Sunto delle lezioni - Informatica

Revisione 1.01

Alessandro Barenghi

March 8, 2018

Questo sunto delle lezioni ha lo scopo di fornire un supporto allo studio *aggiuntivo* rispetto alle lezioni frontali, senza avere la pretesa di sostituirle integralmente. Il testo verrà aggiornato aggiungendo via via i contenuti trattati a lezione. Per quanto sia rivisto con cura, la natura umana del docente fa sì che possano essere presenti errori: non esitate a segnalarli nel caso ne troviate, verranno corretti quanto più in fretta possibile[1]. Per i curiosi, il presente documento è realizzato con LaTeX, ed è rilasciato sotto licenza Creative-Commons Attribution-NonCommercial-ShareAlike 4.0.
`http://creativecommons.org/licenses/by-nc-sa/4.0/`

[1] ovviamente le segnalazioni di falsi positivi non saranno punite in alcun modo, non preoccupatevene

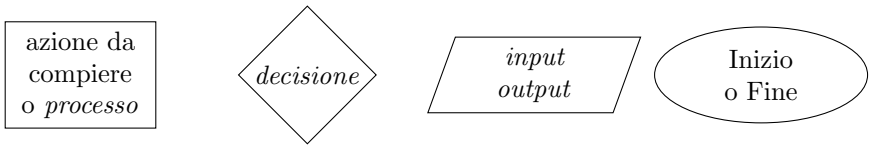

Figure 1: Alcuni dei blocchi usati in un diagramma di flusso

1 Diagrammi di flusso

Tra i formalismi disponibili per la rappresentazione di un algoritmo, uno dei più semplici è quello del diagramma di flusso (flowchart). In un diagramma di flusso viene l' esecuzione di un algoritmo viene suddivisa in una serie di fasi, ognuna delle quali è rappresentata graficamente da un blocco. I blocchi sono collegati da frecce che indicano l' ordine di esecuzione degli stessi.

Sebbene un significativo numero di tipi di blocchi siano stati standardizzati nel documento ISO 5807:1985, per i nostri scopi saranno sufficienti quelli rappresentati in Figura

2 Rappresentazione dei dati in un calcolatore

Un calcolatore moderno elabora i dati utilizzandone una rappresentazione digitale, ovvero rappresentandoli come sequenze *finite* di simboli di un alfabeto anch' esso *finito*. É quindi necessario rappresentare qualunque dato che debba essere elaborato sotto forma di sequenze finite di simboli presi da un alfabeto finito.

A seconda di cosa si desidera rappresentare, è necessario utilizzare una codifica (=convenzione di rappresentazione) opportuna per poterlo fare elaborare dal calcolatore.

L' alfabeto utilizzato per la quasi totalità delle applicazioni è quello binario, ovvero quello costituito da due simboli, 0 e 1.

2.1 Numeri naturali ($\mathbb{N}$)

Il caso dei numeri naturali è il più semplice, essi vengono rappresentati direttamente convertendoli in base 2, ovvero in codifica binaria. Un numero codificato in binario è rappresentato da una sequenza di n cifre $(a_{n-1}a_{n-2}\dots a_1a_0)_2$ che possono essere solamente 0 o 1.

Il significato della codifica è:

$$(a_{n-1}a_{n-2}\dots a_1a_0)_2 = 2^{n-1}\cdot a_{n-1} + 2^{n-2}\cdot a_{n-2} + \dots 2^1\cdot + a_1 + 2^0\cdot a_0$$

Una cifra in codifica binaria è detta *bit*, contrazione di *binary digit*, cifra binaria, appunto. Per quanto possa sembrare poco familiare di primo acchito, non è altro che una variazione sulla comune codifica dei numeri a cui siamo abituati, quella decimale (base 10), dove un numero $(a_{n-1}a_{n-2}\dots a_1a_0)_{10}$ indica:

$$(a_{n-1}a_{n-2}\dots a_1a_0)_{10} = 10^{n-1}\cdot a_{n-1} + 10^{n-2}\cdot a_{n-2} + \dots 10^1\cdot + a_1 + 10^0\cdot a_0$$

+	0	1
0	0	1
1	1	10

(a) Addizione

×	0	1
0	0	0
1	0	1

(b) Moltiplicazione

Table 1: Tabella additiva e moltiplicativa in base 2

Abbiamo familiarità con il fatto che il massimo numero rappresentabile con n cifre decimali sia $10^n - 1$: ad esempio, il massimo numero rappresentabile con 4 cifre decimali è $10^4 - 1 = 10000 - 1 = 9999$ Similmente, osservando il significato della codifica binaria nell'equazione 2.1, si vede come il massimo numero naturale rappresentabile con n cifre binarie sia $2^n - 1$. É possibile utilizzare tutti gli algoritmi per il calcolo a precisione multipla che conoscete anche in codifica binaria, avendo l'accortezza di memorizzare le "tabelline" per la codifica binaria riportate in tabella 1.

Il passaggio da decimale a binario può essere fatto con il consueto algoritmo delle divisioni successive: è sufficiente continuare a dividere per 2 il numero da trasformare annotando il valore dei resti delle successive divisioni in ordine inverso. La sequenza di resti è la codifica binaria del numero.

2.2 Numeri relativi ($\mathbb{Z}$)

Ci sono più strategie possibili per rappresentare gli interi con segno in un calcolatore:

1. Modulo e segno: in questo caso il numero è rappresentato su n bit, usandone $n-1$ per il valore assoluto e uno per il segno (convenzionalmente il bit di segno a 1 rappresenta il $-$, a 0 il $+$). I numeri rappresentabili vanno da $-(2^{n-1} - 1)$ a $(2^{n-1} - 1)$.

    - Vantaggi:
        - La codifica e decodifica del numero convertendo dal decimale è molto semplice
    - Svantaggi:
        - La gestione delle operazioni aritmetiche è va fatta gestendo esplicitamente tutti i possibili casi di segno degli operandi
        - La rappresentazione dello zero è ridondante : viene rappresentato sia come $+0$ che come -0

2. Complemento a 2: il numero a di n bit viene rappresentato codificandolo, se positivo, in maniera naturale, se negativo codificando $2^n - a$. I numeri rappresentabili vanno da -2^{n-1} a $(2^{n-1} - 1)$.

    - Vantaggi:
        - La codifica fa sì che applicando il comune algoritmo di somma a due numeri in complemento a 2, il risultato *se rappresentabile*, venga calcolato correttamente indipententemente dal segno di entrambi

Codifica binaria	Valore interpretato come ...	
	Modulo e Segno	Complemento a 2
011	3	3
010	2	2
001	1	1
000	0	0
111	-3	-1
110	-2	-2
101	-1	-3
100	-0	-4

Table 2: Tabella esemplificativa della codifica su $n = 3$ bit di numeri relativi.

  - É possibile cambiare di segno un numero in modo molto efficiente con le seguenti osservazioni: i) notare che $2^n - a = 2^n - a - 1 + 1 = ((2^n - 1) - a) + 1$ ii) notare che sottrarre qualcosa a $2^n - 1$, che è fatto di soli 1 significa semplicemente prendere un numero e scambiare gli uni con gli zeri, di conseguenza, per fare $(2^n - 1) - a$ è sufficiente scambiare gli uni con gli zeri e viceversa nella codifica binaria di a. Di conseguenza, il cambio di segno viene effettuato cambiando tutti i bit di a e sommando 1 al risultato.
- Svantaggi:
  - La codifica può sembrare meno "naturale" se osservata direttamente

A conseguenza della maggiore efficienza e semplicità nel meccanizzare il calcolo, tutti i calcolatori moderni codificano i numeri relativi in complemento a 2. In Tabella 2 trovate una comparazione del significato della codifica di interi relativi su 3 bit in modulo e segno/complemento a 2

2.3 Numeri razionali ($\mathbb{Q}$)

L' unico modo di codificare esattamente un numero razionale su un calcolatore è di codificare il suo numeratore e denominatore come due interi relativi. É possibile riusare le codifiche viste in precedenza.

2.4 Numeri reali ($\mathbb{R}$) e complessi ($\mathbb{C}$)

Data la densità[2] dei numeri reali, non è possibile codificare neppure un intervallo finito di essi, e.g. $[0, 1]$ con un numero finito di bit. La soluzione a cui si ricorre è una codifica che introduce necessariamente un' approssimazione nel numero rappresentato. Le strategie per gestire i segni usate in precedenza, possono essere utilizzate con profitto anche con i numeri reali e complessi Due strategie possono essere usate:

1. Codifica a virgola fissa: il numero reale viene codificato su n cifre binarie, di cui i codificano la parte intera e f quella frazionaria. Il metodo

[2]da intendersi come la proprietà per cui tra due numeri reali è sempre presente un terzo numero reale, strettamente compreso tra essi

per trasformare un numero decimale in codifica a virgola fissa è identico a quello usato per i numeri naturali, con la differenza che si arresta il procedimento quando si è ottenuto il numero di cifre desiderato e il resto della divisione è da considerarsi senza la parte frazionaria a ogni passo.

- Vantaggi:
  - L' operazione di codifica è semplice ed è effettuabile in modo efficiente
  - Le operazioni aritmetiche possono essere effettuate dalle stesse unità che fanno quelle per i numeri interi
  - L' errore di codifica è fissato per tutti i numeri ed è minore di 2^{-f}
- Svantaggi:
  - L' intervallo di numeri rappresentabili è piuttosto ridotto: va da 0 a $2^i - 1$, nel caso di numeri senza segno e da $-(2^{i-1} - 1)$ a $2^{i-1} - 1$ per quelli in complemento a 2.

2. Codifica a virgola mobile: il numero viene rappresentato codificando separatamente mantissa su m bit ed esponente su e bit, sostanzialmente in maniera simile alla notazione scientifica. Lo standard internazionale per la codifica dei numeri a virgola mobile prevede che il segno viene codificato su un bit separato.

   - Vantaggi:
     - L' intervallo di numeri codificabile è molto più ampio: va da $-(2^m - 1)^{2^e-1}$ a $(2^m - 1)^{2^e-1}$
   - Svantaggi:
     - Le operazioni aritmetiche devono gestire mantissa ed esponente in modo opportuno, sono necessari algoritmi diversi da quelli per la virgola fissa
     - L' errore di codifica è diverso a seconda della grandezza del numero: intuitivamente, errori di codifica sulla mantissa con esponenti più grandi danno origine a errori assoluti più grandi
     - Problema dell' *underflow*: attorno allo zero è presente una zona non rappresentabile, dovuta alla dimensione finita dell' esponente

A causa dell' intervallo di rappresentazione molto ampio, è comune utilizzare la codifica a virgola mobile, per quanto sia necessaria cura nel controllo degli errori di approssimazione. La codifica a virgola fissa offre un' alternativa molto efficiente nel caso non si abbia particolari necessità di intervallo di rappresentazione alto, o si necessiti assolutamente di un errore di approssimazione indipendente dalla dimensione del numero.

2.5 Testo

Nel caso del testo, il problema è quello di codificare una serie di simboli arbitrari, l' alfabeto, in binario. Una volta codificata ogni simbolo, la codifica delle parole si può ottenere giustapponendo quella dei simboli, esattamente come viene fatto nel testo comune.

Considerato che non c'è nessun vincolo semantico a priori (ovvero, non c'è una buona ragione per rappresentare "A" con $(101010)_2$ per quel che vale), in linea di principio ogni tabella di corrispondenze tra i simboli del testo e dei numeri in codifica binaria è buona. L' uso razionale delle risorse del calcolatore[3] ha imposto due criteri nello scegliere questa codifica: i) minimizzare il numero di bit usati per ogni simbolo, ii) avere uno standard comune.

Il punto di convergenza è lo standard ASCII, che rappresenta ogni carattere stampabile come un intero senza segno in codifica binaria su $n = 7$ bit. Potete reperire una tabella con le corrispondenze a questo link: `https://en.wikipedia.org/wiki/ASCII#ASCII_printable_code_chart`, oppure digitando `man ascii` dal terminale della macchina virtuale che usate per gli esercizi.

Un punto degno di nota dello standard ASCII è che la codifica è stata scelta in maniera tale per cui i numeri che codificano le lettere maiuscole sono tra loro contigui e ordinati come le lettere. Ad esempio, il carattere `A` è codificato da $(65)_{10} = (1000001)_2$, `B` da $(66)_{10} = (1000010)_2$ e così via. Lo stesso vale anche per le lettere minuscole, avendo cura di ricordare che la codifica della `a` è $(97)_{10} = (1100001)_2$.

Lo standard ASCII consente anche di codificare i caratteri che rappresentano le cifre di un numero, quando questo è scritto in un testo. Dal punto di vista della codifica ASCII, il carattere `0` viene trattato esattamente come ogni altro carattere, e.g. `a`, e dunque assegnato a un numero intero su 7 bit ($(48)_{10}$, per la precisione). Abbiate cura di non confondere la codifica di una serie di caratteri rappresentanti le cifre di un numero in un testo, con la codifica di quel numero naturale.

Lo standard ASCII ha incontrato alcune limitazioni al momento di rappresentare alfabeti molto ampi, e diversi da quello latino: diverse proposte sono state adottate, tra cui la più comune è UTF-8 che mantiene la stessa codifica per i caratteri stampabili di ASCII, e utilizza più bit per rappresentare i restanti caratteri (fino a 16).

3 Codifica di segnali

La necessità pratica di rappresentare fenomeni naturali in un calcolatore (ad esempio con lo scopo di effettuare esperimenti simulati), richiede di trovare una codifica a valori che sono di per loro natura espressi con numeri reali o complessi. Considerato che è necessario rappresentarli con un numero finito di bit, si avrà una perdita di informazione in questa codifica.

Allo scopo di acquisire misure di una grandezza naturale si utilizzano dispositivi che convertono questa grandezza in una differenza di potenziale, detti sensori. Un esempio molto comune è un microfono piezoelettrico (come quello contenuto in un cellulare), che converte la pressione istantanea esercitata su di esso dall' onda sonora, in una differenza di potenziale. La conversione in segnale elettrico rende possibile la successiva acquisizione del segnale. Restano due problemi per poter acquisire il segnale: esso è continuo nel tempo e (sostanzialmente) continuo nella differenza di potenziale. Dovendo rappresentare l' entità del segnale con un numero finito di bit, così come l' istante temporale in cui esso è stato acquisito, il segnale viene *campionato* e *quantizzato*. Il processo di

[3] in fondo, ogni bit diventerà un pezzo di metallo percorso da corrente, non vale la pena di sprecarli

campionamento prevede il fatto che l' entità del segnale sia memorizzata con una cadenza regolare, detta periodo di campionamento, come un numero, detto appunto campione. L' inverso del periodo di campionamento è detto frequenza di campionamento. La quantizzazione prevede che l' entità del segnale sia rappresentata con un numero finito di bit (tipicamente come un intero con segno). Esempio pratico: quasi tutti i telefoni cellulare campionano il segnale del microfono a 44100 Hz, salvando l' entità del segnale come un intero con segno in complemento a 2 di 16 bit. Quantificare l' effettiva perdita di informazione dovuta a campionamento e quantizzazione è l' oggetto di studio della teoria dei segnali. Il componente elettronico che si occupa di quantizzazione e campionamento (tipicamente è un unico componente integrato) è detto convertitore analogico-digitale (o Analog-to-Digital Converter, ADC).

Una volta campionato e quantizzato, il segnale può essere trattato come una sequenza numerica, e su di esso possono essere effettuate le trasformazioni desiderate. Dopo la sua elaborazione, il segnale elaborato può essere riprodotto, per mezzo di un convertitore digitale-analogico (digital-to-analog converter, o DAC) e un opportuno apparato. Ad esempio, il segnale campionato e quantizzato da un cellulare può essere riprodotto dall' altoparlante del cellulare stesso, che è collegato a un DAC.

3.1 Rappresentazione delle immagini

Sono possibili due approcci, completamente diversi per la loro natura, per rappresentare immagini in un calcolatore.

Formato vettoriale Una immagine in formato vettoriale viene rappresentata in un calcolatore dandone una descrizione sintetica delle forme. Ad esempio, è possibile rappresentare una circonferenza memorizzandone le coordinate dell' origine e la lunghezza del raggio, in un opportuno sistema di riferimento, come 3 numeri, codificati con la precisione desiderata. Il vantaggio della memorizzazione in formato vettoriale è che si ha una rappresentazione esatta della forma desiderata, a meno della precisione dei parametri della stessa. Considerando l' esempio, la circonferenza rappresentata in formato vettoriale ci consente di calcolare la posizione di quanti punti vogliamo di essa, e con la precisione che preferiamo (compatibilmente con le risorse di calcolo). Lo svantaggio principale di un formato vettoriale è che, con rare eccezioni, mal si applica alla rappresentazione di immagini provenienti da fenomeni naturali (e.g., fotografie). In generale, infatti, è difficile andare a derivare una descrizione sintetica (e.g. un'equazione) che descriva la forma esatta di ogni elemento di una fotografia.

Formato raster La rappresentazione raster di un'immagine utilizza la stessa strategia usata per segnali quali il suono, tenendo però conto del fatto che un'immagine è bidimensionale. Dovendola rappresentare come una sequenza di valori è necessario dapprima partizionare l' immagine in una serie di elementi, detti comunemente *pixels* (contrazione di *picture elements*), di prassi quadrati o rettangolari, all' interno dei quali si assume il colore sia costante. Le dimensioni di un immagine sono tipicamente espresse in termini della sua lunghezza e larghezza in numero di pixels. Nel caso si voglia andare a riprodurre un'immagine digitalizzata a grandezza naturale, la sola informazione legata alle

sue dimensioni non è sufficiente: il numero di pixel per riga o colonna infatti non indica a quale dimensione reale corrisponde l' elemento dell' immagine. A questo scopo, si indica con *risoluzione* il numero di pixel per unità di spazio: se i pixel sono quadrati, un' unica misura della risoluzione è sufficiente. Nella pratica, la risoluzione è comunemente espressa in termini di punti per pollice (dots per inch), o punti per centimetro. La suddivisione di un immagine in pixels e il considerare il colore uniforme all' interno del pixel sono a tutti gli effetti la generalizzazione a due dimensioni dei processi di campionamento e quantizzazione.

Allo scopo di rappresentare il contenuto di un pixel, si utilizza una opportuna convenzione per rappresentare i colori (tipicamente salvando 3 valori interi corrispondenti a quanto è intensa la componente rossa, verde e blu) oppure l'intensità luminosa del singolo pixel (tipicamente nel caso di immagini in scala di grigi, con un solo valore).

Scelto il modo di rappresentare il contenuto del pixel, si sceglie una strategia di scansione dell' immagine, ovverosia un ordine in cui le rappresentazioni dei pixel verranno salvate. Ad esempio, un ordine possibile è quello dato dal partire dall' angolo in alto a sinistra, salvando i pixel sulla stessa riga in modo consecutivo. Terminata la prima riga, si riprende salvando da sinistra a destra i pixel della seconda, e così via fino al termine dell' immagine.

4 Cenni di architettura del calcolatore e sistemi operativi

Questa sezione fornisce un sunto della struttura dell' architettura di un calcolatore e del sistema operativo. Nella descrizione sono state operate semplificazioni rispetto alle architetture moderne, senza alterare i principi di funzionamento generali.

4.1 Unità di misura

Considerato che l' informazione nei calcolatori è manipolata e stoccata in codifica binaria, l' unità fondamentale di misura della quantità di informazione è la singola cifra binaria, il **bi**nary digi**t** [b]. Per ragioni di praticità (= un solo bit è poco per codificare un dato significativo) è comune in informatica utilizzare un pacchetto di 8 bit alla volta: con un gioco di parole, 8 bit sono chiamati *byte*[4] [B]. Meno comune, ma a volte usato, è il `nibble`, un insieme di 4 bit contigui. I prefissi e suffissi standard del sistema internazionale (k,M,G,T...) si applicano sia a bit che a byte, ottenendo i multipli opportuni, e.g. un megabyte [MB] è una quantità di informazione pari a 10^6B.

Una nota di carattere pratico è l' osservazione che $2^{10} = 1024 \approx 10^3$, che ci consente di dire che 2^{10}B$\approx$ 1 kB. L' errore causato da questa approssimazione diventa via via più significativo al crescere delle dimensioni dei numeri, fino a toccare il 10% approssimando 2^{40} con 10^{12}.

Per facilitare dunque i calcoli sono stati standardizzati i prefissi per indicare esplicitamente che si stanno indicando potenze di 2 e non di potenze di 10 con un prefisso. I prefissi nuovi aggiungono una "i" minuscola a quelli classici, pertanto

[4]bit significa piccolo morso in inglese, bite morso vero e proprio

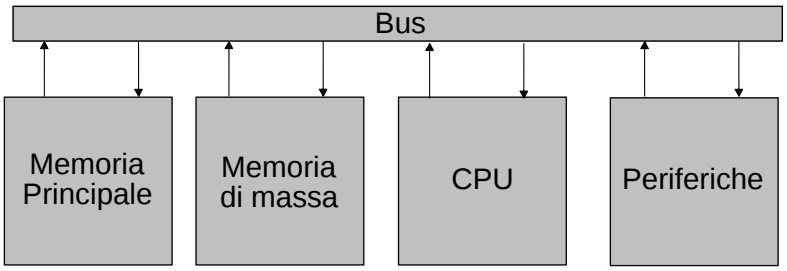

Figure 2: Un semplice schema di una macchina calcolatrice universale con architettura di Von Neumann

un kiB corrisponde a 1024 B, un MiB a 2^{20} B e così via. É utile ricordare che essere in grado di scrivere la prima potenza di 2 che maggiora la quantità che si vuole rappresentare in binario consente di calcolare facilmente il numero di bit necessari per farlo.

4.2 Macchina di Von Neumann

I moderni calcolatori hanno come modello fondamentale quello proposto da Von Neumann (1945): nonostante la loro significativa evoluzione, resta ancora valido per comprenderne il funzionamento. Il modello in questione, schematizzato in Figura 2 è formato da quattro componenti fondamentali comunicanti tramite un un canale comune, il bus (praticamente implementato con degli opportuni cablaggi). A causa delle limitazioni dovute alla presenza di un solo bus, i calcolatori moderni sono tipicamente caratterizzati da più bus, uno per ogni scopo (e.g. trasportare dati, inviare comandi) in modo da evitare l' eccessivo traffico su uno solo di essi. Allo stesso modo, le periferiche sono collegate tramite bus appositi alla macchina, ad esempio l' Universal Serial Bus, o USB.

I componenti del modello di von Neumann sono:

- Central Processing Unit (CPU) o processore: Dispositivo che effettua operazioni aritmetiche, logiche e controlla il comportamento della macchina
- Memoria principale: memoria contenente il codice (= rappresentazione binaria delle istruzioni date alla macchina) del programma da eseguire e i dati ad esso relativi
- Memoria di massa: memoria per lo stoccaggio a lungo termine di dati, non è possibile eseguire direttamente programmi da qui
- Periferiche: dispositivi di supporto per inserire/emettere dati da/verso l' utilizzatore (e.g. tastiera, schermo, microfono)

Analizziamo nel dettaglio i componenti.

Memoria Centrale E' il componente che stocca i dati e i programmi correntemente in esecuzione da parte del calcolatore: la tecnologia con cui è realizzata consente di leggerla e scriverla accedendoci in un qualunque ordine (= anche non in sequenza): viene spesso detta memoria ad accesso casuale o memoria RAM (= Random Access Memory). Fa eccezione una piccola parte di essa, che è solo

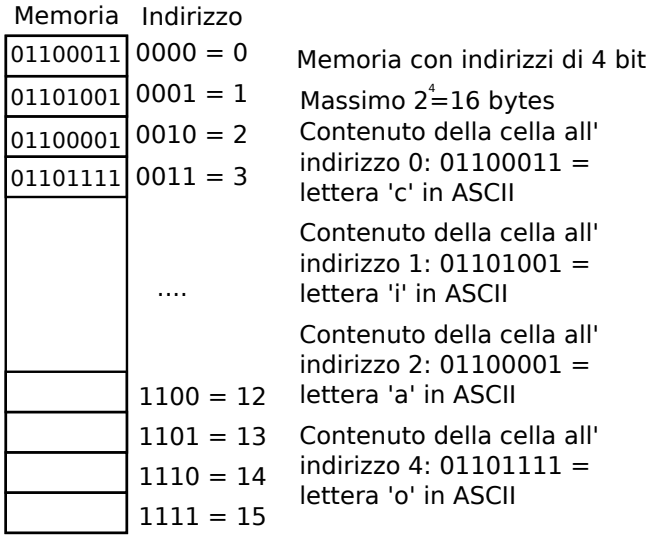

Figure 3: Schema esemplificativo di indirizzamento della memoria principale, con indirizzi da 4 bit

leggibile, e contiene il programma di bootstrap: il primo che il calcolatore esegue per portarsi in uno stato funzionante (= inizializzare le varie componenti). Non venendo scritta di consueto questa piccola porzione è nota come memoria ROM (= Read Only Memory), mentre il programma di inizializzazione è noto come Basic Input/Output System (BIOS).

Per accede alla memoria principale, la si considera logicamente come una sequenza di bytes consecutivi, caratterizzati da un' etichetta numerica, l'*indirizzo*. Tutti i bytes contenuti in memoria hanno quindi una posizione indicata dal loro indirizzo, rappresentato come un numero naturale[5] e, chiaramente, codificato in binario all' interno della macchina.

Essendo necessario avere un indirizzo per ogni byte di memoria principale, dobbiamo scegliere su quanti bit codificare gli indirizzi. Questa scelta limita la quantità massima di memoria a cui possiamo accedere: un indirizzo da k bit $\rightarrow$ ci consente di indirizzare al massimo 2^k bytes di memoria (p.e., indirizzi a 32 bit $\rightarrow$ massimo $2^{32} = 4$ GiB di memoria). La Figura 3 riporta un esempio di memoria da 16 bytes, indirizzata con indirizzi a 4 bit.

Da un punto di vista tecnologico, le attuali memorie principali sono realizzate in modo tale da consentire un tempo d' accesso piuttosto basso, $\sim 60 - 100$ns. Da un punto di vista della capienza, è comune avere da 4 a 32 GiB in un calcolatore nuovo per uso personale e da 512 MiB a 2 GiB a bordo di un cellulare. Un punto chiave della attuale memoria principale è che essa è *volatile*: nel momento in cui si spegne il calcolatore, il suo contenuto viene perso velocemente (in meno di un secondo a temperatura ambiente, il freddo può allungare i tempi a qualche decina di secondi). Per i lettori più interessati: la tipologia più comune è realizzata tramite un circuito integrato, contenente elementi in grado di immagazzinare carica elettrica i.e. dei condensatori di dimensioni molto ridotte. A causa della costante tendenza a scaricarsi di questi, la RAM viene "rinfrescata

[5] in gergo, si dice che la memoria è indirizzata al byte

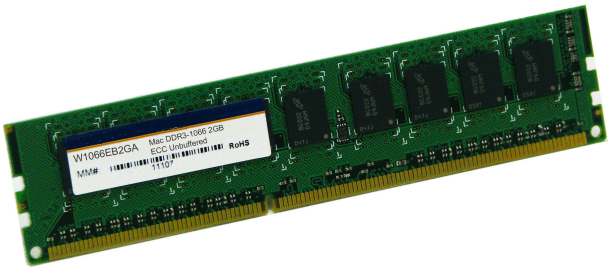

Figure 4: Esempio di memoria RAM moderna: le parti nere sono le capsule plastiche contenenti i circuiti integrati, la parte verde è un supporto meccanico che integra i contatti per poterli collegare al bus

periodicamente" da un circuito apposito, è pertanto detta Dynamic RAM, o DRAM. In Figura 4 è riportata un banco di DRAM di quelli che potete trovare nei vostri calcolatori fissi (è simile a quella di un laptop, semplicemente la parte di adattatore meccanico è più grossa).

Central Processing Unit (CPU). La CPU è il componente effettivamente attivo dell' intero calcolatore, ovvero quello che effettua i calcoli interpretando una sequenza di istruzioni codificate in binario, presenti in memoria principale. In Figura 5 è presente uno schema sommario dell' architettura di una CPU, di cui andiamo ad analizzare i componenti. Il primo componente da esaminare è l' insieme dei *registri* ad uso generale (General Purpose Registers o GPRs): si tratta di un insieme di piccoli componenti di memoria (la loro dimensione tipica è di 4 B o 8 B e il loro numero varia da 4 a 32), che vengono utilizzati dalla CPU per contenere i valori su cui sta effettuando delle elaborazioni (e.g. operazioni aritmetiche). Essi sono tipicamente contraddistinti da un nome simbolico, e.g., `GPR0`, `GPR1`, ... `GPRn`, che viene rappresentato nella macchina con un numero intero. Il secondo componente da prendere in considerazione è l'*Unità Aritmetico-Logica (ALU)*: questa è la porzione del processore dedita effettivamente ai calcoli: può leggere e scrivere il contenuto dei GPR per ottenere i valori su cui effettuare i calcoli e stoccare il risultato. L' attività della CPU è regolata dall' unità di controllo (UC): essa è il componente che pilota le attività della ALU, e richiede caricamenti e salvataggi di dati da memoria, nonchè interpreta l' esecuzione corrente. Per consentire questo, l' UC considera il contenuto di alcuni registri speciali:

- Il registro istruzioni (Instruction Register, IR): esso contiene la rappresentazione binaria dell' istruzione che deve essere eseguita correntemente. Per i lettori interessati: un' istruzione è codificata utilizzando un insieme di bit per descrivere cosa deve fare (e.g. una somma), e i restanti per descrivere su quali registri deve andare ad operare. la porzione di bit dedicata a descrivere la natura dell' istruzione è comunemente detta *codice operativo*, o *opcode*, mentre i registri che usa sono detti operandi dell' istruzione.
- Il contatore di programma (Program Counter, PC): esso contiene l' indirizzo di memoria dove si trova la prossima istruzione da eseguire

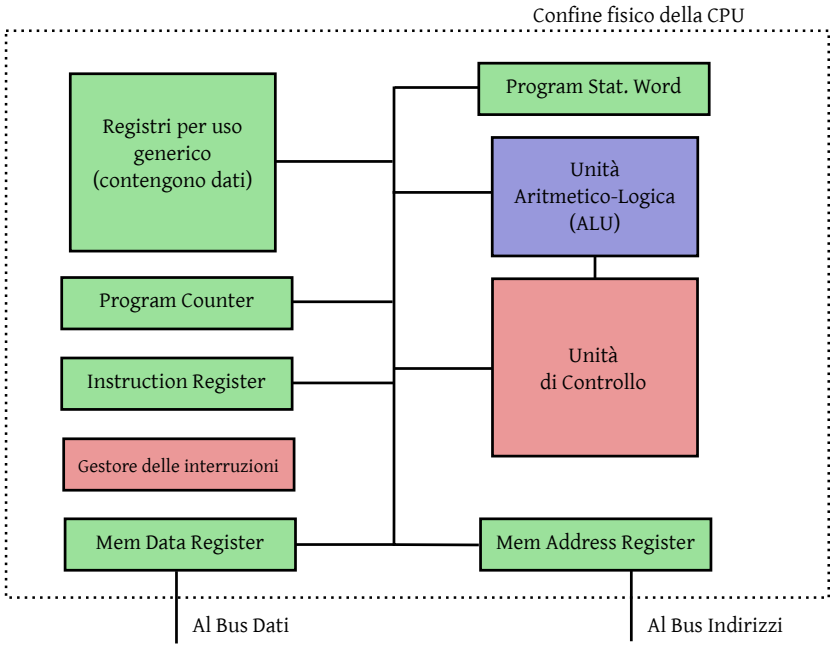

Figure 5: Schema semplificato di una CPU: in verde gli elementi di memoria al suo interno, in rosso quelli che si occupano di regolamentarne il funzionamento, in blu quelli che effettuano la computazione.

- La parola di stato del programma (Program Status Word, PSW): questo registro particolare è da considerarsi come un insieme di registri da 1 bit, che memorizzano le proprietà del risultato dell' ultima istruzione eseguita. Ad esempio, uno dei bit della PSW viene posto a 1 nel caso il risultato precedente sia nullo, ed è detto pertanto *zero-bit*. Analogamente, un altro bit segnala se il risultato dell' ultima istruzione eseguita è negativo.

- Il registro indirizzi e il registro dati verso la memoria (Memory Address/-Data Register, MAR/MDR): questi due registri contengono il valore dell' indirizzo in cui un operazione di memoria deve essere effettuata e il dato corrispondente. Nel caso di una lettura (caricamento) da memoria, il MAR contiene l' indirizzo da cui leggere, l' MDR, alla fine dell' operazione di lettura, conterrà il dato letto. Nel caso di una scrittura, il MAR contiene l' indirizzo a cui scrivere (salvare) il dato contenuto nell' MDR. Al termine dell' operazione di scrittura, il dato nell' MDR sarà stato scritto in memoria all' indirizzo richiesto (sovrascrivendo il dato precedente).

L' unità di controllo regola il funzionamento del processore ripetendo le seguenti 3 fasi all' infinito:

1. **Fetch.** Nella fase di fetch (=recupero), l' UC richiede il caricamento dell' istruzione che si trova in memoria all' indirizzo contenuto nel PC, e una volta che esso è caricato (=è materializzato nel MDR), lo copia nell' IR. A seguito di questo, incrementa il valore di PC in modo che contenga l' indirizzo dell' istruzione successiva.

2. **Decode.** L' UC analizza il contenuto dell' IR, decodifica quindi il tipo di istruzione da compiere, e i suoi operandi.

3. **Execute.** L' UC segnala all' ALU di compiere l' operazione decodificata, segnalando ai GPR che devono essere letti/scritti opportunamente

Le istruzioni eseguite possono essere suddivise in 3 macrocategorie:

1. Istruzioni computazionali: questa categoria di istruzioni comprende tutte quelle che effettuano un calcolo aritmetico o operazioni logiche su operandi e ne salvano il risultato: e.g. moltiplicare il contenuto di un GPR, per quello di un altro e salvare il risultato in un terzo.

2. Istruzioni di accesso alla memoria: queste istruzioni hanno come scopo il caricare un dato da elaborare dalla sua locazione in memoria a un registro ad uso generale, oppure salvare il contenuto di un registro in memoria.

3. Istruzioni di salto: queste istruzioni hanno come unico effetto il modificare il valore del PC. L' effetto ai morsetti sull' esecuzione è quello di cambiare quindi quale sarà la prossima istruzione ad essere eseguita, causando un "salto" in quella che è la naturale sequenza di esecuzione. Un salto in avanti (= a valori maggiori) rispetto al valore corrente del PC causa evita di eseguire un blocco di istruzioni, mentre un salto all' indietro causa la ripetizione delle istruzioni tra il nuovo valore del PC e il vecchio. Il cambio del PC può essere effettuato sia incondizionatamente (l' operazione è detta di salto incondizionato), oppure basandosi su uno o più valori dei bit della PSW. In questo secondo caso si parla di salto condizionato (*branch*), e l' istruzione viene utilizzata per far prendere alla computazione un percorso diverso a seconda del valore del risultato dell' ultima computazione. É possibile, usando opportunamente un salto condizionato, saltare a una certa porzione dell' esecuzione solo se due valori sono uguali, semplicemente effettuando una sottrazione tra loro due e saltando solo se il bit zero della PSW è acceso.

L' ultima porzione della CPU che osserviamo è il sistema di gestione delle interruzioni hardware: esso consente al processore di interrompere l' esecuzione corrente nel momento in cui è segnalata la necessità di attenzione da parte di un evento esterno (e.g. la pressione di un tasto sulla tastiera). Per fare questo, è presente all' interno del processore una porzione di hardware che contiene le posizioni in memoria del codice da eseguire a seconda del tipo di interruzione avvenuta. Nel momento in cui avviene l' interruzione, il processore salta ad eseguire questo codice, detto *gestore dell' interruzione* o *interrupt handler*. Tipicamente questo codice salva lo stato corrente della computazione (valori nei registri, indirizzo del codice dove è arrivata la computazione interrotta), gestisce l' interruzione (e.g., legge il carattere dalla tastiera e lo salva in memoria principale) e ripristina l' esecuzione della computazione precedente. Per mezzo del meccanismo delle interruzioni hardware è quindi possibile per la CPU rispondere alle esigenze delle periferiche in modo tempestivo e coordinare la comunicazione con esse. Un ulteriore sorgente di interruzioni hardware è data dall' orologio di sistema: questo fornisce interruzioni periodiche per consentire alla CPU di avere una cognizione dello scorrere del tempo.

Figure 6: Esempio di disco elettromeccanico:in evidenza i piatti nichelati (3 in foto) e le testine sulla cui estremità è montato il solenoide di lettura/scrittura (non visibile per ragioni di dimensioni)

Periferiche Le periferiche sono componenti del calcolatore che sono, in linea di principio, non indispensabili al procedere della computazione, ma offrono la possibilità di immettere dati all' utente e di visualizzare dati contenuti nel calcolatore. Esempi classici di periferiche sono: mouse, tastiere, monitor, stampanti, attuatori per la domotica. Esse sono collegate a uno o più bus dedicati verso il processore, e sono in grado di segnalare la loro attività tramite cavi dedicati al sistema di gestione delle interruzioni hardware. Tipicamente contengono una piccola quantità di memoria da cui il processore può leggere o in cui può scrivere. Un esempio tipico è la tastiera, che contiene una piccola memoria da qualche byte in cui vengono stoccati i caratteri, codificati in ASCII, corrispondenti ai tasti premuti, fino a quando il processore non ne recupera il contenuto. Un altro esempio è la stampante, che contiene una memoria dove vengono inviati i valori dei dati che devono poi essere stampati su carta. In generale, l' interazione con le periferiche è sempre mediata dal processore, il quale si fa carico di caricare i dati da loro prodotti (gli input dati alla macchina) in memoria principale e copiare dalla memoria principale nelle loro memorie gli output. Per i lettori più curiosi: i calcolatori moderni sono in realtà caratterizzati da alcuni processori "aiutanti" che si fanno carico di effettuare queste copie autonomamente, a fronte di un solo comando di inizio della copia da parte della CPU. Questo tipo di sistema viene detto Direct Memory Access, o DMA.

Memoria di Massa La memoria di massa è il componente che stocca i dati a lungo termine, ma non può essere usata direttamente dalla CPU per i programmi e i dati in uso. Essa è permanente, non viene cancellata quando il calcolatore è spento e, tipicamente è circa 1000 volte più grossa della memoria principale. La tecnologia per realizzarla varia a seconda delle dimensioni, e delle tempistiche di accesso richieste ad essa; al momento attuale le tecnologie prevalenti sono 2:

- **Supporti magnetici.** La memoria di massa magnetica si basa sull' idea di magnetizzare una piccola porzione di materiale ferromagnetico, usando la direzione del campo per indicare il valore del bit. Il suo formato più comune è quello dei dischi fissi elettromeccanici: si tratta di dischi come quello riportato in Figura 6 che vengono letti e scritti con un piccolo elettromagnete posto in cima a una testina di lettura/scrittura. La loro capacità varia tra 1 e 6 TiB correntemente, e il tempo per accedere a un dato va da 2 a 10 ms a seconda della posizione della testina. L' alternativa con capienza più alta è quella dei nastri magnetici: si tratta di nastri di plastica simili alle vecchie musicassette, ricoperti di ferrite. La ferrite sul nastro viene magnetizzata ed essi possono essere letti sequenzialmente. La capacità dei nastri magnetici supera gli 8 TiB per cassetta, ma i tempi di accesso possono raggiungere le decine di secondi nel caso il dato si trovi alla fine del nastro.

- **Memorie di massa a stato solido (solid state disks, SSD).** Si tratta di circuiti integrati, in grado di conservare una carica per tempi molto lunghi (anni). Possono essere collegati alla macchina attraverso lo stesso bus dei dischi elettromeccanici, oppure attraverso il bus dati USB (le cosiddette chiavette). La loro capacità è minore dei dischi elettromeccanici, ha di recente raggiunto il TiB, in compenso il tempo di accesso si aggira nell' ordine delle decine/centinaia di μs per accessi casuali ai dati, circa 100 volte più veloce dei supporti elettromeccanici.

L' accesso fisico alla memoria di massa avviene con le stesse modalità dell' accesso alle memorie delle periferiche in quanto, in origine i calcolatori non erano dotati di memoria di massa, e andavano riinizializzati da capo ad ogni accensione.

4.3 Il sistema operativo

Fino ad ora abbiamo considerato il caso in cui la macchina sta eseguendo un solo programma alla volta. Questo scenario è effettivamente quello che avviene in piccoli calcolatori dedicati (e.g. quello a bordo di una macchina fotografica), ma nella maggior parte dei casi desideriamo eseguire più di un programma "contemporaneamente" su un calcolatore. A questo scopo è sono stati creati i *sistemi operativi*: l' idea chiave di un sistema operativo è quella di gestire direttamente l' hardware, offrendo a tutti gli altri programmi, detti *applicazioni*, devono essere eseguiti l' illusione di avere la macchina completamente per sè. Di conseguenza, sulla stragrande maggioranza dei calcolatori, è il sistema operativo ad avere il controllo delle operazioni della macchina e a decidere cosa viene eseguito, e quando. In particolare, il codice che va a gestire le interruzioni è parte del sistema operativo.

In particolare, le attività fondamentali del sistema operativo sono:

- *Gestione e protezione della memoria principale*: Allo scopo di evitare interferenze tra programmi diversi, il sistema operativo gestisce l' accesso alla memoria (direttamente, o per mezzo di un componente hardware di supporto, la Memory Management Unit (MMU)). L' effetto netto è che ogni applicazione vede uno spazio di indirizzamento proprio, contiguo, e indipendente da cosa stiano facendo gli altri, mentre il sistema operativo,

o l' MMU si occupano di tradurre gli indirizzi di memoria del programma, detti *indirizzi virtuali*, negli indirizzi fisici in memoria di massa. L' intero processo è trasparente dal punto di vista dell' applicazione, mentre consente al sistema operativo di isolare le zone di memoria fisica acceduta dai vari programmi, consentendo di applicare opportune prassi di sicurezza. Il sistema operativo è anche incaricato di caricare i programmi dalla memoria di massa in memoria principale, nel momento in cui devono iniziare la loro esecuzione, e di rimuoverli una volta che la loro esecuzione è terminata.

- *Scheduling*: Il sistema operativo simula l' esecuzione contemporanea di più programmi, eseguendoli per una limitata porzione di tempo a testa (meccanismo di *time-sharing*). A questo scopo, il sistema operativo conserva una lista di tutti i programmi che sono correntemente che devono essere eseguiti a turno, del punto dove è giunta la loro computazione (indirizzo in memoria dell' istruzione), e i valori dei registri della CPU in quell' esecuzione. Nel momento in cui viene segnalata un'interruzione dall' orologio di sistema, il gestore, che è parte del sistema operativo, viene eseguito e decide a chi tocca l' esecuzione per il prossimo quanto di tempo, tenendo in conto il fatto che nessun programma deve restare fermo a tempo indeterminato (*fairness* dello scheduling). Una volta scelto quale programma deve essere posto in esecuzione, il sistema operativo carica il suo contesto di esecuzione (il contenuto dei registri della CPU) e salta all' istruzione che deve essere eseguita di quel programma. Nel caso, come nelle moderne piattaforme, sia presente più di una CPU, operante in parallelo, il sistema operativo decide, per ognuna di esse, quale applicazione deve essere eseguita.

- *Gestione del filesystem*: Per offrire un accesso alla memoria di massa più organizzato e meno prono a conflitti tra diverse applicazioni dell' accesso diretto, il sistema operativo offre ad esse l' astrazione di filesystem.

- *Gestione delle periferiche*: L' inizializzazione delle periferiche in uno stato stabile (e.g. accendere il video, lasciandolo completamente nero all' inizio, accendere i led sulla tastiera) e la loro gestione durante il funzionamento (leggere/scrivere dati da/verso le loro memorie interne) è un' attività che ha in carico il sistema operativo. I dati e, se necessario, la possibilità di interagire con le periferiche, vengono poi offerti secondo un' opportuna astrazione (ad esempio dei files) alle applicazioni.

Un applicazione può richiedere servizi al sistema operativo per mezzo dell' esecuzione di una *chiamata di sistema* o *syscall*. Una syscall, concretamente, è una porzione del codice del sistema operativo che può essere invocata da un programma affinchè effettui per esso operazioni non direttamente consentite. L' esempio più tipico di syscall è quella che viene effettuata per leggere i caratteri che un utente sta fornendo tramite una tastiera. Al momento dell' esecuzione della syscall, il sistema operativo svolge la funzione per cui è stato invocato e, se il quanto di tempo non è terminato, rimette in esecuzione l' applicazione.

4.4 Filesystem

L' astrazione più comune con cui viene offerto l' accesso alle memorie di massa è quella di *filesystem*. Un filesystem è un modo per organizzare i dati su una memoria di massa che assegna a gruppi di zone della memoria di massa, detti *files*, un nome simbolico che li identifica univocamente (e.g. mio_documento.txt). In pratica un filesystem consiste di un elenco di nomi di files, e delle posizioni dei blocchi di dati sulla memoria di massa. Il gestore del filesystem, parte del sistema operativo, offre alle applicazioni la possibilità di richiedere i dati presenti in un file, utilizzando il suo nome simbolico. Indipendentemente dalla posizione dei blocchi di dati sul disco, il file è presentato all' applicazione come una sequenza contigua di bytes, all' interno della quale l' applicazione può scrivere nella posizione che preferisce. L' accesso al file è effettuato in maniera simile a un nastro: il file viene letto sequenzialmente per mezzo di un cursore che può essere spostato a seconda delle necessità dell' applicazione.

Con il crescere del numero di files nei moderni filesystem, si è reso necessario offrire una forma di organizzazione migliore del solo nome. A questo scopo, i filesystem moderni offrono un' organizzazione gerarchica dei files in *directory*. Le directory si comportano come contenitori e possono contenere gruppi di files o altre directory a loro volta. Questo fa sì che si possa convenientemente rappresentare un intero filesystem sotto forma di un albero di directory e files, simile ad un albero genealogico, dove la relazione padre-figlio tra due elementi indica che il figlio è contenuto all' interno del padre. Il vantaggio più evidente, oltre alla capacità di suddividere i files per aree semantiche (una directory per i file contenenti musica, una per le immagini, una per i programmi ...), è la possibilità di avere files con lo stesso nome, a patto che risiedano in directory diverse. Pertando l' identificativo unico per accedere a un file non è più il solo nome, ma il suo *percorso* (o *pathname*) ovvero una codifica testuale dell' intero percorso per giungere al file, a partire dalla radice dell' albero genealogico. Per rappresentare testualmente il passaggio padre-figlio si usa un carattere eletto a separatore di percorso (*path separator*). La scelta del path separator dipende dal sistema operativo, le più comuni sono

- / in sistemi UNIX-like (Linux, MacOS, FreeBSD)
- \in tutti i sistemi Windows

In sistemi UNIX like esiste un solo albero, la cui radice è codificata in testo con un singolo /, mentre in tutti i sistemi Windows possono esserci più radici, tipicamente una per ogni periferica di massa, e sono identificate da lettere maiuscole, seguite da due punti e un \. Ad esempio, la codifica della radice a cui è assegnata la lettera `C` è C:\.

5 Introduzione alla programmazione

Un linguaggio di programmazione è un insieme di regole per poter esprimere sotto forma di testo (codificato nella macchina in ASCII nel nostro caso), un algoritmo. In maniera simile a quello che accade per i linguaggi naturali, anche un linguaggio di programmazione è caratterizzato da tre livelli a cui si può interpretarlo:

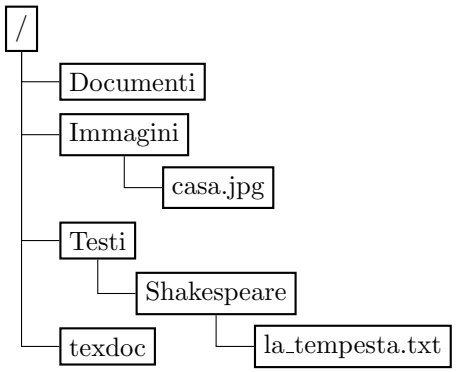

Figure 7: Esempio di filesystem

1. **Lessico.** Il lessico di un linguaggio è l' insieme di regole che caratterizzano la forma delle sue parole, o *lessemi*. Un modo banale di esprimere queste regole è un vocabolario, ovvero una lista di tutte le parole valide del linguaggio. Per contro, è possibile anche esprimere in forma sintetica alcune delle regole che caratterizzano tutti i lessemi. Ad esempio, per l' italiano, tutti i lessemi che finiscono in -gia, con i atona, hanno il plurale che termina in -gie se la g è preceduta da vocale, in -ge se preceduta da consonante. Un esempio di errore lessicale è omettere un accento su una parola che ne necessita. É possibile correggere errori lessicali automaticamente in modo piuttosto efficiente (e.g. il correttore ortografico a corredo di programmi di videoscrittura).

2. **Sintassi.** La sintassi di un linguaggio è l' insieme di regole che stabiliscono come i lessemi vanno combinati per produrre una frase (e.g. quali preposizioni precedano un complemento, l' ordine di soggetto e complemento oggetto in alcune lingue). Il punto fondamentale della sintassi di un linguaggio di programmazione è che essa non deve essere *ambigua*, ovvero, una sequenza di lessemi non deve avere due interpretazioni valide. Per contro, l' ambiguità è un carattere comune dei linguaggi naturali: in italiano la frase "Luigi ha visto un uomo nel parco con il telescopio" presenta un' ambiguità sintattica per cui sono valide due interpretazioni. L' individuazione di errori di sintassi in una frase di un linguaggio, posto che essa non sia ambigua, è fattibile in modo efficiente.

3. **Semantica.** La semantica di un linguaggio è l' insieme di regole che associano a ogni frase un significato, tipicamente come composizione dei significati dei singoli lessemi. L' individuare errori semantici in un testo espresso in un linguaggio di programmazione non è possibile automaticamente nel caso più generale. La conseguenza diretta è che non esiste un modo automatico di dire se un generico programma è corretto o meno (è possibile farlo per alcune classi di programmi).

Una parte di questo corso si occupa di insegnarvi lessico e sintassi dei linguaggi C e Fortran, nonchè la semantica di elementi di base di essi. Comprendere il modo di combinare gli elementi di base in programmi con il valore semantico che volete, ovvero che facciano fare al calcolatore quello che desiderate, è la

seconda parte di questo corso. La combinazione di queste due parti fa sì che siate in grado di programmare, ovverosia di tradurre in un linguaggio di programmazione un algoritmo di vostra concezione.
Per questo scopo, vi serve seguire una serie di passaggi:

0. **Concepire l' algoritmo.** É il passo iniziale in cui inquadrate il problema e lo descrivete a voi stessi, anche informalmente, raffinandone la descrizione fino al punto in cui è suddiviso in operazioni elementari, decisioni, ed eventualmente ripetizioni delle stesse.

1. **Codificare l' algoritmo.** A questo punto potete usare un programma applicativo che vi consenta di scrivere del testo ASCII (e.g. un qualsiasi simil-notepad) per codificare l' algoritmo in un programma nel linguaggio di programmazione che preferite. Il testo ottenuto è comunemente chiamato *codice del programma*.

2. **Compilare il programma.** Il testo da voi prodotto e salvato in un file viene dato in ingresso a un programma, il *compilatore*, il quale ne verifica la correttezza lessicale e sintattica, dopodichè lo traduce in una codifica binaria che è quella che la macchina è in grado di interpretare sia per le istruzioni, che per i dati. Il file risultante da questa traduzione viene detto comunemente *binario eseguibile* o *codice oggetto*.

3. **Avviare il programma.** Una volta ottenuto il binario eseguibile potete dire al vostro calcolatore di iniziare effettivamente la sua esecuzione. Il sistema operativo si farà carico di leggerlo dalla memoria di massa, caricarlo in memoria principale e iniziare ad eseguire i suoi contenuti.

Fatto questo, se la concezione e la codifica del programma sono corrette, avete prodotto il programma che volevate. Purtroppo, non sempre il tutto va come si deve, di conseguenza è necessario avere un modo di ispezionare il funzionamento di un programma mentre sta girando. Per questo sono disponibili dei programmi, detti *debugger*, che prendono in ingresso un *binario eseguibile* e consentono di controllare la sua esecuzione passo passo interattivamente. Ad esempio è possibile interromperla quando esegue un certo passaggio e stampare i valori in memoria. Va notato che è possibile utilizzare un debugger su un qualunque oggetto binario a vostra disposizione: potete quindi esaminare anche i programmi applicativi già presenti sulla vostra memoria di massa, anche se le loro dimensioni non banali potrebbero rendere la comprensione completa un po' faticosa.

6 Il linguaggio C

Analizziamo ora il linguaggio C, comprendendo come codificare degli algoritmi con esso. Nel linguaggio C alcuni dei lessemi hanno una forma fissata, mentre per altri ci sono solo regole che devono essere rispettate affinchè siano validi. I lessemi con una forma unica vengono detti *parole chiave* del linguaggio o *keywords*: essi sono *riservati*, ovvero non possono essere usati per scopi che non siano quello dichiarato. in questo testo le keyword compariranno scritte in `questo modo` , mentre il resto del codice C comparirà in nero `così`.

Il primo punto da comprendere del linguaggio C è che un programma è descritto da un insieme di frasi, o *statements*. Il delimitatore che indica che una di queste è terminata è il carattere punto e virgola `;` in modo da fornire al compilatore un modo univoco per distinguere dove esse terminano (la codifica ASCII del carattere "a-capo" è possibile usando sia il carattere numero 10, che il 14, che entrambi in sequenza). Il corrispettivo di un paragrafo in C è un insieme di statements delimitato da due parentesi graffe { }. Vedremo in seguito qual è la semantica del racchiudere un insieme di statements all' interno di un blocco delimitato da graffe.

In prima istanza, un algoritmo viene rappresentato come un programma scritto in C come un blocco di statements racchiuso tra graffe, e preceduto da quella che viene detta la *firma* o *signature* del programma. La signature indica cosa il programma riceve come input e cosa restituisce come output a chi lo invoca, tipicamente il sistema operativo che lo pone in esecuzione. Allo scopo di comprendere la sintassi di un primo, semplice programma, inizieremo ad analizzare le sue componenti.

In prima battuta, gli statements in C possono essere divisi in due categorie principali: dichiarazione di variabili, e statements di programma. Allo scopo di consentire delle annotazioni all' interno di un programma in C, è possibile specificare dei *commenti* all' interno del codice dato in ingresso al compilatore. Sono possibili due diverse sintassi per indicare i commenti:

- Il testo racchiuso tra un marcatore `/*` (inizio commento) e un marcatore `*/` (fine commento) viene ignorato dal compilatore integralmente. I marcatori possono trovarsi anche su righe diverse
- Il testo compreso tra `//` e la fine della riga (= fino al primo a-capo) viene ignorato dal compilatore

6.1 Dichiarazioni di variabili e tipi di dato base, costanti

Allo scopo di rappresentare in C un algoritmo, la prima domanda da porsi è su quali variabili esso deve agire, e conoscere come esse vengono rappresentate in C. Questo è necessario in quanto il compilatore, nel tradurre il programma in un formato comprensibile dalla macchina, avrà necessità di sapere quali variabili, e di quale tipo, dovranno essere stoccate in memoria principale durante la computazione.

Le variabili con struttura più semplice in C sono singoli valori appartenenti ai cosiddetti *tipi base* del linguaggio. La sintassi di uno statement di dichiarazione di una variabile di un tipo base ha la seguente sintassi

```
nome_del_tipo nome_della_variabile ;
```

In C, i nomi di variabile devono rispettare le seguenti regole:

- Il nome di una variabile è lungo almeno un carattere e non ha lunghezza massima
- Il nome di una variabile deve iniziare con una lettera maiuscola o minuscola, oppure con il carattere _
- Il nome di una variabile può contenere, nei rimanenti caratteri, lettere maiuscole, minuscole, caratteri numerici e il carattere _

Esempi di nomi di variabile validi sono `var`, `_test`, `CaMmElLo`, `NCC_1701`, `_project_2501`. Non sono validi `99bottles` (inizia con un numero), `-segno` (contiene il carattere `-`, che non è consentito), `età` (contiene `à`, non consentito).

Il tipo di dato in C influenza come la variabile verrà trattata durante i calcoli e rappresentata in memoria. I tipi di dato base in C sono

- Intero (`int`): La variabile contiene come un intero, con segno, rappresentato su almeno 32 bit. Un esempio di dichiarazione di variabile intera è `int numero_di_arance;` É possibile ottenere interi rappresentati con meno bit (secondo lo standard non più di `int`) aggiungendo la keyword `short` prima di `int` nella dichiarazione. Simmetricamente, è possibile richiedere interi rappresentati con più bit attraverso la keyword `long`.

- Numero rappresentato a virgola mobile `float`: La variabile contiene un numero con segno, rappresentato in notazione a virgola mobile, su almeno 32 bit (di prassi 8 per l' esponente, 24 per la mantissa e 1 per il segno della mantissa). É possibile ottenere variabili in virgola mobile rappresentate su più di 64 bit (53 per la mantissa 11 per l' esponente, 1 per il segno) utilizzando la keyword `double` al posto di `float`. Se necessario, si possono ottenere variabili con ancora più precisione dichiarandole come `long double`.

- Carattere `char`: La variabile è un singolo carattere in codifica ascii. La dimensione della variabile è garantita essere fissa a 1 byte (8 bit), mentre il segno non è specificato dallo standard (dipende dunque da implementazione a implementazione del compilatore).

É possibile richiedere esplicitamente alla macchina di rappresentare una variabile intera come priva di segno aggiungendo all' inizio della dichiarazione di variabile la keyword `unsigned`. Aggiungendo `unsigned` all' inizio di una dichiarazione di una variabile di tipo `char`, è possibile forzare il programma a considerare la rappresentazione binaria del carattere come rappresentazione di un naturale, nel caso serva. Non è possibile avere variabili a virgola mobile prive di segno in C.

Nel caso si renda necessario dichiarare più variabili dello stesso tipo, è possibile farlo con una dichiarazione sola, con la sintassi:

```
nome_del_tipo nome_var1, nome_var2, nome_var3, nome_var4;
```

Ad esempio, `int x,y,z;` dichiara 3 variabili intere chiamate `x`,`y` e `z`.
In C è possibile assegnare un valore iniziale alla variabile dichiarata. Per fare questo è sufficiente aggiungere il valore in questione secondo la sintassi:

```
nome_del_tipo nome_var1 = valore_iniziale;
```

Il valore iniziale è espresso secondo una sintassi che consente alla macchina di comprendere di cosa si tratti in particolare:

- Per le costanti intere, è sufficiente scrivere il numero, con segno laddove serva. Ad esempio `int a=137;` dichiara una variabile `a` e pone il suo valore iniziale a 137.

- Per le costanti a virgola mobile è necessario mettere in evidenza che il numero specificato è a virgola mobile: questo viene fatto indicando uno 0 come prima cifra dopo la virgola, nel caso serva. Il separatore decimale in C è il *punto* (convenzione americana). Come esempio `int`

`temperatura=20.0;` dichiara una variabile `temperatura` e la inizializza a 20.

- Per i caratteri ASCII, è necessario indicare al linguaggio che il carattere in questione è da leggersi come una costante e non come un nome di variabile. Per questo, la sintassi del C impone di racchiudere tra due caratteri `'` il carattere in questione. Ad esempio `char iniziale = 'A';` dichiara una variabile chiamata `iniziale` il cui valore iniziale è la lettera maiuscola `A`. Per alcuni caratteri ASCII, i quali non hanno una rappresentazione visiva efficace (ad esempio il carattere a-capo) è possibile indicare la costante carattere con quel valore attraverso le cosiddette *escape-sequences*. In particolare, vi saranno di utilità :

  - `\n`: il carattere a-capo
  - `\t`: il carattere tabulazione, che introduce uno spazio bianco "elastico", ovvero in grado di riempire al più 8 caratteri. Se stampato prima di un testo fa sì che esso sia allineato.
  - `\0`: il carattere che ha come rappresentazione nella codifica ascii il valore zero. Nota bene: questo carattere ***NON*** è il carattere `'0'`, il quale è rappresentato dal valore 48 in binario.
  - `\\`: è il carattere \, il quale necessita di essere scritto in modo non ambiguo con l' inizio di una escape-sequence.

  Potete trovare una tabella completa della rappresentazione ASCII incluse le escape sequences del C, digitando `man ascii` dal vostro interprete di comandi testuale nella macchina virtuale.

6.2 Operatori

In C tutte le operazioni su variabili sono simboleggiate da operatori, in modo affine a come viene fatto in matematica. Un insieme di variabili e operatori ben costruito è un' *espressione* in C. Intuitivamente, un' espressione ben costruita associa variabili e operatori insieme in modo che sia possibile calcolarla in modo non ambiguo. Ad esempio `a+b` è un' espressione ben formata (indica, in C, che la macchina deve leggere i contenuti di a e b e sommarli), `a+-` non lo è. Gli operatori agiscono sulle variabili con una semantica che dipende anche dai loro tipi: l' esempio più classico è quello delle operazioni di divisione, le quali agiscono in modo diverso su variabili intere o su variabili a virgola mobile.

Gli operatori del C che compiono le comuni espressioni aritmetiche, con le quali siete già familiari, sono:

- `+` e `-` : sono la comune addizione e sottrazione tra 2 elementi. `-` viene usato anche per indicare il meno unario, per intenderci quello di `-47`.

- `*` e `/` : rappresentano la moltiplicazione e divisione. Le consuete regole di precedenza tra addizione/sottrazione e moltiplicazione/divisione sono rispettate in C. L' operazione di divisione tra due variabili intere fornisce un risultato intero, ottenuto tramite il troncamento di quello frazionario che si otterrebbe. Ad esempio `5/3` dà come risultato 1, mentre `9/10` restituisce 0.

- `%` : è l' operatore resto, restituisce il resto della divisione intera tra le due variabili. es. `7%3` dà come risultato 1.

- Le parentesi tonde `(` e `)` si comportano come le comuni parentesi tonde delle espressioni aritmetiche, ovverosia consentono di forzare una precedenza ben precisa nel calcolo delle espressioni. Nota bene: sono *solo* le parentesi tonde a dover essere utilizzate per questo scopo, le parentesi quadre e graffe in C hanno una semantica completamente diversa. Esempio di espressione con parentesi `(1+(2+3))*6` dà 36 come risultato.

Tutti gli operatori aritmetici si comportano secondo le comuni regole dell' aritmetica con le variabili intere e con quelle a virgola mobile. Nel caso usiate operatori aritmetici con variabili di tipo carattere, essi vengono considerati come l' intero che li rappresenta secondo la codifica ASCII. L' uso principale di questo è convertire un carattere ASCII rappresentante una cifra, ad esempio `'3'` nell' intero corrispondente. A questo scopo, osservate che le codifiche ASCII dei caratteri corrispondenti alle cifre sono tra loro consecutivi (ovvero la codifica di zero `'0'` precede quella di `'1'`, quella di `'1'` a sua volta precede quella di `'2'` e così via). Grazie a questa accortezza del codice ASCII, sottrarre il valore `'0'` alla codifica di un carattere dà come risultato il valore intero corrispondente, e.g. `'9'-'0'` è l' intero 9.

Tra gli operatori del C, uno di essi serve ad effettuare il salvataggio in una variabile del risultato di un espressione. L' operatore di assegnamento è `=` e ha la semantica: calcola l' espressione alla destra dell' operatore, considerando i valori correnti delle variabili in memoria, e salva il risultato nella variabile a sinistra. Pertanto, lo statement:

```
a = a+1;
```

carica il valore di `a` da memoria, somma ad esso 1 e salva il risultato ottenuto all' interno della stessa variabile `a`. In particolare, è importante notare che in questo caso `=` non ha nessuna connotazione di uguaglianza algebrica o numerica. É possibile, per ragioni favorire la compattezza del codice, scrivere un assegnamento che salva nella stessa variabile il risultato di un' operazione compiuta su di essa usando gli operatori `+=`, `-=`, `*=`, `/=`. Ad esempio, il precedente statement può essere riscritto come:

```
a += 1;
```

mantenendo la stessa semantica.

Oltre agli operatori aritmetici citati finora, il linguaggio C contiene anche altri operatori: operatori di confronto, operatori logici, operatori bitwise, l' operatore ternario e l' operatore di typecast , l'operatore sizeof e gli operatori di shift.

Gli operatori di confronto effettuano, per l' appunto, un confronto tra i valori contenuti in due variabili, dando come risultato 1 nel caso il confronto sia vero, 0 se è falso. Gli operatori di confronto offerti dal linguaggio C sono:

- `==` confronta due variabili, restituendo 1 se sono uguali, 0 se diverse

- `!=` confronta due variabili, restituendo 1 se sono diverse, 0 se uguali

- `>` confronta due variabili, restituendo 1 se sono la prima è strettamente maggiore della seconda, 0 in caso contrario

- `<` confronta due variabili, restituendo 1 se sono la prima è strettamente minore della seconda, 0 in caso contrario

- `>=` confronta due variabili, restituendo 1 se sono la prima è maggiore o uguale alla seconda, 0 in caso contrario
- `<=` confronta due variabili, restituendo 1 se sono la prima è minore o uguale alla seconda, 0 in caso contrario

Gli operatori logici effettuano le operazioni tra i valori delle variabili loro operandi trasformandole in valori logici, ovvero "vero" e "falso". La convenzione per la trasformazione è: una variabile non nulla ($\neq 0$) rappresenta il valore logico "vero", una nulla il valore logico "falso". Il risultato di un operatore logico è a sua volta un valore logico rappresentato come $0 \rightarrow$ "falso" e $1 \rightarrow$ "vero". Gli operatori logici disponibili sono i seguenti:

- `&&` congiunzione logica o `and`. L' espressione logica `a && b` è vera se e solo se sia `a` che `b` contengono un valore "vero". La sua semantica è quella del connettivo "e" nel linguaggio naturale.
- `||` disgiunzione logica inclusiva o `or`. L' espressione logica `a || b` è vera se almeno una tra `a` e `b` contiene un valore "vero". La sua semantica è quella del connettivo "o" nel linguaggio naturale, con l' accezione che entrambi gli eventi veri danno una frase vera, simile al connettivo "vel" del latino.
- `!` negazione o `not`. L' espressione logica `!a` è vera se `a` contiene un valore "falso". La sua semantica è quella del connettivo "non" nel linguaggio naturale.

Nota bene: è possibile ottenere la disgiunzione logica esclusiva o `xor` utilizzando semplicemente l' operatore di confronto `!=`. L' espressione logica `a != b` infatti vera se almeno una tra `a` e `b` contiene un valore "vero", il che corrisponde al fatto che i valori logici di `a` e `b` siano diversi. La sua semantica è quella del connettivo "o" nel linguaggio naturale, con l' accezione che entrambi gli eventi veri danno una frase falsa, simile al connettivo "aut ... aut" del latino.

Gli operatori bitwise del C consentono di applicare operazioni logiche ai singoli bit contenuti all' interno delle variabili, effettuando un' operazione logica per ogni coppia dei bit delle variabili in input. La convenzione di rappresentazione è la consueta $1 \rightarrow$ "vero" , $0 \rightarrow$ "falso". Il principio è simile a quello delle "operazioni in colonna" fatte a mano su carta, con la differenza che l' operazione in questione non è una somma, ma quella specificata dall' operatore bitwise. Il linguaggio C ha i seguenti operatori bitwise:

- `&` congiunzione logica o `and`. Il risultato dell' operazione è dato dall' and dei bit delle due variabili. L' effetto netto è che i bit del primo operando corrispondenti
- `|` disgiunzione logica inclusiva o `or`. Il risultato dell' operazione è dato dall' or inclusivo dei bit delle due variabili.
- `~` negazione o `not`. Il risultato è dato dal contenuto della variabile in ingresso con tutti i bit cambiati di valore: dove era presente uno 0, ora è presente un 1.
- `^` disgiunzione logica esclusiva o `xor`. Il risultato dell' operazione è dato dall' or esclusivo dei bit delle due variabili. Notate che `a^ a` dà sempre 0.

6.3 Iniziare a programmare

Dopo aver definito qual è la sintassi del linguaggio C per dichiarare variabili e manipolarle attraverso operatori, è possibile iniziare ad approcciare la traduzione in C di un algoritmo. A questo scopo, è necessario sapere come il C rappresenta l' inizio La sintassi usata dal C per descrivere l' inizio e la fine di un algoritmo è la seguente:

```
int main(int argc, char* argv[]){
/* Qui viene inserita la traduzione in C del vostro algoritmo
    */
}
```

La semantica data alla sintassi precedente è questa : le parentesi graffe hanno in C con lo scopo di racchiudere un insieme di statements. L' algoritmo principale in C prende sempre il nome fissato `main`, e per convenzione produce un risultato intero, indicato dalla keyword `int` preposta al nome. Si dice, in gergo, che l' algoritmo *ritorna* un intero. La lista di variabili, con relativo tipo, specificata tra parentesi tonde, descrive i parametri che il programma si vede consegnare da chi lo invoca: nel caso del `main`, è il sistema operativo, che si occupa di caricare il programma, che prepara il parametro intero `argc`, contenente il numero di parametri passati, e la struttura dati `argv`, che contiene la rappresentazione testuale dei parametri, in un formato che analizzeremo in seguito.

Fondamentale per il corretto funzionamento di un programma è rispettare la promessa del tipo da ritornare. Nel caso del `main`, siamo obbligati a restituire a chi ci ha chiamato (ovvero al sistema operativo) un intero, che, convenzionalmente, è posto a 0 per significare "il programma ha terminato correttamente la sua esecuzione" e un qualunque valore non nullo, per indicare una terminazione abnormale. Questa è semplicemente una convenzione, nulla vi vieta di scrivere un programma che restituisce sempre 42, per quel che vale. Per restituire un valore al chiamante, e dunque interrompere l' esecuzione del programma, si utilizza uno statement con la keyword `return`.

Ad esempio, il seguente codice è un main completo e valido che, all' atto dell' esecuzione, non fa nulla, se non ritornare 0.

```
int main(int argc, char* argv[]){
   return 0;
}
```

Realizziamo, ad esempio, un semplice programma che calcola l' area di un triangolo:

```
int main(int argc, char* argv[]){
  float base=3.0, altezza=5.2;
  float area;
  area=base*altezza/2;
  return 0;
}
```

Potete provare a trascrivere il programma e ispezionarne il corretto funzionamento con `gdb`.

6.4 Vettori o array

Allo scopo di ottenere una maggiore espressività, il linguaggio C consente, oltre alla dichiarazione di variabili appartenenti a un tipo base, di utilizzare variabili più complesse.

Il primo tipo aggregato del C che trattiamo è il cosiddetto *vettore* o *array*: si tratta di un tipo di variabile che viene rappresentato come una sequenza di elementi, tutti uguali tra loro in memoria. Il numero di elementi è fissato e deve essere noto al momento della compilazione (ovvero, deve essere una costante, o un' espressione calcolabile da costanti). La sintassi per dichiarare una variabile di tipo array in C è la seguente:

```
nome_tipo nome_variabile[numero_di_elementi];
```

Ad esempio, `int classifica[10];` dichiara una variabile chiamata classifica, di tipo array di interi, da 10 elementi. Gli elementi di un array sono disposti consecutivamente nella memoria principale del calcolatore e la distanza tra l' indirizzo del primo byte di uno di essi e il successivo viene detta *stride*, o *passo* del vettore, ogni elemento è anche detto *cella* dell' array.
Per accedere ad un elemento del vettore, è possibile utilizzare l' operatore `[]` dopo il nome della variabile, indicando tra le quadre l' *indice* della cella a cui accedere. L' indice di un vettore è sempre un numero naturale, pertanto la numerazione delle celle parte da 0. Data quindi la dichiarazione di un vettore di interi `int a[10];` l' espressione `a[0]` indica la variabile intera contenuta nella prima cella del vettore, `a[2]` la terza, `a[9]` l' ultima. É possibile usare le espressioni che fanno riferimento a una cella di un vettore all' interno di espressioni più complesse: l' operatore `[]` ha priorità più alta di tutti quelli visti finora. Di conseguenza, assumendo che sia stato dichiarato un vettore `int num[10];` lo statement:

```
num[3]=num[0]+2*num[2];
```

fa sì che la macchina legga il contenuto della prima cella di `num` e gli somma il doppio del contenuto nella terza cella, per poi scrivere il risultato nella quarta.

L' uso tipico di vettori di numeri interi e a virgola mobile è quello di rappresentare collezioni di valori numerici, ad esempio l' elenco dei pesi di diversi veicoli. I vettori di caratteri giocano un ruolo particolare in C: essi sono il modo di rappresentare un testo, codificato in ASCII, all' interno del calcolatore. Ad esempio è possibile rappresentare l' Amleto di Shakespeare[6] come `char hamlet[180849];` all' interno del calcolatore. Nell' approfondimento dello studio del linguaggio faremo spesso uso di una convenzione del C sulla rappresentazione del testo: i vettori di caratteri che rappresentano blocchi contigui di testo stampabile hanno come ultimo carattere del testo `'\0'`. Il ruolo del carattere nullo è quello di terminatore del testo stampabile: viene dunque interpretato da tutte le parti del linguaggio che trattano con la stampa dei testi come la fine di quello che va stampato a video.

In aggiunta agli array come descritti finora, il C offre la possibilità di dichiarare dei cosiddetti *array multidimensionali*. L' array multidimensionale è la naturale

[6] in particolare, questo `https://www.gutenberg.org/cache/epub/1524/pg1524.txt`

estensione dell' array monodimensionale, in cui i valori in memoria sono indicizzati come se fossero disposti su una matrice a più dimensioni. Iniziamo con l' analizzare un array bidimensionale, la sintassi per la dichiarazione è la seguente:

```
nome_tipo nome_variabile[lunghezza_dim1][lunghezza_dim2];
```

Ad esempio lo statement:

```
int altitudine[100][100];
```

dichiara un vettore bidimensionale di interi, contenente 100 righe di interi, ognuna da 100 elementi. L' accesso ad un elemento è possibile secondo la stessa prassi con cui si accede a un vettore bidimensionale: ad esempio `altitudine[3][5]` rappresenta l' intero che si trova nella quarta riga, sesta colonna del vettore bidimensionale. A prima vista, l' idea di un vettore bidimensionale non è compatibile con il fatto che la memoria sia una lista di celle con un solo indirizzo numerico (mentre qui si usano due indici per l' accesso). Quello che accade in realtà è che il vettore viene *linearizzato*[7] in memoria, ovvero il C attua una strategia per impacchettare gli elementi di questa griglia di interi in memoria. Nel caso del C, si dice che l' array è linearizzato per righe, ovvero in memoria vengono salvati per primi, consecutivamente, tutti gli elementi della prima riga, ovvero da `altitudine[0][0]` ad `altitudine[0][99]`, poi quelli della seconda `altitudine[1][0]` – `altitudine[1][99]`, e così via. Fortunatamente, non dovete occuparvi di calcolare dove si trovi esattamente in memoria un elemento, l' operatore `[]` farà il lavoro per voi. Potete dichiarare vettori con un numero arbitrario di dimensioni, il C non ha una limitazione formale su di esse.

Una caratteristica fondamentale dei vettori in C è quella di *non* essere cosiddetti "cittadini di prima classe", ovvero, di non essere utilizzabili direttamente all' interno di espressioni. In pratica questo si concretizza nel fatto che, nella seguente porzione di codice:

```
int a[10],b[10];
a=a+b;
b=a;
```

lo statement nella seconda riga ***NON*** somma elemento per elemento i vettori, nè tantomeno salva il risultato nel vettore `a`. Similmente, l' ultimo statement ***NON*** copia il vettore `b` in `a`.

L' unico caso in cui è possibile assegnare tutti i valori di un vettore in un solo statement di assegnamento è al momento della dichiarazione della variabile. Il linguaggio C consente infatti di usare la seguente sintassi:

```
tipo_di_dato nome[dimensione]={costante,costante, costante,
    costante};
```

ovvero, consente di specificare il valore iniziale di tutte le celle di un vettore come una lista di costanti separate da virgola, e racchiuse tra graffe. Ad esempio lo statement:

```
float useful_numbers[4]={3.141,1.412, 1.054, 42.0};
```

dichiara un vettore di float e lo inizializza. Nel caso particolare si voglia inizializzare il contenuto di tutte le celle con lo stesso valore è possibile indicare lo stesso, da solo, tra graffe. Ad esempio:

[7] Questo termine ha una triste ambiguità semantica con il concetto di linearizzazione proprio dell' analisi.

```
int numeri[10]={0};
```

inizializza a 0 tutte le celle del vettore `numeri`.

In maniera affine è possibile inizializzare vettori multidimensionali: la sintassi è quella di insiemi di insiemi, ovvero

```
float useful_numbers[2][2]={ {3.141,1.412}, {1.054, 42.0} };
int other_numbers[2][2][2]={ {{1,2},{3,4}} , {{5,6},{7,8}} };
```

mentre le inizializzazioni con un valore unico diventano:

```
float useful_numbers[2][2]={{37.0}};
int other_numbers[2][2][2]={{{0}}};
```

É presente anche una sola eccezione al formato per l' inizializzazione al momento della dichiarazione di vettori: si tratta del caso di vettori di caratteri che possono essere inizializzati con la sintassi che fa uso di dei doppi apici " " , ovvero:

```
char nome[11]="Alessandro";
```

La semantica della dichiarazione precedente fa sì che il vettore da 11 caratteri nome sia riempito con i caratteri presenti tra gli apici, seguiti dal terminatore \0.

6.5 Strutture

In aggiunta a vettori mono- e multidimensionali, il linguaggio C consente anche di creare dati strutturati come aggregati di dati eterogenei. Questo è possibile grazie al concetto di *struttura*: una struttura è un aggregato di variabili di tipo diverso, dette campi, a cui è possibile accedere separatamente. Per utilizzare le strutture in C è necessario dapprima descrivere cosa contiene la struttura con la seguente sintassi:

```
struct etichetta_struttura {
  tipo nome_campo;
  ...
  tipo nome_campo;
};
```

ovvero si tratta di una lista di statement che descrivono i campi, racchiusi tra graffe e preceduti dalla parola chiave `struct` e dall' etichetta del tipo di struttura, o *tag* della struttura. Questa descrizione ha la stessa funzione di una planimetria di un edificio: descrive che forma deve avere una volta costruito. Un esempio di dichiarazione di struttura, è la seguente, pensata come elemento di una rubrica telefonica:

```
struct persona{
  char nome[30];
  char cognome[30];
  int eta;
  char numero[14];
}
```

Per dichiarare una variabile di tipo struttura, una volta specificata la descrizione della struttura, è sufficiente usare la parola chiave `struct`, seguita dal tag della struttura come tipo, all' interno della comune dichiarazione di variabili. Ad esempio:

```
struct persona pers1;
struct persona rubrica[20];
```

sono due dichiarazioni valide, la prima di una variabile struttura chiamata `pers1`, la seconda di un vettore di strutture, chiamato rubrica, contenente 20 strutture. Le strutture sono cittadini di prima classe in C, di conseguenza l' operatore di assegnamento funziona come atteso; il seguente codice:

```
struct persona p1,p3;
p1=p3;
```

copia, campo per campo, il contenuto della struttura `p1` nella struttura `p3`. Tutti i campi della struttura sono disposti in maniera contigua in memoria al momento dell' allocazione. Per accedere ad un campo della struttura si utilizza l' operatore `.` seguito dal nome del campo; il tipo del campo è quello dichiarato nella descrizione della struttura. Un esempio di uso è il frammento di codice seguente:

```
struct persona p1;
p1.eta= p1.eta+1;
```

in questo codice, il valore del campo `eta` della struttura `p1` viene letto, viene sommato 1 al suo contenuto e il risultato viene salvato nello stesso campo, sovrascrivendo il vecchio valore, ovvero facendo invecchiare la persona di un anno. Gli operatori `[]` e `.` hanno priorità più alta di tutti gli altri operatori aritmetico/logici in C, di conseguenza vengono calcolati prima che avvengano i calcoli aritmetici, con il risultato netto che è possibile scrivere il seguente frammento di codice:

```
struct persona rubrica[20];
rubrica[0].nome[0]='A';
rubrica[1].eta=rubrica[1].eta+2;
```

per cambiare la prima lettera del nome della prima persona nel vettore di strutture che funge da rubrica, e far invecchiare di due anni la seconda persona contenuta in essa.

6.6 Tipi personalizzati

Allo scopo di incrementare la leggibilità dei programmi, il linguaggio C offre la possibilità di definire dei propri tipi di dato personalizzati. In pratica si tratta di una caratteristica di cosiddetto *zucchero sintattico* (*syntactic sugar*), ovvero, una caratteristica del linguaggio che consente di tradurre in C in modo più chiaro lo stesso concetto (la dichiarazione di una variabile con un dato tipo).

Un tipo personalizzato può essere dichiarato con la seguente sintassi:

```
typedef descrizione_del_tipo nome_tipo;
```

di cui un esempio pratico sono le definizioni di tipo personalizzato:

```
typedef float massa;
typedef float accelerazione;
```

che definiscono due tipi di dato personalizzati, `massa` e `accelerazione` come nomi alternativi per il tipo float. Il nome del tipo personalizzato è di vostra scelta, ma, per essere valido, deve rispettare le stesse regole dei nomi di variabile (iniziare con una lettera o con _ e proseguire con lettere, numeri o _). L' effetto

netto è che è possibile, dopo i due statement precedenti, scrivere il seguente codice:

```
massa m1=10.0, m2=20.0;
accelerazione G=9.81;
```

e il compilatore considererà legittime le dichiarazioni delle variabili `m1`, `m2` e `G`, in quanto abbiamo prima definito i tipi `massa` e `accelerazione`.

É possibile anche dichiarare tipi di dato personalizzato formati da array o strutture; nel dettaglio:

```
typedef int classifica_t[20];
typedef struct mia_struttura {
            int x;
            int y;
            char nome[20];
          } punto_t;
```

rappresentano due dichiarazioni di tipi personalizzati: un tipo `classifica_t`, costituito da un vettore di interi, e un tipo `punto_t`, costituito da una struttura a tre campi (due interi per rappresentare le coordinate e un vettore di caratteri per il nome del punto). É buona abitudine utilizzare `_t` come suffisso del nome di un tipo personalizzato, in questo modo è facile distinguere i nomi di tipi dai nomi di variabili. Dopo queste dichiarazioni di tipo è quindi possibile scrivere codice come:

```
classifica_t classifica1;
punto_t p,q,r;
p.x=q.x;
```

Questo codice dichiara una variabile di tipo `classifica_t` (quindi in pratica un vettore da 20 interi) e tre variabili di tipo `punto_t` (ovvero strutture a tre campi come descritto sopra).

A causa del fatto che, una volta definito un tipo personalizzato basato su una struttura, è possibile direttamente direttamente usare il nome del nuovo tipo per le dichiarazioni di variabile, è molto comune omettere del tutto il tag della struttura nella typedef, usando a tutti gli effetti una struttura anonima. La dichiarazione di tipo personalizzato `punto_t` precedente può quindi essere riscritta, senza cambiamento nella semantica, come:

```
typedef struct {
            int x;
            int y;
            char nome[20];
          } punto_t;
```

6.7 Costrutti di selezione: lo statement `if`

I costrutti di selezione consentono di tradurre in C il concetto di "operazioni che devono essere eseguite solo a fronte del verificarsi di una determinata condizione". Il primo e più semplice costrutto di selezione è il costrutto `if`. La sintassi del costrutto `if` è la seguente:

```
if (espressione) {
   codice da eseguire
}
```

La semantica è la seguente: l' espressione all' interno delle parentesi tonde (obbligatorie in questo caso) viene valutata considerandola falsa se uguale a zero o vera altrimenti. Nel caso non si tratti di un espressione intera, questa viene promossa a intero con le consuete regole (troncamento all' unità per i `float`, utilizzo del valore della codifica ASCII come intero per i `char`). Nel caso l' espressione risulti vera, il codice racchiuso tra le graffe viene eseguito, in caso contrario viene saltato. Questo è effettivamente tradotto dal compilatore in termini di un' istruzione di salto condizionato che bypassa il codice risultante dalla traduzione di ciò che c'è tra parentesi. Si dice in gergo tecnico che l' `if` è *preso* se viene eseguito il codice, *non preso* in caso contrario. É possibile non racchiudere il codice da eseguire tra graffe, nel caso si tratti di un singolo statement in C. Questo tuttavia penalizza la leggibilità e dà problemi di manutenibilità del codice, quindi è fortemente sconsigliato (in pratica, mettete sempre le parentesi).

Il linguaggio C offre anche la possibilità di specificare un blocco di codice che deve essere eseguito nel caso l' `if` non sia preso attraverso la seguente sintassi

```
if (espressione) {
   codice da eseguire
} else {
   codice da eseguire
}
```

In questo caso, il blocco di codice che segue la parola chiave `else` viene eseguito solo quando l' `if` non è preso.

L' introduzione dell' `else` porta a un naturale problema di ambiguità sintattica: in particolare, nel caso del codice

```
if (a==1)
if (a==2) a=a*2;
else a=a+1;
```

É possibile interpretarlo sia come

```
if (a==1){
   if (a==2) a=a*2;
} else a=a+1;
```

che come

```
if (a==1) {
   if (a==2) a=a*2;
     else a=a+1;
}
```

Dato che non deve esserci alcuna ambiguità nella sintassi di un linguaggio di programmazione (altrimenti il compilatore dovrebbe decidere arbitrariamente quello che vogliamo cosa non possibile in modo automatico), questa ambiguità, detta del *dangling else* va risolta. La soluzione è accettare per convenzione che l' else dangling si riferisce sempre all' ultimo if, ovvero l' interpretazione corretta è la seconda.

6.8 Cicli: lo statement `do while` e `while`

É molto frequente il dover ripetere una seri di statements in un algoritmo. A questo scopo, il linguaggio C consente di rappresentare la ripetizione di una

serie di statements, fino a quando una data condizione si mantiene vera con i cosiddetti costrutti iterativi o *cicli*.

Il primo e più semplice tra essi è il costrutto `do-while`, che ha la seguente sintassi:

```
do {
  /* qui il codice da ripetere*/
} while ( espressione );
```

Il costrutto do-while ripete il codice all' interno delle parentesi graffe fino a quando l' espressione è vera (ovvero diversa da zero). Comunemente, la porzione di codice tra le graffe è detta *corpo* del ciclo, mentre l' espressione è anche detta *condizione*, in quanto rappresenta la condizione che, verificata, impone la continuazione del ripetersi del codice. Il costrutto do-while esegue sempre almeno una volta il corpo, prima di arrivare a valutare la condizione. I possibili casi per quello che riguarda l' espressione sono rappresentati da:

```
do {
  /* Blocco di codice A*/
} while(42);
```

e da:

```
do {
  /* Blocco di codice B*/
} while(0);
```

Nel primo costrutto do-while, l' espressione è sempre diversa da zero (essendo una costante): la conseguenza è che il primo ciclo ripete l' esecuzione del corpo all' infinito. Un esempio di utilità pratica di un ciclo di questo tipo è il programma che invia le pagine di un sito web via rete, in seguito alle richieste del browser. Questo programma, comunemente detto *server HTTP* ripete in eterno un ciclo in cui si occupa della richiesta correntemente fatta. Il secondo costrutto do-while, per contro, ha una condizione che è sempre falsa (essendo uguale alla costante 0). La conseguenza è che il corpo viene eseguito una sola volta: in pratica il costrutto non ha effetto.

Nel caso non sia necessario eseguire sempre almeno una volta il corpo, il linguaggio C fornisce un secondo tipo di costrutto iterativo, il costrutto while. La sintassi del costrutto while è la seguente:

```
while ( espressione ) {
   /* qui il codice del corpo */
}
```

Il costrutto while effettua come prima operazione il controllo del valore della condizione, eseguendo il corpo del ciclo solo se essa è vera (non nulla). La diretta conseguenza è che, nel caso la condizione sia nulla sin da subito, il corpo del ciclo non viene mai eseguito, a differenza di ciò che accade nel costrutto do-while.

6.9 Cicli: lo statement `for`

I costrutti `while` e `do-while` sono utilizzati per tradurre il concetto di una ripetizione di un blocco di codice per un numero non esplicitamente definito di volte. Tuttavia, è piuttosto frequente, specie quando si vuole ripetere un

compito per tutti gli elementi di un vettore, sapere esattamente quante volte andrà ripetuto un ciclo. Inoltre, in questi casi, è anche necessario mantenere una variabile intera, che verrà usata come indice dell' elemento da estrarre del vettore, o, più in generale per scandire il numero di iterazioni. Allo scopo di facilitare la traduzione di questo concetto, il linguaggio C offre un ulteriore costrutto ciclico, il costrutto `for`. La sintassi del costrutto `for` è la seguente

```
for ( inizializzazione ; condizione ; passo ){
  /* codice da ripetere */
}
```

Nella sintassi del costrutto `for`, l' inizializzazione è uno statement C, così come il passo, mentre la condizione di terminazione è un' espressione. La semantica del costrutto è la seguente: lo statement di inizializzazione viene eseguito una sola volta, prima di iniziare il ciclo; viene quindi controllata la condizione e, se è vera, viene eseguito una volta il corpo. Al termine dell' esecuzione del corpo, il costrutto `for` esegue il passo, quindi controlla la condizione. Se essa è ancora verificata, ripete l' esecuzione del corpo e del passo, altrimenti l' esecuzione passa oltre. L' effetto pratico è la possibilità di riscrivere il seguente frammento di codice

```
int i=0, a[10];
while(i<10){
  a[i]=i;
  i=i+1;
}
```

in maniera più compatta e leggibile come

```
int i, a[10];
for(i=0;i<10;i++){
   a[i]=i;
}
```

Nel costrutto `for` è possibile omettere uno o più degli elementi tra parentesi tonde. Gli effetti sono i seguenti:

- L' omissione dell' inizializzazione fa sì che il ciclo parta senza effettuare nessuna operazione prima del primo controllo della condizione
- L' omissione della condizione di terminazione è equivalente a specificare una condizione costantemente *vera*: il ciclo non termina mai
- L' omissione del passo fa sì che venga ripetuta la sola esecuzione del corpo del ciclo, senza nessuna operazione aggiunta

É comune prassi di programmazione chiamare la variabile che scandisce il numero di iterazioni del `for` con una lettera minuscola dell' alfabeto, a partire da `i`. La variabile è comunemente detta *variabile d' induzione* del ciclo.

6.10 Costrutti di selezione: il costrutto `switch-case`

Una porzione dell' algoritmo da tradurre in C può richiedere di effettuare una selezione in cui il valore dell' espressione che pilota la selezione deve essere valutato in modo più ricco del semplice vero/falso (nullo/non nullo). In questi casi, è possibile utilizzare il costrutto `switch-case`, il quale ha la seguente sintassi:

```
switch (espressione){
case valore1: /*Codice A*/
              /*Ancora Codice A*/
case valore2: /*Codice B*/
case valore3: /*Codice C*/
}
```

La semantica del costrutto `switch-case` è la seguente: dapprima viene calcolato il valore dell' espressione, considerato come un intero (i float vengono troncati, i caratteri considerati come tali). Successivamente, l' esecuzione salta al `case` con il valore corrispondente a quello appena calcolato. L' esecuzione quindi prosegue lungo tutto il codice rimanente valido fino alla fine del blocco racchiuso tra graffe. Questo effetto, detto di *fall-through* attraverso i casi può essere sfruttato opportunamente nel caso in cui gli statement necessari per gestire uno dei casi siano un sottoinsieme stretto di quelli di un altro. Un esempio possibile è dato dal seguente codice che valuta il resto della divisione per 3 del contenuto di una variabile intera, e rende lo stesso divisibile per 3 arrotondando per eccesso:

```
int a;
switch(a%3){
  case 1: a=a+1;
  case 2: a=a+1;
  case 0:
}
```

É possibile, nel caso ci sia la necessità di avere un `case` che comprende tutti i valori non esplicitamente catturati dagli altri case, utilizzare la keyword `default` per indicarlo.

6.11 Deviazioni del flusso d' esecuzione: `break` e `continue`

Il linguaggio C fornisce la possibilità di deviare il flusso di controllo uscendo dal blocco di codice correntemente in esecuzione, sia che si tratti del corpo di un ciclo, sia che si tratti del corpo di un costrutto `switch-case`, attraverso la keyword `break`. La keyword `break` è usata da sola in uno statement che causa l' uscita istantanea dal blocco di codice, ad esempio il seguente ciclo:

```
int a=5,b=1;
while(1){
  if (a<b){
    break;
  }
a=a-1;
}
```

terminerà, nel momento in cui la variabile `a` vale 1 grazie all' effetto dello statement `break`. Notate che è possibile riscrivere il precedente frammento di codice in modo più leggibile:

```
int a=5,b=1;
while(a<b){
a=a-1;
}
```

L' uso del `break` va quindi limitato ai casi di effettiva necessità, in quanto peggiora la leggibilità del codice.

Un caso tipico di utilizzo corretto del `break` è quello di far sì che il codice dei vari `case` di un costrutto `switch-case` siano eseguiti in modo mutuamente esclusivo, avvicinandosi così a un' idea più naturale del costrutto di selezione. Ad esempio, una riscrittura del precedente `switch-case` è:

```
int a;
switch(a%3){
  case 0: break;
  case 1: a=a+2;
          break;
  case 2: a=a+1;
          break;
}
```

L' utilizzo del `break` in questo caso rende sostanzialmente irrilevante l' ordine in cui vengono scritti i vari `case` del costrutto, in quanto solo uno di essi viene effettivamente eseguito.

6.12 Funzioni

La soluzione di problemi complessi viene comunemente affrontata suddividendoli in porzioni più piccole e risolvendo ognuna di esse. Questo concetto viene realizzato nell' algoritmica descrivendo la soluzione a un problema come un algoritmo che a sua volta può fare uso altri algoritmi per costituire parte della propria descrizione. In C questo concetto viene rappresentato tramite il costrutto sintattico detto *funzione*. Una funzione in C rappresenta la codifica di un algoritmo che può essere invocato da altri algoritmi, evitando quindi la necessità di reinventare la ruota da capo ogni volta. A questo scopo, la funzione opera su dei dati forniti in ingresso, detti *parametri* e restituisce un valore prodotto, detto *valore di ritorno*. Sintatticamente, una funzione è divisa in due parti, il suo *prototipo*, o *firma*, e la sua implementazione. Il prototipo di una funzione a due parametri in C ha la seguente sintassi

```
tipo_valore_ritorno nome_funzione (tipo_par1 nome_par1,
    tipo_par2 nome_par2)
```

ovvero, descrive il tipo del valore ritornato (un qualunque tipo valido per il C , inclusi quelli personalizzati), fornisce un nome alla funzione (deve rispettare lo stesso formato del nome delle variabili), ed elenca i *parametri* accettati dalla funzione, preceduti dal loro tipo, come una lista separata da virgole. Ad esempio

```
int somma(int a, int b);
```

dichiara il prototipo di una funzione chiamata `somma`, che accetta 2 parametri interi `a` e `b`, e restituisce un valore intero, che è il risultato della sua computazione. L' intero prototipo è l' identificativo unico per la funzione, di conseguenza le funzioni

```
int somma(int a, int b);
int somma(float a, float b);
```

sono entità diverse in C.

L' implementazione di una funzione è un semplice blocco di codice C (ovvero un insieme di statements racchiuso tra due parentesi graffe) che viene scritto di seguito al prototipo. Nel caso in cui venga fornita l' implementazione della funzione, non è necessario il punto e virgola dopo il blocco di codice della sua implementazione.

All' interno dell' implementazione è possibile utilizzare i parametri, semplicemente facendo riferimento a loro con lo stesso nome con cui sono stati dichiarati nel prototipo. All' interno del prototipo, seguendo la stessa prassi che abbiamo finora seguito per la funzione speciale `main`, possiamo utilizzare la keyword `return` per indicare che l' esecuzione della funzione termina in quel punto e che deve ritornare il valore contenuto nell' espressione che segue `return`.

Una possibile implementazione della funzione somma tra due interi, il cui prototipo è stato dichiarato in precedenza, è la seguente:

```
int somma(int a, int b){
 return a+b;
}
```

Notate che lo statement return si aspetta un' espressione, dello stesso tipo dichiarato come tipo ritornato nel prototipo. L' espressione che segue `return` può anche essere un' espressione banale fatta di una sola variabile, o, al limite anche una costante, a patto che sia del tipo corretto, promesso come ritorno nel prototipo. Ad esempio, la seguente implementazione alternativa della funzione somma:

```
int somma(int a, int b){
  int ris;
  ris = a+b;
  return ris;
}
```

è ugualmente valida.

Una volta implementato il sottoalgoritmo che preferiamo all' interno di una funzione, il passo successivo è essere in grado di usare la funzione. L' uso di una funzione è detto in gergo *chiamata* o *invocazione* e, sintatticamente viene fatto semplicemente indicando il nome della funzione e, tra parentesi, le variabili che le devono essere fornite come parametri. Il seguente frammento di codice, usa la funzione somma

```
int num=20, num2=22,risultato;
risultato = somma(num, num2);
```

La semantica in C della chiamata a funzione è la seguente: come primo passo vengono effettuate **copie** del contenuto delle variabili passate come parametri ad uso unico della funzione. Per maggiore chiarezza, si chiamano *parametri attuali* le variabili di cui viene effettuata una copia ad uso della funzione al momento della chiamata (`num` e `num2` nel nostro caso), mentre si chiamano *parametri formali* quelli usati nella dichiarazione del prototipo della funzione e nella sua implementazione. La ragione del nome *parametri formali* risiede nel fatto che essi sono semplicemente delle "etichette" cieche per le zone di memoria che verranno riempite con la copia dei *parametri attuali* (*actual parameters* in inglese, una migliore traduzione sarebbe *veri parametri*, o *parametri effettivi*) al momento della chiamata. In seguito al passaggio di parametri per copia, il flusso di esecuzione del programma passa al codice della funzione, che viene eseguito.

L' esecuzione della funzione termina nel momento in cui essa raggiunge uno statement `return`, e il valore ritornato viene copiato all' interno dello spazio di memoria utilizzato dal suo chiamante. Questa prassi consente di utilizzare il risultato di una chiamata a funzione direttamente all' interno di una qualunque espressione in C. Nell' esempio precedente, il risultato della funzione somma viene assegnato alla (=copiato nella) variabile `risultato`. Notate che la copia dei parametri della funzione viene effettuata solo per i first class citizens in C (tipi base e strutture). É possibile dichiarare variabili all' interno dell' implementazione di una funzione (del resto, è quello che abbiamo fatto all' interno del `main` finora): lo spazio per esse viene allocato in memoria al momento della chiamata a funzione e viene deallocato nel momento in cui l' esecuzione della funzione termina. Vengono pertanto dette variabili *locali* della funzione.

Vediamo quindi un programma completo, che utilizza una funzione:

```
int mul(int a, int b){
   return a*b;
}

int main(int argc, char* argv[]){
  int i,espon=3, base=4, ris=1;
  for(i=0;i<espon;i++){
   ris=mul(ris,base);
  }
  return ris;
}
```

Questo programma utilizza la funzione `mul` allo scopo di calcolare un' esponenziazione. Dato che il compilatore traduce il C in binario in una singola passata sul sorgente, è necessario che almeno il prototipo della funzione compaia prima della sua chiamata: nell' esempio indicato sia il prototipo che l' implementazione della funzione `mul` apaiono prima del punto in cui viene chiamata all' interno.

Una possibile alternativa è quella di porre il solo prototipo prima, in modo che il compilatore sia in grado di emettere correttamente la porzione del codice che effettua la chiamata a funzione (che al netto risulta essere tradotta nell' allocazione dello spazio per parametri e risultato, nella copia dei parametri e nella sequenza di istruzioni che salta al punto nel codice dove è presente la funzione, previo il salvataggio del punto dove tornare).

Un esempio di programma con dichiarazione del solo prototipo della funzione prima è il seguente:

```
int mul(int a, int b);

int main(int argc, char* argv[]){
  int i,espon=3, base=4, ris=1;
  for(i=0;i<espon;i++){
   ris=mul(ris,base);
  }
  return ris;
}

int mul(int a, int b){
   return a*b;
```

```
}
```

Notate che il compilatore vi segnalerà un errore se tentate di fornire più di un' implementazione per la stessa funzione. Questo è la conseguenza diretta del fatto che un programma con due implementazioni per la stessa funzione è ambiguo: non è possibile decidere quale delle due deve essere chiamata.

Nel caso una funzione accetti come parametro dei vettori monodimensionali è possibile omettere la lunghezza del vettore nella dichiarazione. Ad esempio, la funzione `massimo`:

```
int massimo(int input[])
```

accetta come parametro in ingresso un vettore di interi. Questo in pratica vuol dire che, al momento della chiamata, quello che viene copiato nel parametro `input` della funzione è l' *indirizzo* della prima cella del vettore che le viene passato come parametro attuale. La conseguenza è che, nel caso la funzione `massimo` scriva nelle celle del vettore `input`, questo effettivamente altererà il contenuto del vettore del chiamante, in quanto quello che le è stato passato è una copia dell' indirizzo in memoria dove si trova.

La differenza con il comportamento di una funzione che riceve due interi come parametro al posto di un vettore da due interi è evidente nel seguente frammento di codice:

```
int max_swap(int a, int b){
  int tmp;
  if (a < b){
    tmp=b; b=a; a=tmp;
  }
  return a;
}

int max_swap_v(int v[]){
  int tmp;
  if (v[0] < v[1]){
    tmp=v[1]; v[0]=v[1]; v[0]=tmp;
  }
  return v[0];
}

int main(int argc, char* argv[]){
  int vett[2] = {9,7};
  int num = 9, num2 =7, max;
  max = max_swap(num,num2);
  max = max_swap_v(vett);
  return 0;
}
```

Nell' esempio proposto, l' effetto della chiamata a funzione `max_swap` all' interno del `main` è semplicemente quello di ritornare 9, ma non scambia i contenuti di `num` e `num2`, in quanto agisce su loro copie. Per contro, la funzione `max_swap_v` scambia effettivamente i contenuti della prima e della seconda cella di `vett`, in quanto il parametro attuale che le viene passato per copia è l' indirizzo della prima cella di `vett` in memoria.

6.13 Regole di visibilità

Il linguaggio C regolamenta la visibilità delle variabili dichiarate all' interno del sorgente secondo semplici regole, le cui conseguenze sono già state osservate in qualche caso (e.g. non è possibile dichiarare due variabili con lo stesso nome consecutivamente).

La visibilità di una variabile, ovverosia la porzione di codice in cui essa è utilizzabile, ha inizio dallo statement successivo a quello della sua dichiarazione. Gli elementi sintattici che regolano la visibilità di una variabile sono i delimitatori dei blocchi di codice, ovvero le parentesi graffe. La conseguenza della possibilità avere blocchi di codice annidati (ad esempio, il blocco di codice di un `if`, all' interno dell' implementazione di una funzione), è che le regole di visibilità tengono in considerazione la *gerarchia* degli annidamenti. Un buon modello mentale è quello delle scatole cinesi, ovvero scatole che contengono altre scatole.

Le regole di visibilità per le variabili in C sono quindi le seguenti:

- Non è possibile dichiarere due variabili con lo stesso nome all' interno dello stesso blocco.
- La variabile che viene vista dal codice è quella corrispondente alla dichiarazione più *interna* nella gerarchia dei blocchi.

Una prima conseguenza di queste regole è la possibilità di dichiarare più di una variabile con lo stesso nome, a patto che risiedano a livelli diversi della gerarchia dei blocchi. Ad esempio in:

```
int fun(int a){
  int tmp=2;
  if (a==1){
     int tmp = 3;
     a = a+tmp;
   }
  a= a-tmp;
  return a;
}
```

nel caso in cui l' espressione `a+tmp` sia calcolata, essa utilizzerà la dichiarazione più interna di `tmp`, ovvero quella che è stata inizializzata a 3 al momento della dichiarazione. Questo fenomeno è detto di *overloading* o sovraccarico del nome.

Una seconda conseguenza delle regole di visibilità è il fatto che le variabili locali di una funzione siano visibili solamente al suo interno, in quanto dichiarate nel blocco di codice della funzione stessa. Per quanto riguarda la visibilità, i parametri formali di una funzione sono da considerarsi variabili dichiarate all' interno della funzione, ovvero visibili in tutto il suo codice.

Un' ulteriore conseguenza delle regole di visibilità è che è possibile dichiarare variabili all' esterno di tutti i blocchi di codice esistenti, inclusi quelli che delimitano le implementazioni delle funzioni. Queste variabili sono quindi per definizione visibili all' interno di tutti i blocchi (salvo ridichiarazioni interne), e vengono per questo chiamate *variabili globali*. L' uso delle variabili globali è pesantemente sconsigliato: rendono particolarmente difficile la lettura del codice dato che il loro ciclo di vita inizia con l' inizio del programma e qualunque funzione può modificarle.

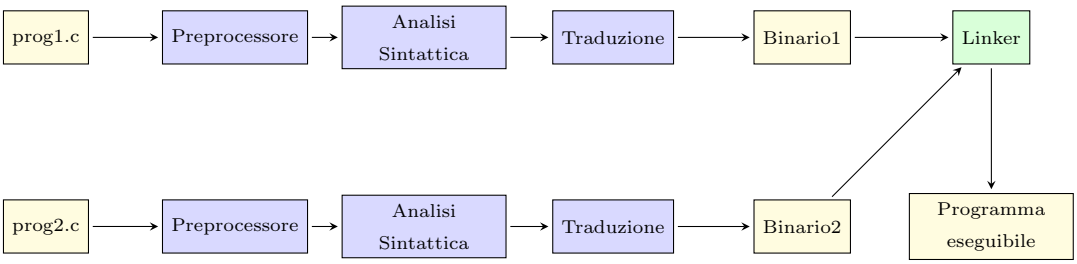

Figure 8: Struttura semplificata del processo di compilazione e collegamento (linking). I rettangoli blu rappresentano parti del compilatore, in verde il ruolo del linker

Date le regole di visibilità, il linguaggio C non consente di dichiarare due variabili con lo stesso nome se esse si trovano nello stesso blocco. Ad esempio, in C, la seguente funzione è errata:

```
int fun(int a){
  int tmp=2;
  int c;
  int c;
  return a;
}
```

6.14 Il preprocessore e gli headerfiles

Fino ad ora, data la limitata dimensione dei programmi realizzati, è stato possibile scrivere l' intero sorgente del programma all' interno di un singolo file, detto anche *compilation unit*. Tuttavia, è molto frequente la necessità di suddividere programmi complessi in più compilation unit diverse, per favorire l' organizzazione e la manutenzione dei sorgenti. Un esempio tipico dell' utilità di questa strategia è il riuso di alcune funzioni all'interno di programmi diversi, senza doverle copiare e incollare a mano.

Allo scopo di rendere tutto questo possibile, mantenendo la capacità di compilare ogni compilation unit separatamente, e riunire i binari ottenuti di conseguenza in un solo eseguibile, si utilizzano le funzionalità offerte dal cosiddetto *preprocessore* del compilatore.

In Figura 8 è riportato il flusso di compilazione che porta ad ottenere un eseguibile funzionante a partire da sorgenti in C scritti in 2 programmi. Entrambi i programmi vengono dati in input al compilatore, all' interno del quale il preprocessore li manipola per primo. In seguito alle manipolazioni del preprocessore, viene effettuata la fase di analisi sintattica e, se la compilation unit non contiene errori, la sua traduzione. Le traduzioni producono due files contenenti binario eseguibile dalla macchina, che vengono collegate insieme dal linker, producendo quindi il programma completo.

Il preprocessore agisce dunque prima che l' analisi sintattica del programma abbia inizio, pertanto, quello che produce in uscita è del codice C valido. É possibile per il programmatore pilotare il preprocessore attraverso le cosiddette *direttive per il preprocessore*. Sintatticamente, esse sono contraddistinte da un

carattere # seguito da una keyword. Per le necessità di questo corso vedremo solamente due keyword: `define` e `include`.

La direttiva `#define` consente al programmatore di specificare una sostituzione lessicale come:

```
#define valore_da_trovare valore_da_sostituire
```

ad esempio

```
#define PI 3.1415952
```

fa sì che il preprocessore, a partire dalla linea in cui è presente la direttiva `#define`, sostituisca tutte le occorrenze di `PI` con `3.1415952`. Questa sostituzione è fatta in modo cieco, il preprocessore non conosce la sintassi C, semplicemente si limita a svolgere il lavoro che fareste a mano usando la funzione "trova e sostituisci" del vostro editor di testo. L' utilità principale della direttiva `#define` è quella di rendere il codice più leggibile, ad esempio, consentendo di sostituire costanti numeriche con una stringa che rappresenta in modo più chiaro il loro scopo.

Chiaramente, è possibile abusare della direttiva `#define`, come ad esempio nel seguente frammento di codice:

```
#define if while

int fun (int a){
  if (a > 0){
    a=a-1;
  }
  return a;
}
```

In questo caso, il codice visto dal passo di analisi sintattica del processo di compilazione è

```
int fun (int a){
  while(a > 0){
    a=a-1;
  }
  return a;
}
```

il quale ha un comportamento significativamente diverso da quello che ci si può attendere da una lettura superficiale del sorgente con la primitiva `#define`. In generale, usi di questo tipo (ovvero coinvolgere keyword del linguaggio nella `#define`) vanno assolutamente evitati allo scopo di conservare la leggibilità del codice.

Per i più curiosi, è anche possibile utilizzare una versione parametrica della primitiva `#define`, dove nella stringa da sostituire sono specificate delle porzioni che vengono copiate all' interno della stringa sostituita. La sintassi in questo caso assomiglia a quella di una funzione, anche se non ha nulla a che vedere con essa. Ad esempio in

```
#define VETTORE(n) int vett[n]
VETTORE(3);
```

l' uso della define parametrica fa sì che, dopo la sostituzione operata dal preprocessore, il compilatore veda semplicemente `int vett[3];` come unica riga rimasta di sorgente.

La seconda direttiva che consideriamo è la direttiva `#include`: accetta un parametro, che è il percorso di un file, secondo questa sintassi:

```
#include "/home/utente/my_functions.h"
```

L' effetto della direttiva `#include` è di far sì che il contenuto del file presente al percorso specificato venga incollato al posto della direttiva, prima che il sorgente venga passato alla fase di analisi sintattica del compilatore.

```c
int sum(int a, int b){
  return a+b;
}
int prod(int a, int b){
  return a*b;
}
```

(a) Contenuto del file funzioni.c

```c
int sum(int a, int b);
int prod(int a, int b);
```

(b) Contenuto del file funzioni.h

```c
#include "funzioni.h"
int main(){
 int a=2,b=4,s,p
 s=sum(a,b);
 p=prod(a,b);
return 0;
}
```

(c) Contenuto del file programma.c prima dell' azione del preprocessore

```c
int sum(int a, int b);
int prod(int a, int b);
int main(){
  int a=2,b=4,s,p
  s=sum(a,b);
  p=prod(a,b);
 return 0;
}
```

(d) Contenuto del file programma.c dopo l' azione del preprocessore

Figure 9: Contenuto di 3 files strutturati in modo da implementare le funzioni somma e prodotto in un file diverso e consentire il suo utilizzo da parte del codice all' interno di `programma.c`.

6.15 Utilizzo di librerie

L'utilizzo più pratico della direttiva `#include` è quello che consente di andare a suddividere il proprio codice in porzioni, atte ad essere riusate, dette *librerie*.

Questa suddivisione è possibile in quanto, allo scopo di tradurre una singola compilation unit, è necessario al compilatore conoscere solamente i prototipi delle funzioni chiamate al suo interno. Di conseguenza, suddividendo le implementazioni di funzioni in diversi files, e avendo cura di copiare i prototipi delle funzioni implementate in files separati, detti *headerfiles* o files di intestazione, è possibile utilizzare la direttiva `#include` allo scopo di copiare i prototipi delle funzioni contenuti nell' headerfile all' interno della compilation unit che andrà ad usarlo. Convenzionalmente, l' headerfile ha lo stesso nome della compilation unit che implementa le funzioni in esso contenute, e ha come estensione `.h`.

Ad esempio, si considerino i 3 files distinti rappresentati in Figura 9. L' effetto del passaggio del preprocessore sul file `programma.c`, rappresentato in

```
unsigned int getint(void){
  unsigned int res=0;
  char buf;
  buf=getchar();
  while((buf <='0') && (buf
      <='9')){
    res=res* buf-'0';
  }
  return res;
}
```

```
void putint(int n){
char out_buf[10] = {'\0'};
int p=0;
do {
  out_buf[p]= n%10;
  n = n/10;
  p++
} while(n>0);
while(p>=0){
 putchar(out_buf[p]+'0');
 p--;
}
return;
}
```

Figure 10: Due implementazioni esemplificative di `getint`, e `putint`

Figura 9(c) è quello di andare a inserire il contenuto dell' headerfile, facendo sì che il compilatore possa quindi tradurre le chiamate a `sum` e `prod`, in quanto il loro prototipo è presente prima.

Allo scopo di consentire l' inclusione dei files, è necessario prestare attenzione nel fornire alla direttiva `#include` il percorso completo dell' headerfile nel caso esso non si trovi nella stessa directory dei rimanenti sorgenti.

La specifica del linguaggio C include anche un insieme di librerie standard, atte ad offrire agli sviluppatori funzionalità di uso comune nella programmazione, tra cui funzioni matematiche più complesse di quelle offerte dagli operatori, funzioni di manipolazione delle stringhe e funzioni che consentono di gestire input/output verso l' utente e verso la memoria di massa. Essendo queste librerie standard, e presenti sul sistema al momento dell' installazione degli accessori del compilatore, è possibile utilizzare una sintassi alternativa della direttiva `#include` che consente al programmatore di non curarsi di quale sia il percorso in cui esse sono installate. Un esempio della sintassi in questione, volendo includere la libreria che contiene funzioni matematiche accessorie, `math.h` è il seguente:

```
#include <math.h>
```

Questa sintassi fa sì che sia responsabilità del preprocessore localizzare il file `math.h` e includerlo.

6.16 Input / Output verso l' utente

L' utilizzo delle librerie standard C ci consente di effettuare input e output da parte dei nostri programmi: l' headerfile contenente le funzioni necessarie è quello della libreria di STandard Input/Output, `stdio.h`.

Al suo interno, sono contenuti i prototipi delle seguenti due funzioni:

```
int getchar(void);
int putchar(int c);
```

La funzione `getchar` legge un carattere che viene inserito via tastiera e lo restituisce all' interno di una variabile intera. Il carattere è rappresentato con i bit

meno significativi dell' intero, ed è quindi un valore sicuramente positivo. La conversione automatica, tramite troncamento, di un intero in un carattere consente di salvare il risultato di `getchar` direttamente all' interno di un carattere. La funzione `putchar`, simmetricamente, stampa a video il valore ricevuto come parametro, previa la sua conversione a char tramite troncamento. Il valore ritornato da `putchar` è il carattere stampato a video in caso di successo, o un valore negativo rappresentante un errore.

Tramite l' uso di getchar e putchar, possiamo a nostra volta implementare funzioni che ci consentono di leggere e stampare numeri interi senza segno, `getint` e `putint`. Trovate un' implementazione possibile per entrambe in Figura 10.

Contents

1 Diagrammi di flusso **2**

2 Rappresentazione dei dati in un calcolatore **2**
2.1 Numeri naturali ($\mathbb{N}$) . . . 2
2.2 Numeri relativi ($\mathbb{Z}$) . . . 3
2.3 Numeri razionali ($\mathbb{Q}$) . . . 4
2.4 Numeri reali ($\mathbb{R}$) e complessi ($\mathbb{C}$) . . . 4
2.5 Testo . . . 5

3 Codifica di segnali **6**
3.1 Rappresentazione delle immagini . . . 7

4 Cenni di architettura del calcolatore e sistemi operativi **8**
4.1 Unità di misura . . . 8
4.2 Macchina di Von Neumann . . . 9
4.3 Il sistema operativo . . . 15
4.4 Filesystem . . . 17

5 Introduzione alla programmazione **17**

6 Il linguaggio C **19**
6.1 Dichiarazioni di variabili e tipi di dato base, costanti . . . 20
6.2 Operatori . . . 22
6.3 Iniziare a programmare . . . 25
6.4 Vettori o array . . . 26
6.5 Strutture . . . 28
6.6 Tipi personalizzati . . . 29
6.7 Costrutti di selezione: lo statement `if` . . . 30
6.8 Cicli: lo statement `do while` e `while` . . . 31
6.9 Cicli: lo statement `for` . . . 32
6.10 Costrutti di selezione: il costrutto `switch-case` . . . 33
6.11 Deviazioni del flusso d' esecuzione: `break` e `continue` . . . 34
6.12 Funzioni . . . 35
6.13 Regole di visibilità . . . 39
6.14 Il preprocessore e gli headerfiles . . . 40
6.15 Utilizzo di librerie . . . 42
6.16 Input / Output verso l' utente . . . 43