Capitolo 1

Introduzione al linguaggio C

Il linguaggio C è ancora oggi, nonostante siano passati all'incirca quarant'anni dalla sua prima apparizione, uno straordinario linguaggio di programmazione; ricco, espressivo, potente e flessibile. Esso ha influenzato moltissimi altri linguaggi di programmazione che hanno preso in "prestito" molti aspetti della sua filosofia, della sua sintassi e dei suoi costrutti principali. Tra questi è sufficiente citare alcuni dei maggiori linguaggi *mainstream* come Java, C# e, naturalmente, C++ che, per certi versi è un C "potenziato" con l'astrazione della programmazione orientata agli oggetti e della programmazione generica attraverso il meccanismo dei *template*.

NOTA

Anche se C++ è spesso definito come un *superset* del linguaggio C, è comunque un linguaggio "diverso" che ha sia delle caratteristiche aggiuntive (per esempio, fornisce dei costrutti propri della programmazione a oggetti) sia, per alcuni costrutti, delle regole diverse (per esempio, non consente di assegnare un puntatore a `void` a un puntatore a un altro tipo senza un opportuno cast). Tuttavia, imparare C permetterà di padroneggiare C++ almeno nei suoi aspetti essenziali e nei suoi costrutti basici; infatti, con la dovuta attenzione e senza l'utilizzo delle astrazioni proprie di C++, un programma scritto in C potrà compilarsi senza problemi con un compilatore C++.

In questo capitolo

- **Storia della nascita di un "mito"**
- **Caratteristiche**
- **Cenni sull'architettura di un elaboratore**
- **Paradigmi di programmazione**
- **Concetti introduttivi allo sviluppo**
- **Il primo programma**
- **Compilazione ed esecuzione del codice**
- **Problemi di compilazione ed esecuzione?**

Allo stesso tempo, però, tali linguaggi hanno anche "eliminato" alcuni aspetti di C che sono complessi, a basso livello e richiedono una certa attenzione e conoscenza come per esempio i *puntatori* che sono, in breve, un potente meccanismo attraverso il quale è possibile accedere in modo diretto alla memoria per leggerla e manipolarla.
In buona sostanza linguaggi come Java, C# e così via, se è vero che da un lato hanno reso la scrittura del software più "controllata" e "sicura" è anche vero che, dall'altro lato, l'hanno resa maggiormente vincolata e meno permissiva in termini di libertà operativa ed espressiva. In definitiva, il programmatore ha sì meno margini di errore, ma ha anche più lacci e vincoli.
In ogni caso, giova da subito dire che è proprio la totale libertà di azione offerta dal linguaggio C che ne rappresenta la sua grande forza e, perché no, lo straordinario fascino che continua a riscuotere sui programmatori sia alle prime armi sia in quelli più esperti. In sostanza C dà grande responsabilità al programmatore e assume che questi sappia cosa sta facendo e sia consapevole degli effetti delle sue azioni.
C, come detto, malgrado l'avanzata di altri linguaggi di programmazione, è, in modo netto e assoluto, uno straordinario e moderno strumento di programmazione con il quale è possibile scrivere programmi per i più svariati dispositivi hardware, da quelli altamente performanti e multiprocessore a quelli *embedded* dove, in taluni casi, le risorse di memoria e la potenza della CPU sono limitate.

TERMINOLOGIA

Per sistema *embedded* si intende un sistema elettronico che ha del software e dell'hardware "incorporati" deputati a eseguire in modo esclusivo un task o una serie di task per i quali il sistema è stato ideato e progettato. In pratica, è pensabile come un sistema hardware/software dedicato e non general-purpose come i comuni PC, che può essere indipendente oppure parte di un sistema più grande. Un sistema embedded ha, generalmente, i seguenti principali componenti incorporati: dell'hardware simile a quello di un computer; il software dell'applicazione principale; un sistema operativo real time (RTOS). In più si caratterizza anche per dei limiti o vincoli che possono influire sulla scelta della progettazione e del design, quali la quantità di memoria e la potenza del processore disponibili, le dimensioni richieste, i costi di produzione e così via. I sistemi embedded trovano applicazione in svariati settori, come quelli delle telecomunicazioni e del networking (router, firewall...), della medicina, dei satelliti, delle automobili (*motor control system*, *cruise control*...), dell'elettronica digitale di consumo (*set top box*, fotocamere digitali, lettori MP3...) e via dicendo.

C è, in definitiva, un importante e potente linguaggio di programmazione e ciò lo dimostra anche il fatto che la sua conoscenza è ancora uno *skill* molto richiesto nel mondo del lavoro e che la sua popolarità è ancora ai primi posti, come dimostrato, per esempio, dal famoso indicatore di popolarità TIOBE (Figura 1.1) aggiornato a gennaio del 2015.

DETTAGLIO

Il *TIOBE Programming Community index* è un indicatore che misura la popolarità di un linguaggio di programmazione *Turing completo*. Esso è aggiornato mensilmente in base al numero dei risultati delle ricerche effettuate con 25 search engine di una query avente il seguente pattern: +"<language> programming".

Jan 2015	Jan 2014	Change	Programming Language	Ratings	Change
1	1		C	16.703%	-1.24%
2	2		Java	15.528%	-1.00%
3	3		Objective-C	6.953%	-4.14%
4	4		C++	6.705%	-0.86%
5	5		C#	5.045%	-0.80%
6	6		PHP	3.784%	-0.82%
7	9	^	JavaScript	3.274%	+1.70%
8	8		Python	2.613%	+0.24%
9	13	^^	Perl	2.256%	+1.33%
10	17	^^	PL/SQL	2.014%	+1.38%
11	15	^^	MATLAB	1.390%	+0.62%
12	26	^^	ABAP	1.273%	+0.80%
13	27	^^	COBOL	1.267%	+0.81%
14	24	^^	Assembly	1.171%	+0.68%
15	12	v	Ruby	1.130%	+0.07%
16	11	vv	Visual Basic .NET	1.074%	-0.48%
17	-	^^	Visual Basic	1.074%	+1.07%
18	44	^^	R	1.042%	+0.79%
19	10	vv	Transact-SQL	0.874%	-0.68%
20	20		Delphi/Object Pascal	0.837%	+0.24%

Figura 1.1 Tabella risultati TIOBE.

Storia della nascita di un "mito"

Il linguaggio C, come oggi lo conosciamo, è frutto di un'affascinante storia che inizia diversi decenni or sono durante uno *startup* temporale (siamo all'incirca negli anni Settanta) di grandi innovazioni tecniche soprattutto nel campo dei computer (per esempio, il 15 novembre del 1971 Intel svela la nascita del primo microprocessore, con ciò sancendo il passaggio dalla terza generazione di computer, quella dei mainframe e dei minicomputer, alla quarta generazione di computer, quella dei microcomputer e dei personal computer), dei sistemi operativi (per esempio, nel 1971 nasce la First Edition di Unix) e del networking (per esempio, nel 1973 furono attivate la prime due connessioni internazionali di ARPANET, progenitrice di quella che sarebbe poi diventata Internet, rispettivamente in Inghilterra e in Norvegia).
Tutto iniziò verso la fine del 1960, quando presso i Bell Telephone Laboratories, famoso centro di ricerca e sviluppo operante nel settore dell'information technology dove sono

state sviluppate tante straordinarie tecnologie (transistor, laser, il sistema operativo Unix e così via), decisero di abbandonare il progetto del sistema operativo in time-sharing denominato Multics (*Multiplexed Information and Computing Service)*, avviato congiuntamente al MIT (*Massachusetts Institute of Technology*) e a GE (*General Electric)*, ritenendo che le idee da implementare non fossero praticabili in tempi relativamente brevi, oltre a essere piuttosto dispendiose in termini economici.
Al progetto del Multics, tra gli altri, lavorava un giovane scienziato informatico, Kenneth Thompson, il quale, dopo l'abbandono del progetto medesimo, iniziò a nutrire ambizioni per la creazione di un nuovo sistema operativo che avrebbe dovuto incorporare gli aspetti più innovativi di Multics e, nel contempo, eliminare quelli più problematici.
Iniziò, così, nel 1968, su un mainframe GE-635 e su un minicomputer DEC PDP-7, lo sviluppo, in linguaggio assembly, del sistema operativo che sarebbe divenuto Unix e che avrebbe visto poi la partecipazione anche di straordinari *hacker* del codice quali Dennis MacAlistair Ritchie, Malcolm Douglas McIlroy, Joe Ossanna e Rudd Canaday.
Nel 1969, si ebbe il *first run* di Unics, ortografia del nome scelto in origine prima di Unix su suggerimento di Brian Wilson Kernighan e come gioco di parole rispetto al sistema Multics (*Multiplexed Information and Computing Service*). Unics era l'acronimo di *UNiplexed Information and Computing Service*.
Ken Thompson aveva, tuttavia, anche un'altra ambizione: desiderava infatti creare un linguaggio di programmazione ad alto livello per quello che poi si sarebbe chiamato Unix, sollecitato anche dagli sforzi di altri sviluppatori come Douglas McIlroy, che aveva partecipato alla scrittura di un compilatore per il linguaggio PL/I (*Programming Language One*) per il sistema Multics.
Sicché ideò e sviluppò un nuovo linguaggio di programmazione denominato B, basato a sua volta sul linguaggio di programmazione BCPL (*Basic Combined Programming Language)* ideato nel 1966 da Martin Richards, e il cui nome, secondo altre ipotesi, potrebbe non essere una contrazione di BCPL bensì una derivazione di un altro linguaggio di programmazione, Bon, creato dallo stesso Thompson durante la sua collaborazione al progetto Multics.
Dopo lo sviluppo del linguaggio B, sul sistema Unix del PDP-7, furono scritte con esso solo poche utility e fu scartata l'idea di riscrivere Unix stesso in B a causa della scarsa potenza ed efficienza computazionale del PDP-7.
Si arriva così al 1970, quando i Bell Labs acquistarono un minicomputer a 16 bit DEC PDP-11 per il progetto Unix, sicuramente più potente e avanzato del PDP-7, che fu subito impiegato da Thompson per riscrivere, sempre in assembly, il kernel Unix e altre utility di base. Sullo stesso PDP-11 fu poi effettuato anche il porting del linguaggio B.
Tuttavia, sul PDP-11, il linguaggio B presentava delle problematiche e inadeguatezze dovute a scelte progettuali e di design che erano anche strettamente legate all'hardware del PDP-7. Infatti, prima di tutto B era *typeless* o, per meglio dire, aveva un solo tipo di dato definito come *word* o *cell* e si adattava bene per il PDP-7, che era una macchina *word-addressable*. Il PDP-11, di converso, era una macchina *byte-addressable*, e ciò causava un overhead con i puntatori poiché per B un puntatore era una variabile che poteva contenere un indirizzo di memoria rispetto a un array di indirizzi di memoria disposti secondo un ordine di tipo word addressing; pertanto a run-time un puntatore doveva essere "convertito" per contenere il byte address atteso dall'hardware del PDP-11.

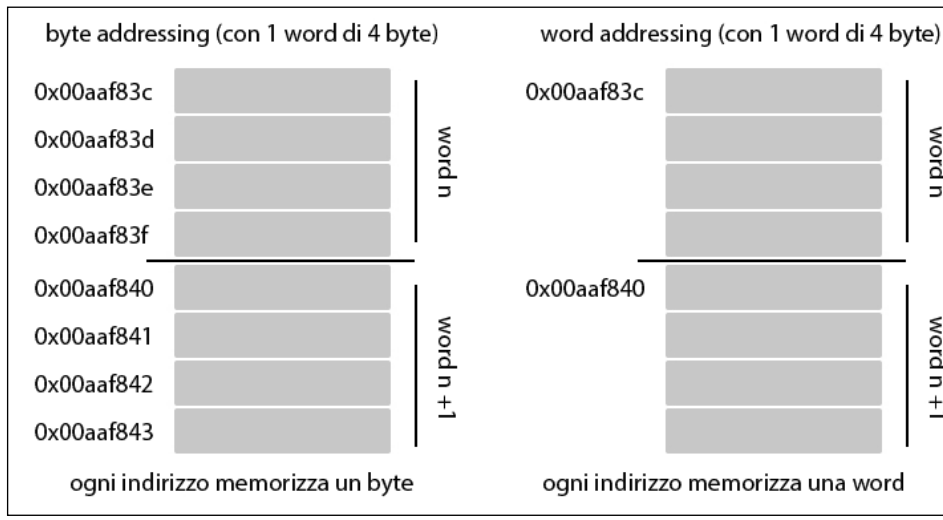

Figura 1.2 Differenza tra byte addressing e word addressing.

In secondo luogo, il linguaggio B era lento: un sorgente scritto con esso, infatti, non produceva direttamente codice eseguibile; da questo punto di vista B era un linguaggio interpretato che veniva "tradotto" a run-time.
Interviene così, nel 1971, Dennis Ritchie, il quale iniziò a lavorare sul linguaggio B, estendendolo con l'aggiunta dei tipi `int` e `char`, con gli array e i puntatori a quei tipi. In più i valori dei puntatori contenevano ora indirizzi di memoria misurati in byte, e l'accesso per *indirezione* al valore della cella di memoria referenziata non implicava alcun overhead per scalare il puntatore da un word offset a un byte offset.
Produsse inoltre un compilatore in grado di generare direttamente codice in linguaggio macchina nativo del PDP-11, rendendo così i programmi prodotti efficienti, compatti e veloci quasi quanto quelli prodotti con il linguaggio assembly.
Inizialmente decise di denominare questa versione estesa di B come NB, che stava per *New B*; successivamente, quando il type system e il compilatore furono pronti, scelse di cambiarne il nome in C (la motivazione di questa scelta di denominazione è stata lasciata dallo stesso Ritchie ambigua: egli, infatti, non ha mai specificato se tale singola lettera è una semplice progressione alfabetica rispetto a B oppure rispetto a BCPL).
Tra il 1972 e il 1973 vi furono altre importanti aggiunte sia strettamente legate al linguaggio C, come il preprocessore e le strutture, sia dal punto di vista delle librerie, come la scrittura da parte di Michael Lesk di una serie di funzionalità di I/O portabili.
In più, durante questo periodo, accaddero altre due cose di notevole importanza: la prima, riferita alla riscrittura in C del sistema Unix sul PDP-11 (nell'estate del 1973); la seconda riferita al porting di C su altri sistemi come l'Honeywell 635 e l'IBM 360/370. L'importanza del primo aspetto si commenta da sola: nasceva, infatti, l'edizione di Unix che avrebbe cambiato la storia dei sistemi operativi e lo avrebbe reso "immortale", e ciò soprattutto grazie alla sua stretta integrazione con il linguaggio C. Il secondo aspetto è altresì importante perché, di fatto, apriva la possibilità di estendere Unix su altri sistemi che non fossero solo i PDP-11 estendendo la penetrazione sia di tale sistema operativo sia del linguaggio C. Esempi di quanto detto si hanno, quando, all'incirca tra il 1977 e il

1978, Ritchie, Thompson e Stephen Curtis Johnson, quest'ultimo autore dell'importante compilatore portabile pcc (*portable C compiler*), iniziarono il porting di Unix sul minicomputer Interdata 8/32 e poi Tom London e John Reiser su un sistema DEC VAX 11/780. Tra il 1973 e il 1980 il linguaggio C ottenne altre importanti aggiunte, come il tipo `long`, le `union`, le *enumerazioni* e così via, mentre nel 1978 uscì *The C Programming Language*, il primo libro su C (detto il K&R o il "White Book"), a opera di Brian Kernighan e Ritchie stesso, che divenne una sorta di guida al linguaggio utilizzata come standard *de facto* per gli implementatori dei compilatori. Questo testo fondamentale, tuttavia, portò alla luce anche dei rilevanti problemi legati a incompatibilità tra i compilatori su alcune caratteristiche che erano state scarsamente dettagliate. A questo si aggiunse il fatto che dopo la pubblicazione ci furono modifiche al linguaggio che il testo non descriveva; pertanto si iniziò ad avvertire l'esigenza di formalizzare C con uno standard ufficiale che desse conto della crescente importanza del linguaggio.
Nel 1983 l'ANSI (*American National Standards Institute*), un'organizzazione che produce standard industriali per gli Stati Uniti, istituì il comitato X3J11 con lo scopo di produrre uno standard per C con questo preciso e ambizioso obiettivo:

> to develop a clear, consistent, and unambiguous Standard for the C programming language which codifies the common, existing definition of C and which promotes the portability of user programs across C language environments.

I lavori sullo standard terminarono nel dicembre del 1989 e produssero la specifica X3.159-1989, riferita anche come C89 o ANSI C, che standardizzava, per l'appunto, sia il core del linguaggio C sia una serie di funzionalità o API (*Application Programming Interface*) che andavano a costituire il core di una libreria di base.
Questo standard fu poi, nel 1990, approvato e ratificato dall'ISO (*International Organization for Standardization*) un'organizzazione indipendente non governativa, composta da membri provenienti da oltre 160 paesi, che produce standard internazionali per svariati settori (agricoltura, informatica, sanità e così via). Il documento di specifica prodotto dall'ISO fu denominato ISO/IEC 9899:1990 (è riferito anche come C90 o ISO C, e rispetto alla specifica C89 non ha alcuna differenza se non per la diversa formattazione). Successivamente, l'ISO diffuse documenti con correzioni ed emendamenti denominati rispettivamente ISO/IEC 9899/COR1:1994, ISO/IEC 9899/AMD1:1995 (questo riferito anche come C95) e ISO/IEC 9899/COR2:1996, cui seguì nel 1999 la pubblicazione di un nuovo documento di specifica del linguaggio C, denominato ISO/IEC 9899:1999 e riferito come C99, che introdusse molte importanti aggiunte al linguaggio sia nel core (*variable-length arrays*, *flexible array members*, *restricted pointers*, *compound literals*, *designated initializers*, *inline functions*, *extended identifiers* e così via) sia nella libreria standard (header `<complex.h>`, `<stdbool.h>`, `<tgmath.h>` e così via).
Seguirono, quindi, negli anni successivi i documenti correttivi ISO/IEC 9899:1999/Cor 1:2001, ISO/IEC 9899:1999/Cor 2:2004 e ISO/IEC 9899:1999/Cor 3:2007, che culminarono nella creazione del nuovo e attuale standard denominato ISO/IEC 9899:2011 e riferito come C11, che ha apportato ulteriori migliorie al linguaggio (*anonymous structures*, *anonymous unions*, *type-generic expressions* e così via) e alla libreria standard attraverso, soprattutto, gli header `<stdatomic.h>` e `<threads.h>` per il supporto alla programmazione concorrente.

C99 e C11

Lo standard C99 ha indubbiamente introdotto innumerevoli modifiche al linguaggio C che l'hanno migliorato in molti aspetti e, per certi versi, l'hanno "avvicinato" a C++ con l'aggiunta di caratteristiche come le funzioni *inline*, i commenti a singola riga tramite i caratteri `//`, le dichiarazioni di variabili inseribili in qualsiasi punto di un blocco di codice e così via. Tuttavia, il supporto da parte dei vendor dei compilatori alle nuove caratteristiche è stato molto lento, e a oggi vi sono ancora alcuni di essi che le hanno implementate solo parzialmente. Ecco, dunque, che nel documento di specifica C11, per rendere meno difficoltoso l'adozione di tale standard, sono state esplicitate alcune caratteristiche del linguaggio che i vendor dei compilatori possono implementare solo se lo desiderano o se lo ritengono essenziale per il loro ambiente target (in pratica, il supporto ad alcune caratteristiche come, per esempio, quelle relativa al multithreading o ai numeri complessi sono opzionali). In ogni caso devono essere implementate delle macro come `__STDC_NO_VLA__`, `__STDC_NO_COMPLEX__`, `__STDC_NO_THREADS__` e così via, che consentono di verificare se quella particolare feature è supportata o meno. Per esempio, se la macro `__STDC_NO_THREADS__` ha come valore `1`, ciò indicherà che l'attuale compilatore non supporta l'header `<threads.h>` e, dunque, la programmazione concorrente.

Caratteristiche

C, come in più parti già evidenziato, è uno straordinario e moderno linguaggio di programmazione contraddistinto dalle seguenti caratteristiche.

- *Efficienza*. Come in precedenza detto, C è stato ideato, tra le altre cose, anche per soppiantare l'assembly come linguaggio di programmazione *a basso livello* per lo sviluppo del sistema operativo Unix. Esso, infatti, tende a produrre codice compatto e con elevata velocità di esecuzione, quasi quanto quella di codice prodotto da un assemblatore, che ben si adatta al sistema hardware per il quale il relativo compilatore è stato progettato e implementato.
- *Portabilità*. C fa della portabilità uno dei suoi massimi punti di forza. Il linguaggio è portabile perché un sorgente scritto secondo lo standard C e con la libreria di funzioni standard può essere compilato senza particolari problemi su altre piattaforme hardware che hanno il relativo compilatore. C, infatti, si è talmente diffuso che è possibile trovare compilatori per i più svariati dispositivi hardware, dai sistemi embedded con poca memoria e poca capacità di calcolo ai sofisticati e potenti supercomputer.
- *Potenza*. C è tra i pochi linguaggi di programmazione che consentono di manifestare una reale potenza, sia espressiva, grazie a pochi e mirati costrutti sintattici, sia algoritmica, grazie alla presenza di determinati operatori che permettono l'accesso e la manipolazione dei singoli bit della memoria.
- *Flessibilità*. A oggi non esiste praticamente alcun dominio applicativo che C non possa coprire. Esso è, infatti, utilizzato per sviluppare qualsiasi tipologia di software, come compilatori, sistemi operativi, driver hardware, motori per grafica 3D, editor di testi, player musicali, videogame e così via. Da questo punto di vista C si è "evoluto" da mero linguaggio di programmazione di sistema qual era *ab origine* a vero e proprio linguaggio di programmazione general-purpose.
- *Permissività*. Rispetto ad altri linguaggi di programmazione, C è stato pensato per essere un linguaggio per programmatori e pertanto lascia agli stessi una grande libertà

di costruzione dei programmi con gli strumenti offerti (si pensi ai puntatori) senza fornire, quindi, soprattutto in fase di compilazione, una *cattura* e un rilevamento di determinati tipi di errori. In pratica C dà piena fiducia e grande responsabilità al programmatore; presuppone che lo stesso, se sta facendo una cosa, sappia a cosa va incontro e sia in grado di valutarne gli effetti.

NOTA

I compilatori moderni sono oggi in grado di effettuare un check del sorgente alla ricerca sia dei comuni errori in violazione delle regole sintattiche sia di altri tipi di errori, evidenziati tramite dei messaggi di diagnostica o *warning*, che nonostante non producano una mancata compilazione del sorgente potrebbero essere fonte di problemi durante l'esecuzione del programma. Per esempio, con GCC possiamo usare l'opzione `-Warray-bounds` per ottenere un warning se accediamo a un indice di un array fuori dal suoi limiti, `-Waddress` per ottenere un warning per utilizzi sospetti degli indirizzi di memoria, `-Wreturn-type` per ottenere un warning se, per esempio, in una funzione con un tipo di ritorno non `void` si scrive un'istruzione di `return` senza un valore di ritorno o si omette di scriverla e così via per altri warning.

Tra le altre caratteristiche di C, possiamo altresì evidenziare che esso è considerabile come un linguaggio di programmazione di *medio livello* perché mette a disposizione del programmatore sia facility di *alto livello* (si pensi al costrutto di *funzione* o ad altre astrazioni come il costrutto di *struttura*), che garantiscono una maggiore efficienza e flessibilità nella costruzione dei programmi, sia facility di *basso livello* (si pensi al costrutto di *puntatore*) che garantiscono, al pari dei linguaggi *machine-oriented* come l'assembly, una maggiore efficienza nello sfruttamento diretto dell'hardware sottostante.

TERMINOLOGIA

Un linguaggio di programmazione è definibile di alto livello se offre un alto livello di astrazione e indipendenza rispetto ai dettagli hardware di un elaboratore. Ciò implica che un programmatore utilizzerà keyword e costrutti sintattici di facile comprensione che gli permetteranno di scrivere il programma in modo relativamente semplificato con la possibilità di concentrarsi solo sulla logica dell'algoritmo da implementare. Per questo sono definibili come linguaggi *closer to humans*. Di converso, un linguaggio di programmazione è definibile di basso livello quando non offre alcun layer di intermediazione/astrazione rispetto all'hardware da programmare, e il programmatore deve non solo avere una profonda conoscenza di tale hardware ma deve anche lavorare direttamente con esso (registri, memoria e così via). Per questo sono definibili come linguaggi *closer to computers*.

Infine, C è un linguaggio ricco di funzionalità, ovvero ha un set di API predefinite, fornite tramite la sua libreria standard, che vanno dalla gestione dell'input/output e delle stringhe alla gestione della memoria, delle date e degli orari, degli errori e così via.

Cenni sull'architettura di un elaboratore

Prima di addentrarci nello studio del linguaggio C appare opportuno delineare alcuni concetti teorici che riguardano la struttura di un generico calcolatore elettronico, soffermandoci in modo più approfondito su due componenti in particolare, ossia la CPU

e la memoria centrale. Ciò si rende necessario perché C consente, tra le altre cose, di sfruttare a basso livello l'hardware di un sistema target. Pertanto, comprendere, seppure a grandi linee, come un computer è progettato e costituito può sicuramente aiutare a scrivere programmi che *dialogano* con l'hardware sottostante con maggiore consapevolezza dei loro effetti (capire, per esempio, come è strutturata la memoria centrale può aiutare a gestirla in modo più sicuro ed efficiente).

Il modello di von Neumann

I linguaggi imperativi, di cui C è un esponente, come vedremo meglio poi, condividono un modello computazionale che rappresenta un'astrazione del sottostante calcolatore elettronico dove, in breve, la computazione procede modificando valori memorizzati in locazioni di memoria. Questo modello è definito come *von Neumann architecture* dal nome dello scienziato ungherese John von Neumann che, nel 1945, lo ideò, e fu, ed è ancora, alla base del design e della costruzione dei computer e quindi dei linguaggi imperativi che vi si rifanno e che sono delle astrazioni della macchina di von Neumann. Nella sostanza, nel modello di von Neumann (Figura 1.3) un computer è costituito da una CPU (*Central Processing Unit*) per il controllo e l'esecuzione dell'elaborazione, al cui interno si trovano l'ALU (*Arithmetic Logic Unit*) e una *Control Unit*; da celle di memoria identificate da un indirizzo numerico atte a ospitare i dati coinvolti nell'elaborazione; da dispositivi per l'input e l'output dei dati da e verso l'elaboratore; da un bus di comunicazione tra le varie parti per il transito di dati, indirizzi e segnali di controllo. In più, sia i dati sia le istruzioni di programmazione sono memorizzati nella memoria. In pratica, nel modello di von Neumann abbiamo due elementi caratterizzanti: la memoria, che memorizza le informazioni anzidette, e il processore, che fornisce operazioni per modificarne il contenuto, ossia lo stato.
In definitiva, un computer digitale modellato secondo l'architettura di von Neumann non è altro che un sistema di componenti quali processori, memorie, device di input/output e così via tra di loro interconnessi (*bus oriented*) che cooperano congiuntamente al fine di svolgere i processi computazionali per i quali sono stati progettati.

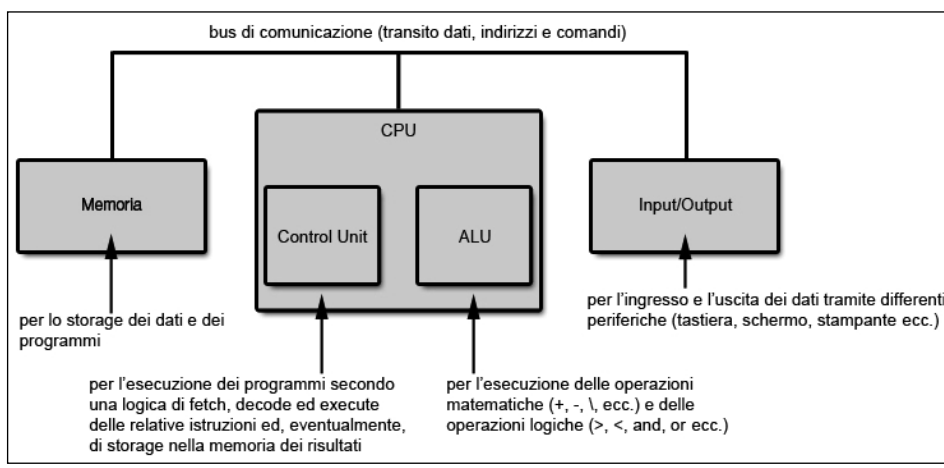

Figura 1.3 Architettura semplificata di un computer basata sul modello di von Neumann.

La CPU

La CPU (*Central Processing Unit*) è il "cervello" di ogni computer ed è deputata principalmente a interpretare ed eseguire le istruzioni elementari che rappresentano i programmi e a effettuare operazioni di coordinamento tra le varie parti di un computer.
Questa unità di processing, dal punto di vista fisico, è un circuito elettronico formato da un elevato numero di transistor (varia da diverse centinaia di milioni fino ai miliardi delle più potenti attuali CPU) che sono ubicati in un circuito integrato (chip) di ridotte dimensioni (pochi centimetri quadrati).
Dal punto di vista logico, invece, una CPU è formata dalle seguenti unità (Figura 1.4).

- ALU (*Arithmetic Logic Unit*), ossia l'unità aritmetico-logica. Quest'unità è costituita da un insieme di circuiti deputati a effettuare calcoli aritmetici (addizioni, sottrazioni e così via) e operazioni logiche (boolean AND, boolean OR e così via) sui dati. Il suo funzionamento si può così sintetizzare ponendo, per esempio, un'operazione di somma tra due numeri: preleva da appositi registri di memoria di input gli operandi trasmessi (diciamo `X` e `Y`); effettua l'operazione di addizione (`X + Y`); scrive il risultato della somma in un registro di memoria di output. In pratica possiamo vedere una ALU come una sorta di calcolatrice "primitiva" che riceve dati sui quali effettuare calcoli al fine di produrre un risultato.
- *Control Unit*, ossia l'unità di controllo. Questa unità coordina e dirige le varie parti di un calcolatore in modo da consentire la corretta esecuzione dei programmi. In pratica essa, in una sorta di ciclo infinito, preleva (*fetch*), decodifica (*decode*) ed esegue (*execute*) le istruzioni dei programmi. Attraverso la fase di prelievo acquisisce un'istruzione dalla memoria, la carica nel registro istruzione e rileva la successiva istruzione da prelevare. Attraverso la fase di decodifica interpreta l'istruzione corrente da eseguire. Attraverso la fase di esecuzione esegue ciò che l'istruzione indica: è un'azione di input? Allora comanda l'unità di input relativa al trasferimento dei dati nella memoria centrale. È un'azione di output? Allora comanda l'unità di output relativa al trasferimento dei dati dalla memoria centrale verso l'esterno. È un'azione di processing dei dati? Allora comanda il trasferimento dei dati nell'ALU, comanda l'ALU alla loro elaborazione e al trasferimento del risultato nella memoria centrale. È un'azione di salto? Allora aggiorna il registro *contatore di programma* con l'indirizzo cui saltare. Infine, le operazioni svolte dall'unità di controllo sono regolate da un orologio interno di sistema (*system clock*) che genera segnali o impulsi regolari a una certa frequenza, espressa in Hertz, che consentono alle varie parti di operare in modo coordinato e sincronizzato. Maggiori sono questi segnali per secondo, detti anche *cicli di clock*, maggiore è la quantità di istruzioni per secondo che una CPU può processare e quindi la sua velocità.
- *Registers*, ossia i registri. I registri sono unità di memoria, estremamente veloci (possono essere letti e scritti ad alta velocità perché sono interni alla CPU) e di una certa dimensione, utilizzati per specifiche funzionalità. Vi sono numerosi registri, tra cui quelli più importanti sono: l'*Instruction Register* (registro istruzione), che contiene la corrente istruzione che si sta eseguendo; il *Program Counter* (contatore di programma), che contiene l'indirizzo della successiva istruzione da eseguire; gli *Accumulator* (accumulatori), che contengono, temporaneamente, gli operandi di un'istruzione e alla fine della computazione il risultato dell'operazione eseguita dall'ALU.

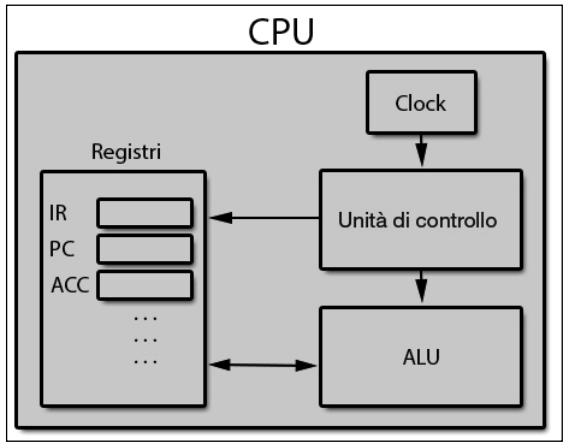

Figura 1.4 Vista logica dell'interno di una CPU.

La memoria centrale

La memoria centrale, detta anche primaria o principale, è quella parte del computer dove sono memorizzate le istruzioni e i dati dei programmi.
Essa ha diverse caratteristiche: è *volatile* perché il contenuto è perso nel momento in cui il relativo calcolatore è spento; è *veloce*, ossia il suo accesso in lettura/scrittura può avvenire in tempi estremamente ridotti e minimi; è ad *accesso casuale* perché il tempo di accesso alle relative celle è indipendente dalla loro posizione e dunque costante per tutte le celle (da questo punto di vista è definita come RAM, che sta per *Random Access Memory*).
L'unità basica di memorizzazione è la cifra binaria, o *bit*, che è il composto aplologico delle parole *binary digit*. Un bit può contenere solo due valori, la cifra 0 oppure la cifra 1, e il correlativo sistema di numerazione binario richiede pertanto solo quei due valori per la codifica dell'informazione digitale.

NOTA

Consultare l'Appendice C per un approfondimento sul sistema di numerazione binario.

La memoria primaria, dal punto di visto logico, è rappresentabile come una sequenza di celle o locazioni di memoria ciascuna delle quali può memorizzare una certa quantità di informazione (Figura 1.5). Ogni cella, detta parola (*memory word*) ha una dimensione fissa e tale dimensione, espressa in multipli di 8, ne indica la lunghezza, ossia il suo numero di bit (possiamo avere, per esempio, word di 8 bit, di 16 bit, di 32 bit e così via). Così se una cella ha k bit allora la stessa può contenere una delle 2^k combinazioni differenti di bit.
Inoltre la lunghezza della word indica anche che quella è la quantità di informazione che un computer durante un'operazione può elaborare allo stesso tempo e in parallelo.
Altra caratteristica essenziale di una cella di memoria è che ha un *indirizzo*, ossia un numero binario che ne consente la localizzazione da parte della CPU al fine del reperimento o della scrittura di contenuto informativo anch'esso binario. La quantità di indirizzi referenziabili dipende dal numero di bit di un indirizzo: generalizzando, se un indirizzo ha n bit allora il massimo numero di celle indirizzabili sarà 2^n.

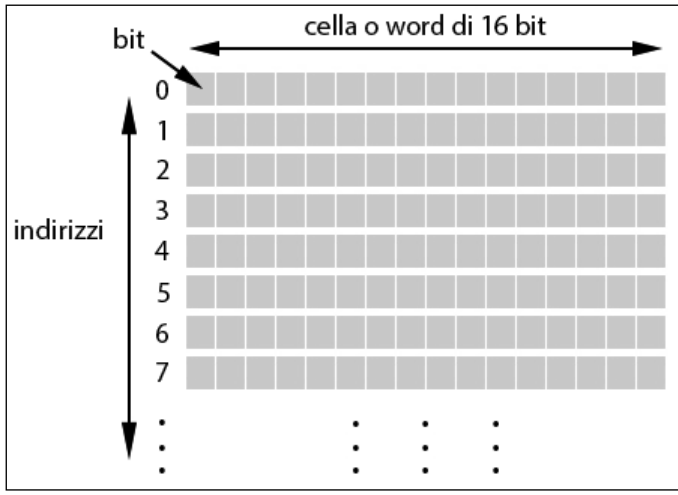

Figura 1.5 Memoria primaria come sequenza di celle.

Per esempio, la Figura 1.6 mostra tre differenti layout per una memoria di 160 bit rispettivamente con celle di 8 bit, 16 bit e 32 bit. Nel primo caso sono necessari almeno 5 bit per esprimere 20 indirizzi da 0 a 19 (infatti 2^5 ne permette fino a 32); nel secondo caso sono necessari almeno 4 bit per esprimere 10 indirizzi da 0 a 9 (infatti 2^4 ne permette fino a 16); nel terzo caso sono necessari almeno 3 bit per esprimere 5 indirizzi da 0 a 4 (infatti 2^3 ne permette fino a 8).

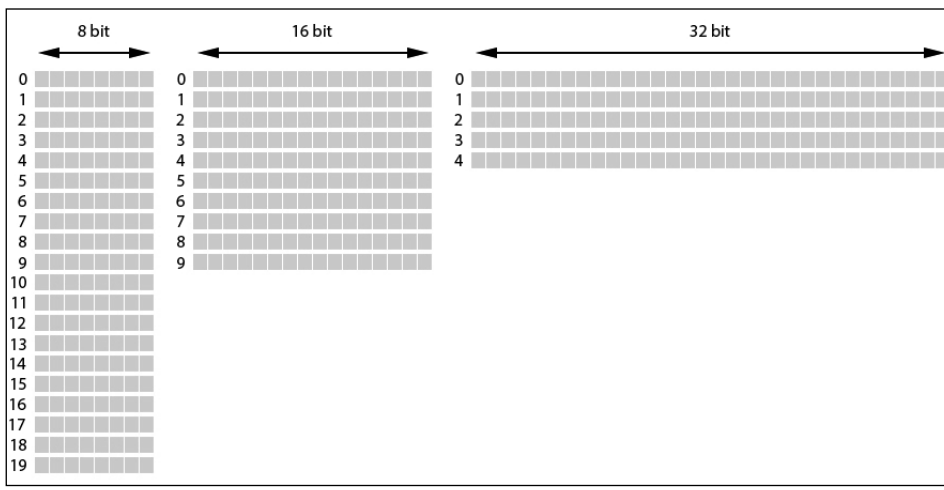

Figura 1.6 Tre differenti layout per una memoria di 160 bit.

In più è importante dire che il numero di bit di un indirizzo è indipendente dal numero di bit per cella: una memoria con 2^{10} celle di 8 bit ciascuna e una memoria con 2^{10} celle di 16 bit ciascuna necessiteranno entrambe di indirizzi di 10 bit.

La dimensione di una memoria è dunque data dal numero di celle per la loro lunghezza in bit. Nei nostri tre casi è sempre di 160 bit (correntemente, un computer moderno avrà una dimensione di 4 Gigabyte di memoria se avrà almeno 2^{32} celle indirizzabili di 1 byte di lunghezza ciascuna).

Ordinamento dei byte

Quando una word è composta da più di 8 bit (1 byte), tipo 16 bit (2 byte), 32 bit (4 byte) e così via vi è la necessità di decidere come ordinare o disporre in memoria i relativi byte. Oggi esistono due modalità di ordinamento largamente utilizzate dai moderni elaboratori elettronici che sono denominate come segue.

- *Big endian* (architetture Motorola 68k, SPARC e così via), dove, data una word, il byte più significativo (*most significant byte*) è memorizzato all'indirizzo più basso (*smallest address*) e a partire da quello, da sinistra a destra (*left-to-right*), sono memorizzati i byte successivi.
- *Little endian* (architetture Intel x86, x86-64 e così via),dove, data una word, il byte meno significativo (*least significant byte*) è memorizzato all'indirizzo più basso (*smallest address*) e a partire da quello, da destra a sinistra (*right-to-left*), sono memorizzati i byte successivi.

CURIOSITÀ

Questi termini si devono allo scrittore e poeta irlandese Jonathan Swift, che nel suo famoso libro *I viaggi di Gulliver* prendeva in giro i politici che si facevano guerra per il giusto modo di rompere le uova sode: dalla punta più piccola (*little end*) oppure dalla punta più grande (*big end*)? I due termini vennero poi ripresi da Danny Cohen, uno scienziato informatico, in un significativo articolo del 1980, dal titolo *ON HOLY WARS AND A PLEA FOR PEACE*, rintracciabile ancora all'URL `http://www.ietf.org/rfc/ien/ien137.txt`.

Per comprendere quanto detto, consideriamo come viene memorizzato un numero come 2854123 (binario, 00000000001010111000110011101011) in una word di 32 bit secondo un'architettura big endian e secondo un'architettura little endian (Figura 1.7): nel primo caso il byte più a sinistra (più significativo) è memorizzato nell'indirizzo 0 e poi a seguire, da sinistra a destra, gli altri byte negli indirizzi 1, 2 e 3; nel secondo caso il byte più a destra (meno significativo) è memorizzato nell'indirizzo 0 e poi a seguire, da destra a sinistra, gli atri byte negli indirizzi 1, 2 e 3.

TERMINOLOGIA

Se vediamo una word come una sequenza di bit (Figura 1.8) piuttosto che come una sequenza di byte, possiamo altresì dire che essa avrà un bit più significativo (*most significant bit*) che sarà ubicato al limite sinistro di tale sequenza (*high-order end*) e un bit meno significativo (*least significant bit*) che sarà ubicato al limite destro sempre della stessa sequenza (*low-order end).*

In definitiva, se due sistemi adottano i due differenti metodi di ordinamento in memoria dei byte e si devono scambiare dei dati, allora vi possono essere dei problemi di congruità tra i dati inviati da un sistema low endian verso un sistema big endian se non sono previsti appositi accorgimenti oppure se non sono forniti idonei meccanismi di conversione.

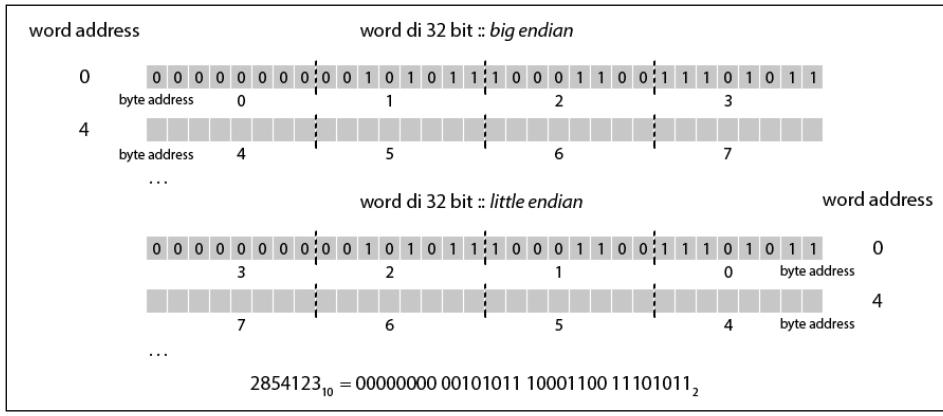

Figura 1.7 Ordinamento della memoria: differenza tra big endian e little endian.

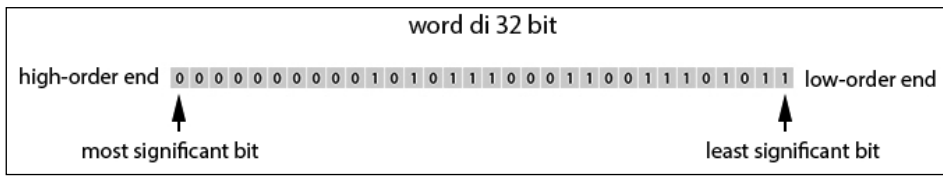

Figura 1.8 Word come sequenza di bit.

Per esempio, se un'applicazione scritta su un sistema SPARC memorizza dei dati binari in un file e poi lo stesso file è aperto il lettura su un sistema x86, si avranno dei problemi di congruità perché il file è stato scritto in modo *endian-dependent*.
Per evitare tale problema si possono adottare vari accorgimenti come, per esempio, quello di scrivere i dati in un formato *neutrale* che prevede file testuali e stringhe, oppure adottare idonee routine di conversione (*byte swapping*) che, a seconda del sistema in uso, forniscano la corretta rappresentazione dell'informazione binaria.
Quando si deve scrivere software per diverse piattaforme hardware che hanno sistemi di *endianness* incompatibili lo stesso deve essere sempre pensato in modo portabile, non assumendo mai, dunque, un particolare ordinamento in memoria dei byte.

Paradigmi di programmazione

Un paradigma o stile di programmazione indica un determinato modello concettuale e metodologico, offerto in termini concreti da un linguaggio di programmazione, al quale fa riferimento un programmatore per la progettazione e scrittura di un programma informatico e dunque per la risoluzione del suo particolare problema algoritmico. Si conoscono numerosi paradigmi di programmazione, ma quelli che seguono rappresentano i più comuni.
Nel *paradigma procedurale* l'unità principale di programmazione è, per l'appunto, la procedura o la funzione che ha lo scopo di manipolare i dati del programma. Questo paradigma è talune volte indicato anche come *imperativo*, perché consente di costruire

un programma indicando dei comandi (assegna, chiama una procedura, esegui un loop e così via) che esplicitano quali azioni si devono eseguire, e in quale ordine, per risolvere un determinato compito. Questo paradigma si basa, dunque, su due aspetti di rilievo: il primo è riferito al cambiamento di stato del programma che è causa delle istruzioni eseguite (si pensi al cambiamento del valore di una variabile in un determinato tempo durante l'esecuzione del programma); il secondo è inerente allo stile di programmazione adottato che è orientato al "come fare o come risolvere" piuttosto che al "cosa si desidera ottenere o cosa risolvere". Esempi di linguaggi che supportano il paradigma procedurale sono FORTRAN, COBOL, Pascal e C.
Nel *paradigma a oggetti* l'unità principale di programmazione è l'oggetto (nei sistemi basati sui prototipi) oppure la classe (nei sistemi basati sulle classi). Questi oggetti, definibili come virtuali, rappresentano in estrema sintesi astrazioni concettuali degli oggetti reali del mondo fisico che si vogliono modellare. Questi ultimi possono essere oggetti più generali (per esempio un computer) oppure oggetti più specifici, ovvero maggiormente specializzati (per esempio una scheda madre, una scheda video e così via). Noi utilizziamo tali oggetti senza sapere nulla della complessità con cui sono costruiti e comunichiamo con essi attraverso l'invio di messaggi (sposta il puntatore, digita dei caratteri) e mediante delle interfacce (il mouse, la tastiera). Inoltre, essi sono dotati di attributi (velocità del processore, colore del case e così via) che possono essere letti e, in alcuni casi, modificati. Questi oggetti reali vengono presi come modello per la costruzione di sistemi software a oggetti, dove l'oggetto (o la classe) avrà metodi per l'invio di messaggi e proprietà che rappresenteranno gli attributi da manipolare. Principi fondamentali di tale paradigma sono i seguenti.

- L'*incapsulamento*, che è un meccanismo attraverso il quale i dati e il codice di un oggetto sono protetti da accessi arbitrari (*information hiding*). Per dati e codice intendiamo tutti i membri di una classe, ovvero sia i dati membro (come le variabili), sia le funzioni membro (definite anche *metodi* in molti linguaggi di programmazione orientati agli oggetti). La protezione dell'accesso viene effettuata applicando ai membri della classe degli specificatori di accesso, definibili come: *pubblico*, con cui si consente l'accesso a un membro di una classe da parte di altri metodi di altre classi; *protetto*, con cui si consente l'accesso a un membro di una classe solo da parte di metodi appartenenti alle sue classi derivate; *privato*, con cui un membro di una classe non è accessibile né da metodi di altre classi né da quelli delle sue classi derivate ma soltanto dai metodi della sua stessa classe.
- L'*ereditarietà*, che è un meccanismo attraverso il quale una classe può avere relazioni di ereditarietà nei confronti di altre classi. Per relazione di ereditarietà intendiamo una relazione gerarchica di parentela *padre-figlio*, dove una classe figlio (definita *classe derivata* o *sottoclasse*) deriva da una classe padre (definita *classe base* o *superclasse*) i metodi e le proprietà pubbliche e protette, e dove essa stessa ne definisce di proprie. Con l'ereditarietà si può costruire, di fatto, un modello orientato agli oggetti che in principio è generico e minimale (ha solo classi base) e poi, man mano che se ne presenta l'esigenza, può essere esteso attraverso la creazione di sottomodelli sempre più specializzati (ha anche classi derivate).
- *Il polimorfismo*, che è un meccanismo attraverso il quale si può scrivere codice in modo generico ed estendibile grazie al potente concetto che una classe base può riferirsi a tutte le sue classi derivate cambiando, di fatto, la sua *forma*. Ciò si traduce, in pratica,

nella possibilità di assegnare a una variabile `A` (istanza di una classe base) il riferimento di una variabile `B` (istanza di una classe derivata da `A`) e, successivamente, riassegnare alla stessa variabile `A` il riferimento di una variabile `C` (istanza di un'altra classe derivata da `A`). La caratteristica appena indicata ci consentirà, attraverso il riferimento `A`, di invocare i metodi di `A` che `B` o `C` hanno ridefinito in modo specialistico, con la garanzia che il sistema run-time del linguaggio di programmazione a oggetti saprà sempre a quale esatta classe derivata appartengono. La discriminazione automatica, effettuata dal sistema run-time di un tale linguaggio, di quale oggetto (istanza di una classe derivata) è contenuto in una variabile (istanza di una classe base) avviene con un meccanismo definito *dynamic binding* (binding dinamico).

Esempi di linguaggi che supportano il paradigma a oggetti sono Java, C#, C++, JavaScript, Smalltalk e Python.

- Nel *paradigma funzionale* l'unità principale di programmazione è la funzione vista in puro senso matematico. Infatti, il flusso esecutivo del codice è guidato da una serie di valutazioni di funzioni che, trasformando i dati che elaborano, conducono alla soluzione di un problema. Gli aspetti rilevanti di questo paradigma sono: nessuna mutabilità di stato (le funzioni sono *side-effect free*, ossia non modificano alcuna variabile); il programmatore non si deve preoccupare dei dettagli implementativi del "come" risolvere un problema ma piuttosto di "cosa" si vuole ottenere dalla computazione. Esempi di linguaggi che supportano il paradigma funzionale sono: Lisp, Haskell, F#, Erlang e Clojure.
- Nel *paradigma logico* l'unità principale di programmazione è il *predicato logico*. In pratica con questo paradigma il programmatore dichiara solo i "fatti" e le "proprietà" che descrivono il problema da risolvere lasciando al sistema il compito di "inferirne" la soluzione e dunque raggiungerne il "goal" (l'obiettivo). Esempi di linguaggi che supportano il paradigma logico sono: Datalog, Mercury, Prolog e ROOP.

Dunque, il linguaggio C supporta pienamente il paradigma procedurale dove l'unità principale di astrazione è rappresentata dalla funzione attraverso la quale si manipolano i dati di un programma. Da questo punto di vista, si differenzia dai linguaggi di programmazione che sposano il paradigma a oggetti come, per esempio, C++ o Java, perché in quest'ultimo paradigma ci si concentra prima sulla creazione di nuovi tipi di dato (le classi) e poi sui metodi e le variabili a essi relativi.
In altre parole, mentre in un linguaggio procedurale come C la modularità di un programma viene fondamentalmente descritta dalle procedure o funzioni che manipolano i dati, nella programmazione a oggetti la modularità viene descritta dalle classi che incapsulano al loro interno metodi e variabili. Per questa ragione si suole dire che nel mondo a oggetti la dinamica (metodi) è subordinata alla struttura (classi).

TERMINOLOGIA

Nei linguaggi di programmazione si usano termini come funzione (*function*), metodo (*method*), procedura (*procedure*), sotto-programma (*subprogram*), sotto-routine (*subroutine*) e così via per indicare un *blocco* di codice posto a un determinato indirizzo di memoria che è invocabile (chiamabile), per eseguire le istruzioni ivi collocate. Dal punto di vista pratico, pertanto, significano tutte la stessa cosa anche se, in letteratura, talune volte sono evidenziate delle differenze soprattutto in base al linguaggio di programmazione che si prende in esame. Per esempio, in Pascal una funzione ritorna un valore, mentre una

procedura non ritorna nulla; in C una funzione può agire anche come una procedura, mentre il termine metodo non è esistente; in C++ un metodo è una funzionalità "associata" all'oggetto o alla classe dove è stato definito ed è anche denominato *funzione membro*.

Concetti introduttivi allo sviluppo

Prima di affrontare lo studio sistematico della programmazione in C, e preliminarmente all'illustrazione di un semplice programma che ne darà una panoramica generale, appare opportuno soffermarci su alcuni concetti propedeutici allo sviluppo di C che sono trasversali al linguaggio e che servono quindi per inquadrarlo meglio nel suo complesso.

Fasi di sviluppo di un programma

Un programma scritto in C prima di poter essere eseguito sul sistema di interesse passa attraverso le seguenti fasi, che ne definiscono un generico *ciclo* operativo.

- *Analisi*. È la fase che sottende all'individuazione delle informazioni preliminari allo sviluppo di un software quali la sua fattibilità in senso tecnico ed economico (analisi costi/benefici), il suo dominio applicativo, i suoi requisiti funzionali (cosa il software deve offrire) e così via.
- *Progettazione*. È la fase di *design* dove si inizia a ideare in modo più concreto come si può sviluppare il software che è stato oggetto di una precedente analisi. Il software viene scomposto in moduli e componenti, si definisce la loro interazione e anche il loro contenuto (dettaglio interno). La progettazione indica, pertanto, *come* il software deve essere implementato piuttosto di *cosa* deve fare (appannaggio della fase di analisi).
- *Codifica*. È la fase dove si implementa concretamente il software oggetto della progettazione. In pratica, attraverso l'utilizzo di un editor di testo, si scrivono gli algoritmi, le funzioni e le istruzioni del programma codificate secondo la sintassi propria del linguaggio C. Il codice scritto nell'editor si chiama *codice sorgente* (*source code*), mentre il file prodotto si chiama *file di codice sorgente* (*source code file*).
- *Compilazione*. È la fase dove un apposito programma, il compilatore (*compiler*), traduce e converte il codice sorgente presente nel relativo file in codice eseguibile (*executable code*), ovvero in codice nativo del sistema hardware riferito. Questo codice nativo è scritto in un apposito file (*executable code file*). Ricordiamo che il codice nativo è il codice espresso nel linguaggio macchina (*machine language*) del sistema hardware scelto e che tale linguaggio è anche l'unico che lo stesso può comprendere direttamente, senza alcuna intermediazione. Inoltre, in questa fase, il compilatore si occupa di verificare che il codice sorgente sia scevro da errori sintattici e, in caso contrario, avvisa l'utente e non procede alla compilazione e alla generazione del codice eseguibile.
- *Esecuzione*. È la fase dove il file contenente il codice eseguibile è caricato nella memoria ed è eseguito, ovvero produce gli effetti computazionali per i quali è stato progettato e sviluppato. Solitamente, in una shell testuale, per caricare in memoria ed eseguire un programma scritto in C, è sufficiente digitare nel relativo ambiente di esecuzione il nome del corrispondente file eseguibile. Lo stesso file eseguibile,

invece, in un ambiente dotato di GUI, può essere caricato ed eseguito facendo clic due volte di seguito sulla corrispondente icona.

- *Test e debug*. È la fase dove si verifica la correttezza funzionale del programma in esecuzione e se lo stesso presenta errori o comportamenti non pertinenti con la logica che avrebbe, invece, dovuto seguire. A tal fine esistono appositi programmi chiamati *debugger* che consentono di analizzare il flusso di esecuzione di un programma *step by step*, fermare la sua esecuzione al raggiungimento di un determinato *breakpoint* per esaminarne il corrente stato, rilevare il valore delle variabili in un certo momento e così via.

> **NOTA**
>
> L'Appendice A contiene un breve tutorial introduttivo sull'utilizzo del debugger GDB.

- *Mantenimento*. È la fase dove un programma che è in produzione, a seguito di richieste od osservazioni degli utenti, può subire modifiche migliorative che impattano, per esempio, sulla performance, oppure modifiche integrative che riguardano l'aggiunta di moduli che coprono caratteristiche funzionali supplementari rispetto a quelle previste in origine.

Codifica, compilazione ed esecuzione: dettaglio

Quando scriviamo un programma in linguaggio C, il relativo codice sorgente è scritto in un file la cui *basename*, ovvero la parte del nome che precede il carattere punto (.), può avere un numero di caratteri il cui massimo valore è uguale a quello stabilito dal sistema dove il programma deve girare (per esempio, in un sistema MS-DOS la basename può essere costituita da massimo 8 caratteri), mentre l'*extension*, ovvero la parte del nome che segue il carattere punto . (detta anche *suffix*, *type* o *format*), deve essere indicata obbligatoriamente per molti compilatori con il carattere `c` (per esempio, `client.c`, `widgets.c`, `time.c` e così via sono tutti nomi che esprimono file di codice sorgente C). In più è importante rammentare che nei sistemi operativi *Unix-like* vi è distinzione tra lettere maiuscole e lettere minuscole che compongono il nome di un file C. Per esempio, mentre in Windows non c'è alcuna differenza tra, diciamo, `client.c` e `Client.c`, in GNU/Linux c'è; entrambi, infatti, rappresentano file differenti.

> **ATTENZIONE**
>
> Se denominate un file di codice sorgente C con estensione `.C` e non `.c`, un compilatore, come per esempio quello proprio della suite GCC, può interpretarlo come file di codice sorgente C++. Infatti, per C++ le comuni estensioni valide sono `.cpp`, `.cc`, `.cxx` e, per l'appunto, `.C`.

Dopo la scrittura di un file di codice sorgente denominato secondo le regole indicate, lo stesso deve essere dato come input a un compilatore C al fine di fargli produrre il relativo file di codice macchina per il sistema hardware target. Il compilatore, tuttavia, esegue tale lavoro producendo un file di codice intermedio definito *file di codice oggetto* (*object code file*), che nonostante contenga codice macchina non è ancora eseguibile sul sistema target. Infatti, interviene poi, durante il processo di compilazione, un altro

programma denominato *linker* che "combina", "collega", il codice dell'object file con il codice di *startup* del sistema target corrispondente e con il codice dei file oggetto delle eventuali librerie utilizzate. Al termine del suo lavoro il linker produce il relativo file di codice eseguibile.
Questa caratteristica di generazione del file eseguibile in una sorta di processo a "due fasi" ha una spiegazione che è legata al concetto di modularizzazione del codice. Dati più file oggetto compilati, è possibile grazie al linker combinarli tutti insieme per produrre un unico file eseguibile. Successivamente, se uno dei file oggetto necessita di modifiche o aggiustamenti, è necessario compilare solo tale file oggetto e poi ricombinarlo insieme agli altri file oggetto senza che questi ultimi abbiamo subìto un'altra compilazione.

NOTA

Generalmente il programma linker è invocato automaticamente dal compilatore durante la fase di compilazione. Il compilatore invoca anche un altro programma denominato *preprocessor* (preprocessore) che elabora dei particolari "comandi" scritti all'interno di un file di codice sorgente, definiti come *preprocessor directives* (direttive del preprocessore), che hanno lo scopo di compiere determinate manipolazioni prima dell'effettuazione della compilazione (per esempio includere all'interno di un file sorgente contenuto presente in un altro file detto *header file*).

L'intero processo di generazione di un file di codice eseguibile dato un file di codice sorgente è visualizzabile nella Figura 1.9.

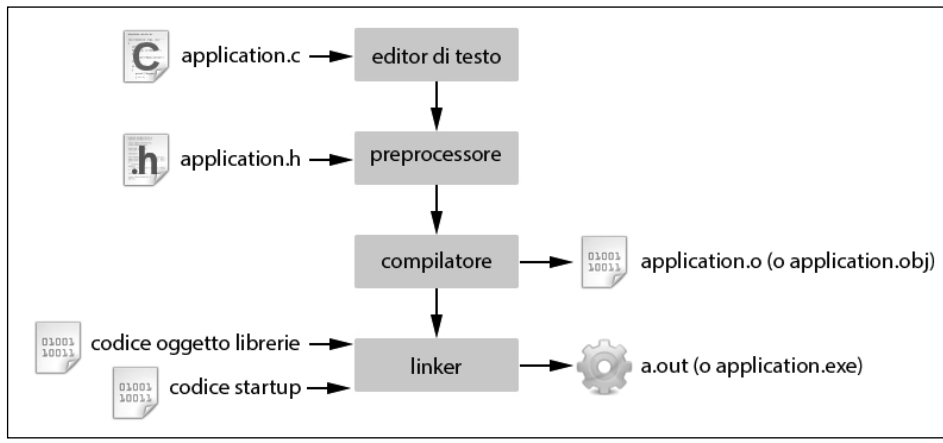

Figura 1.9 Flusso di generazione di un file eseguibile dato un file sorgente.

Sistemi di compilazione e sistemi target

Per scrivere codice in linguaggio C è necessario utilizzare tool di sviluppo che mettono a disposizione per un determinato sistema target, sia hardware (architetture ARM, x86, PowerPC, MIPS e così via) sia software (sistemi Unix, Windows e così via), il relativo compilatore, le funzionalità della libreria standard e via discorrendo.

In linea generale nei sistemi Unix è presente il comando `cc`, che invoca il relativo compilatore e produce, in assenza di opzioni, un file eseguibile denominato `a.out` (produce altresì un file oggetto con estensione `.o` e con la stessa basename del file di codice sorgente, che però viene eliminato dal linker quando il file eseguibile è stato creato).

CURIOSITÀ

Il nome `a.out` è un'abbreviazione di *assembler output* e la sua etimologia si fa risalire ai tempi di Ken Thompson e del PDP-7, quando il file sorgente di un programma si faceva processare dal relativo assemblatore che produceva un file di output, con nome fisso, che era direttamente eseguibile. Dopo di allora è rimasta tradizione dei compilatori Unix generare tale nome di output per un file eseguibile in mancanza di diversa indicazione.

Il comando `cc`, è solitamente un alias verso il reale comando di compilazione che può essere, per esempio, `gcc`, presente in genere come compilatore di default nei sistemi GNU/Linux con installata la suite GCC (*GNU Compiler Collection*) oppure `clang`, presente come compilatore di default nei sistemi FreeBSD con installata la suite LLVM (*Low Level Virtual Machine*).
Questi comuni compilatori, tuttavia, non sono gli unici utilizzabili in ambienti Unix-like; è possibile sceglierne altri, quali, solo per citarne alcuni, SAS/C, Amsterdam Compiler Kit, LCC (*Little C Compiler*) e PCC (*Portable C Compiler*).
Per quanto attiene ai sistemi Windows, di default, non sono mai installati né un compilatore né la relativa suite di strumenti e librerie. È possibile, comunque, usare la suite GCC tramite i progetti Cygwin e MinGW (M*inimalist GNU for Windows*), che forniscono rispettivamente un ambiente runtime Unix/POSIX-like sotto Windows e un ambiente di compilazione per applicazioni native sotto Windows.
In ogni caso è possibile utilizzare, così come visto per i sistemi Unix, anche altri compilatori C, quali quello fornito da Microsoft tramite l'installazione dell'IDE Visual Studio e denominato `cl` (in questo caso il file oggetto prodotto avrà la stessa basename del file sorgente ma l'estensione `.obj`, mentre il file eseguibile avrà la stessa basename del file sorgente ma l'estensione `.exe`), Digital Mars e Pelles C.

NOTA

Ricordiamo, come anticipato nella Prefazione, che il compilatore di elezione scelto per la didattica del corrente testo è quello fornito dalla suite GCC utilizzabile nativamente con il sistema operativo GNU/Linux. Per il sistema operativo Windows, invece, GCC è utilizzabile, nel nostro caso, mediante l'installazione dell'environment MinGW. L'Appendice A spiega come installare e utilizzare GCC sia sotto sistemi GNU/Linux sia sotto sistemi Windows. In ogni caso verrà anche utilizzato il compilatore `cl` di Microsoft per evidenziare eventuali differenze tra differenti implementazioni del linguaggio. Per quanto concerne invece il sistema target hardware, esso è una piattaforma x86-64, ovvero una versione a 64 bit dell'*instruction set* della piattaforma x86 e ciò sia per il sistema GNU/Linux sia per il sistema Windows (in quest'ultimo caso, comunque, la suite MinGW sarà a 32 bit e ciò per evidenziare le differenze tra compilatori a 64 bit, come quello del sistema GNU/Linux, rispetto a quelli a 32 bit).

Il primo programma

Vediamo, attraverso la disamina del Listato 1.1, quali sono gli elementi basilari per strutturare e scrivere un programma in C, con l'avvertenza che tutte le informazioni didattiche successivamente enucleate su alcuni costrutti del linguaggio saranno introduttive ai concetti che tratteranno e giocoforza non dettagliate. Le stesse saranno trattate con dovizia di particolari nei relativi capitoli di pertinenza.
Abbiamo, in questa sede, infatti, inteso perseguire solamente i seguenti obiettivi: illustrare una struttura di massima degli elementi costitutivi di un programma in C; fornire una terminologia basica applicabile ai principali costrutti del linguaggio.

Listato 1.1 PrimoProgramma.c (PrimoProgramma).

```
/* PrimoProgramma.c :: Struttura di un generico programma :: */
#include <stdio.h>
#include <stdlib.h>

#define MULTIPLICAND 10
#define MULTIPLIER  20

// prototipo di una funzione
int mult(int a, int b);

// entry point del programma
int main(void)
{
    // dichiarazione e inizializzazione contestuale
    // di più variabili di diverso tipo
    char text_1[] = "Primo programma in C:",
         text_2[] = " Buon divertimento!";
    int a = 10, b = 20;

    float f; // dichiarazione
    f = 44.5f; // inizializzazione

    // stampa qualcosa...
    printf("%s%s\n", text_1, text_2);
    printf("Stampero' un test condizionale tra a=%d e b=%d:\n", a, b);

    if (a < b) // se a < b stampa quello che segue...
    {
        printf("a < b VERO!");
    }
    else /* altrimenti stampa quest'altra stringa */
    {
        printf("a > b VERO!");
    }

    printf("\nStampero' un ciclo iterativo, dove leggero' ");
    printf("per 10 volte il valore di a\n");

    /*
     * ciclo for
     */
```

```
    for (int i = 0; i < 10; i++)
    {
        printf("Passo %d ", i);
        printf("--> a=%d\n", a);
    }

    printf("Ora eseguiro' una moltiplicazione tra %d e %d\n", MULTIPLICAND, MULTIPLIER);
    int res = mult(MULTIPLICAND, MULTIPLIER); // invocazione di una funzione
    printf("Il risultato di %d x %d e': %d\n", MULTIPLICAND, MULTIPLIER, res);

    /*
        // esce dalla funzione main
     */
    return (EXIT_SUCCESS);
}

/**************************************
 * Funzione: mult                     *
 * Scopo: moltiplicazione di due valori *
 * Parametri: a, b -> int             *
 * Ritorno: int                       *
 **************************************/
int mult(int a, int b)
{
    return a * b;
}
```

Il programma del Listato 1.1 inizia con la definizione di un'istruzione di commento, ovvero con la scrittura, tra il carattere di inizio commento `/*` e il carattere di fine commento `*/`, di testo che viene ignorato dal compilatore e che serve a documentare o chiarire parti del codice sorgente. Il testo di questo tipo di commento può anche essere suddiviso su più righe e, infatti, per tale ragione è spesso anche definito commento *multiriga*.
In ogni caso non è possibile innestare commenti multiriga (Snippet 1.1), e ciò perché il marker di commento finale `*/` del commento innestato "pareggia" con il marker di commento iniziale `/*` del commento che innesta. In questo modo, quindi, il marker di commento finale `*/` del commento che innesta si trova senza un marker di commento iniziale `/*` con cui corrispondere, inducendo il compilatore a generare un errore di compilazione del tipo `error: unknown type name 'CCCC'`.

Snippet 1.1 Commenti multiriga innestati.

```
/* // commento che innesta
 AAAA
 /* // commento innestato
     BBB
  */
  CCCC
*/
```

Infine, i commenti multiriga possono essere anche scritti di fianco, a lato di una porzione di codice (*winged comment*), come mostra quello definito dopo l'istruzione `else`, così come essere scritti in *forma* di riquadro di contenimento (*boxed comment*), come mostra quello scritto prima della definizione della funzione `mult`, oppure scrivendo degli asterischi `*` per ogni riga di separazione, come mostra quello indicato prima del ciclo `for`.

In ogni caso, a parte le forme indicate di scrittura dei commenti multiriga che sono quelle più comuni, il programmatore è libero di scegliere la formattazione che più gli aggrada a condizione, però, che via sia sempre una corrispondenza tra un marcatore di commento iniziale `/*` e un marcatore di commento finale `*/`.
C99 ha introdotto, inoltre, un secondo tipo di commento, che è indicato tramite l'utilizzo di due caratteri slash `//` cui si fa seguire il testo da commentare. Questo commento, poiché termina in automatico alla fine della corrispettiva riga, è definito commento *a singola riga* e ha l'importante caratteristica che può essere innestato all'interno di un commento multiriga, come mostra in modo evidente il commento posto prima dell'istruzione `return` all'interno della funzione `main`. In più, anche con questo tipo di commento, è possibile scrivere commenti multiriga semplicemente scrivendo su ogni riga i caratteri `//` e la porzione di testo da commentare. Un esempio di quanto detto è visibile nei primi due commenti posti subito dopo la parentesi graffa di apertura ({) della funzione `main`.

DETTAGLIO

Il compilatore quando trova dei commenti, prima della compilazione, li rimuove tutti sostituendo ciascuno di essi con un carattere di spazio.

Seguono il primo commento del sorgente e le istruzioni `#include` e `#define`, che rappresentano delle direttive per il preprocessore, ovvero dei *comandi* che saranno da quest'ultimo eseguiti prima che la compilazione vera e propria venga avviata.
La direttiva `#include` consente di includere, a partire dal punto dove è stata definita, il contenuto di un file indicato. Per esempio, `#include <stdio.h>` e `#include <stdlib.h>` includeranno rispettivamente il contenuto del file `stdio.h`, che fornisce le funzionalità per effettuare delle operazioni di input/output, e il contenuto del file `stdlib.h`, che fornisce funzionalità di carattere generale.
Il contenuto di questi file, detti anche *header file*, è essenziale per la corretta compilazione del codice sorgente quando si utilizzano, per l'appunto, le loro funzionalità.
Nel nostro caso è impiegata intensivamente la funzione `printf`, che visualizza nello *standard output* la stringa di caratteri indicata come argomento, la quale è dichiarata nel file `stdio.h` che deve, per l'appunto, essere incluso per permetterne l'utilizzo. Lo stesso meccanismo di inclusione avviene per l'identificatore `EXIT_SUCCESS` dichiarato nel file `stdlib.h`, che viene anch'esso incluso.

NOTA

I file header contengono informazioni per il compilatore che sono essenziali per consentire la corretta compilazione di un programma (per esempio il nome delle costanti, i prototipi delle funzioni e così via). Tuttavia, il codice implementativo delle funzioni è posto in altri file di libreria precompilati (file oggetto) che sono collegati dal linker al file oggetto del programma principale per produrre il codice eseguibile finale.

La direttiva `#define` consente di definire, invece, delle *macro*, cioè degli identificatori che possono essere utilizzati all'interno di un programma come *nomi costanti* che il preprocessore sostituisce con gli effettivi valori collegati. Il programma in esame, dunque, definisce le macro `MULTIPLICAND` e `MULTIPLIER` con i valori relativi di `10` e `20` e, pertanto, quando il preprocessore le individuerà nel sorgente le sostituirà con i predetti valori.

Abbiamo poi la scrittura o dichiarazione dell'identificatore `mult`, che rappresenta il cosiddetto *prototipo di funzione* con il quale si indicano le sue proprietà quali il tipo di ritorno e gli eventuali parametri che può processare quando invocata. Nel complesso il nome di una funzione, il tipo di ritorno e i tipi dei suoi parametri rappresentano il suo *header*. Il prototipo di una funzione è un aspetto fondamentale perché è utilizzato dal compilatore per verificare la congruità delle relativa definizione (per esempio se i tipi dei parametri corrispondono) e il suo impiego corretto durante una sua chiamata (per esempio se il tipo di un argomento fornito è compatibile con il tipo del parametro dichiarato).
Il prototipo di `mult` specifica che ritorna un tipo di dato intero e accetta due argomenti sempre di tipo intero. Nell'ambito della funzione `main` notiamo come `mult` sia invocata con due argomenti di tipo intero (i valori `10` e `20`) e ritorni un valore di tipo intero assegnato alla variabile `res`. La stessa funzione `mult` ha infatti una sua definizione, ossia una sua implementazione algoritmica scritta dopo la definizione della funzione `main` che evidenzia come ritorni un valore di tipo intero che è il risultato della moltiplicazione tra i parametri `a` e `b` sempre di tipo intero.
La funzione `main`, invece, è la funzione principale di un qualsiasi programma in C e deve essere sempre presente perché ne rappresenta l'*entry point*, ossia il "punto di ingresso" attraverso il quale viene eseguito. In pratica la funzione `main` è invocata automaticamente quando il programma è avviato e poi attraverso di essa vengono eseguite le relative istruzioni e invocate le altre funzioni eventualmente indicate. Da questo punto di vista un programma in C è composto sempre almeno da una funzione `main` e poi da una o più funzioni ausiliarie; non è altro, quindi, che una collezione di una o più funzioni.
Tale funzione può essere definita con un tipo di ritorno `int` e con nessun parametro (keyword `void`) oppure con un tipo di ritorno `int` e con due parametri l'uno di tipo `int` e l'altro di tipo array di puntatori a carattere (Snippet 1.2 e 1.3).

Snippet 1.2 Modalità di definizione di main. Prima definizione.

```
int main(void) { /* ... */ }
```

Snippet 1.3 Modalità di definizione di main. Seconda definizione.

```
int main(int argc, char *argv[]) { /* ... */ }
```

In pratica quando si utilizza la prima definizione si indica che `main` non accetta argomenti dalla riga di comando, mentre quando si utilizza la seconda definizione si esprime la volontà di processare gli argomenti forniti dalla relativa shell.

NOTA

Scorrendo codice in C è possibile vedere altre due modalità di definizione della funzione `main`: `main() { /* ... */ }` e `void main() { /* ... */ }`. Tuttavia, anche se il compilatore in uso può tollerare lo standard C11, esplicita solo quelle indicate negli Snippet 1.2 e 1.3.

Il tipo di ritorno `int` di `main` indica un valore numerico che deve essere ritornato al sistema operativo e che assume per esso un certo significato: per esempio, il valore `0` (espresso dal nostro programma attraverso la macro `EXIT_SUCCESS` definita nel file header `<stdlib.h>`), indica una terminazione corretta del corrente programma. Nel caso della funzione `main`, inoltre, l'istruzione `return` termina anche l'esecuzione di un programma (spesso,

in alcuni programmi, si trova, al posto dell'istruzione `return 0` l'istruzione `exit(0)`, che termina allo stesso modo l'esecuzione di un programma).
È altresì possibile omettere l'istruzione `return` dal `main`, poiché quando il programma raggiunge la parentesi graffa di chiusura `}` della funzione viene in automatico ritornato il valore `0` (in C89 il valore è non definito, in C11 il valore è comunque non definito se il `main` è definito con un tipo di ritorno diverso da `int`).
Per quanto attiene al contenuto della funzione `main` notiamo subito una serie di istruzioni che definiscono delle variabili, ossia delle locazioni di memoria modificabili deputate a contenere un valore di un determinato tipo di dato. Così gli identificatori `text_1` e `text_2` indicano variabili che possono contenere caratteri, gli identificatori `a`, `b` e `res` indicano variabili che possono contenere numeri interi e l'identificatore `f` indica una variabile che può contenere numeri decimali, cioè numeri formati da una parte intera e una parte frazionaria separati da un determinato carattere (in C tale carattere è il punto .).
Una variabile, solitamente, può essere prima dichiarata e poi inizializzata (è il caso della variabile `f`) oppure può essere dichiarata e inizializzata contestualmente in un'unica istruzione (è il caso delle altre variabili).
A parte le varie istruzioni di stampa su console dei valori espressi dalle corrispettive stringhe delle funzioni `printf`, notiamo l'impiego: di un'istruzione di selezione doppia `if/else` che valuta se una data espressione è vera o falsa eseguendone, a seconda del risultato della valutazione, il codice corrispondente (quello del ramo valutato vero oppure quello del ramo valutato falso); di un'istruzione di iterazione `for` che consente di eseguire ciclicamente una serie di istruzioni finché una data espressione è vera.
Pertanto l'istruzione `if/else` valuta se il valore della variabile `a` è minore del valore della variabile `b` e nel caso stampa la relativa stringa, altrimenti, in caso di valutazione falsa, stampa l'altra stringa a esso relativa. L'istruzione `for`, invece, stampa su console per 10 volte informazioni sul valore delle variabili `i` e `a`.

Un "assaggio" di printf

La funzione della libreria standard `printf` è dichiarata nel file header `<stdio.h>` e consente di visualizzare sullo standard output (generalmente a video) il letterale stringa fornitogli come argomento. All'interno del letterale stringa è possibile inserire degli appositi caratteri prefissi dal simbolo percento (`%`), definiti nel complesso come *specifiche di conversione* o *specificatori di formato*, che rappresentano dei *segnaposto* di un determinato tipo di dato che saranno *sostituiti*, in quella precisa locazione, con i valori delle relative variabili fornite come ulteriori argomenti alla funzione.
Così il segnaposto `%d` permette di stampare il valore di una variabile come intero in base 10, `%f` permette di stampare il valore di una variabile con un determinato numero di cifre dopo il punto decimale di separazione, `%s` permette di stampare il valore di una variabile come stringa di caratteri e così via per altri specificatori.
Quando si utilizza la funzione `printf` è essenziale sapere che essa non stampa altresì i caratteri di delimitazione della stringa (i doppi apici ") e non avanza in automatico sulla successiva riga di output al termine della visualizzazione dei suoi caratteri.
Per istruire `printf` in modo che avanzi alla successiva riga di output, a partire dalla quale riprendere (oppure iniziare) la visualizzazione di altri caratteri, è possibile utilizzare una *sequenza di escape* formata dalla combinazione dei caratteri `\n` (*newline character*).

Elementi strutturali di un programma

Un sorgente in C è modellato idealmente in più parti strutturali e semantiche che sono combinate con lo scopo di formare un determinato programma (Figura 1.10). Abbiamo: gli elementi lessicali (keyword, identificatori, letterali stringa, costanti, segni di punteggiatura, commenti e così via); le istruzioni (*statement*) e i blocchi (*compound statement*); le espressioni (operandi più operatori); le dichiarazioni (di variabili, di funzioni e così via); le definizioni esterne; le direttive per il preprocessore.
A partire dagli elementi lessicali si costruiscono le istruzioni, che possono formare delle espressioni, delle dichiarazioni, delle definizioni esterne e delle direttive per il preprocessore.

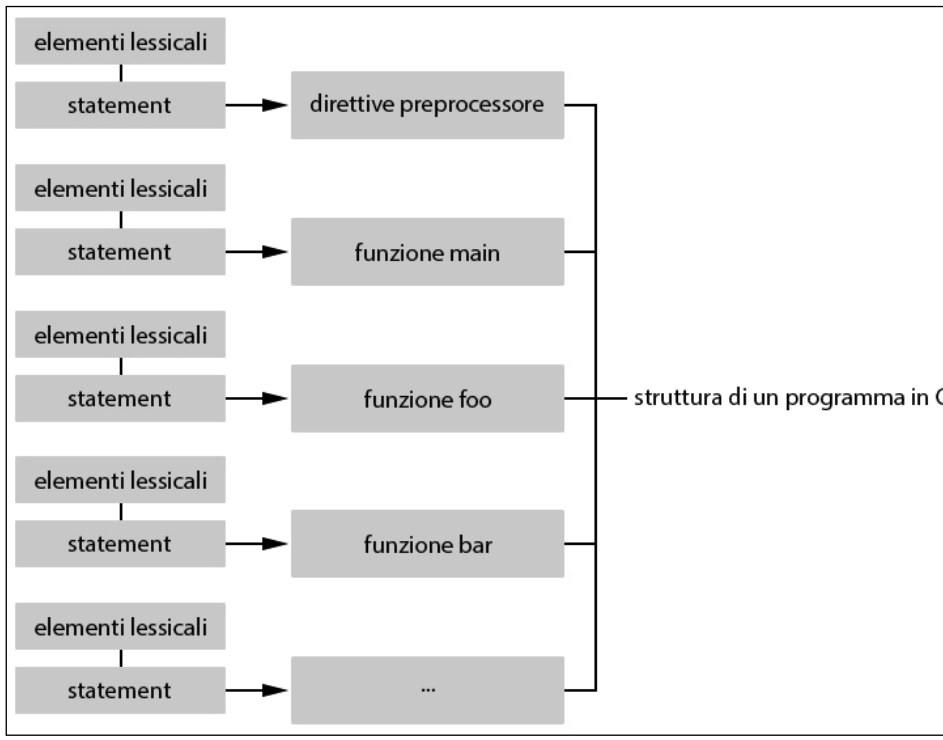

Figura 1.10 Elementi strutturali e semantici di un generico programma in C.

Degli elementi lessicali indicati, tutti, tranne i commenti, sono definiti come *token* e rappresentano dunque delle entità minimali significative per il linguaggio C.
La Tabella 1.1 mostra tutte le keyword del linguaggio aggiornate allo standard C11; la Tabella 1.2 mostra i segni di punteggiatura utilizzabili; lo Snippet 1.4 evidenzia alcuni identificatori, letterali stringa e costanti.

Tabella 1.1 Keyword del linguaggio C (sono case sensitive e riservate nell'uso).

auto	else	long	switch	_Atomic
break	enum	register	typedef	_Bool
case	extern	restrict	union	_Complex
char	float	return	unsigned	_Generic
const	for	short	void	_Imaginary
continue	goto	signed	volatile	_Noreturn
default	if	sizeof	while	_Static_assert
do	inline	static	_Alignas	_Thread_local
double	int	struct	_Alignof	

Tabella 1.2 Segni di punteggiatura.

[	]	(	)	{	}	.	->	++	--
&	*	+	-	~	!	/	%	<<	>>
<	>	<=	>=	==	!=	^	|	&&	||
?	:	;	...	=	*=	/=	%=	+=	-=
<<=	>>=	&=	^=	|=	,	#	##	<:	:>
<%	%>	%:	%:%:						

Snippet 1.4 Esempi di identificatori, letterali stringa e costanti.

```
...
// identificatori
int number, temp, status;
void foo(void) { /*...*/ }

int main(void)
{
    // identificatori
    int number, temp, status;
    void foo(void) { /*...*/ }

    int a = 100; // 100 è una costante intera
    float f = 120.78f; // 120.78 è una costante in virgola mobile
    char c = 'A'; // 'A' è una costante carattere

    char name[] = "Pellegrino"; // "Pellegrino" è un letterale stringa
    ...
}
```

L'ammontare di caratteri di spaziatura (spazio, tabulazione, invio e così via) atti a fungere da separazione tra i token non è obbligatorio: ogni programmatore può scegliere quello che più gli aggrada secondo il suo personale stile di scrittura. Tuttavia un token non può essere "diviso" senza causare un probabile errore e un cambiamento della sua semantica. Lo stesso vale per un letterale stringa con il seguente distinguo: al suo interno è sempre possibile inserire dei caratteri di spazio, ma è un errore separarlo all'interno dell'editor su più righe mediante la pressione del tasto Invio.

Per quanto riguarda invece le istruzioni, esse rappresentano azioni o comandi che devono essere eseguiti durante l'esecuzione del programma. In C, ogni statement deve terminare con il carattere punto e virgola (;) e due o più statement possono essere raggruppate insieme a formare un'unica entità sintattica, definita *blocco*, se incluse tra la parentesi graffa aperta ({) e la parentesi graffa chiusa (}).

Compilazione ed esecuzione del codice

Dopo aver scritto il programma del Listato 1.1, con un qualunque editor di testo o con un IDE di preferenza (per noi l'IDE sarà NetBeans), vediamo come eseguirne la compilazione che, lo ricordiamo, è quel procedimento mediante il quale un compilatore C legge un file sorgente (nel nostro caso `PrimoProgramma.c`) e lo trasforma in un file (per esempio `PrimoProgramma.exe`) che conterrà istruzioni scritte nel linguaggio macchina del sistema hardware di riferimento.

NOTA

Per i dettagli su come eseguire la compilazione di un programma in C, sia in ambiente GNU/Linux sia in ambiente Windows e senza l'ausilio di alcun IDE, consultare l'Appendice A.

Shell 1.1 Invocazione del comando di compilazione (GNU/Linux).

```
[thp@localhost MY_C_SOURCES]$ gcc -std=c11 PrimoProgramma.c -o ~/MY_C_BINARIES/
PrimoProgramma
```

Shell 1.2 Invocazione del comando di compilazione (Windows).

```
C:\MY_C_SOURCES>gcc -std=c11 PrimoProgramma.c -o \MY_C_BINARIES\PrimoProgramma
```

Alla fase di compilazione segue la fase di esecuzione, nella quale un file eseguibile (nel nostro caso `PrimoProgramma.exe` o `PrimoProgramma`), memorizzato per esempio su una memoria secondaria come un hard disk, viene caricato nella memoria principale (la RAM) da un apposito *loader* dove la CPU corrente preleva, decodifica ed esegue le relative istruzioni che compongono il programma medesimo (Figura 1.11).

Shell 1.3 Avvio del programma (GNU/Linux).

```
[thp@localhost MY_C_BINARIES]$ ./PrimoProgramma
```

Shell 1.4 Avvio del programma (Windows).

```
C:\MY_C_BINARIES>PrimoProgramma.exe
```

Output 1.1 Esecuzione di Shell 1.3 o di Shell 1.4.

```
Primo programma in C: Buon divertimento!
Stampero' un test condizionale tra a=10 e b=20:
a < b VERO!
Stampero' un ciclo iterativo, dove leggero' per 10 volte il valore di a
```

```
Passo 0 --> a=10
Passo 1 --> a=10
Passo 2 --> a=10
Passo 3 --> a=10
Passo 4 --> a=10
Passo 5 --> a=10
Passo 6 --> a=10
Passo 7 --> a=10
Passo 8 --> a=10
Passo 9 --> a=10
Ora eseguiro' una moltiplicazione tra 10 e 20
Il risultato di 10 x 20 e': 200
```

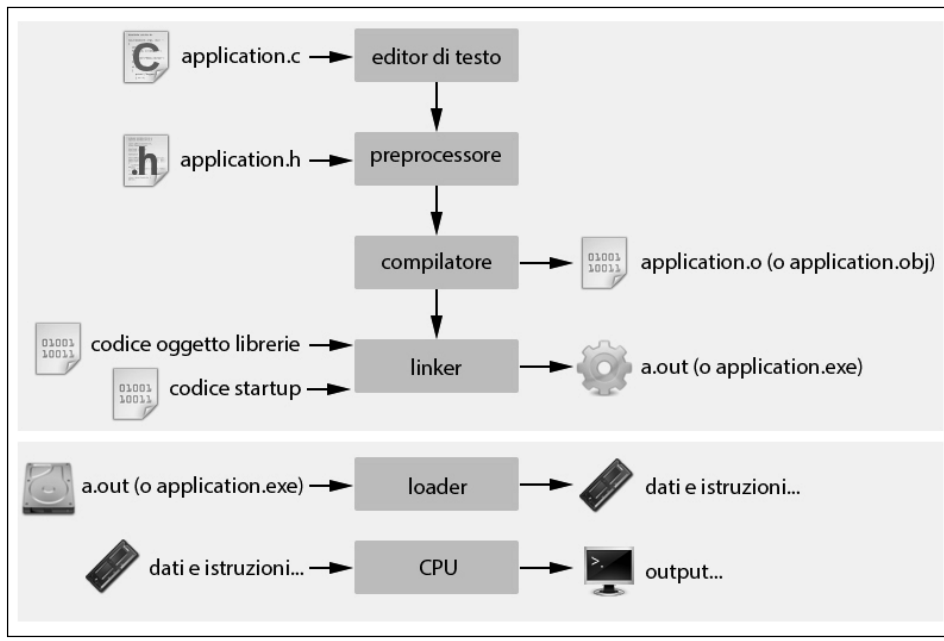

Figura 1.11 Flusso completo di generazione di un file eseguibile e sua esecuzione.

Problemi di compilazione ed esecuzione?

Elenchiamo alcuni problemi che si potrebbero incontrare durante la fase di compilazione oppure di esecuzione del programma appena esaminato.

- Il comando di compilazione `gcc` è inesistente? Verificare che sotto Windows l'environment MinGW sia stato correttamente installato oppure che la corrente distribuzione di GNU/Linux scelta presenti il package GCC. Si rimanda all'Appendice A per i dettagli su come installare e configurare sia MinGW sia GCC.
- Il compilatore `gcc` non trova il file `PrimoProgramma.c`? Verificare che la directory corrente sia `c:\MY_C_SOURCES` (per Windows) oppure `~/MY_C_SOURCES` (per GNU/Linux, laddove in quest'ultimo caso il carattere tilde `~` è sostituito dalla directory home dell'utente corrente, che per noi sarà `/home/thp`).

- Il file `PrimoProgramma.exe` o il file `PrimoProgramma` non viene trovato? Verificare che la directory corrente sia `c:\MY_C_BINARIES` (per Windows) oppure `~/MY_C_BINARIES` (per GNU/Linux, laddove in quest'ultimo caso il carattere tilde `~` è sostituito dalla directory home dell'utente corrente, che per noi sarà `/home/thp`).