliorare le prestazioni dei programmi è inutile spendere energie ottimizzando la velocità del 90% di codice che non gira mai, ma è meglio focalizzare l'attenzione su quel 10% che gira sempre. Per individuare quella parte di codice che esegue per la maggior parte del tempo può essere utile un "**profiler**".
Per profiler si intendono due tipi di applicazione che hanno lo scopo di valutare la "qualità" del software prodotto. L'accezione più comune del termine profiler è quella di "software che valuta le prestazioni di un programma". Esso si installa "nascosto" quando il programma viene fatto eseguire e cattura continuamente informazioni durante l'esecuzione. È quindi in grado di realizzare un "profilo" dell'applicazione, individuando le parti del programma che sono eseguite più spesso, quelle che sono eseguite raramente e le parti di codice più lente nell'esecuzione.
Il secondo significato del termine profiler è usato meno frequentemente. Infatti si usa questo termine anche quando si parla di un programma che analizza il codice sorgente e produce un rapporto calcolando diversi parametri che danno un'indicazione sulla qualità del software scritto.

4 Per ottenere le informazioni simboliche con MASM, TASM e MASM32 bisogna usare l'opzione /Zi; esempio: TASM MIOPROG.ASM /ZI
con TLINK l'opzione /V; esempio: TLINK MIOPROG.OBJ /V

1.4.8 Altri strumenti per lo sviluppo di software e hardware

Quelli visti finora sono tutti strumenti software. Come avevamo accennato esistono anche strumenti hardware che possono aiutare nello sviluppo dei programmi e nell'integrazione del software con l'hardware su cui deve girare.
L'uso tipico di questi strumenti è nello sviluppo di sistemi dedicati a microcontrollore (embedded).
Il modo più semplice per realizzare un programma per un microcontrollore embedded è usare un cross compilatore su PC; compilare il programma, poi programmare una EPROM che contiene il codice binario e provare il sistema complessivo, con CPU, EPROM e I/O. Questo modo poco costoso non è però praticabile per sistemi un minimo complessi.
Infatti per essere sicuri del funzionamento del programma non basta vedere se funziona "dall'esterno", bisogna provarlo con un debugger.
Inoltre in aziende in cui in molti lavorano allo sviluppo di uno stesso progetto ci sono persone che si dedicano alla progettazione ed allo sviluppo dell'hardware ed altre che lavorano per la scrittura del software. Per minimizzare i tempi di sviluppo queste persone dovranno lavorare contemporaneamente, perciò è probabile che chi fa il software debba cominciare a provarlo ancora prima che l'hardware dedicato sia pronto.
Queste prove sono possibili utilizzando un "sistema di sviluppo".

Sistemi di sviluppo

Evaluation board
Il "sistema di sviluppo" più semplice è la "scheda di valutazione" (evaluation board). Essa è un computer "tipico", che di solito viene venduto dal produttore del microcontrollore. Contiene, oltre alla CPU, anche dispositivi di I/O tipici e software di base che è in grado di comunicare con un PC e di far esaminare al PC il contenuto dei suoi registri e delle sue locazioni di memoria.
Nel PC, con un cross compilatore, si svilupperà il software, che poi sarà scaricato già compilato nell'evalutation board, per la sua verifica.
Il microcontrollore della scheda eseguirà il programma, sotto il controllo del PC.
Sul PC gira il debugger che mostrerà all'utente la situazione della scheda. Il PC spedirà alla scheda i comandi che permettono le funzioni tipiche dei debugger, come l'esecuzione passo passo, lo stabilire od esaminare il contenuto di memoria e registri ed il piazzamento dei breakpoint.
In questo modo le fasi di sviluppo e le modifiche vengono realizzate tutte su PC. Solo quando il programma è pronto verrà trasferito su EPROM e messo nel sistema dedicato.

Emulatori in circuito (ICE)
Un emulatore in circuito (**ICE** In Circuit Emulator) ha tutte le caratteristiche di una scheda di valutazione, in più ne ha altre molto speciali, che lo rendono un sistema molto costoso.
La cosa più importante è che possiede uno speciale connettore, simile ad una proboscide, che si sostituisce al chip della CPU, nello stesso dispositivo che si deve provare.
L'emulatore quindi sta "al posto" della CPU, nel sistema che si sta realizzando, non è una scheda separata come la evaluation board. Questo permette di verificare anche l'hardware.
L'ICE è unico sistema tramite il quale è possibile mettere dei breakpoint legati a linee hardware. Si può per esempio interrompere l'esecuzione del programma ogni volta che una determinata linea del control "bus" viene modificata.
Inoltre l'emulatore contiene considerevoli quantità di memoria temporanea, chiamata "trace buffer". In questa memoria viene tenuta traccia delle ultime le istruzioni che il programma ha eseguito (il trace buffer può ricordare l'esecuzione di milioni di istruzioni).
Questo permette di capire cosa è successo quando si presenta un errore intermittente. Quando l'errore si evidenzia, a programma sospeso da un breakpoint, si può andare a cercare la causa dell'errore "nel passato", analizzando il contenuto del trace buffer ("playback" del buffer).

Analizzatore di stati logici
L'analizzatore di stati logici è un dispositivo che serve soprattutto agli addetti alla verifica ed al test hardware, ma può essere utile anche al programmatore, nel caso di problemi del software legati all'hardware.
Si tratta di un dispositivo che controlla a grande velocità lo stato di parecchie linee digitali e ne memorizza tutte le variazioni con un'indicazione del tempo in cui esse sono occorse. In questo modo è possibile ricostruire la storia dei valori assunti nel tempo da ogni piedino di ogni circuito digitale.

1.5 Classificazione del software secondo la distribuzione

A parte alcune eccezioni, tutto sommato rare, il software si paga. Il software è difficile da sviluppare (il mio lettore si renderà conto di persona quanto è difficile programmare!), richiede applicazione e competenza. Altrettanto difficile e costoso è "produrlo", cioè organizzare le condizioni perché sia realizzato bene e "distribuirlo", cioè farlo giungere all'utente finale. È giusto che chi lo sa fare venga per questo ricompensato.
Succede anche che alcuni programmatori, spesso molto bravi, mettano a disposizione il loro software gratis, per chiunque lo voglia usare. Solo quel software non si paga.
Di solito il software non viene mai venduto. Precisiamo meglio: non viene venduto il programma né il suo sorgente, ma piuttosto una sua "licenza" d'uso.

Il programma rimane di proprietà del produttore, che accorda, a pagamento, al suo cliente un permesso per usarlo (una "licenza" d'uso, appunto), limitato dalle condizioni scritte sul contratto di licenza. Il compratore deve sottostare a quelle condizioni se vuole che il contratto rimanga valido.
Se il contratto non è valido l'utente non ha diritto ad usare il programma e può incorrere nelle sanzioni previste dalla legge "norme per la tutela del diritto d'autore in campo informatico", e/o a quelle di altre leggi internazionali.

La Licenza è un contratto e, come tutti i contratti, stabilisce tutte le regole cui si devono attenere i contraenti. Su quelle regole si impegna chi usa il software ed anche, sebbene di solito con impegni non molto "forti", chi lo produce.
Immaginiamo che ci avvicini un tipo equivoco dicendo sottovoce "ho un motorino da venderti, ad un decimo del suo valore commerciale ..". Chi comprasse quel motorino saprebbe cosa gli è stato offerto ed avrebbe anche una idea ben precisa sulle qualità morali del venditore e del compratore.
La stessa cosa vale per il software. Anche se è molto più facile da rubare, ciò non vuol dire che sottrarlo senza pagarlo non sia un furto. Eppure il giudizio morale che ci si fa su un trafficante di software, anche su quelli che diventano ricchi con questi furti, è oggi molto più benevolo del giudizio che ci si fa su un ladro di polli!
C'è da dire che nella società postindustriale è divenuta sempre più importante la protezione della "proprietà intellettuale", perché i beni sono sempre più "smaterializzati" (il software è il caso più tipico, come dice anche il suo nome).
Se non si vuole un tracollo totale delle economie dei paesi più sviluppati la legislazione dovrà sempre più assimilare i beni materiali con quelli non materiali, per cui, tra le altre cose, sarà sempre più "pericoloso" copiare il software.
Dopo questo predicozzo, che speriamo utile, procediamo ora ad una classificazione del software secondo il tipo di licenza e di distribuzione commerciale.

1.5.1 Software commerciale tradizionale

E' prodotto da aziende che hanno lo scopo del profitto e lo perseguono con la produzione e commercializzazione di programmi per computer. Se queste aziende non facessero profitto non esisterebbero: né loro né i loro programmi.
La distribuzione del software commerciale avviene attraverso i normali canali del commercio: produttore, filiale nazionale, distributore, grossista, dettagliante, utente.
Le licenze del software commerciale sono del tipo "prima paghi poi provi", per cui l'utente deve farsi un'idea del software prima di acquistarlo, in genere consultando la letteratura tecnica. Le licenze prevedono che il programma non possa essere copiato, né distribuito in alcun modo, se non da alcuni "anelli" della rete di vendita del produttore.
La licenza d'uso del software commerciale può anche prevedere l'accesso al suo sorgente, ma in casi molto rari e sempre con costi molto elevati e con clausole molto restrittive sulle possibilità di modifiche da parte del cliente. Con queste licenze di solito le modifiche appartengono al venditore anche quando siano realizzate dal cliente.

1.5.2 Software shareware e "trialware"

Questo tipo di software è commerciale, nel senso che è stato scritto per ricavarne un compenso economico, ma la sua distribuzione è più libera e permette di provare prima di acquistare.
Lo shareware, come implica il suo nome che viene dal verbo "to share" ("condividere"), può essere distribuito liberamente da chiunque non lo modifichi in alcun modo e lo distribuisca integralmente.
Esso può essere anche usato liberamente solo per il periodo di tempo indicato nella licenza. Dunque il programma può essere provato prima di acquistarlo e lo si può comprare solo se serve e piace. Alla fine del periodo di prova l'utente, se non procede all'acquisto, ha l'obbligo morale di disinstallare il prodotto. Molti shareware sono limitati nel tempo o nelle funzioni e possono funzionare pienamente solo quando si effettui la loro "registrazione" attraverso un codice che ne certifica l'acquisto.
Quindi lo shareware non è software gratuito, ma si paga! Di solito si paga meno del software commerciale tradizionale, perché il prezzo non raddoppia ad ogni passaggio nella catena di distribuzione, e perché non è necessario fare grosse campagne pubblicitarie che spieghino com'è il prodotto. Infatti il programma si spiega da sé, provandolo!
La distribuzione dello shareware non avviene attraverso i canali "classici", che terminano al negozio, ma attraverso Internet e le riviste di informatica.
Di solito si intende per "trial software" o "trialware" il software distribuito per gli stessi canali dello shareware, ma che "scade" e dopo un certo periodo d'uso non è più utilizzabile. A quel punto l'utente deve acquistare il prodotto, od uncodice di "sblocco", attraverso la catena commerciale o in un "Internet shop".

1.5.3 Software gratis (freeware)

Di solito si definisce "freeware" quel software per il quale l'autore non richiede alcun compenso e che è liberamente distribuibile. Del software "freeware" l'autore di solito non distribuisce il sorgente, riservandosi il diritto di modificarlo solo di persona e di cambiare il tipo di licenza. Molti autori rendono gratuito il loro lavoro solo per scopi non commerciali e chiedono di essere contattati in caso di distribuzione commerciale.
Anche in questo caso è comunque il documento di licenza che spiega quali sono i limiti entro i quali quel software è gratuito.

In taluni casi le aziende distribuiscono come freeware vecchie versioni dei loro programmi, per invogliare gli utenti a passare alle ultime versioni a pagamento.

1.5.4 Software "open source"

Si definisce "open source" il software che viene distribuito insieme al suo sorgente. Le licenze che regolano il software open source sono molto diverse, ed esiste anche una definizione "legale" del termine, che è stato registrato come marchio commerciale allo scopo di mantenerlo libero da ambiguità. Le licenze "open source" possono essere più o meno "aperte" e richiedere impegni più o meno gravosi a chi utilizza e/o modifica il programma.
Il nodo centrale delle varie licenze open source è la proprietà dei sorgenti, che può rimanere all'autore (licenze commerciali a sorgente aperto, che non rispondono al termine "open source" legale), può essere vincolata alla distribuzione libera del programma (libre software) o può essere del tutto condivisa (pubblico dominio).

1.5.5 "Free software" (software libero o "libre software")

l concetto di free software si è sviluppato intorno a Richard Stallman ed alla sua "Free Software Foundation" (FSF, http://www.fsf.org), che ha reso disponibili una quantità impressionante di strumenti di sviluppo ed applicativi a tutti gli sviluppatori ed agli utenti interessati. Questi programmi, raccolti sotto il nome di "progetto GNU", sono distribuiti con la GNU GPL ("General Public License") detta anche "copyleft" (vedi: http://www.fsf.org/copyleft/gpl.html).
La licenza GPL, che definisce il free software, è la più usata delle tante licenze Open Source.
Il free software ha distribuzione **libera**. Infatti il nome free in questo caso significa "libero", non necessariamente "gratis".
Dato che la parola "free" in Inglese significa sia "libero" che "gratis" essa è ambigua ed in Europa si preferisce chiamare questi programmi "libre software", in Italia "software libero".
Chi ottiene un software libero, difeso dalla GPL, non ottiene su di esso alcuna garanzia ed ha il diritto di usarlo, di modificarlo e di distribuirlo come meglio crede, anche vendendolo al prezzo che vuole.
Ciò che **non** può fare è renderlo "chiuso", cioè incorporarlo in un proprio programma senza rendere pubblici i sorgenti delle proprie modifiche.
La GPL garantisce la "libertà" di un software perché obbliga chiunque lo distribuisca, in qualsiasi forma, ad includere nella distribuzione anche il sorgente, **comprensivo delle modifiche** che ha apportato.
Inoltre non si possono includere **porzioni** di free software in programmi commerciali che non includano anche il sorgente.
I programmi scritti interamente da un soggetto, ma che usano componenti di software libero, come per esempio un Sistema Operativo, possono essere distribuiti commercialmente e con licenze chiuse, anche senza sorgente.
Questo modo di intendere il software "libero" ha creato una grande comunità virtuale di sviluppatori, che prende forma su Internet.
Con la GPL vengono distribuiti gli strumenti di sviluppo GNU, praticamente per tutti i linguaggi esistenti, il Sistema Operativo Linux, il Web server per Internet Apache, il programma per la grafica ed il fotoritocco GIMP e molti applicativi di vario tipo.
E' chiaro che il free software non costerà mai molto. Dato che la distribuzione è libera, se qualcuno chiede grosse cifre per un programma che un altro potrebbe distribuire gratis o per molto poco, riuscirà a venderlo solo a chi si fida molto di lui, cioè in definitiva non venderà il prodotto, ma un servizio di consulenza e di assistenza al cliente, ed eventualmente la garanzia, se vorrà offrirla.
E' importante far rilevare che il software libero **ha** un "copyright" che è specificato legalmente dalla GPL e che rimane prerogativa dell'autore originario. Se l'autore originario accerta che il suo software libero è stato "chiuso" da qualcuno può intentargli causa legale. La FSF lo potrà assistere nel processo. Se l'autore originario ha rilasciato il suo programma sotto la GPL non può più "chiuderlo" e reclamarne la proprietà esclusiva.

1.5.6 Software di pubblico dominio

Di solito si definisce di "pubblico dominio" quel software per il quale l'autore rinuncia ad esercitare ogni diritto di licenza, non richiede alcun compenso e rende il software liberamente distribuibile ed incorporabile in programmi proprietari. In questo caso dunque, gli Autori "regalano" il loro lavoro anche a chi lo distribuisce per scopi commerciali.
La famosa licenza Unix BSD (Berkeley Software Distribution), che è Open Source e sotto la quale vengono distribuiti i Sistemi Operativi FreeBSD, OpenBSD e NetBSD ha caratteristiche che si possono definire di "pubblico dominio".
Chi fa modifiche ai software con licenza BSD può distribuire o meno il sorgente delle modifiche che ha realizzato ed anche cambiarne la licenza.

Il termine " freeware"
Da notare che questo termine viene usato oggi in modo ambiguo, in alcuni casi per intendere il software gratis o di pubblico dominio, in altri casi per intendere il free software.