

PROGRAMMAZIONE 1 - RICCARDI

Obiettivi, Cos'è l'informatica e Introduzione all'informatica:

Obiettivo: conoscenza di base dell'informatica.

Informatica: dal francese, informazione automatica. Termine coniato da Ph. Dreyfus nel 1962

Questo significa che è l'insieme degli aspetti scientifici e tecnici che sono specificatamente applicati alla raccolta e al trattamento dell'informazione, in particolare all'elaborazione automatica dei dati. Ma anche lo studio sistematico degli algoritmi che descrivono e trasformano l'informazione.

Algoritmi e linguaggi di programmazione:

Un **algoritmo** è una sequenza precisa di operazioni, comprensibili da un esecutore, che definisce una sequenza finita di passi che portano alla realizzazione di un compito (*task o problema*).

L'algoritmo ha alcune caratteristiche: (2 e 3 sono le più importanti)

1. deve essere **comprendibile** al suo esecutore (linguaggi di programmazione nel caso di un calcolatore). L'algoritmo così codificato viene chiamato *programma*.
2. deve essere **corretto** (l'algoritmo ottiene la soluzione del compito cui è preposto, senza errori in nessun passo fondamentale).
3. deve essere **efficiente** (l'algoritmo ottiene la soluzione usando la minor quantità di risorse).

DA NOTARE CHE: il concetto di algoritmo non è prerogativa dei calcolatori, ma semplicemente il calcolatore (in questo caso un computer) ha capacità di calcolo tali da gestire e lavorare con quantità di dati altrimenti intrattabili.

Gli algoritmi sono il modo in cui, implicitamente o esplicitamente, affrontiamo ogni problema nella vita di tutti i giorni. Questo problema viene chiamato **task**, e l'algoritmo serve a risolverlo.

Il problema dell'algoritmo della scelta invece crea un problema, il cosiddetto **il problema della segretaria**, che contiene una semplificazione matematica dell'algoritmo della scelta ottimale dal punto di vista matematico.

Il concetto di questo problema è che dovendo scegliere di assumere una tra 100 segretarie, con l'unica remora che una volta assunta una, non si continua a cercare. L'algoritmo in questo caso direbbe di scegliere solo passati i primi 37% elementi (in questo caso segretarie), dopodiché, passato questo step, il primo che si dimostra all'altezza del compito anche in base alle persone viste in precedenza viene assunto e si chiude la ricerca.

(Vedi: <https://www.smartworld.it/tecnologia/formula-matematica-decisioni-difficili-della-vita.html>).

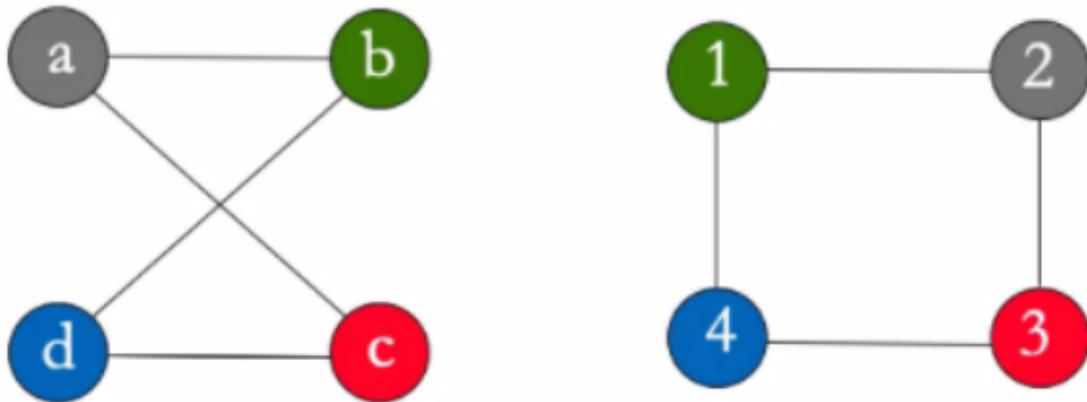
Dal concetto di ottimizzazione e rendimento dell'algoritmo si incontra anche il concetto di **reward**: ho due scelte, una usuale (**exploit**), un'altra che cambia e esce dagli schemi (**explore**). La migliore dipende dal migliore risultato che le due rendono possibile.

Il grafo

Per la rappresentazione di alcuni logaritmi è molto utile l'utilizzo del **grafo**. Il grafo è uno schema che collega tra loro le informazioni possedute, ottenute anche fonti diverse. Le informazioni così raccolte hanno il lato positivo di essere facilmente ricercabile, distinguibile e visibile. Inoltre rende evidenti i collegamenti che hanno tra loro.

Questo tipo di schematizzazione, essendo molto ottimizzata e mettendo subito in risalto le informazioni che hanno contatti tra di loro, vengono ampiamente utilizzati nell'intelligenza artificiale, permettendo appunto grandi elaborazioni. Il grafo quindi è un oggetto logico molto potente.

Un accortezza: con questa struttura è facile prendere due grafi uguali come diversi. I grafi si dicono **isomorfi** se hanno le stesse caratteristiche in termini di nodi e archi.



A questo punto però si crea un problema, infatti in alcuni casi, usando un dato algoritmo ottengo risultati diversi se non qualifico un qualche tipo di **parametro**. Proprio il concetto di parametro (inteso nel senso informatico, cioè elemento che caratterizza un determinato algoritmo e che l'algoritmo si aspetta per eseguire un compito). I parametri quindi possono **condizionare l'esecuzione del task**.

Come descrivere un algoritmo:

L'algoritmo per definizione è una sequenza precisa di passi, di conseguenza una lista non ordinata di elementi non è l'oggetto giusto per la sua rappresentazione. Inoltre il fatto che attraverso un parametro si possa ottenere un risultato diverso rende la sua codifica difficile.

Innanzitutto è essenziale che sia descritto come un programma, cioè come una sequenza di istruzioni scritte in un linguaggio comprensibile al calcolatore. Il compito del programmatore è proprio quello di scrivere del codice che porti alla soluzione ottimale del problema.

Per iniziare bisogna inizialmente **definire esattamente il problema**, che può essere:

1. problema di **puro calcolo e conversione**, cioè risolvibile attraverso conoscenze matematiche.
2. problemi di **decisione**, cioè capire se una certa proprietà è propria di un certo elemento. (es. 15467 è primo?).
3. problemi di **ricerca**, cioè trovare un elemento con determinate caratteristiche in un determinato insieme.

Il passaggio da definizione del problema a esecuzione del task poi passa da altri step, che sono necessari alla codifica e alla comprensione dei vari step:

1. Problema: linguaggio **L_p**, ovvero linguaggio naturale (per descrivere il problema).
2. Algoritmo: linguaggio **L_a**, ovvero lo pseudocodice (lascia meno incomprensione ma più difficile da leggere).
3. Programma: linguaggio **L_t**, ovvero linguaggio macchina (es. C/C++).

Una volta eseguiti questi **macropassaggi** si può passare alla rifinitura del processo, attraverso dei **micro-algoritmi** (alcune volte il programma e l'algoritmo necessario al suo funzionamento sono così semplici da non necessitare la stesura di micro-algoritmi di supporto).

Su algoritmi molto complessi si può applicare la tecnica del **deconstruct**, ovvero decostruire il problema finché non si ottengono tanti problemi di facile soluzione.

Nel caso della stesura di un algoritmo si devono prendere in considerazione tutte le casistiche. Ad esempio se si scrive un algoritmo per cercare un libro bisogna anche tenere in considerazione la possibilità che questo libro non sia presente nell'indice della ricerca. Allo stesso modo non si può pensare di avere una differenza abissale di prestazioni se il libro è presente nella prima parte di indice o invece è in fondo. Ci sono di solito sempre più di un modo per risolvere problemi, quindi il problema principale diventa quello di trovare il modo più efficiente disponibile.

La programmazione e il linguaggio di programmazione

Programmare innanzitutto significa analizzare un problema, progettare un algoritmo per ottenere una soluzione, esprimere l'algoritmo in un linguaggio di programmazione, mettere la macchina nella situazione di eseguire il programma e infine correggere eventuali errori.

L'algoritmo deve essere tradotto da linguaggio formale e reso comprensibile all'esecutore. L'insieme dei comandi dati al calcolatore in un certo linguaggio viene chiamato **programma**.

Il linguaggio di programmazione deve essere **rigoroso e preciso** dal punto di vista di:

1. **sintassi**: le regole che descrivono le stringhe di parole riconosciute dal linguaggio.
2. **semantica**: regole per l'interpretazione delle stringhe e che descrivono i processi computazionali dell'esecutore (o più semplicemente un errore logico nella stesura del programma, cioè un errore nella comprensione logica del programma). Questo errore è più difficile da individuare perché il compilatore non dà errore, ma i risultati saranno incoerenti.

Parlando di linguaggi di programmazione, bisogna dire anche qualcosa sull'astrazione del linguaggio:

1. se il linguaggio è di bassissimo livello, quindi troppo vicino alla macchina è difficile programmare.
2. se il linguaggio è troppo di alto livello, quindi molto vicino al linguaggio del programmatore, i programmi diventano inefficienti.

- Linguaggio C

TOT=PAGA+STRAORD ;

- Linguaggio assembler

**LOAD PAGA
ADD STRAORD
STORE TOT**

- Linguaggio macchina

**0100001111
1100111001
0110001111**

Livello di astrazione ↑

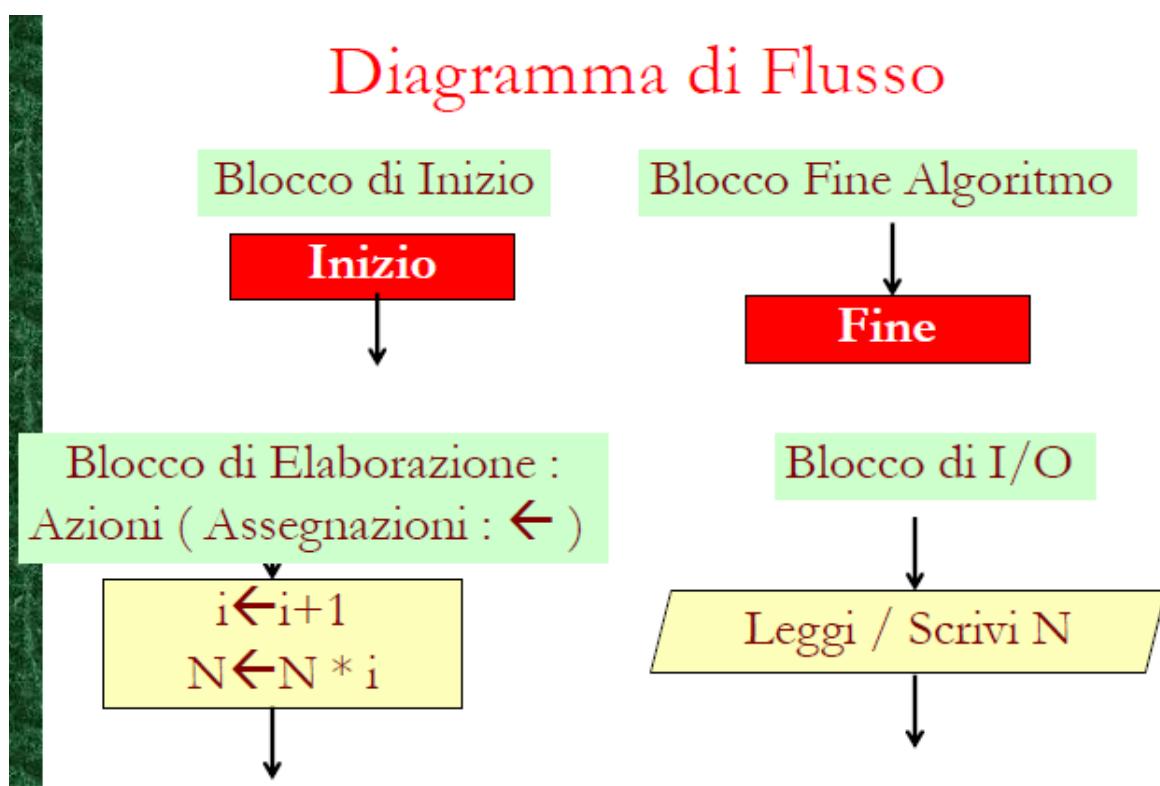
Agli albori dell'informatica si programmava in linguaggio macchina, cioè compreso direttamente dalla macchina.

Dagli anni '50 iniziano a nascere i primi linguaggi di programmazione di alto livello, data la sempre più alta complessità dei programmi.

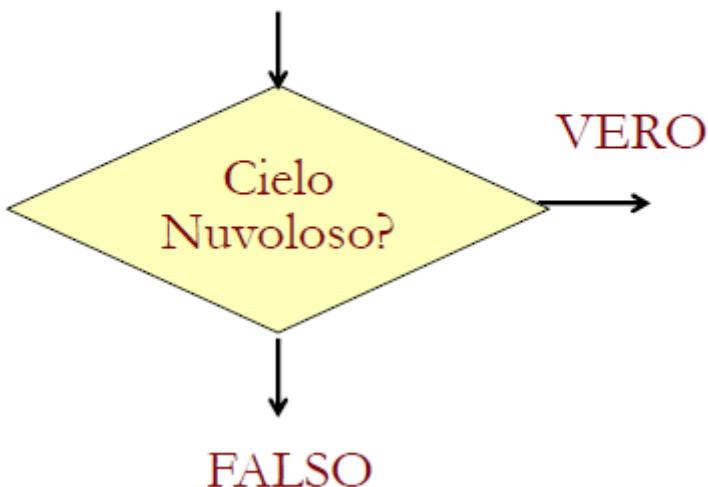
Ad oggi esistono tantissimi linguaggi di programmazione, ognuno con il suo scopo e inventato per esigenze diverse.

Metodo grafico per creare algoritmi:

Gli algoritmi possono essere schematizzati anche attraverso diagrammi di flusso, cioè blocchi orientati che hanno un significato proprio.



Blocco di Controllo



CARATTERISTICHE:

- **BLOCCO INIZIO:** unico nodo del grafo da cui si può solo partire infatti ha una sola freccia uscente. Rettangolo rosso per convenzione. Di solito si definisce con '1' l'inizio.
- **BLOCCO FINE:** blocco in cui si può solo arrivare infatti ha una sola freccia entrante. Rettangoli rossi per convenzione. Di solito si definisce con '0' la fine.
- **BLOCCO ISTRUZIONE:**
 - **assegnazione:** dà il valore che c'è a sinistra alla variabile di destra.
 - **I/O:** (input/output), si sta leggendo o scrivendo
 - **controllo:** (con più uscite, in questo caso vero e falso)

Architettura hardware di un compilatore:

Storia del Calcolatore:

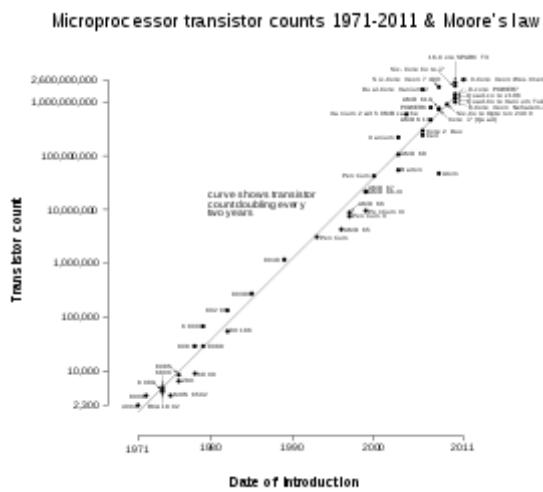
Il primo calcolatore definito tale è ENIAC, creato nel 1945 in America con il principale scopo di calcolare la traiettoria dell'artiglieria. Poi donato al University of Pennsylvania, dove invece venne usato per il calcolo ingegneristico e scientifico. Pesava circa 27 tonnellate e per programmarlo erano necessarie diverse persone, infatti si basava su una *plugboard* cioè un sistema di "prese" che, usate per fare il contatto giusto, fornivano risultati alle operazioni date come input. Non esisteva un linguaggio di programmazione per questo computer e i cosiddetti "programmatori" dell'epoca probabilmente non lo hanno mai visto in tutta la loro vita, dato che loro facevano solo i calcoli per farlo funzionare e poi qualcun altro si occupava di creare i giusti collegamenti. La sua potenza di calcolo era tale da permettere di calcolare quello che un uomo avrebbe fatto in 20 ore in soli 30 secondi. Aveva un clock di circa 100kHz, e garantiva una vasta quantità di calcoli possibili.

Dopo la seconda guerra mondiale inizia effettivamente lo sviluppo di una macchina più vicina all'immaginari comune. Sempre più comuni diventano:

1. **MAINFRAME:** computer di grandi dimensioni ed elevata potenza, solitamente condiviso fra più persone/uffici.

2. PERSONAL COMPUTER: computer dotati di tastiera e schermo separati, di memoria di massa interni o esterni alla memoria centrale e molto più economico.
3. LAPTOP: computer portatile, nonché evoluzione portabile del personal computer. Diventa molto famoso e utilizzato. Diventano così sofisticati che inizia il processo di miniaturizzazione dato che si poteva fare con le tecnologie disponibili ormai.
4. HANDHELD COMPUTER (SMARTPHONE): cellulari dotati di touchscreen, molto portabili e maneggevoli. Anche con questi come con i laptop avviene il processo di miniaturizzazione.
5. WHERABLE COMPUTER: sono computer così miniaturizzati da poter essere messi all'interno di un orologio, o un altro oggetto facilmente indossabile. Questi solamente sono caratterizzati da una gamma di sensori solitamente molto avanzati che possono servire per monitorare lo stato di salute o altri scopi.

La miniaturizzazione che è avvenuta è dovuta alla possibilità di dimezzare la grandezza del transistor per un certa quantità di volte, data dalla legge di Moore.

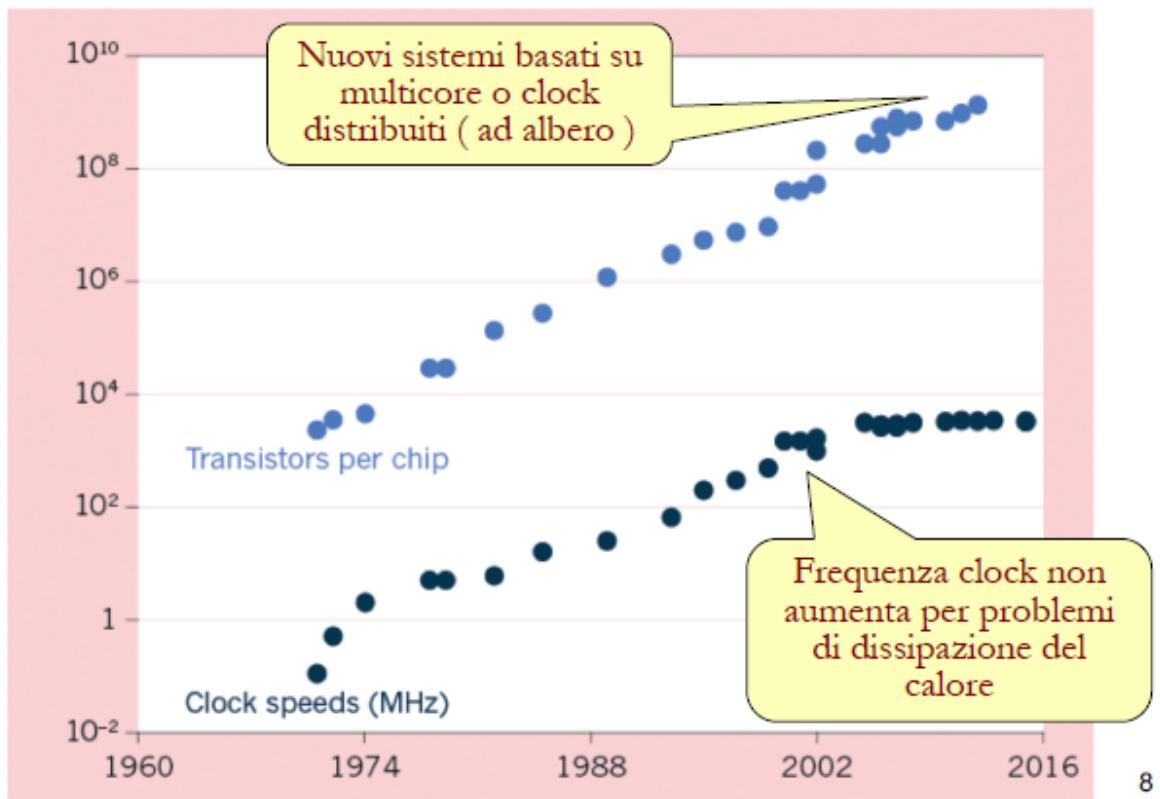


Moore infatti nel 1965 ipotizza che il numero di transistor inseribili nei microprocessori sarebbe raddoppiato ogni anno. La sua previsione si rivela corretta, finché verso la fine degli anni '80 viene riformulata: raddoppio del numero di transistori ogni 18 mesi.

Per aumentare ancora più le prestazioni si cerca di aumentare il clock dei processori stessi. Il **clock** di un processore è il dato che indica la quantità di calcoli può fare al secondo. Al momento il dato del clock è pressoché fermo intorno al dato di 3GHz. Questo stop è dovuto al problema fisico per cui aumentando troppo il clock, il calore emesso diventa esponenzialmente maggiore. Di conseguenza quando si sono provati clock più alti si ha avuto problemi di gestione delle temperature eccessivamente alte. Per ovviare questo problema si apre un'altra strada: il multithreading o clock distribuito ad albero. Questo permette di passare il clock in tutte le parti del processore e rendere quindi i processi sincronizzati, utilizzando la potenza di tutte le parti assieme e quindi garantendo una maggior potenza.

Legge di Moore

Transistors e Clock Speed



Organizzare il software di un calcolatore

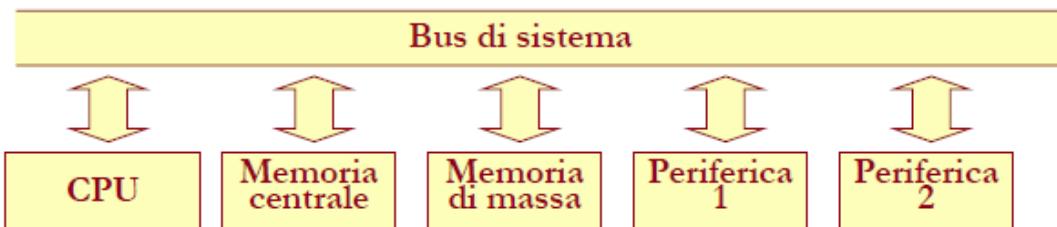
Il normale schema di un calcolatore è (generalmente) così costruito:



Per leggere questo schema bisogna capire che:

1. Tutto si basa su quello che ha sotto di se e può comunicare solo con i blocchi a se adiacenti. (ad es. il software di sistema non si interfaccia direttamente con l'hardware, passa attraverso il sistema operativo).
2. L'unico che può operare con l'hardware è il sistema operativo. Questo ne gestisce tutte le componenti.

Architettura Hardware:



L'architettura hardware sopra è nota con il nome di **Macchina di Von Neumann** ed è così costituita:

1. **CPU** (central processing unit): svolge l'elaborazione eseguendo i programmi logici.
2. **MEMORIA CENTRALE** : memoria utilizzata per memorizzare dati e istruzioni, volatile (usata quindi solo per l'esecuzione di programmi).
3. **MEMORIA DI MASSA** : usata per memorizzare grandi quantità di dati e programmi in maniera persistente, al contrario della memoria centrale.
4. **PERIFERICHE**: sono di vario tipo e servono a tutto quello che permette l'interazione con il calcolatore. (ad es. tastiera, monitor, stampante, ..).
5. **BUS DI SISTEMA**: elemento che interconnette gli altri componenti consentendo lo scambio di dati, quindi anche il collegamento fra periferiche e hardware.

Macchina di Von Neumann:

CPU: unità di elaborazione

La CPU è l'unità di elaborazione del calcolatore, si occupa di caricare le istruzioni in memoria centrale, interpretarle e eseguirle.

E' altamente specializzata, perché è pensata per eseguire pochi tipi di operazioni ma molto velocemente.

Il suo lavoro è scandito dal **clock** (orologio interno). La potenza del calcolatore dipende in parte dal clock, infatti quanto più questo è alto, tante più sono le istruzioni che riesce ad eseguire al secondo. La sua frequenza infatti viene misurata in Hz, e 1 Hz = 1 ciclo/s. Ad oggi le CPU riescono a lavorare in parallelo, il cosiddetto "lavoro condiviso", grazie al clock. Infatti il segnale del clock che arriva a tutti sincronizzato permette di eseguire azioni contemporaneamente.

Memoria centrale:

La memoria centrale è destinata a accogliere dati e programmi sui quali opera il calcolatore. (ad es. quando usiamo il computer qui vengono memorizzati i dati di programmi che stiamo usando, ...). La memoria centrale è velocissima ma volatile (cioè una volta che il programma finisce, tutti i dati di un programma vengono cancellati e lo spazio viene liberato). Questa memoria accoglie i

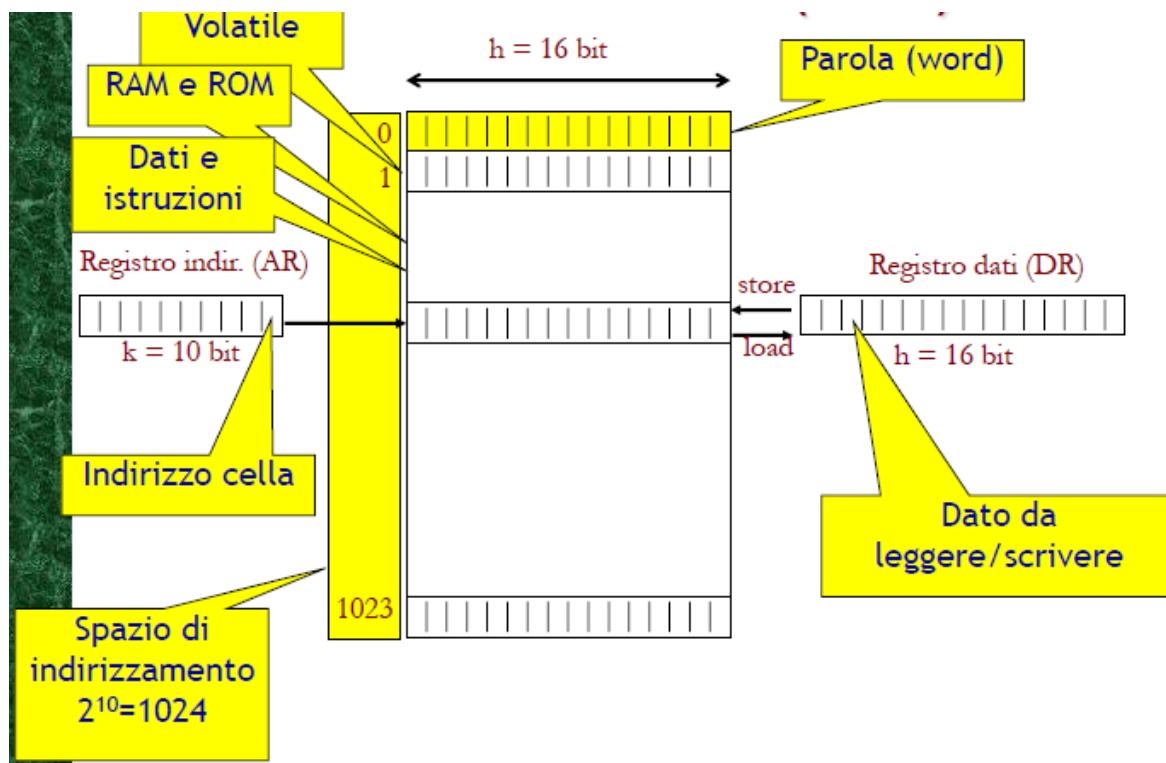
dati necessari a far funzionare i programmi. Concettualmente è composta da una sequenza di celle ognuna delle quali contiene una parola (**word**). Ad ogni cella si può accedere direttamente specificandone l'**indirizzo**, e accedendone si può cambiare il suo contenuto (read/write). La quantità di bit da cui è composta la word dipende dalla macchina, infatti è caratteristico del microprocessore e attraverso questo si identifica lo spazio di indirizzamento.

La memoria centrale solitamente è realizzata da una **RAM** (random access memory). La sua caratteristica è che ogni cosa è accessibile e non devo scansionare tutti gli indirizzi per cercare la cella che ho bisogno. (per es. non è come il nastro magnetico che devono fisicamente essere spostate avanti e indietro per trovare l'informazione cercata). Questo ha ricadute dirette sul tempo di accesso, che è indipendente dall'indirizzo della word che si vuole accedere. Questa è una memoria volatile quindi i dati presenti qui sopra si perdono quando si spegne la macchina. Esistono SRAM (static RAM), molto veloce, può contenere pochi dati acceduti molto frequentemente, e DRAM (dynamic RAM) che contiene più dati, ma più lentamente.

Altri tipi di memoria presenti sono le **ROM** (read-only memory). Hanno caratteristiche simili alle RAM, hanno un accesso veloce ai dati (ma non come quello delle RAM) e si tratta di **memorie permanenti** su cui però non si può scrivere. Tipicamente sono utilizzate per memorizzare dati e programmi che servono prima del caricamento del Sistema Operativo (ad es. per il caricamento del BIOS = Basic Input-Output System).

La memoria quindi com'è fatta? Appare come una lista di word cioè un mattoncino composto da 16bit, di conseguenza posso memorizzare tanti dai quanti 2^{16} valori binari. All'interno della memoria ci posso mettere tante word come gli indirizzi di memoria che sono 1024, e sono assegnati da 0 a 1023. Se questi fossero i limiti del tuo microprocessore allora devi fare in modo che all'interno di questo spazio di memoria ci stiano i dati e i programmi necessari al tuo scopo.

Per questo motivo in memoria non si può accedere a valori più grandi di 2^{16} e non si può accedere all'indirizzo 1024.



Però poi a questa memoria si deve poter accedere, quindi nella CPU esistono dei registri, che sono word di memoria chiamate così per distinguerle da quelle in memoria centrale. Questi registri possono essere:

1. AR (Address Register): di almeno 10 bit, perché in questo caso posso indirizzare da 0 a 1023

2. DR (data Register): posto in cui i dati di cui la CPU ha bisogno per fare operazioni che vengono copiati in questa posizione dalla memoria centrale per rendere le informazioni direttamente disponibili alla CPU.

I registri sono elementi di supporto al calcolo.

Gerarchia di memoria

Ci sono a disposizione memorie con caratteristiche diverse in base al loro scopo:

1. Registri: presenti nella CPU, piccoli ma veloci, utili per contenere dati temporanei per le operazioni della CPU
2. Memoria cache (SRAM): veloce, contiene pochi dati usati/richiesti frequentemente
3. Memoria principale (DRAM): meno veloce ma contiene più dati
4. Memoria secondaria o di Massa: molti dati, indicativamente più lenti (anche se gli SSD consentono un vantaggio prestazionale rispetto agli HD o agli HHD).

Rappresentazione dell'informazione:

Tutti i dati vengono solitamente rappresentati in maniera **binaria**, ovvero il bit che può prendere il valore 0 o 1. Questo è molto utile perché si basano strutturalmente basati su dispositivi bistabili (corrente ON o corrente OFF). Per questo l'elaboratore elettronico può operare solo su sequenze di simboli binari.

Il **BIT** (derivato da Bynary digIT) è quindi l'unità elementare dell'informazione. Comandi e dati nel computer vengono quindi rappresentati con lunghe sequenze di numeri binari.

Essendo tutto codificabile attraverso il bit, allora sono state inventate strutture per contenere la rappresentazione figurata della realtà. Ad esempio la convenzione ASCII, poi diventata extended-ASCII, ha codificato i caratteri come sequenza di 8bit, il **byte**.

Carattere 'a' rappresentato da:

$$(97)_{10} = (01100001)_2$$



BYTE = 8 bit

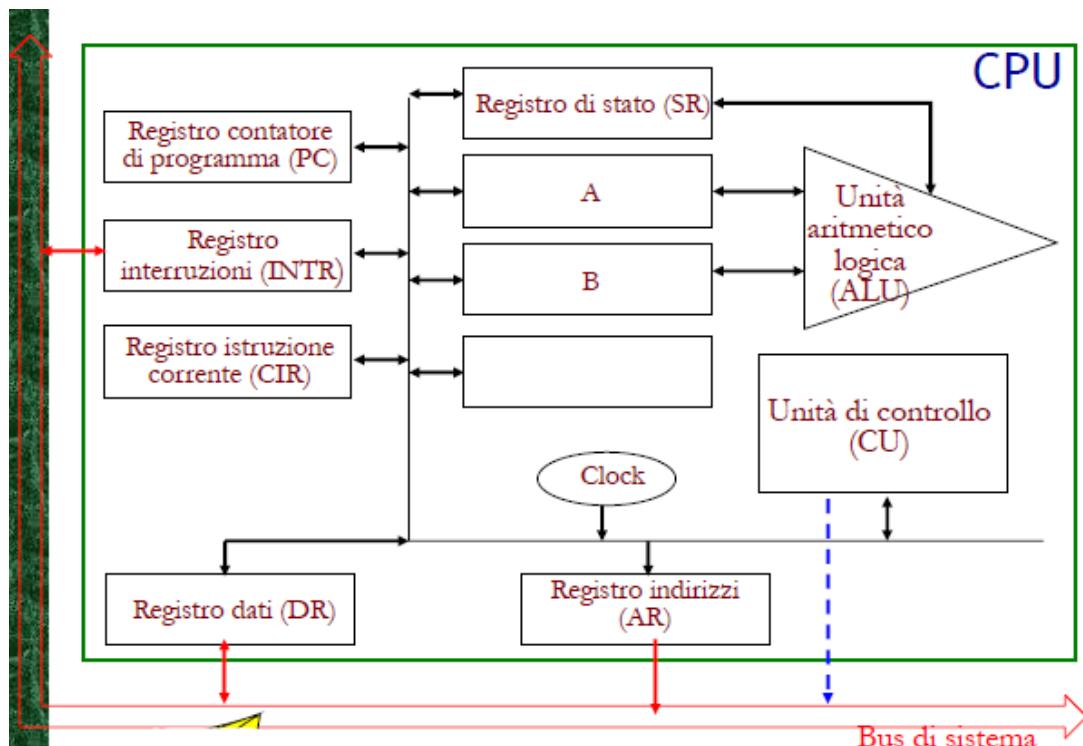
Per i multipli del byte si adottano gli stessi simboli del sistema decimale, ma visto che gli elementi sono in base 2, il fattore di scala è leggermente diverso:

KILOBYTE	KB	$2^{10} = 1024$ byte	$\sim 10^3$ byte
MEGABYTE	MB	$2^{20} = 1024^2 = 1024$ KB	$\sim 10^6$ byte
GIGABYTE	GB	$2^{30} = 1024^3 = 1024$ MB	$\sim 10^9$ byte
TERABYTE	TB	$2^{40} = 1024^4 = 1024$ GB	$\sim 10^{12}$ byte
PETABYTE	PB	$2^{50} = 1024^5 = 1024$ TB	$\sim 10^{15}$ byte
EXABYTE	EB	$2^{60} = 1024^6 = 1024$ PB	$\sim 10^{18}$ byte

Funzionamento della CPU:

Innanzitutto bisogna dire che:

1. il trasferimento dei dati avviene con il bus di sistema.
2. Le fasi di elaborazione si susseguono in modo sincrono rispetto all'orologio di sistema (clock).
3. Durante ogni intervallo di tempo **l'unità di controllo** (che fa parte del processore) stabilisce la funzione da svolgere.
4. L'intera macchina opera in maniera sequenziale (però architetture più evolute prevedono l'esecuzione parallela delle istruzioni).



Come si vede dall'immagine la CPU ha a disposizione vari elementi:

1. i registri (PC, INTR, CIR, DR, AR, ...)
 - registro di stato della CPU (**SR**) (FLAG: C, Z, S, V), serve a far capire al compilatore lo stato delle periferiche e delle varie parti del compilatore stesso.
 - Il program counter (**PC**) indica gli indirizzi dove andare a prendere la prossima istruzione.
 - registro istruzioni (**INTR**) segnale lo stato di funzionamento delle periferiche.
2. unità di controllo: capacità di controllare quello che sta succedendo, in pratica manda segnali per impartire degli ordini, in particolare possono provenire da questa unità il segnale di prelievo, di decodifica e d'esecuzione dell'istruzione.
3. Unità ALO: oggetto che esegue le operazioni aritmetico-logiche.
4. si nota il BUS di SISTEMA, responsabile di collegarsi ai vari registri e alle varie periferiche.
(sotto)

Il Ciclo base di funzionamento della CPU: (visione ad alto livello)

1. FETCH

1. La CU manda un segnale affinché il PC sia spostato nel AR (cioè la prossima istruzione viene indicizzata).
2. segnale controllo (read) alla memoria centrale posto all'indirizzo in AR.

3. il dato letto viene messo a disposizione nel DR (registro dei dati).
4. la CU manda il segnale di controllo affinché il contenuto di DR sia spostato nel CIR (registro istruzione corrente).
2. INTERPRETAZIONE: le informazioni sul CIR vengono decodificate dalla CU.
3. ESECUZIONE: la CU genera una sequenza di segnali di controllo necessari a eseguire l'istruzione.
4. il PC viene incrementato per puntare alla prossima istruzione.

Durante l'esecuzione la CPU può eseguire 3 macro-tipologie di istruzioni:

1. Istruzioni aritmetiche.
2. istruzioni di controllo.
3. istruzione di trasferimento dei dati (sia registro-registro, sia memoria-memoria, sia memoria-registro, sia registro-memoria).

Il Bus di Sistema

Il bus di sistema è l'elemento che interconnette le varie periferiche e i vari elementi del calcolatore. In ogni istante il bus è dedicato al collegare due unità, una che trasmette e una che riceve. Il processore esegue il *bus mastering*, ovvero seleziona le connessioni da attivare e indica l'operazione da svolgere. Il bus è suddiviso in tre insiemi di linee: *bus dati*, *bus indirizzi* e *linee di controllo* (quest'ultime trasportano informazioni relative alla modalità di trasferimento e alla temporizzazione).

Il Bus di sistema ha l'organizzazione della comunicazione cosiddetta *master/slave*, ovvero ci sono parti (*slave*) che devono sempre ascoltarne altre (i *master*). Generalmente il master è l'elemento di controllo, nonché quello che manda i segnali di esecuzione.

Le periferiche: memorie di massa

Con il termine memoria di massa ci si riferisce a un dispositivo di memorizzazione permanente capace di contenere grosse quantità di dati.

Possono essere: fissi o rimovibili, ad accesso sequenziale o casuale, dispositivi in sola lettura(RO), in lettura scrittura (RW), o WORM (Write Once Read Many), dispositivi magnetici, ottici o magnetico-ottici.

1. HARD DISK

Sfruttano le proprietà magnetiche di alcuni materiali (sostanze ferromagnetiche) di poter assumere a comando una certa direzione di magnetizzazione. A ciascuna direzione associa un simbolo binario. Sono costituiti da micro-celle magnetizzabili indipendentemente. La magnetizzazione è semipermanente, cioè rimane anche in assenza di mancanza di corrente ma può essere modificata.

2. MEMORIE DI MASSA: HDD

Usa uno schema di memorizzazione creato in fase di formattazione di basso livello. Ogni superficie è divisa in tracce concentriche. I dati sono memorizzati in maniera sequenziale. Ogni traccia è divisa in settori. L'insieme delle tracce omologhe poste su diverse facce è detto cilindro.

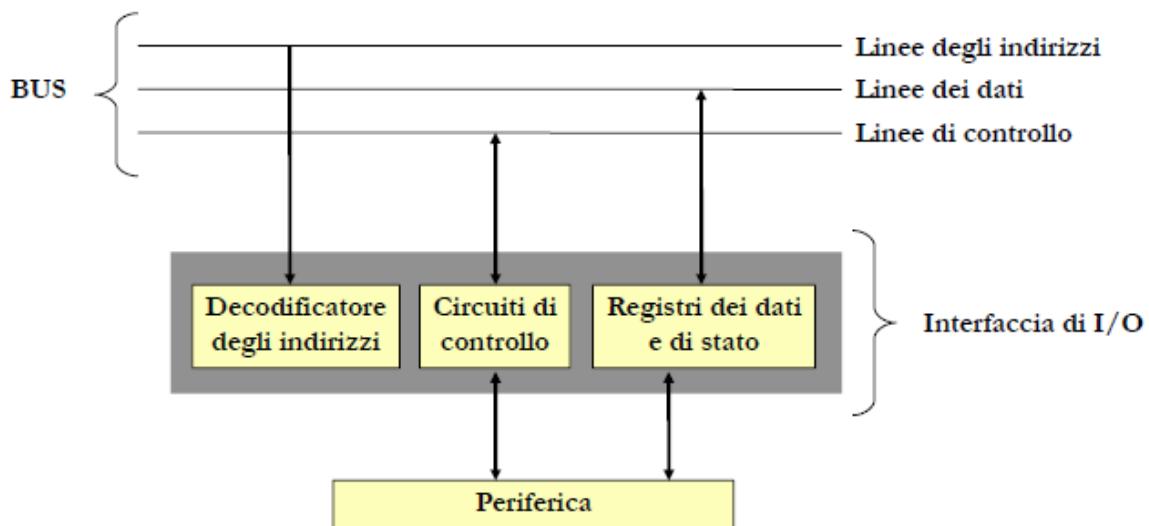
3. SSD: MEMORIE ALLO STATO SOLIDI

Accesso completamente elettronico (invece che elettromagnetico) ai dati. Al momento costi maggiori per unità di memoria. Permettono uno start-up immediato, latenza molto bassa e velocità di trasferimento circa un ordine di grandezza maggiore rispetto agli hard disk.

4. RAID

Per aumentare le prestazioni dei sistemi a disco è possibile raggruppare più dischi in un sistema RAID (Redundant Array of Inexpensive Disk). Questo sistema suddivide i file in blocchi registrati su dischi diversi per aumentare le prestazioni (data striping). Questo sistema è anche utilizzato per incrementare l'affidabilità dei sistemi a disco attraverso un meccanismo di ridondanza.

L'interfaccia delle Periferiche



Comunicare con le periferiche può essere complicato, infatti se si dovesse avere del codice per parlare con ogni singola periferica il lavoro del programmatore diventerebbe impossibile. Per questo esiste un'interfaccia, che fa come da traduttore tra il linguaggio della periferica e la macchina.

E' possibile avere un interfaccia diversa per ogni periferica, ma è più logico avere delle interfacce standard per periferiche simili. (Ad es. USB: Universal Serial Bus).

Questa interfaccia si occupa concettualmente della gestione e dello scambio di dati tra il processore e le periferiche. In generale contiene:

1. un registro dati della periferica
2. un registro di comando della periferica
3. un registro di stato (questo talvolta collegato al registro delle interruzioni del processore)

A seconda del processore e dei registri delle periferiche, le interfacce possono:

1. condividere lo spazio di indirizzi della memoria (memory mapped I/O)
2. adottare uno spazio di istruzioni/indirizzi distinti (port mapped I/O)

Le periferiche attraverso le interfacce possono essere gestite con due metodi principalmente:

1. POLLING: il processore invia sul bus il comando di lettura e si mette in attesa che il dato sia disponibile sul registro della periferica attraverso continui cicli. (PRO: facile implementazione e gestione; CONTRO: tiene sospeso il processore nel ciclo di attesa del dato).

2. INTERRUPT: il processore invia il comando di lettura alla periferica e poi continua le sue operazioni. Quando il dato è disponibile sul registro della periferica, la periferica stessa "solleva" un *interrupt*, un'interruzione. Il processore così interrompe le sue operazioni, salva il proprio stato ed esegue una opportuna routine per la gestione delle interruzioni (compito del sistema operativo). Questa routine serve a verificare la presenza del dato sulla periferica del dato e a iniziare il trasferimento nel registro interni al processore, fino ad arrivare in memoria. Alla fine dell'*interrupt* il processore ritorna alle sue operazioni normali. (PRO: lascia più libero il processore di operare; CONTRO: gestione, implementazione e controllo più complicati).

Tra le periferiche sono presenti anche i terminali, cioè qualunque dispositivo di puntamento (tastiera, mouse, video ...), e le stampanti.

Architettura software di un compilatore:

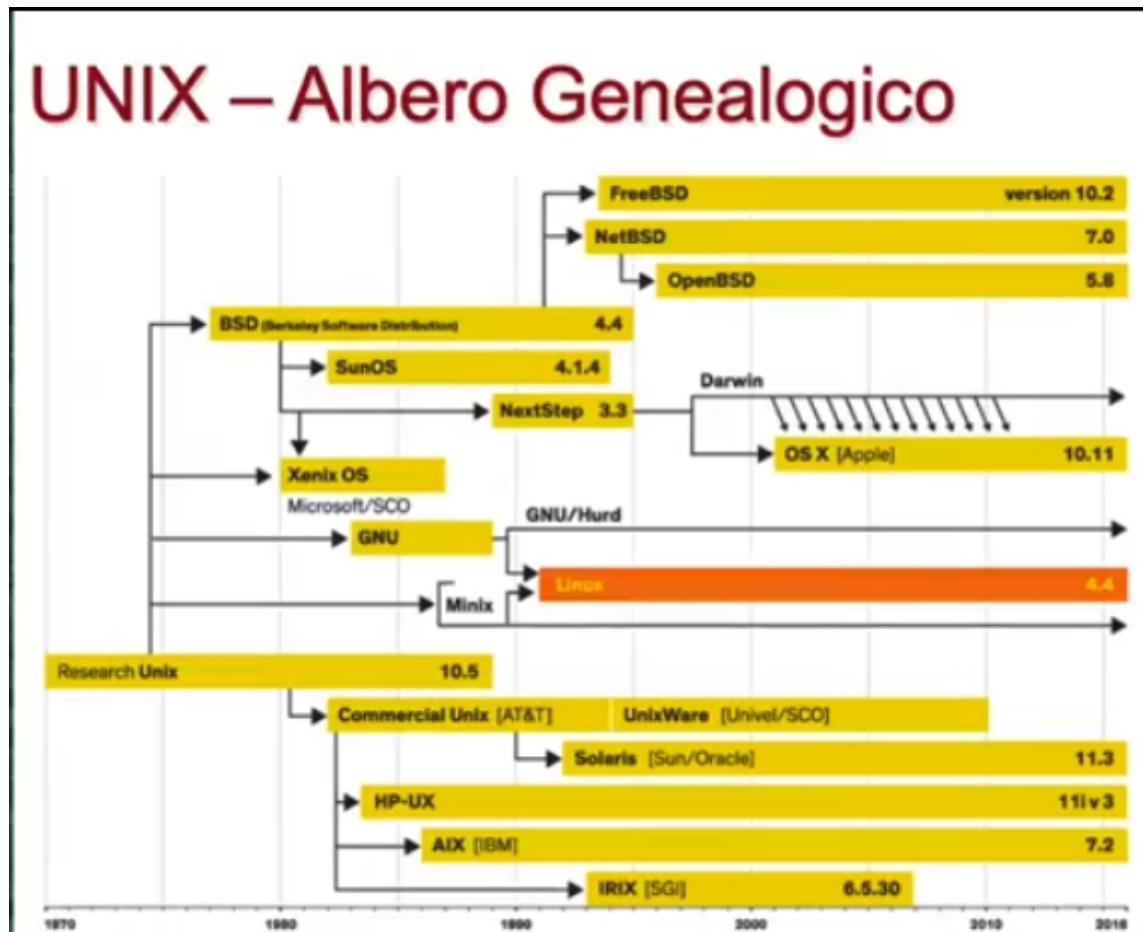


Con il termine *sistema operativo* intendiamo l'insieme di programmi che opera direttamente sulla macchina fisica, fornendo interfacce di alto livello e mascherandone le caratteristiche specifiche delle periferiche e del processore. Questo ad esempio è molto utile nella gestione delle periferiche, perché permette di fornire un metodo *consistente* delle periferiche, ovvero una modalità standard di interfacciarsi con le periferiche disponibili senza dover eseguire comandi di basso livello. Importante notare inoltre che il sistema operativo è l'unico elemento con accesso diretto alle risorse hardware, e si riesce ad accedervi in altro modo direttamente probabilmente abbiamo un problema.

Il sistema operativo è il modo che consente ai programmi di ottenere risorse dal calcolatore. La traduzione da programma a calcolatore avviene ad alto livello, in modo che non si debba scontrarsi con la gestione degli indirizzi o dei registri. Il sistema operativo quindi, dovendo gestire tutte le funzionalità di basso livello attraverso dei controlli di alto livello, opera un alto livello di astrazione del linguaggio macchina.

Storia dei sistemi operativi:

Nel 1982 Kernigham introduce Unix, prodotto dai laboratori AT&T, che al tempo avevano grandi esigenze di un ecosistema stabile e uguale su cui sviluppare applicazioni. Caratteristica principale: poteva servire più utenti contemporaneamente, usando il multitasking. Tutti gli utenti infatti hanno un ordine di esecuzione al processore, ma sono tutti nell'ordine del proprio utente. Da Unix poi si svilupperanno tutti i sistemi operativi che tuttora conosciamo (Come MS-DOS o Linux).



Architettura di un SO, che ad oggi è organizzato secondo un architettura a *strati* (anche detta *onion skin architecture*). Ogni strato fornisce un astrazione dello strato sui cui si appoggia e permette una chiara separazione tra interfaccia e implementazione delle diverse funzionalità, oltre che a fornire l'insieme di programmi e librerie.



Il *kernel* del sistema operativo, ovvero il cuore dell'OS si occupa di:

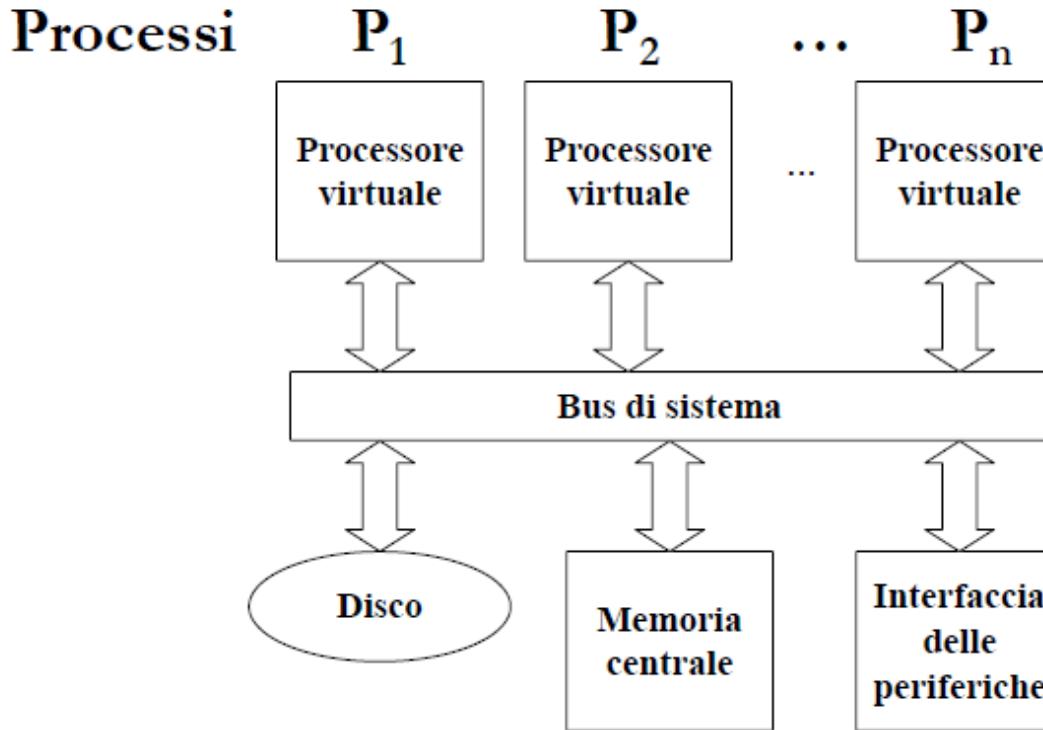
- gestione periferiche (I/O informazioni)
- gestione memoria (Fornisce/non fornisce al memoria in base al carico del momento, stabilisce il tipo di memoria da fornire)
- gestione dei processi

Un processo è un entità dinamica, contrariamente al programma, infatti viene causato dal codice in esecuzione (*programma*) e dal suo stato di esecuzione (es. valore delle sue variabili).

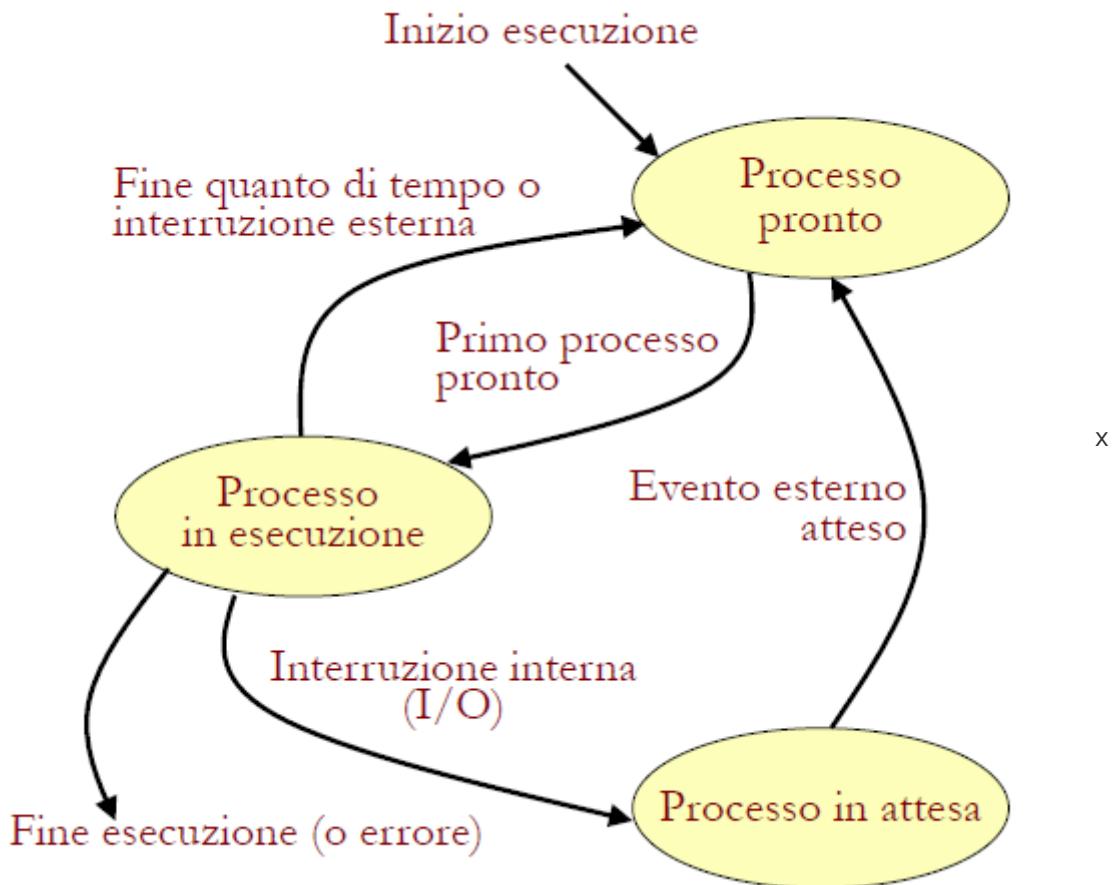
Un processo è quindi un insieme di due elementi:

- E: il codice eseguibile del programma (questo fa partire il fetch per creare un processo)
- S: lo stato del processo

Logicamente il processo non viene gestito direttamente dal processore reale, se no verrebbe meno il concetto di multitasking. Il processore crea dei processori virtuali che si comportano similmente e che possono essere assegnati a un processo.



Nel processore è possibile che ci sia un solo processo in esecuzione in ogni istante, mentre gli altri processi sono pronti o in attesa. Ad ogni processo viene assegnato un valore massimo di tempo di esecuzione, scaduto tale viene revocato il processore virtuale e assegnato a un altro processo. Questa tecnica è detta di time-sharing, viene eseguito dal nucleo che decide da quali processi devono andare in esecuzione determinando lo *scheduling*, che è sequenziale. La soluzione tipica per la gestione del tempo di esecuzione di processi è a turno (*round-robin*, tutti i processi in questo hanno la stessa priorità ad essere eseguiti, nella realtà non funziona perché alcuni processi hanno la priorità). La CPU viene rilasciata anche quando un processo sta aspettando un I/O da/verso una periferica.



In questa complicata gestione delle risorse prende parte anche la gestione della memoria, che pensa al partizionamento della memoria tra i vari processi che la richiedono garantendo la protezione/separazione fra le diverse zone allocate. Il gestore memoria gestisce anche la memoria che va assegnata ai processi, e quando i processi finiscono la memoria fisica si occupa di creare una memoria virtuale (più lenta di solito, chiamata SWAP o semplicemente memoria virtuale) e assegnare quella memoria.

THE C PROGRAMMING LANGUAGE:

Ci sono principalmente 3 tipi di programmazione:

- programmazione per HARDWARE, ovvero la programmazione di dispositivi fisici/logici il cui linguaggio di programmazione coincide con il linguaggio macchina.
- programmazione per FIRMWARE, ovvero la programmazione che usa un'insieme comune di istruzioni mediante assembly, ovvero linguaggio di programmazione molto basso
- programmazione per SOFTWARE, ovvero programmare in un linguaggio intermedio che simula le funzionalità della macchina fisica, che per questo permette maggiore flessibilità (nel senso che posso realizzare un gran numero di programmi diversi e con funzioni diverse), ma che ha un utilizzo delle risorse meno efficiente.

Il sistema operativo prende parte molto nell'ultimo caso, infatti è suo il compito di fornirci le periferiche necessarie a fornire la traduzione da linguaggio macchina (che usano ad esempio nella programmazione firmware) e il codice. L'OS si occupa anche dell'astrazione degli oggetti e/o delle istruzioni complesse di alto livello.

Diffusione dei linguaggi e Perché il C

Ci sono tantissimi linguaggi di programmazione creati o utilizzati per specifici utilizzi in cui sono molto apprezzati. Linguaggi più standard di medio-basso livello come il C o il C++ permettono una maggior comprensione del funzionamento della maggior parte degli altri linguaggi, che in alcuni casi forniscono astrazioni alle strutture di basso livello ancora presenti in questi.

Il C presenta una serie di elementi che lo rendono importante da imparare:

- permette l'allocazione dinamica e altri aspetti normalmente di alto livello, a un livello basso.
- gestione della memoria molto manuale
- linguaggio apprezzato/richiesto dalle aziende
- ha un astrazione che è posizionata tra il medio e il basso livello, motivo per cui è molto utile per ad. es. embedded system.

Il linguaggio C è stato creato nel 1972 da Kernighan e Ritchie ai Bell Tel. Labs.

Operazioni Logiche (algebra di Boole)

L'algebra di Boole è basata su tre operatori logici (NOT, AND, OR). Gli operandi possono assumere due valori: VERO e FALSO.

Gli operatori godono delle proprietà

- commutativa (es. A OR B = B OR A)
- distributiva (es. A AND (B OR C) = (A AND B) OR (A AND C))

Le tabelle di verità associano a tutti i possibili valori degli operandi il risultato

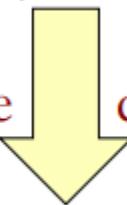
NOT(A) !A	
A	NOT A
0	1
1	0

A	B	A AND B	A	B	A OR B
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	1	0	1
1	1	1	1	1	1

soltanente NOT viene rappresentata con !, AND con && e OR con ||.

quando si valuta un'espressione l'ordine è il seguente (NOT, AND, OR), per esempio.

NOT Y AND Y OR NOT X

Regole  di Precedenza

((NOT Y) AND Y) OR (NOT X)

Per calcolare il risultato di un'espressione (per esempio NOT Y AND (Y OR NOT X)) si crea una tabella di verità, prima con i valori singoli per poi raggrupparli fino ad arrivare alla formula di partenza.

X	Y	NOT X	NOT Y	Y OR NOT X	NOT Y AND (Y OR NOT X)
1	0	0	1	0	0
0	1	1	0	1	0
0	0	1	1	1	1
1	1	0	0	1	0

Leggi di De Morgan

- A AND B = NOT ((NOT A) OR (NOT B))
- A OR B = NOT ((NOT A) AND (NOT B))

si possono dimostrare compilando la tabella di verità e osservando che le tabelle delle espressione ai due lati dell'uguale hanno stessi risultati a parità di input.

Tautologia: espressione che è sempre vera

Contradizione: espressione sempre falsa

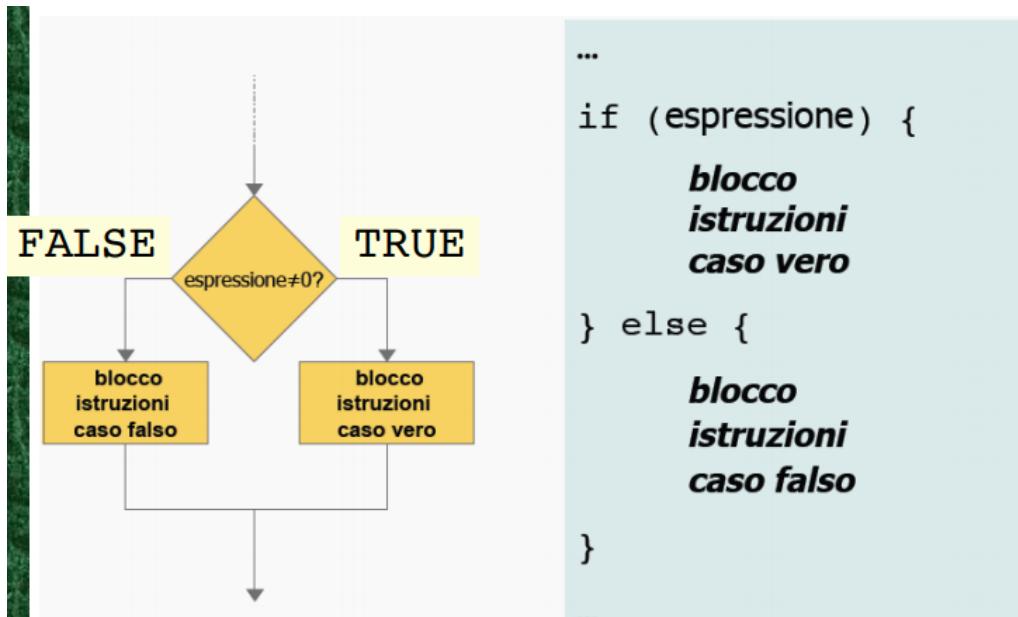
Codifica Semplici Algoritmi in C

istruzione di assegnamento:

```
1 | x = 23;  
2 | w = 'a';  
3 | y = z;
```

Costruttore if-else

diagramma di flusso:



L'espressione tra parentesi viene valutata, se vera viene eseguito il primo blocco, se falsa l'altro. Naturalmente si può utilizzare anche l' if da solo senza else. All'interno dell'espressione posso utilizzare gli operatori logici (&&, ||).

è sempre meglio utilizzare indentazione e parentesi graffe per una migliore leggibilità e per non commettere errori.

*Operatore ternario `? :` *è un altro modo di scrivere if-else, la sintassi è la seguente

```
1 | espressione1 ? espressione2 : espressione3; //questo equivale al seguente if-  
2 | else  
3 | if (espressione1)  
4 | { espressione2; }  
5 | else  
6 | { espressione3; }
```

dopo le parentesi graffe il ; non è necessario, ma se viene messo non c'è errore.

Precedenza degli operatori

In un'espressione vengono eseguiti prima gli operatori con precedenza superiore, se gli operatori sono dello stesso gruppo si usano le regole di associatività (da destra o da sinistra), le parentesi posso essere usate per modificare la precedenza.

Associatività da sinistra a destra significa che a parità di priorità l'espressione viene eseguita partendo da sinistra a destra.

Gruppo	Operatori	Associatività
Notazione Suffissa	() [] . ->	Da sinistra a destra
Unari	! ++ -- * & sizeof - (type)	Da destra a sinistra
Moltiplicativi	* / %	Da sinistra a destra
Additivi	+ -	Da sinistra a destra
Shifting	>> <<	Da sinistra a destra
Relazionali	< <= > >=	Da sinistra a destra
Uguaglianza	== !=	Da sinistra a destra
AND Logico	&&	Da sinistra a destra
OR Logico		Da sinistra a destra
Condizionali	? :	Da destra a sinistra
Assegnamento	= += -= ...	Da destra a sinistra

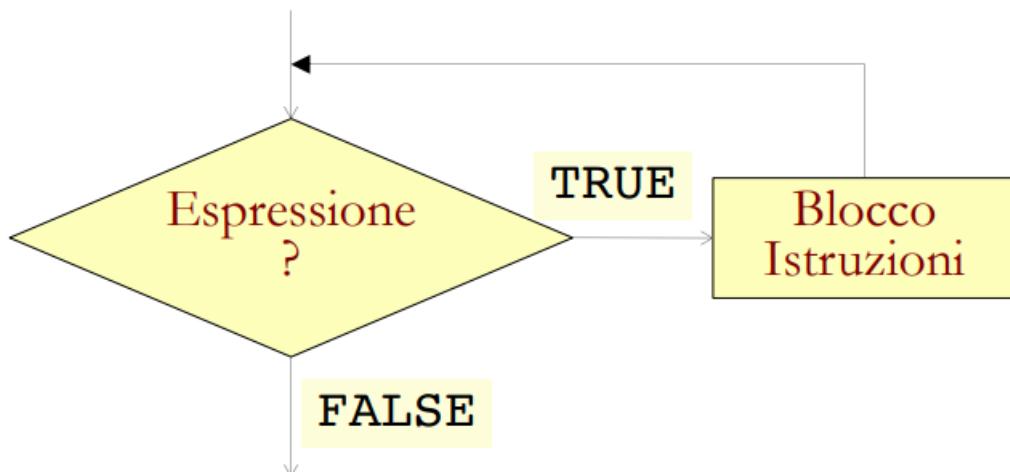
```

1 | if (a + b - 4 <= 9 && x < tot -1) // questa espressione è equivalente a
2 | quella sotto
3 | if (((a + b - 4) <= 9) && (x < tot -1) )

```

Istruzione Iterativa (ciclo)

il diagramma di flusso è il seguente (il ciclo si chiama while). Il blocco istruzioni viene ripetuto fino a quando l'espressione non diventa falsa.



(da pagina 38 a 56 un po' di esercizi noiosi)

Getchar e Putchar

```
1 //getchar legge il prossimo carattere inserito da tastiera  
2 c = getchar();  
3 //putchar stampa il carattere nello standard output  
4 putchar(c)
```

Esercizio scale:

Sia data una scala di N gradini. Si supponga di salire l'intera scala con passi da 1 , 2 o 3 scalini. In quanti possibili modi si può salire l'intera scala ?

risoluzione in modo semplice:

Devo capire in quanti modi possibili posso salire una scala con n gradini. Posso compiere passi da 1, 2 o 3 gradini.

Mi calcolo in quanti modi posso salire una scala formata da 1 2 o 3 gradini e basta.

Un gradino = 1 modo

Due gradino = 2 modo

Tre gradino = 4 modo

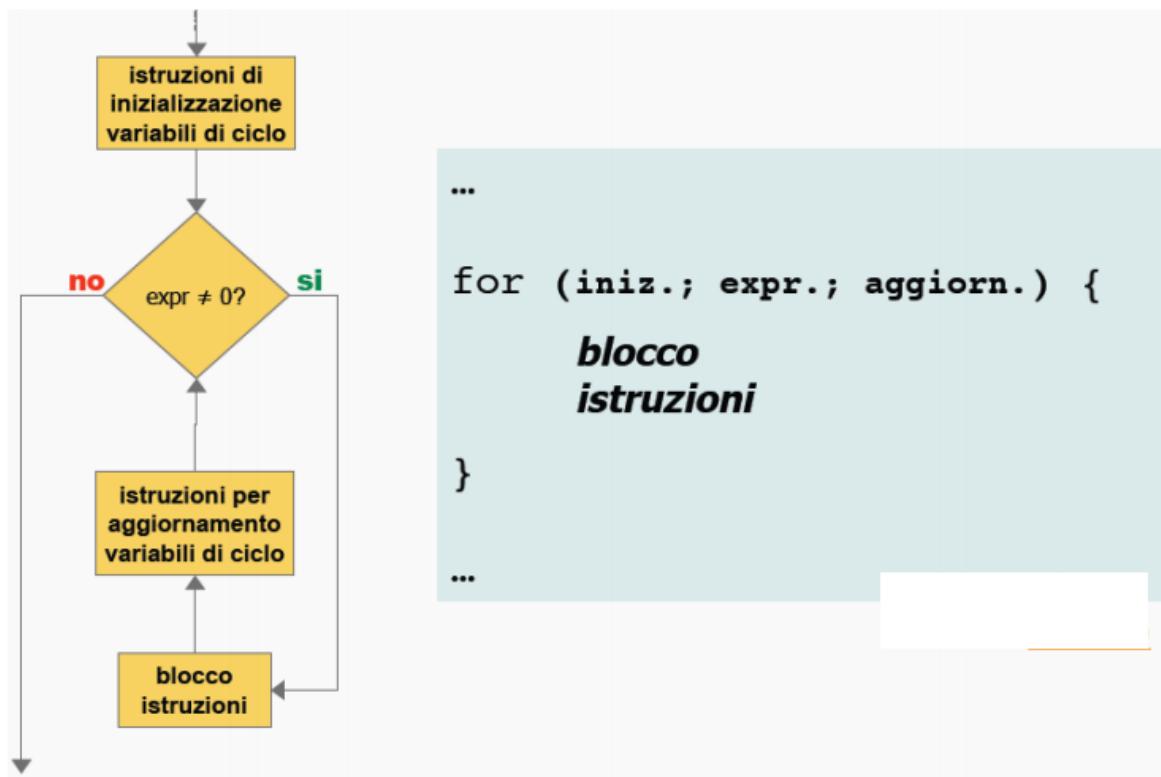
Per arrivare al quarto gradino ho solo tre possibilità: fare un passo da uno, da due o da tre, quindi devo arrivare al gradino 4-1, 4-2, 4-3, siccome so già quanti passi ci vogliono per arrivare in questi posti, basta sommarli assieme per capire il numero di passi per arrivare al quarto.

Aggiorno il numero di passi per il gradino n-1, n-2, n-3 e vado avanti. (se non si capisce chiedetemi che vi spiego meglio).

Strutture di controllo

Istruzione di ciclo: FOR

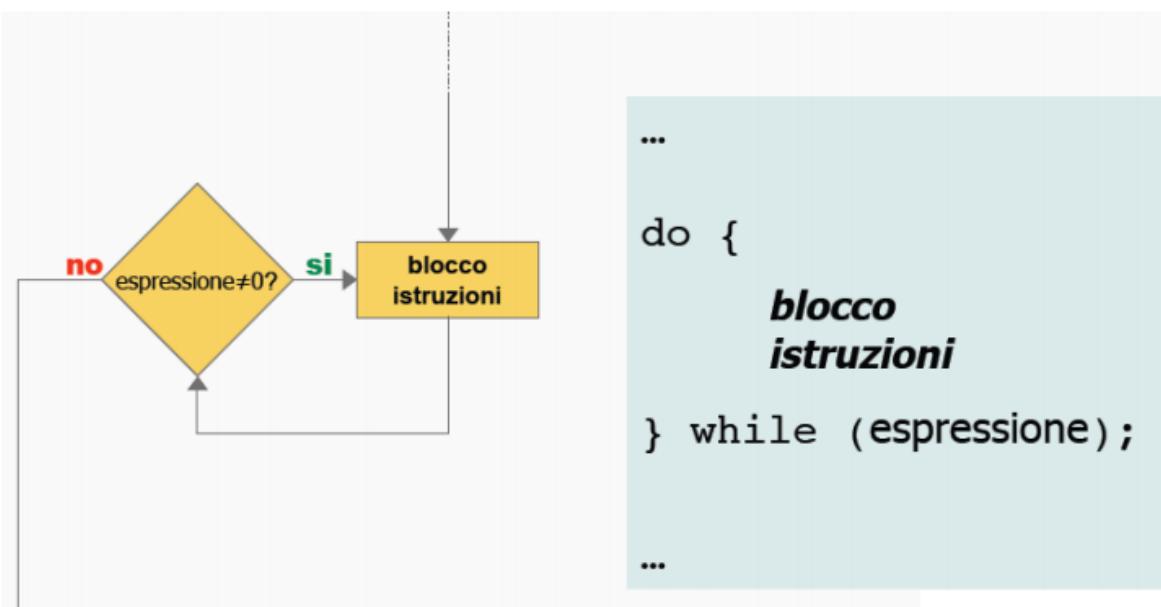
schema a blocchi e sintassi:



questo operatore è utile quando so a priori quante operazioni devo fare, in quei casi è più compatto rispetto ad un while.

Il costrutto DO-WHILE

schema a blocchi e sintassi:



```

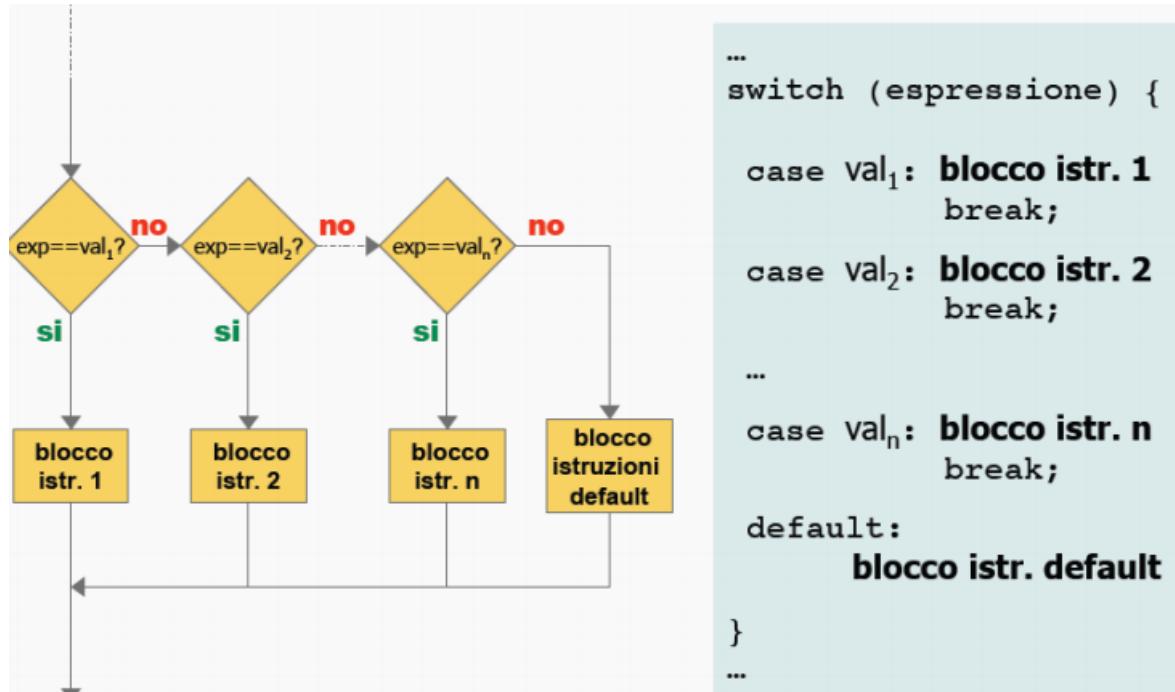
1 // un esempio
2 Contatore = 0;
3 do
4 {
5     scanf ("%c", &Dato);
6     Contatore++;
7 } while (Dato != '%');

```

si utilizza quando voglio eseguire un blocco di istruzioni almeno una volta, può essere utile quando devo fare un controllo sull'input da tastiera.

Il costrutto SWITCH

Va a sostituire un if-else multiplo, schema a blocchi e sintassi:



Il **default** (le istruzioni che vengono eseguite in caso che nessuna delle altre sia vera) è opzionale. I singoli case vengono eseguiti quando il valore dell'espressione è uguale a quello scritto appena dopo l'istruzione case. ATTENZIONE: bisogna mettere il break dopo il blocco istruzioni altrimenti si rimane all'interno dello switch (verranno valutati i case seguenti ed eseguito il default se presente).

Valuta solo variabili di tipo INT, quindi l'espressione deve avere come risultato un int.

Break: quando viene eseguito all'interno di un while, for, do, switch provoca l'uscita dall'istruzione

Continue: quando viene eseguito all'interno di un ciclo passa alla iterazione successiva

un linguaggio di programmazione può codificare qualsiasi algoritmo se ha: sequenza di istruzioni, if-else e while.

Array in C

Gli array possono essere paragonati a vettori e matrici in matematica. Da un punto di vista più concreto sono una sequenza di celle di memoria consecutive e omogenee. L'array è quindi un contenitore per *variabili dello stesso tipo*.

A ciascun elemento dell'array si accede tramite indice (esempio $a[i]$ è l'elemento alla posizione i -esima). Le parentesi quadre sono operatori ad alta precedenza (sono al primo livello della tabella).

Il primo elemento dell'array è quello in posizione 0, la macchina astratta prende l'indice e lo somma all'indirizzo della prima cella dell'array.

Prima di utilizzare le array bisogna dichiararle:

```
1 | int a[100]; // dichiara un contenitore a (array) che potrà contenere 100  
elementi di tipo int, il primo elemento lo si trova in a[0], l'ultimo in  
a[99]
```

il compilatore va a riservare la memoria per tutti questi elementi

Elemento	a[0]	a[1]	a[2]	a[3]	...
Indirizzo	1000	1001	1002	1003	...

Indici che spaziano da 0 a 99

In generale l'ultimo elemento dell'array è nella posizione n-1, dove n è la lunghezza dell'array stessa. All'interno delle parentesi si possono mettere delle espressioni. Un veloce esercizio:

```
int a[5];  
int i=1, b=5;  
a[i--] = ++b; //a[0] ? a[1]=?
```

SOLUZIONE: a[0]= indeterminato, a[1] = 6

ATTENZIONE: se vado oltre l'indice massimo dell'array accedo a celle di memoria che non appartengono all'array e il cui valore è indeterminato.

L'array in C non è un tipo, ma un costruttore di tipo.

Inizializzazione e stampa

si può inizializzare direttamente al momento della dichiarazione

```
1 | int a[5] = {5, 2, -5, 10, 234};  
2 | int b[4] = {5, 2, -5}; //un elemento non è inizializzato  
3 | int c[2] = {5, 2, -5}; // ERRORE: inizializzato un elemento che non  
appartiene all'array
```

per array grandi questo metodo diventa scomodo, quindi si usano i cicli per inizializzare.

anche per stampare un array devo utilizzare un ciclo

```
1 | printf("%d", a); // errato perchè a è un array  
2 |  
3 | int i=0; // questo è il procedimento corretto  
4 | while (i<5){  
5 |     printf("%d",a[i]);  
6 |     i++;  
7 | }
```

esercizi sulle array dalla slide 31 a 41.

Array dinamici: il C permette inizializzare la dimensione di un array durante l'esecuzione di un programma (per esempio chiedendo la dimensione da tastiera).

Array multidimensionali

Le array di due dimensioni corrispondono alle matrici in matematica. Si dichiarano nel seguente modo:

```
1 int a[N][M]; //N numero righe M numero colonne  
2  
3 //è anche possibile definire più dimensioni  
4 int a[10][5][20];
```

Come per gli array ad 1 dimensione, anche questi possono essere inizializzati nella fase di dichiarazione:

```
1 int a[4][5] = { {2, 5, -8, 7, 6},  
2                 {3, 10, 7, 6, 1},  
3                 {-1, 8, -8, 5, 3},  
4                 {2, 5, 8, 4, 2}  
5             };
```

Per semplicità possiamo immaginare l'array in due o più dimensioni, ma la macchina astratta del C memorizza gli elementi uno dietro l'altro. Per esempio l'array creato sopra verrà memorizzata nel seguente modo:

a_{00}	a_{01}	a_{02}	a_{03}	a_{04}	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{20}	a_{21}	a_{22}	a_{23}	a_{24}	a_{30}	a_{31}	a_{32}	a_{33}	a_{34}
2	5	-8	7	6	3	10	7	6	1	-1	8	-8	5	3	2	5	8	4	2

altri esempi di inizializzazione corretta e sbagliata:

```
1 int D[][]={1,2,3,4}; // errata  
2 int E[2][]={1,2,3,4}; //errore non viene specificato il numero di colonne  
3  
4 int F[][4]={{1,2,3,4}}; //va bene  
5 // in C nella dichiarazione di un array bisogna valorizzare tutte le  
// dimensioni, si può fare a meno di quella più a sinistra
```

le array possono anche essere inizializzate con dei cicli

```
1 int main(int argc, char *argv[]){  
2     int matrice[10][5];  
3     int i=0,j=0;  
4     while (i<10)  
5     {  
6         j=0;  
7         while (j<5)  
8         {  
9             printf("%d ",matrice[i][j]);
```

```
10         j++;
11     }
12     printf("\n");
13     i++;
14 }
15 }
```

esercizi da pagina 61 in poi

Stringhe in C

Un array di char può essere rappresentata con una stringa (per esempio "hello"). L'ultimo carattere deve essere il carattere nullo '\0'. Questo carattere serve alle varie funzioni per capire dove terminerà la stringa. Quindi quando vado a creare una stringa per memorizzare n caratteri ne serviranno n+1 (uno lo uso per il carattere nullo).

Esiste un modo semplificato per inizializzare un'array di caratteri come stringa:

```
1 | char mia_stringa[] = "Ciao a tutti!";
```

Questo mi memorizza automaticamente lo spazio per il miei caratteri più il carattere terminatore. Quindi il risultato sarà:

Codice ASCII

C	i	a	o		a		t	u	t	t	i	!	
67	105	97	111	32	97	32	116	117	116	116	105	33	0

L'inizializzazione vista sopra è molto più veloce ed è equivalente ad inizializzare nel seguente modo:

```
1 | char mia_stringa[] = {'c', 'i', 'a', 'o', ' ', 'a', '
, 't', 'u', 't', 'i', '!', '\0'};
```

Se non specifico il numero all'interno delle parentesi quadre quando dichiaro l'array il compilatore va a riservare uno spazio pari al numero degli elementi con cui l'array viene inizializzato. Nel caso delle stringhe posso anche dichiarare esplicitamente la dimensione di memoria da riservare:

```
1 | char frase[20] = "Ciao a tutti!";
```

Bisogna stare attenti che un elemento (dei 20 messi a disposizione per l'array) sarà occupato da '\0' e poi gli elementi in più saranno lasciati vuoti.

ATTENZIONE: se non specifico né il numero di caratteri (all'interno delle parentesi quadre) né assegno alla stringa un valore, il compilatore da un errore, perché non sa quanta memoria riservare.

```
1 | char parola[]; // ERRORE
```

per stampa le stringhe si usa %s:

```
1 | printf("%s", mia_stringa); //questo non sarebbe possibile se non ci fosse il
carattere terminatore perché non saprei dove fermarmi
```

quando faccio scanf non bisogna mettere la & perché la stringa è un array e la variabile con il suo nome è già un indirizzo.

```
1 | scanf("%s", parola);
```

Rappresentazione di informazioni

In un calcolatore le informazioni vengono rappresentate sotto forma di dati, codificati in un linguaggio comprensibile al calcolatore. Per permetterci di interpretare le informazioni i dati devono essere decodificati. Quindi ci sono diversi livelli di decodifica che partono dall'hardware fino ad arrivare ad informazioni interpretabili a noi umani.



I tipi di dato che il calcolatore può interpretare direttamente sono :

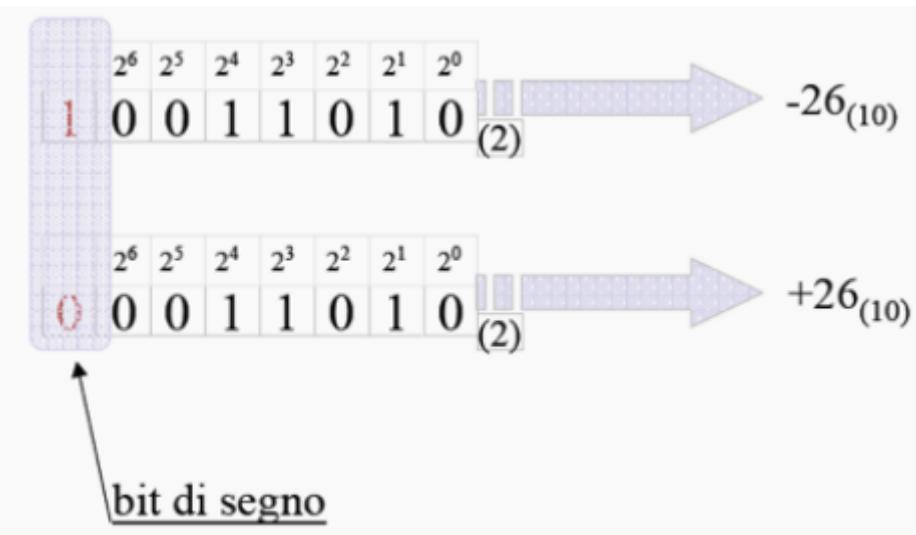
- booleani
- numeri interi
- numeri frazionari
- caratteri

Per questi dati la codifica è gestita direttamente dall'HW, per tipi di dato più complessi si usa una rappresentazione di tipo software.

Interi

Sono rappresentati da una sequenza finita di bit. 8 bit (un byte) permettono di rappresentare i valori da 0 a 255. Solitamente per gli interi positivi si usano 4 byte (32 bit), quindi i numeri vanno da 0 a 4.294.967.295. (questo implica che all'interno dei calcolatori i numeri sono finiti).

Per rappresentare anche i numeri negativi, si utilizza il primo bit come bit di segno (0 per i numeri negativi, 1 per i positivi)



In realtà nei calcolatori non si usa questa rappresentazione ma quella in complemento a due con i seguenti vantaggi: non c'è un doppio zero, non c'è bisogno di una circuiteria specifica. Esempio:

dec	binario
7	0111
6	0110
5	0101
4	0100
3	0011
2	0010
1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

Per decodificare i valori positivi si procede nel modo normale, per quelli negativi si decodifica e poi si sottrae 2^N-1 . Per invertire i numeri si invertono gli zeri con gli uno e si somma uno.

Numeri frazionari

I dati con numeri dopo la virgola vanno rappresentati in maniera opportuna, ci sono due tecniche:

- **virgola fissa:** si dividono i bit che rappresentano i valori interi da quello per i valori dopo la virgola
- **virgola mobile:** la maggior parte dei bit viene usata per le cifre rappresentative del numero, gli altri per sapere dove mettere la virgola.

Esempio codifica virgola fissa:

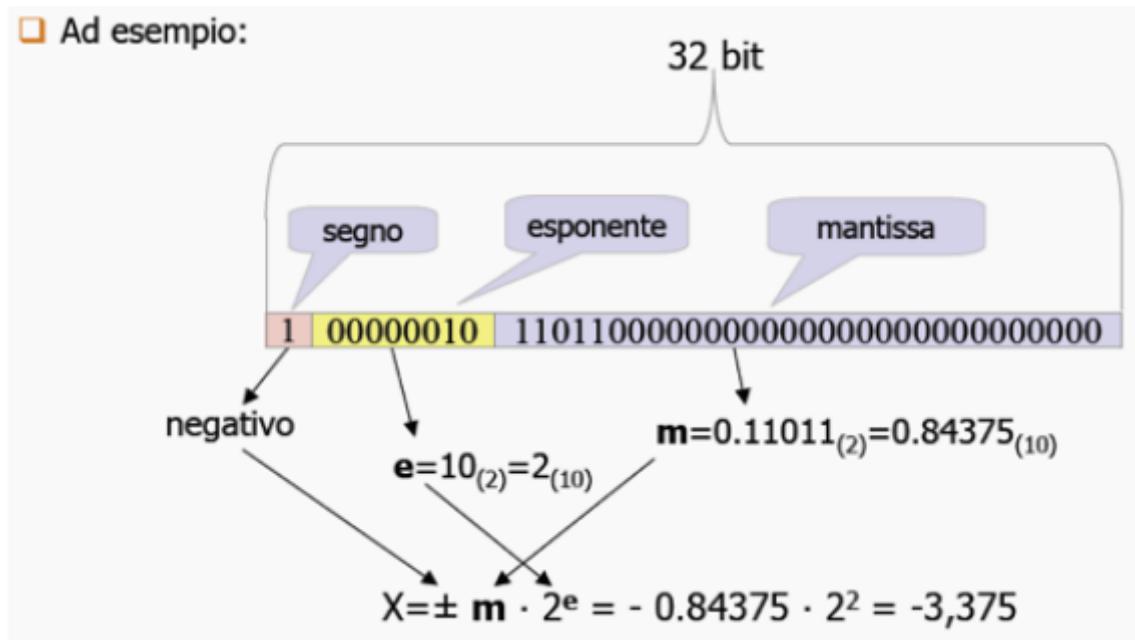
Si ha ad esempio $1001.1010_{(2)} = 9.625_{(10)}$ secondo la codifica:

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
8	4	2	1	0.5	0.25	0.125	0.0625
1	0	0	1	1	0	1	0

(2)

Per la virgola mobile solitamente vengono utilizzati 32 bit, 1 per il segno, 8 per l'esponente e il resto per la mantissa

□ Ad esempio:



Quindi la mantissa rappresenta numeri da 0 a 1, che verranno moltiplicati per 2^e in modo da ottenere il numero desiderato.

$$m = 0.11011_{(2)} = 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} = 0.84375$$

un numero si dice normalizzato se l'esponente è diverso da 0, la mantissa è compresa tra 1 e 2 l'intervallo dei numeri è

$$(-2^{128}, -2^{-126}] [2^{-126}, 2^{128})$$

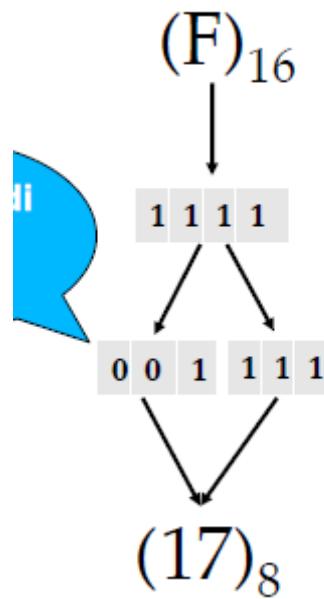
Caratteri

per codificare i caratteri si utilizza la tabella ASCII, i primi 128 valori sono fissi i successivi rappresentano la tabella ASCII estesa con caratteri più specifici (per esempio c'è una tabella ASCII estesa con i caratteri è, ò, à...).

attualmente si utilizza l'UNICODE che utilizza 2 bytes per ogni carattere e permette di non avere tabelle diverse per ogni regione del mondo.

Conversione di Basi

Come si fa a cambiare da base Esa a Ottale? Prima trasformo in binario e poi in ottale, infine l'ultimo bit lo attacco agli altri gruppi in modo da creare dei gruppi di bit tutti da 3.



Raggruppo fino a che posso, poi aggiungo zero. in questa maniera riesco sempre a costruire triplette.

$0 + 0 = 0$
$0 + 1 = 1$
$1 + 0 = 1$
$1 + 1 = 0$ con riporto (<i>carry</i>) di 1

Tipi di Dato

Il tipo di dato è un insieme di valori che può assumere una variabile, i cui elementi possono fare operazioni esserne l'oggetto. In base all'oggetto di cui ho bisogno e in base alle operazioni che devo eseguire con quei dati allora dichiaro tipo di dati diversi, in base allo scopo e alla funzione.

Classificazione:

- **built-in** (predefiniti): interi, caratteri, tipi strutturati (array o struct), etc.
- **user-defined**: definiti dal programmatore, creati a piacere per uno scopo preciso.
- ogni linguaggio offre diversi tipi di dati che possono essere predefiniti o semplici.

I tipi sono necessari per interpretare una serie di bit in memoria, e in base al quale il compilatore o calcolatore lo traduce. Questa è un'operazione necessaria per l'utente finale che ha bisogno di un'astrazione, ma poco utile a bassissimo livello, infatti tutte le informazioni sono codificate in binario. In secondo luogo i tipi sono importanti anche perché permettono di comunicare quanta memoria si necessita, e quindi quanta e che richiesta di memoria posso fare.

Un lato positivo di questi tipi di dato è che possono catturare qualche errore logico, senza causare problemi durante l'esecuzione(ad esempio se il tipo ha dei parametri che vengono utilizzati nel modo sbagliato è più facile trovare l'errore dopo, ma allo stesso modo è più facile evitarlo direttamente). Poi per il fatto che il linguaggio C è statically typed, allora se si utilizza un oggetto come una cosa diversa da quella dichiarata, l'errore viene beccato direttamente dal compilatore.

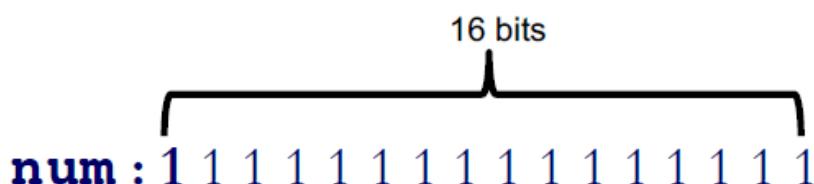
Nonostante questo, e per un motivo pratico di utilizzo delle variabili, c'è il *permesso di casting* ovvero la possibilità di programmaticamente forzare un tipo su altro tipo.

Ci sono una serie di tipi di dato predefiniti:

- *int*:
 - concetto matematico numero intero
 - ha operatori come `+-*%, %==!=><=<=`
 - normalmente usa una parola di memoria (1byte, normalmente, ma poi cambia anche da macchina fisica)
- *float* : equivalente dei numeri frazionari
- *double* : (float con doppia precisione sui frazionari, di conseguenza la sua lunghezza di memoria dovrebbe essere il doppio di quella del float)
- *char*

Un ulteriore modo per caratterizzare il tipo di valori è attraverso la modifica di memoria riservata per la variabile:

- *short & long* : (minor/maggiore memoria)
- *signed & unsigned*: (con/senza segno. Ad esempio se ho bisogno di un contatore allora posso evitare di avere il bit che si occupa del segno perché sarà sempre positivo)
 - generalmente il tipo di variabile *int* e derivate sono *signed int* di default, quindi serve specificare questa caratteristica solo se si vuole una variabile *unsigned*. Questo è importante per come vengono rappresentati i numeri in virgola mobile, che contengono il segno sul primo bit, facendo quindi una variabile *unsigned int* si possono avere valori più grandi nello stesso numero di bit.



La variabile **num** può essere interpretata come:

-1 se **num** dichiarata come **signed int**
oppure
65535 se **num** dichiarata come **unsigned int**

- Da notare come tutti i valori massimi dei tipi di dati predefiniti è contenuto nella libreria *limits.h*

```

1 #include <limits.h>
2 // racchiude i valori minimi e massimi della mia variabile
3 // dichiara il valore massimo e minimo del valore della variabile in una
4 // certa architettura
5 sizeof(nomevariabile); // restituisce il numero di byte
6 // es. char = 1, in generale restituisce il numero di byte assegnati a
7 // una variabile passata al costrutto.
8 // anche se il valore unsigned viene decrementato di uno, riceverò un
9 // valore positivo, nonostante fosse stato inizializzato a zero.
10 unsigned long int x;
11 printf("%d", x); // print valore intero della variabile e con il
12 // rispettivo sengno, anche se dichiarata unsigned. Questo avviene perché
13 // converte il valore da long a int.
14

```

Si noti anche la differenza fra float e double:

- float (6 cifre di precisione)
- double (15 cifre di precisione)
- Lo spazio di memoria sarà quindi (in ordine crescente): *float < double < long double*

NOTA BENE CHE: è rischioso usare un `if (a==b)`, in cui `a` di tipo *int* e `b` di tipo *float*, perché la condizione potrebbe non risultare mai corretta, siccome generalmente il tipo *float* ha molta più precisione del tipo *int* e anche che l'uguaglianza per tornare vera, deve necessariamente avere tutte cifre di `a` uguali a tutte le cifre di `b`.

Inoltre il tipo di dato *char* viene usato come *ununsigned int* e quindi ha senso a scrittura `A, B` in cui si usa il valore numerico del carattere ASCII.

Conversioni di Tipo

Come si diceva precedentemente il tipo di dato consente di fare un controllo sulla variabile, ma può anche avvenire di aver la necessità di una conversione fra variabili, per cui si è citata l'operazione di *casting*. Questa operazione segue un meccanismo automatico, in modo che io non debba preoccuparmi di quello che succede a basso livello per la conversione, e permette una corretta conversione tra variabili non uguali.

Il C fornisce un metodo per il *casting* che però necessita di attenzione al momento dell'utilizzo, per fare in modo che le variabili non perdano valore. Generalmente il C lavora con le regole della precedenza al momento dell'assegnazione, quindi viene convertito il valore a sinistra dell'operatore di assegnamento(`=`).

NOTA BENE: le conversioni automatiche possono essere molto particolari, quindi prestare molta attenzione al tipo di dato che tornano. Il problema è sempre la codifica di oggetti, come ad esempio *int* o *float* (che si presuppone una perdita di cifre dopo la virgola), che sono anche piuttosto simili, ma quando si inizia a dover comparare un *int* con un *char* inizia a essere più complicato.

Per fare questo si seguono delle regole che il C segue e a cui noi quindi ci atteniamo.

```
int i=10,j=3; float f=3.0, f1;  
f1 = i + f;//conversione implicita  
f = j/i; //conversione implicita
```

Facendo una conversione implicita, il compilatore si occupa di convertire le variabili in modo che siano dello stesso tipo.

- **Regole di conversione implicita:**
 - Si converte temporaneamente l'operando di tipo inferiore facendolo divenire di tipo superiore;
 - ogni variabile di tipo **char** o **short** (incluse le rispettive versioni **signed** o **unsigned**) viene convertita in variabile di tipo **int**;
 - se dopo l'esecuzione del passo 1 l'espressione risulta ancora eterogenea rispetto al tipo degli operandi coinvolti, rispetto alla gerarchia:

**int < long < unsigned < unsigned long <
float < double < long double**

Il risultato dell'espressione avrà tipo uguale a quello di più alto livello gerarchico.

Si eseguono tutte le azioni in ordine in modo che le variabili di tipo inferiori diventino superiori, di conseguenza se devo fare:

```
1 int a == float b;  
2 int a => float a;  
3 float a == float b;
```

Per rendere meglio comprensibile/visibile o per rendere il codice più ordinato/leggibile, si usa la keyword (*cast*).

```
1 float media;  
2 int num1=5, num2=3;  
3 average = (num1+num2)/2; // in questo caso ci affidiamo alla conversione  
implicita, il valore mancherà di cifre decimali  
4  
5 // se invece  
6 average = (float) (num1+num2)/2; // risultato corretto, ma anche più  
leggibile.
```

L'operatore di casting è uno di quelli che ha la precedenza su molto, nell'ordine in cui si eseguono le operazioni (tabella di precedenza).

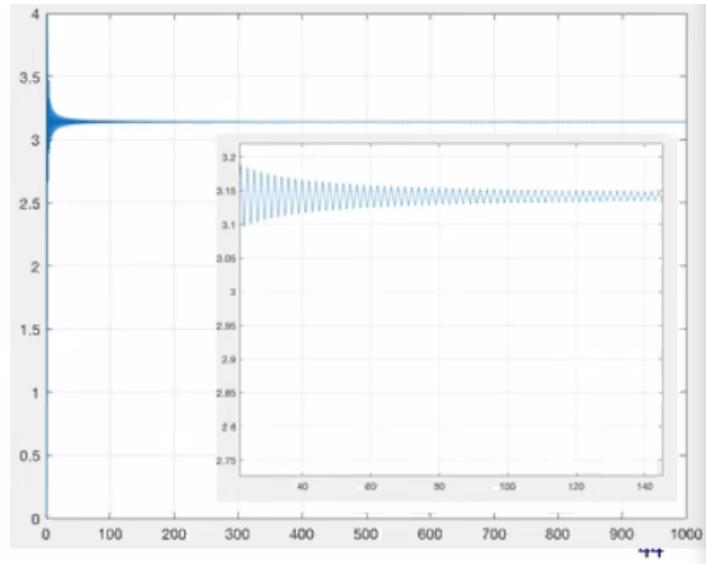
Ad Esempio:

Per calcolare il valore di

$$\pi$$

con precisione a piacere usando l'approssimazione di Leibniz:

$$\frac{\pi}{4} = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}$$



Questo metodo (ma soprattutto dal grafico di questo metodo) si comprende che è possibile calcolare il pigreco con un certo grado di precisione. Per avere 4 cifre piuttosto convergenti bisogna, come si vede appunto dal grafico, usare almeno un centinaio di cifre.

Implementazione del metodo di Liebnitz: (nel main)

```
1 int i;
2 int precisione;
3 int segno = -1;
4 float pigreco = 1;
5
6 scanf("%d", &precisione);
7
8 for (i=1; i<=precisione; i++){
9     pigreco += segno * 1/(2*i+1);
10
11    printf("%f\n", pigreco*4);
12    segno = (-1) * segno
13 }
```

In questo programma, in cui si usa la conversione implicita, che porta a un errore di approssimazione, perché la conversione implicita nel caso di divisione tra due variabili **int** causa un numero intero che logicamente manca del resto dalla divisione stessa.

Con il casting in modo esplicito, il programma funziona meglio, e soprattutto non lascia posto alla possibilità che il compilatore possa ogni tanto fare correttamente e in altri casi sbagliando la conversione implicita.

Tipi user-defined in C

Per usare variabili più complesse che siano definite puramente dal programmatore e contengano al loro interno più variabili e di più tipi, sia *user-defined* sia predefinite si utilizza uno *struct*, ovvero variabili strutturate.

Tutto ciò permette di creare variabili molto complesse, con una serie di caratteristiche diverse fra di loro.

```
1 struct Impiegato{  
2     char Nome[20];  
3     char Cognome[20];  
4     float Stipendio;  
5     char CodiceFiscale[16];  
6 }
```

Alla fine di questa definizione di struttura, che descrive come sarà questa variabile, non viene allocata nessuna memoria, ma è solo un passo formale del C.

Però com'è che si fa ad accedere a queste variabili? Ad esempio negli array sapevo come erano i campi e l'unica cosa a cui dovevo stare attento era di non sorpassare il numero di spazi assegnati nel richiamare i caratteri. Se non più seguire una dichiarazione questa funzione *struct* non ha valore (vuol dire che la dichiaro e poi posso iniziare a usarla).

```
1 struct Impiegato{  
2     char Nome[20];          // DEFINIZIONE  
3     char Cognome[20];        // DEFINIZIONE  
4     float Stipendio;         // DEFINIZIONE  
5     char CodiceFiscale[16];   // DEFINIZIONE  
6 } amministratore, segretario; //DICHIARAZIONE  
7  
8 struct Impiegato staff1,staff2; //DICHIARAZIONE  
9 //Da qui in poi posso iniziare a usarla, siccome la ho dichiarata
```

All'interno dello *struct* non importa l'ordine (a meno che una variabile dipenda da un'altra, in questo caso la variabile indipendente va dichiarata prima). infatti al compilatore non cambia se metti prima il nome o il cognome, perché poi la variabile viene richiamata in ugual modo. (Nessun cambio di ordine della definizione porta a miglioramento delle prestazioni, il programma ci mette ugual tempo a definire prima un campo di un altro).

Per indicizzare i campi all'interno dello *struct* viene usata la *dot notation*:

```
1 impiegato.Stipendio = 0.0;    // assegno al campo stipendio dello struct  
2 impiegato uguale a zero  
3 // dopo aver inizializzato queste cose, poi posso stilare l'albero sintattico  
4 // per usare lo struct come variabile normale:  
5  
6 if (impiegato.Stipendio < 2000){  
7     impiegato.Stipendio = 1.5 * impiegato.Stipendio;  
8 }
```

Il *dot operator* è in cima alla tabella delle precedenze di traduzione, in linea con l'ordine di precedenza delle parentesi.

Il constructo *struct* occupa in memoria esattamente la somma della memoria occupata dagli elementi che lo compongono.

Strutture complesse

Le strutture complesse sono variabili strutturate che usano a loro volta variabili strutturate al loro interno. Ad esempio:

```
1 struct lato{  
2     int lunghezza;  
3     int x;  
4     int y;  
5 }  
6 struct rettangolo{  
7     lato lato-lungo;  
8     lato lato-corto;  
9 }
```

Se all'interno dello *struct* è dichiarato un campo che a sua volta è uno *struct*, allora il dot operator sarà riutilizzato fino ad ottenere il campo ricercato.

```
1 struct Data{          // definizione Data  
2     int giorno;  
3     int mese;  
4     int anno;  
5 };  
6  
7 struct Data mia-data; // dichiarazione Data  
8  
9 struct Agenda{        // definizione Agenda  
10    char[10] NomeEvento;  
11    mia-data DataEvento;  
12 }  
13  
14 struct Agenda mia-agenda; // dichiarazione Agenda  
15  
16 mia-agenda.mia-data.mese = 12;  
17           // accedo al campo di Agenda, che è sua volta una variabile  
           // strutturata.
```

Utilizzo del *typedef*:

L'idea è quello di creare un meccanismo sintattico formale in modo che io possa usare degli altri nomi per definire e chiamare le variabili, che possono essere variabili semplici o strutturate.

Di solito il *typedef* è posto tra la definizione di costanti e il main().

Il *typedef* funziona creando una tabella di sinonimi che rappresentano in realtà tipi di dati predefiniti o user-defined, ma che hanno il lato positivo di comparire al programmatore con un nuovo nome, appunto un sinonimo. Il nuovo tipo eredita le operazioni del tipo di partenza e ne è uguale, eccetto per il nome a cui ci si riferisce per richiamarlo. Ad esempio:

```
1 | typedef int intero; //definisco alias per 'int' chiamato intero
2 | intero numero; // definisco un tipo 'intero' esattamente come fosse un
   | qualsiasi altro tipo
```

Il *typedef* non si basa sulla ereditarietà, cioè non funziona che gli elementi prendono il valore di quelli di partenza perché lo "ereditano", ma proprio perché il compilatore crea una tabella di sinonimi che viene usata al momento della compilazione per sostituire direttamente il tipo di dato originale al posto del suo alias presente nel codice.

Dove si usa:

- nel caso di variabili che ricorrono spesso e necessitano di essere più comprensibili/utilizzabili.
- evitare di definire più volte lo stesso tipo di variabile

```
1 | typedef char[10] NomePersona;
2 | typedef NomePersona CognomePersona;
3 | // adesso il tipo NomePersona è uguale al tipo CognomePersona che a sua volta
   | è uguale a dire char[10]. Ma allo stesso modo posso anche dichiarare:
4 | NomePersona Persona1, Persona2, Persona3;
```

A questo punto il programmare diventa sempre più a pensare a che cosa scegliere, in modo che il programma sia il più comprensibile e utilizzabile possibile.

```
1 | typedef int vettore[20];
2 | typedef vettore Vettore20per20;
3 | typedef vettore Nome;
4 | // osso dichiarare una serie di cose usando sempre il tipo vettore
5 |
6 | // a questo punto posso io a priori dichiare delle costanti a un tipo di dato
   | in modo che mi venga più comodo il suo uso.
```

Utilizzo del *enum*:

Un sottotipo di variabile *struct* è l'*enum* per indicare una variabile strutturata che può prendere solo un valore finito e predeterminato di valori.

Questo tipo di variabile è molto utile ad esempio se si vuole usare una lista di cose di cui si conosce tutto, e che deve essere presa come variabile comprensibile al compilatore all'interno del programma. Un'altra caratteristica è che all'interno della variabile *enum* il compilatore assegna a ogni valore, un'alias di valore *int*.

Ad esempio un *enum* potrebbero essere i giorni della settimana a cui ad ogni giorni si associa un valore *int*, in modo da poter fare facilmente cicli con i numeri.

```
1 | enum Settimana {
2 |     lun, mar, merc, giove, sab, dom
3 |         // lun = 0; mar = 1; merc = 2, giove = 3; ...
4 | }
```

Puntatori

I puntatori sono un elemento necessario per la programmazione in C per:

- costruzione di funzioni
- allocazione dinamica
- efficienza dei programmi in memoria

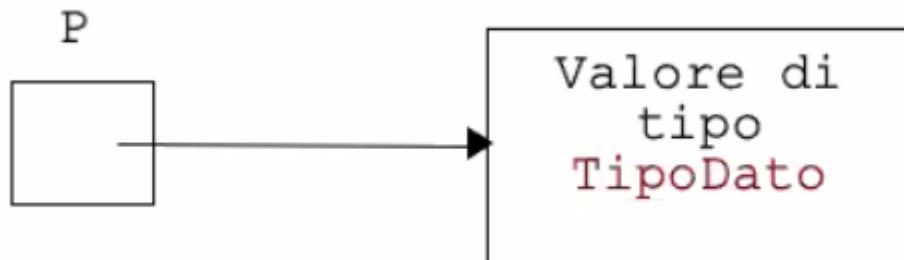
I puntatori hanno il lato positivo nel fatto che permettono un grandissimo uso della memoria e quindi aiutano a rendere il programma più efficiente, al contrario però il loro utilizzo potrebbe causare errori di allocamento di memoria o di sovrascrittura di un luogo di memoria. Di conseguenza i puntatori sono un punto di forza del C, ma anche punto di debolezza.

il concetto di puntatore si deve vedere come una variabile che contiene essa stessa l'indirizzo di memoria. Adesso la variabile puntatore contiene invece solo l'indirizzo, lasciando quindi libero lo spazio necessario all'allocazione di un valore.

Questo concetto è molto diverso da quello di una normale variabile, nella quale era contenuto sia il tipo di dato sia il suo indirizzo di memoria.

ATTENZIONE CHE: il puntatore non dà garanzie che all'interno di quello spazio di memoria ci sia qualcosa o se quello che è presente sia comprensibile.

La rappresentazione più efficace del puntatore è come quella che segue: il quadrato rappresenta il puntatore, la freccia invece indica a cosa punta il valore. Sopra il quadrato è presente l'identificatore del puntatore. La freccia arriva in una zona con un solo tipo di dato e poi identificata da un nome.



```
1 // sintassi per la dichiarazione di un puntatore:  
2 TipoDato *Puntatore;
```

Il tipo di dato che viene assegnato al puntatore è necessario affinché il compilatore del C riesca a tradurre quello che viene indicato dalla variabile stessa. Di fatto, il puntatore non ha un tipo predefinito.

Un puntatore è definibile anche attraverso un *typedef*:

```
1 typedef TipoDiDato *TipoDiDatoPuntato;
```

Operatore ***** e **&** (dereferenziazione e indirizzo di)

Questi operatori servono per operare con i puntatori e con gli indirizzi di memoria.

L'operatore di **dereferenziazione** ha principalmente due funzioni:

- nella dichiarazione del puntatore, indica che la variabile dichiarata è un puntatore

- in qualsiasi altro uso, per recuperare il valore della variabile, ovvero il contenuto dello spazio a cui sta puntando.

Utilizzo generale:

```

1 // nella dichiarazione, indica che si sta indicando un puntatore
2 TipoDiDato *PuntatoreTipoDiDato;
3
4 // Presa la variabile 'IndirizzoAlTipoDiDato' come un indirizzo
5 PuntatoreTipoDiDato = IndirizzoAlTipoDiDato;
6
7 // viene assegnato un TipoDiDato, al valore del punatore
8 *PuntatoreTipoDiDato = valoreTipoDiDato;
9
10 // Per accedere al valore del puntatore si deve usare ancora deferenziare:
11 printf(*PuntatoreTipoDiDato); // Stampa: ValoreTipoDiDato

```

Da come si vede nell'esempio è importante specificare il tipo di dato del puntatore, perché l'operatore di dereferenziazione non fornisce queste informazioni. Se non si specifica il tipo di dato non si stanno fornendo al compilatore informazioni necessarie per *tradurre* il contenuto del suo valore.

L'operatore **indirizzo di** al contrario permette di calcolare un indirizzo di memoria di una data variabile.

```

1 Puntatore *P;
2 P = &x;
3 // L'operatore '&'amp; fa in modo da restituire l'indirizzo della variabile x,
   che poi viene assegnato al puntatore P

```

Questo metodo è molto conveniente per evitare di dover inserire manualmente il valore dell'indirizzo di memoria di una certa variabile. Questo modo è molto comodo anche per cambiare il right-value al puntatore.

Esempio:

```

1 // Dichiaro Puntatori P, Q e inizializzo a NULL, e un TipoDato y
2 Tipopuntatore P, Q;
3 TipoDato y = 10;
4 P = NULL;
5 Q = NULL;
6
7 printf("%d ", y);
8
9 // Inizializzo il puntatore sull'indirizzo di y
10 P = &y;
11
12 // Cambio il valore della variabile puntata
13 *P = 14;
14
15 printf("%d", y); //14
16 printf("%d", *P); // 14
17
18 y++;
19
20 printf("%d", y); //15

```

```

21
22 Q = P;
23 // Sto dicendo che all'indirizzo di memoria che era indicato in P adesso
  punta anche Q. Adesso ci sono 3 modi per accedere/modificare quella
  variabile.
24
25 printf("%d", *Q); // 15

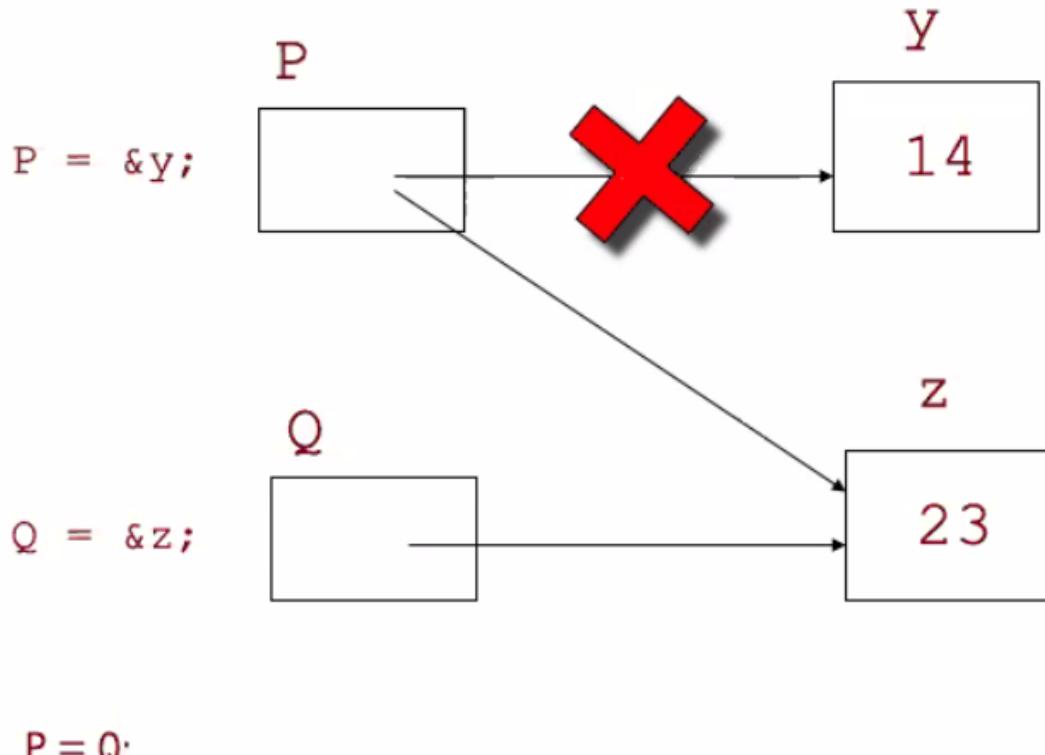
```

Entrambi gli operatori `*` e `&` sono inseriti al secondo posto nella tabella delle precedenze, superati delle parentesi e dagli operatori `.` e `->`.

Problematiche e Rischi con i puntatori:

L'utilizzo dei puntatori è facile ma anche potenzialmente pericoloso, principalmente stare attenti che:

- esiste il problema dell'accesso multiplo (più puntatori allo stesso luogo di memoria), questo problema è detto **aliasing**
- il codice perde leggibilità e immediatezza nella comprensione quasi immediatamente, quindi diventa più difficoltoso ottenere un codice pulito e ordinato.
- non vengano persi valori, eliminando un puntatore, come si vede sotto. Il valore di `y` infatti non è più recuperabile dopo questa operazione, e occuperà sempre un valore di memoria inutilizzabile dal resto del programma.



Puntatori, Dot-notation e Variabili Strutturate:

Esempio di Struct con l'uso di puntatori:

```

1 // dico uno struct
2 typedef struct {
3     int PrimoCampo;
4     char SecondoCampo;

```

```

5 }TipoDato;
6
7 // nel MAIN, dichiaro x e *P come TipoDato, e assegno al puntatore
8 // l'indirizzo di x
9 TipoDato x, *P;
10 P = &x;
11
12 // Per accedere alle classi della variabile strutturata, rispettando le
13 // precedenze degli operatori:
14
15 (*P).PrimoCampo = 12; // Anche se le parentesi tonde non sono necessarie
16
17 *P.PrimoCampo = 12; // Perchè la Dot-notation ha massima priorità
18
19 // Esiste anche una sintassi abbreviata per il puntatore a una struttura:
20 P -> PrimoCampo = 12; // questo operatore si trova in cima priorità

```

Da ricordare poi l'operatore unario **sizeof()** che restituisce il numero di byte di una variabile.

I tipi di dato con i puntatori:

il puntatore come detto è una variabile che non ha un tipo predefinito, ma gliene viene assegnato uno al momento della dichiarazione, azione necessaria affinché il compilatore capisca quello che contiene il valore del puntatore stesso.

Per questo motivo, i puntatori possono essere utilizzati anche con le altre keyword che definiscono le variabili, in particolare si fa notare **const**.

La keyword *const* può diventare necessaria nel caso in cui le variabili debbano essere accessibili in sola lettura, quindi evitare che vengano modificate.

La keyword *const* non va scambiato con *define*:

- *define* è un'operazione che avviene a livello del linker, operando semplicemente una sostituzione attraverso tabella di sinonimi.
- *const* crea una tabella dei sinonimi indicando che la data variabile non si può modificare

Esistono due modi per proteggere il puntatore, nel primo proteggiamo il puntatore, nonostante la variabile puntata possa ancora cambiare:

```
1 | int * const ptr = &x;
```

Se invece voglio un puntatore che contenga una variabile non modificabile devo usare un doppio *const*:

```
1 | const int * const ptr = &x;
```

Aritmetica dei puntatori:

L'aritmetica dei puntatori è necessaria per sommare indirizzi, o modificare il valore puntato dell'indirizzo.

Operatori di autoincremento con i puntatori o di modifica della variabile puntata:

```
1 // Dichiaro e assegno il puntatore
2 int x=1;
3 int *ip;
4 ip = &x;
5
6 // vado ad incrementare di 10 il valore che trovo dentro il banco di memoria
7 *ip = *ip+10;
8 // valore x: 11
9
10 // incrementa la variabile puntata da *ip di uno
11 *ip += 1;
12 // valore x: 12
13
14 ++*ip;
15 (*ip)++;
16 // In questo caso i due operatori hanno lo stesso risultato, ma:
17 // NB: gli operatori * e di autoincremento(++x, x++) hanno stessa
precedenza, di conseguenza il compilatore avrebbe associato da destra e il
risultato potrebbe risultare diverso.
```

Aritmetica degli indirizzi:

```
1 int y[] = {1, 2, 3, 4, 5};
2 int *p = &y; // puntatore inizializzato a y[0]
3
4 *(p+1) == 2; // Al puntatore viene aggiunto l'indirizzo dello spazio di un
intero, siccome il puntatore è inizializzato ad intero. Questo significa
praticamente che si prende il valore intero +1 rispetto al puntatore.
5 *(1+p) == *(p+1);
```

Per questo motivo si può dedurre che:

```
1 // Dichiaro un array di numeri (un vettore), e poi assegno al puntatore a
intero l'indirizzo del primo elemento del vettore.
2 int a[10];
3 int *p;
4
5 a[i] == *(a+i);
6 // *(a+i) è l'indirizzo di 'a' più 'i', quindi ne risulta l'elemento
appartenente all'insieme 'a' in posizione 'i'
7 // 'i' in realtà non incrementa propriamente di i, ma di
i*sizeof(TipoPuntatore)
8
9 // Questo vuol dire che: inizializzare p al primo elemento del vettore in
questi due modi è equivalente
10 (p = &a) == (p = &a[0]);
11 (p = &(a+1)) == (p = &a[1]);
12
```

```

13 // Al contrario NON sono ammessi: (perchè sto sbagliando l'assegnamento a
14 a = p;
15 a = p+1;
16
17 // Sottrazione con puntatori:
18 int *q;
19
20 p-q; // Restituisce il numero di elementi di differenza tra i due puntatori,
       ovvero il numero di spazi di memoria tra i due. ATTENZIONE: NON RESTITUISCE
       LA DIFFERENZA FRA I VALORI DEI DUE PUNTATORI

```

Si possono anche costruire array di puntatori:

```

1 const char *semi[4] = {
2     "Cuori",
3     "Picche",
4     "Fiori",
5 }

```

Questo è un array fatto con valori di stringhe costanti. Si può anche fare con valori di stringhe variabili, attraverso un array di puntatori non costanti.

Relazione tra Array e Puntatori in C

Se si provasse a scrivere un programma che usasse una variabile a puntatore come array di caratteri cosa succederebbe?

```

1 int main (){
2
3     char stringa[] = "Ciao Mondo";
4     char *stringaP = "Ciao Terra";
5
6     printf("Stringa = %s \n", stringa);
7     printf("StringaP = %s \n", stringaP);
8
9 }

```

Questo funziona perché l'operatore `[]` nel C viene usato come il puntatore al primo elemento dell'array, quindi se ne deduce anche che, per l'aritmetica dei puntatori:

```

1 char *stringa == char stringa[];
2
3 stringa[5] == *(stringa+5);
4
5 stringa[5] == *(5+stringa);
6
7 stringa[5] == 5[stringa];

```

Attenzione alla modifica del puntatore a stringa, perché modificandolo si può perdere l'inizio della stringa, facendo l'errore dell'aliasing (sopra).

Quindi per scorrere i valori di una stringa implementata con un puntatore abbiamo bisogno di un puntatore di servizio:

```
1 | char *stringa = "Il sale della vita";
2 | char *p;
3 | int len = 0;
4 | // variabile ptr p, con il valore di stringa, presa a variabile intermedia.
5 | p = stringa;
6 |
7 | while (*p++ != '\0')
8 |     ++len;
9 | printf("Lunghezza stringa %s: %d\n", stringa, len)
```

Le funzioni in C

Le funzioni nel C hanno diverse motivazioni:

- evitare di riscrivere intere operazioni **ripetute**, con il rischio di introdurre errori.
- scrivere blocchi **riutilizzabili**, da programmi/programmatori differenti.
- **incapsulamento** di algoritmi e possibilità di creare un **interfaccia** tra i programmi (ovvero differenza tra compito da eseguire e come questo venga eseguito).
- si possono scrivere **librerie**, ovvero insiemi di funzioni per eseguire determinate azioni definite

Le funzioni nel C:

- hanno bisogno di risorse per essere eseguiti
- devono avere tutti gli elementi di sintassi formale necessari affinché la funzione effettivamente funzioni e restituisca un valore corretto.
- si scorpora il codice in un **programma chiamante** e un **programma chiamato**.

Funzionamento della macchina astratta del C e l'asservimento delle funzioni

I **sottoprogrammi** sono asserviti a sottoprogrammi chiamanti. Affinché questo sottoprogramma funzioni, c'è bisogno che all'interno del programma chiamato siano presenti tutti i parametri necessari a funzionare, sia che questi debbano essere passati da un altro sottoprogramma, sia che debbano solo essere accessibili.

Esistono due tipi di sottoprogrammi:

- *funzioni*, che restituiscono un valore al chiamante.
- *procedure*, che svolgono un compito per il chiamante, ma non restituiscono nessun valore (definiti con *void*, che è la keyword che indica nessun valore o tipo e serve per rispettare la sintassi formale della dichiarazione delle funzioni).

Struttura di un programma C

In un programma solitamente, è presente:

- il sottoprogramma *main*, ovvero il principale, tutti quello che è preceduto da `#` è un'indicazione al linker.

```
1 #include <libreria>
2 #include "programma.h"
3
4 #define MACRO valoreMacro
5
6 int main() {
7     // codice
8 }
```

- `#include <libreria>` serve per aggiungere delle librerie standard
- `#include "programma.h"` serve per aggiungere librerie presenti nella stessa cartella del file che contiene il *main*.
- `#define MACRO valoreMacro` crea un alias che sostituisce il valore della macro al nome della macro.
- Una serie di sottoprogrammi che si possono chiamare

```
1 int funzione(int valore1, int valore2){ // questa linea è chiamata
  testata
2     // codice
3 }
```

- `int` all'inizio è il *parametro di ritorno della funzione*. Può essere un tipo di variabile predefinito o uno user-defined
- `funzione` è il *nome della funzione*
- `(int valore1, int valore2)` sono i *parametri o valori formali*. Indicano i parametri che vanno forniti alla funzione e il tipo di questo parametro. C'è la possibilità di aver tipo di parametri sia predefiniti che strutturati.
- `{ //codice }` è il luogo dove il programma si svolge quando la funzione viene chiamata, ovvero il *corpo della funzione*. Questa può contenere una *parte dichiarativa locale*, dove vengono definite le variabili necessarie all'esecuzione, e una *parte programmatica* che contiene l'algoritmo del programma.

Nel C i sottoprogrammi hanno:

- la *definizione* del programma, che comprende sia la testata che il corpo della funzione
- la *dichiarazione* della funzione, chiamata anche *prototipo*, che comprende solo la testata.

```
1 int funzione(int valore1, int valore2); // questa è una dichiarazione o
  prototipo
2
3 int funzione(int valore1, int valore2){ // questa una definizione del
  programma
4     // codice
5 }
```

Collegamento tra funzione e chiamante

Al momento della restituzione del valore al programma chiamante possono succedere due cose. O il programma ha un ritorno di tipo `void` quindi non ritorna nulla, oppure ha un tipo di variabile che deve restituire. In questo caso, allora:

- la funzione non restituisce nessun valore, quindi la funzione termina con l'ultima istruzione prima delle parentesi graffe
- la funzione termina con `return;`
- la funzione termina con `return espressione;`, allora il valore dell'*espressione* viene passato al programma chiamante. L'*espressione* può essere:
 - tipo di dato predefinito o user-defined, che coincide con il *parametro di ritorno*
 - **non** può restituire un array
 - **può** restituire puntatori, così rende possibile passare strutture molto grandi molto velocemente
 - può essere restituito solo un valore

Nel sottoprogramma si possono avere delle variabili locali, che vengono dichiarate normalmente e che servono per l'effettivo svolgimento del programma. Il loro valore può essere restituito anche con il `return valore;`.

Parametri formali e effettivi

Ci sono due tipi di parametri:

- parametri *formali* sono quelli elencati nella testata
- parametri *effettivi* sono quelli con la quale la funzione viene invocata

I parametri formali vengono inizializzati con valori dei parametri effettivi. Di fatto il programma chiamante si occupa di creare i parametri effettivi per chiamare la funzione attraverso i parametri formali della funzione. Al momento in cui questi parametri vengono passati, l'ordine conta. Questo vuol dire che i parametri effettivi devono essere i corrispettivi dei parametri formali dichiarati nel sottoprogramma. Questo vuol dire anche che prima di usare una funzione, è necessario che questa sia *definita*, ovvero che siano stati dichiarati la testata e il codice della funzione.

Modello di esecuzione e il concetto di ambiente

Prendiamo un esempio di programma che utilizza funzioni chiamanti e sottoprogrammi:

```

1 int x = 0;
2 int f1(int p);
3 void f2();
4
5 main(){
6     f2();
7 }
8 int f1(int p){
9     return p+x;
10}
11 void f2(){
12     printf("%d", f1(x));
13}

```

Ogni sottoprogramma crea in memoria uno spazio, chiamato *ambiente*, che contiene le variabili locali, i parametri passati e il risultato. Questo spazio di memoria viene rilasciato alla fine dell'esecuzione del programma.

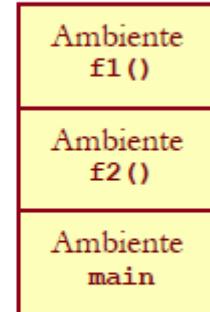
Posso pensare come se avessi una macchina dedicata che viene creata a ogni esecuzione di quella funzione. Questa macchina viene allocata e messa a disposizione quando la funzione viene istanziata, e poi viene rilasciato deallocando tutta la memoria occupata quando ho finito il suo utilizzo.

In realtà una sola macchina virtuale può eseguire questo comportamento. L'immagine sotto è un ottima rappresentazione di quello che accade.

```

int x=0;
int f1(int p) {
    return p+x;
}
void f2() {
    printf("%d", f1(x));
}
main() {
    f2();
}

```



Quando allocco memoria per un programma io di fatto sto costruisco una pila, in cui i pezzi sono in contatto tra di loro e contengono i vari ambienti delle funzioni, che a loro volta contengono le variabili necessarie al funzionamento della funzione stessa. Questa struttura può essere rappresentata dallo **stack**, chiamato anche pila, che fa riferimento alla struttura LIFO (Last In First Out), una delle strutture complesse viste più avanti.

Quando ho finito l'esecuzione di un ambiente posso rilasciarlo. Siccome gli ambienti si sono accumulati dal basso verso l'alto, logicamente rilasciando un ambiente alla volta, si smonterà lo stack dall'alto verso il basso.

Quando una funzione viene invocata si crea una copia delle variabili che vengono passate dalla funzione chiamante, mentre si deve ritornare il risultato dell'ambiente prima che venga deallocated e quindi perdere tutte le variabili non sono state passate. La copia delle variabili permette di avere tutte le informazioni necessarie al funzionamento della funzione corrente. Questo è utile e necessario, perché come abbiamo visto, al momento della creazione di una funzione, si crea l'ambiente necessario a quella funzione, che per forza di cose è diverso dall'ambiente del programma chiamante. Passando il valore per copia, quindi, ci si assicura che tutte le variabili necessarie al corretto funzionamento siano presenti nell'ambiente corrente.

Lo svantaggio di questo metodo è la lentezza nello scrivere grosse strutture di dati, un ingombro di memoria esagerato, soprattutto se la variabile da copiare è molto grande, o è ad esempio un intero database, che non si può replicare tante volte se no la memoria finirebbe.

Per ovviare a questo problema si usano i puntatori. Il puntatore infatti è un oggetto molto piccolo che include al suo interno un indirizzo di memoria, rendendolo così accessibile alla funzione. In questo modo di fatto si sta modificando la variabile presente in un altro ambiente, senza bisogno di avere una copia all'interno dell'ambiente corrente. Questo però ha anche dei problemi, infatti in questo caso i cambiamenti sulle variabili sono permanenti, e non si sta passando nessun valore alla funzione chiamante per eseguire questi cambiamenti.

Il rischio nell'utilizzo dei puntatori è che nel caso di un errore nella gestione del puntatore si possono creare gravi problemi di accesso a sezioni di memoria non normalmente accessibili e c'è il rischio di corrompere sezioni di memoria. Per cercare di risolvere in parte questo problema si deve cercare di indicare i valori costanti nei puntatori.

Un esempio:

```
typedef struct {
    int actualSize;
    array contents;
} Vettore;
Vettore vector;
void valoreFiltro(Vettore *A)
{
    A->contents[1]=10;
}

int main(int argc, char *argv[])
{
    Vettore      v;
    v.contents[1]=7;
    printf("%d PRIMA CHIAMATA FUNZIONE\n",v.contents[1]);
    valoreFiltro(&v);
    printf("%d DOPO CHIAMATA FUNZIONE\n",v.contents[1]);
}
```

In questa implementazione:

- Non ho copiato tutto il vettore V, ma solo il suo indirizzo e quindi siamo riusciti a diminuire lo spazio in memoria necessario all'esecuzione del programma.
- Con questo uso siamo riusciti a modificare un valore appartenente al *main*.

- `&v` al momento del passaggio del vettore serve perché è richiesto un puntatore nella definizione della funzione, e questo operatore restituisce l'indirizzo della variabile.

Il vantaggio di usare i puntatori in C consente nel fatto che si può passare qualsiasi tipo di dato. Infatti usando uno *struct* posso gestire tutti i parametri formali e passo un'intera struttura con un solo puntatore.

Passaggio di parametri pro e contro

Il passaggio per *valore* in una funzione:

- esegue una copia inefficiente se il parametro è ingombrante
- parametro effettivo e formale occupano zone distinte di memoria
- fornisce valori di ingresso

Il passaggio per *indirizzo* in una funzione:

- permette la copia di un indirizzo di una variabile semplice o strutturata
- la modifica avviene direttamente sul parametro effettivo
- sono possibili effetti collaterali

Ambiti di visibilità

Per comprendere il concetto di visibilità delle variabili, si deve prima comprendere quello di *blocco*. Il blocco di codice nel C si dichiara con `{ }` e può comparire in ogni sintassi che consente un'istruzione (nel corpo di una funzione, o in un qualsiasi ciclo).

Due blocchi fra loro possono essere:

- *annidati*, cioè l'uno dentro l'altro
- *paralleli*, cioè entrambi all'interno di un terzo blocco.

Per essere utilizzate le variabili devo essere visibili, ovvero il programma deve sapere a cosa corrisponde l'identificatore della variabile.

Se la dichiarazione avviene nella **parte dichiarativa locale** allora le variabili sono visibili ovunque all'interno del suo blocco e i blocchi che sono contenuti al suo interno.

Se la dichiarazione avviene nella **parte dichiarativa globale**, quindi esterno a qualsiasi blocco, la variabile è visibile da tutte le funzioni, incluso *main* e procedure e in tutti i blocchi del programma.

Può avere *side effect* o *effetti collaterali*, ovvero l'esecuzione di una funzione non è più confinato all'interno del suo ambiente e interessa variabili di altre funzioni, oppure tutto è accessibile, anche quando non lo dovrebbe essere. Per questo motivo l'uso di variabili globali è solitamente sconsigliato (da non confondere con le macro per il linker, che invece possono essere molto utili).

Da ricordare il *mascheramento*: se due variabili una globale e una locale hanno lo stesso nome, il linker assegna il valore di quella locale, di fatto mascherando eventuali omonimi.

Ciclo di vita delle variabili

Il normale ciclo di vita di una variabile va dal momento della creazione/allocamento della memoria, fino alla distruzione/deallocamento della memoria.

Esistono due classi di variabili:

- **statiche:**
 - allocate una volta e poi distrutte al termine dell'esecuzione del programma
 - lo sono le variabili *globali*
 - persistenti all'interno e all'esterno di invocazioni di funzioni
 - si può dichiarare una variabile o un blocco *static* al momento dell'allocamento, che diventano persistenti all'interno ed all'esterno di invocazioni di funzioni.
- **dinamiche/automatiche:**
 - possono essere create e distrutte *automaticamente*/implicitamente o esplicitamente dal programmatore con appositi costrutti.
 - sono dichiarate quando il flusso di esecuzione entra nel loro ambito di visibilità e distrutte all'uscita di tale ambito.
 - sono dichiarate nelle funzioni e nei blocchi, anche come parametri
 - *NB:* le variabili automatiche appartenenti allo stesso blocco che viene più volte ripetuto occupano indirizzi sempre differenti e non contengono il valore precedente

Gestione dei Files in C

Il C attraverso una libreria fornisce un interfaccia consistente per gestire i files. Questo viene eseguito ad altro livello, interfacciandosi direttamente con il sistema operativo, infatti le risorse richieste "vivono" fuori dal programma stesso.

Per scrivere e leggere files, è necessario interfacciarsi a delle periferiche. In generale si può pensare anche che esistano due tipi di periferiche:

- quelle *fisiche*: ovvero l'effettivo elemento con cui il sistema operativo si interfaccia, che può essere uno schermo o una stampa o qualsiasi altra cosa, e che crea con il sistema operativo un flusso di lettura e scrittura di dati
- quelle *logiche*: cioè l'elemento che permette di interfacciarsi nello stesso modo univoco con tutte le periferiche fisiche. Questo avviene attraverso il sistema operativo, che fornisce l'astrazione necessaria tra le periferiche e il programma in esecuzione.

Nel C le periferiche sono di tipo logico, quindi forniscono un astrazione attraverso il sistema operativo, creando dei *flussi* o *streams*, che sono i responsabili di creare queste interfacce consistenti, ovvero quelle interfacce necessarie al C per creare un livello di astrazione che prescinde dalla periferica vera e propria.

I flussi si comportano nello stesso modo e si possono eseguire le stesse funzioni su di loro. I flussi possono permettere un accesso casuale o lineare in lettura o scrittura, cioè lettura e scrittura di byte, che possono partire dall'inizio e arrivare alla fine nel caso di un accesso lineare, o invece possono partire da un determinato punto con l'accesso casuale.

Tutto questo è molto utile perché adesso il nostro modo di comunicazione con una periferica necessiterà solo di inizializzare un flusso, e poi interfacciarsi con questo. Si dice che un flusso è aperto se si è associata una certa periferica e si riesce a leggere e scrivere da e su questa.

I flussi possono essere di due tipi:

- *BINARIO*: lettura e scrittura di byte

- Le sequenze di byte possono essere pacchettizzate in vari formati
- scriviamo o leggiamo byte per byte, quindi abbiamo una corrispondenza 1:1 tra quello che è rappresentato e quello che viene letto/scritto
- nessuna traduzione dell'informazione che vuol dire nessuna perdita di informazioni dovuti all'aggiunta o la diversa codifica delle informazioni
- di tipo *TESTO*: sequenza solo di caratteri in lettura/scrittura
 - Sequenze di righe con zero o più caratteri delimitata da `\n` (carattere a capo).
 - Alcuni caratteri potrebbero non aver giusta corrispondenza nella lettura o scrittura perché provengono da macchine diverse, e che quindi usano codifica in byte diverse. (problemi nella codifica dell'informazione)
 - Bisogna quindi tenere in considerazione che la corrispondenza di caratteri scritti o letti e quelli memorizzati *non è garantita*.

Concetto di Files:

I file sono contenitore di informazioni accessibili e manipolabili con operazioni di *read & write*. La loro gestione è sottoposta al sistema operativo, che si occupa di controllare lo stato della periferica e di esporre una parte di questa periferica al linguaggio di alto livello. Di conseguenza il sistema operativo è necessario nella gestione dei files.

L'operazione di associazione di un file e uno stream avviene con una operazione di *open* che crea un flusso tra periferica e il programma e che permette lo scambio di informazioni (keyword *open* non standard, ma usata nel C). Il C permette questa associazione in modo consistente tramite la libreria `stdio.h`. Nel momento in cui si include questa libreria nel programma, allora si creeranno i flussi necessari al suo funzionamento. In questa libreria sono presenti i **tre flussi standard**. Questi vengono aperti al momento dell'esecuzione del programma e sono:

- sulla periferica video: `stdout` e `stderr`
- sulla periferica da tastiera: `stdin`

Questo è anche il motivo per cui non c'è bisogno di aprire manualmente lo stream per gestire l'operazione di `printf()`, infatti la libreria si incarica di aprire, gestire i flussi e poi chiuderli.

Quasi tutto è astratto per la gestione della periferica attraverso questa libreria come ad esempio le azioni di apertura, di lettura, di scrittura, e di chiusura. Alcune cose dell'aspetto operativo però va conosciuto, infatti mentre gli stream operano sempre nella stessa maniera, le periferiche non sono assolutamente simili: possono avere diverse caratteristiche di accesso ai files. Infatti

- se una periferica consente l'accesso sequenziale, allora bisognerà sempre leggere e scrivere in ordine.
- se una periferica consente l'accesso random, allora si potrà iniziare a leggere e scrivere in modo casuale, posizionandoci all'interno del file a piacere

Per la gestione del flusso è necessario anche saper "staccare" il file dallo *stream*, ovvero chiudere il flusso stesso. E' importante perché il sistema operativo utilizza delle risorse anche per gestire la periferica, lasciando quindi una periferica libera si possono usare per altre azioni.

La scorretta chiusura di uno stream può causare problemi. Infatti generalmente gli stream funzionano attraverso la bufferizzazione dello stream, cioè quando non gestiscono singolarmente tutti byte dello stream, ma si raggruppano vengono mandati in blocchi. Quando non viene chiuso lo stream con un buffer non vuoto le informazioni che questo contiene non vengono passate al programma causando una perdita di dati. Per risolvere questo problema si può eseguire un'operazione di *flushing*, cioè la capacità di caricare tutto quello che contiene il buffer del flusso in

modo programmatico. Generalmente è un'operazione che esegue prima della chiusura dello stream stesso perché si occupa di assicurarsi che il buffer sia vuoto, e in caso contrario passare tutti i dati al programma, in modo che tutto venga letto/scritto e poi svolto, senza rischiare la perdita di dati. Nel C il comando per eseguire questa azione è `fflush(<stream>)`.

Il C possiede la capacità di poter di accedere a delle funzioni e dei flussi di medio livello del sistema operativo che altri linguaggi non possiedono, fornendo inoltre la stessa interfaccia consistente.

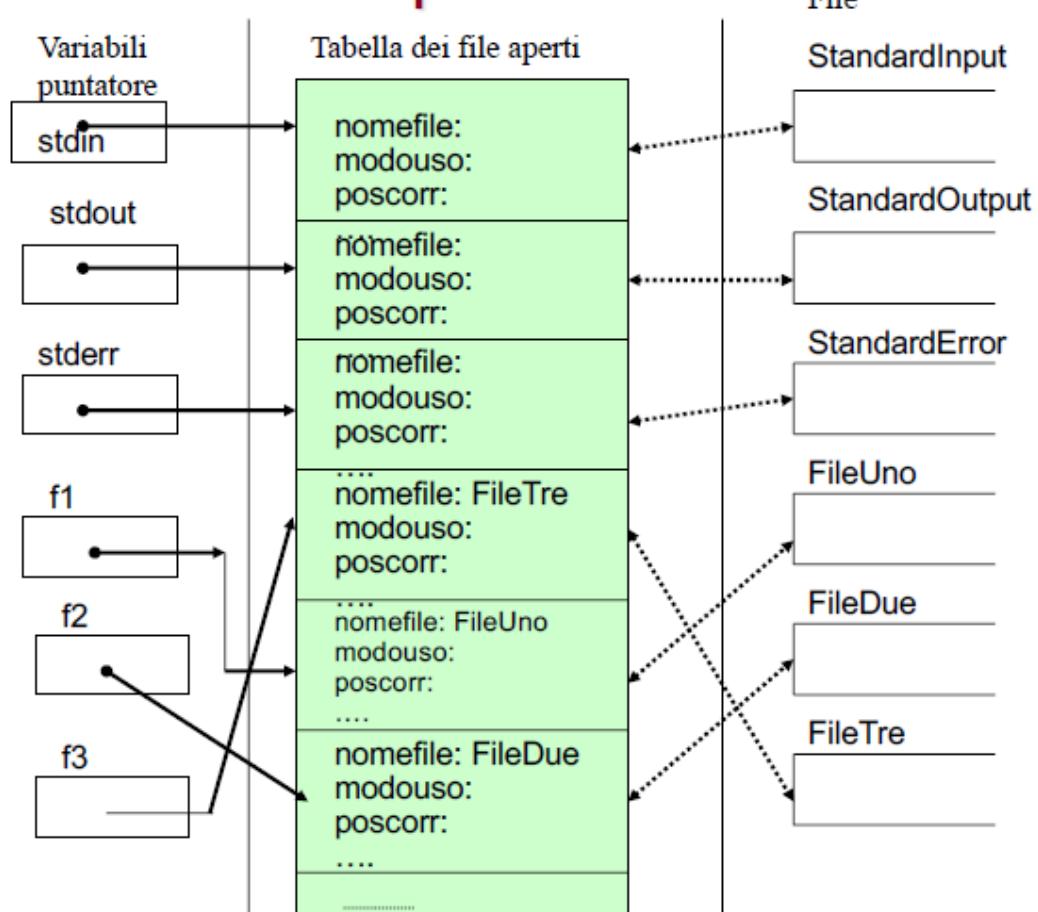
Variabile di tipo *FILE*

La variabile di tipo *FILE* può essere pensata come una qualsiasi altro tipo di variabile, a cui si assegna un puntatore. Questo puntatore serve a indicare la sezione di memoria che è incaricata di leggere e accogliere il buffer del file.

```
1 | FILE *file_da_aprire;
```

Il puntatore di tipo *FILE* conterrà un campo di byte per la lettura e scrittura e per lo stato della periferica. Questo è un oggetto profondamente legato al sistema operativo, che si occupa di istanziare tutti i campi necessari al suo funzionamento, come ad esempio lo stato della periferica. Un esemplificazione di cosa avviene in un programma quando vengono aperti più files.

Tabella file aperti



Operazione di gestione dei file:

```
1 FILE *fopen (nomefile, modalità);
2 // Header della funzione fopen apre lo stream con un file <nomefile> e con
3 // modalità (vedi dopo)
4
5 int fclose (FILE *fp);
6 // Header della funzione che termina l'associazione tra il flusso e file di
7 // una periferica
8
9 // Per usare questa funzione in un programma devo
10 FILE *fp;
11 fp = fopen("MioFile", "r");
12 //se il file è nella cartella in cui sto lavorando, mentre invece se ho
13 //bisogno di un file in un'altra posizione devo selezionare tutto il percorso
14 //del file
```

Per esempio:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[]){
5     FILE *fp;
6     char c;
7
8     fp = fopen("/tmp/MioFile.txt", "r");
9
10    if (fp == NULL){
11        printf("Il file non può essere aperto\n");
12        exit(1);
13    }
14
15    fclose(fp);
16 }
```

In questo esempio si deve notare che:

1. `if (fp == NULL)` è necessario a controllare che l'azione di apertura del file sia andata a buon fine, infatti in caso di errore il puntatore assume valore `NULL`
2. Nel caso avvenga un errore è necessario che il programma termini con `exit()`, prima di cercare di lavorare su una cosa che non esiste
3. ricordare `fclose(<puntatore_a_file>)` per chiudere la periferica. Attenzione alle operazioni di `fflush()` prima di chiudere. Questa funzione deve **SEMPRE ESSERE PRESENTE** alla fine, perché la sua mancanza può provocare perdita di dati, perdita di files e altri errori.

Modalità di accesso ai files

Come visto sopra la funzione `fopen()` accetta come secondo parametro la modalità con cui deve accedere al file. Le modalità possibili sono:

- `r` : lettura modalità testo
- `w` : scrittura modalità testo a inizio file
- `a` : scrittura modalità testo alla fine del file (*append*)
- `rb` : lettura in modo binario
- `ab` : scrittura in modalità binaria a fine file (*append*)
- `wb` : scrittura in modalità binaria a inizio file
- `r+` : apertura per modalità testo per lettura e scrittura
- `w+` : crea file in modalità testo per lettura e scrittura. Si il file già esisteva viene sovrascritto
- `a+` : append o crea un file in modalità testo per lettura e scrittura a fine file
- `r+b` : apertura in modalità binaria per lettura e scrittura
- `w+b` : crea un file in modalità binaria per scrittura e lettura
- `a+b` : append o crea file in modalità binario per lettura e scrittura.

Ricordare che se un file è aperto in una modalità non si può usarne un'altra senza chiuderlo e riaprirlo.

Altre funzioni utili alla gestione dei file

Esistono funzioni per la lettura e scrittura di caratteri: (tutti restituiscono *EOF - end of file* in caso di errore)

```
1 int getchar (void); // legge da stdin
2
3 int putchar(int c); // scrive su stdout
4
5 int fget (FILE *fp); // legge carattere da FILE
6
7 int fputc (int c, FILE *fp); // scrivere carattere su FILE
8
```

Per la lettura e scrittura di stringhe su file:

```
1 char *fgets (char *s, int length, FILE *fp);
2
3 int *fputs (char *s, FILE *fp);
4
5 int puts(char *s);
6
```

- `fgets()` e `fputs()` scrive stringhe su *stream* specificato, ad eccezione del carattere *NULL*. Restituiscono *EOF* come errore.
- `fgets()` legge i caratteri finché non trova il *EOF*, il valore nullo o al più `length-1`, il carattere `\n` viene letto e incluso nella stringa
- `puts()` scrive stringhe su *stdout* aggiungendo *NEWLINE* e restituisce *EOF* come errore.

Operazioni sulla gestione dei files

```
1 int remove (nomeFile);
2
3 int rename (vecchionome, nuovonome);
```

- le azioni `remove()` e `rename()` servono a rimuovere e rinominare i file.
- entrambe queste funzioni dipendono dal sistema operativo: se non si dispongono i permessi necessari a accedere alle risorse e a modificare lo stato dei file, o se il file risulta aperto da un altro *stream* allora l'azione non è consentita e si restituisce un errore.
- il controllo dei permessi e dello stato di utilizzo del file è sottoposto all'implementazione delle funzioni, che devono prevedere questi due casi.

Operazione di lettura e scrittura di testo da/su file:

```
1 int printf(stringa, elementi);           // Scrittura in stdout
2
3 int scanf(stringa, indirizzo_elementi);   // Lettura in stdin
4
5 int fprintf(FILE *fp, stringa, elementi); // Scrittura su stream di tipo FILE
6
7 int fscanf(FILE *fp, stringa, indirizzo_elementi); // Lettura su stream tipo FILE
```

Operazione di lettura e scrittura in binario da/su file:

```
1 int fread(void *ptr, dimElemento, numElementi, FILE *fp);
2
3 int fwrite(void *ptr, dimElemento, numElementi, FILE *fp);
```

- `fread()` legge un blocco di dimensione maggiore di un byte e li memorizza all'indirizzo di `ptr`. Restituisce il numero di elementi effettivamente letti, che potrebbe essere diverso da `dimElemento*numElementi`, quindi bisogna controllare con `ferror()` se sono ritornati errori.
- `fwrite()` scrive blocchi di byte prelevandoli dall'indirizzo di `ptr`. Restituisce il numero di elementi effettivamente scritti, quindi sorge la possibilità di controllo anche in questo caso.

Accesso Random a FILE:

```
1 int fseek(FILE *fp, long offset, int refpoint);
2
3 void rewind(FILE *fp);
```

- `fseek()` permette di accedere a posizioni nel file `fp` per operazioni di lettura e scrittura:
 - `fp` è il puntatore a tipo *FILE*
 - `refpoint` è il parametro che indica da dove bisogna calcolare l'`offset`. Questo parametro può variare tra:
 - `SEEK_SET`: l'inizio del file
 - `SEEK_CUR`: posizione corrente del cursore, cioè l'elemento incaricato di scorrere lo stream

- SEEK_END : fine del file
- offset è il numero di bytes per il calcolo della nuova posizione corrente a partire da refpoint
- da notare che la funzione `rewind()` non è altro che la funzione macro per eseguire `fseek(fp, 0, SEEK_SET)`, cioè iniziale a leggere lo stream dall'inizio con offset uguale a zero.

Pro e Contro di File Binari e di Testo

Entrambi i tipi di file hanno lati positivi e negativi:

FILE BINARI:

- PRO: occupano meno spazio, hanno la possibilità di un accesso random
- CONTRO: non c'è portabilità tra calcolatore, non si può leggere con un normale editor di testo

FILE di TESTO:

- PRO: completa portabilità fra calcolatori
- CONTRO: più complicata la modifica e l'accesso random, perché bisogna conoscere la struttura del file, dimensione maggiore a parità di contenuto, possibilità di corruzione e incompatibilità dei caratteri

Gestione degli errori

Durante l'utilizzo dei files, si possono causare degli errori. Questi devono essere correttamente gestiti per evitare un comportamento non corretto del programma.

Quando la funzione deve restituire una variabile di tipo *FILE*, c'è la possibilità di un errore generico nell'inizializzazione dello stream, ovvero quando si restituisce il puntatore con un valore *NULL*, e la possibilità di raggiungere l'*EOF*, quindi terminare la lettura del file.

Per controllare e in caso prevenire questi errori, sono presenti tre funzioni:

```

1 int ferror(FILE *fp);      // return TRUE se errore in un azione di r/w
2
3 int feof(FILE *fp);       // return TRUE se viene raggiunta la fine del file
4
5 void clearerr(FILE *fp);  // esegue la cancellazione dei segnali di errore

```

Programmazione Ricorsiva

da recuperare appunti dal tablet

Allocazione dinamica

L'allocazione normalmente avviene in maniera statica, ovvero conosco la dimensione in partenza e con questa informazione creo uno spazio di memoria che possa contenere queste informazioni. Questo limite nella quantità però può essere un limite, ad esempio in una variabile che deve gestire un input:

- se ho una variabile troppo piccola c'è il rischio di *overflow*, ovvero di riempirla troppo e in caso causare un crash del programma o un comportamento anomalo
- se ho una variabile troppo grande rischio di occupare un sacco di spazio per niente, togliendolo invece a altri processi a cui potrebbe servire.

Questo quindi riporta il problema alle capacità del programmatore, che dovevano essere molto bravi a stimare la quantità di memoria corretta. Ma in realtà per risolvere questo problema in un modo molto più elegante si può usare la programmazione con tipi di dato a memoria dinamica, ovvero creare oggetti che non possono essere descritti in modo statico dal programma. Questo può servire anche nel caso la variabile sia vuota, e quindi ci sia la necessità di liberare completamente lo spazio.

Nel C queste funzioni sono presenti nella `<stdlib.h>`, una libreria che contiene la strategia per un'allocazione e de-allocazione della memoria dinamica. Si interfaccia con le risorse del sistema operativo e si occupa della richiesta e del rilascio della memoria.

Ricordo che di solito le variabili vengono allocate all'inizio del programma e viene de-allocata alla fine dell'esecuzione del programma. Tutte queste operazioni, insieme ai record di attivazione sono presenti nello stack, ovvero quella struttura che si occupa della gestione delle funzioni.

Nel C questa libreria ha una funzione nativa che si occupa dell'allocazione dinamica:

```
1 | void * malloc(size_t size); // Header della funzione malloc
```

In cui il parametro formale è di tipo `size_t` che praticamente è un `unsigned int`. La funzione torna un puntatore alla sezione di memoria assegnata, ma questo puntatore è sprovvisto di tipo, che va quindi assegnato al momento del ritorno della funzione attraverso un casting.

Il suo funzionamento è il seguente:

```
1 | #include <stdlib.h>
2 |
3 | int main(){
4 |
5 |     int *P; // Definisco un puntatore a intero
6 |
7 |     P = malloc(sizeof(int));
8 | }
```

L'operazione che si sta eseguendo è di *portabilità*, infatti così facendo si può definire la quantità di byte per il tipo *int* su ogni macchina che il codice viene eseguito, rendendo il codice stesso molto flessibile.

Nell'esempio bisogna notare che:

1. viene lanciata la funzione `malloc` che si occupa di creare uno spazio di memoria della dimensione passata, in questo caso la dimensione è `sizeof(int)`.
2. La funzione `sizeof()` restituisce il valore intero di byte che un certo tipo di dato necessita. Può sembrare inutile nel caso di un valore *int*, di cui è conosciuta la dimensione, ma risulta molto comoda quando abbiamo un tipo di dato user-defined.

3. La funzione `malloc` torna un indirizzo alla posizione di memoria in cui è stato creato lo spazio per la variabile. È molto importante che il puntatore sia definito dello stesso tipo di dato di cui si sta creando lo spazio di memoria, in modo che il compilatore riesca a "tradurre" quello che è contenuto in quell'indirizzo.

Potrebbe succedere che questa assegnazione di memoria non vada a buon fine, e quindi devo verificare che il puntatore non torni `NULL`. Se si trova il valore nullo del puntatore però si sa che non si è assegnata la memoria, ma non se ne conosce il motivo.

Dopo queste operazioni, però bisogna ricordare di rilasciare la memoria quando si smette di usarla. Se non si rilascia una zona di memoria non avvengono errori di sintassi, però potrebbe succedere che il sistema operativo si rifiuti di fornirci altra memoria poiché ne abbiamo troppa allocata e inutilizzata. Per eseguire questa operazione si esegue:

```
1 | TipoPuntatore* P;  
2 | free(P);
```

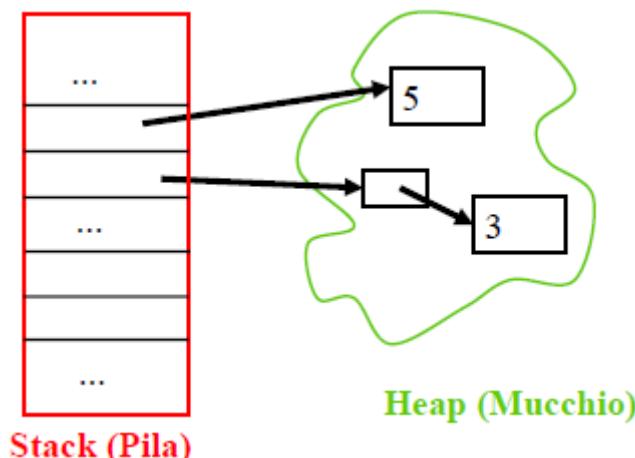
In cui il puntatore è l'indirizzo alla zona di memoria che è stata precedentemente allocata. Per questo è importante non perdere mai l'indirizzo originale della zona di memoria creata, in modo che si possa sempre de-allocare lo spazio.

Gestione della memoria con l'allocazione dinamica

Sappiamo che generalmente le funzioni lavorano sulla struttura dello **stack**, che contiene le variabili delle funzioni create staticamente all'interno delle funzioni e i record di attivazione. Questo spazio viene poi anche liberato automaticamente al momento della fine della funzione, oppure alla fine del programma.

Quando invece si dichiarano tipi di dato in modo dinamico, il compilatore ci assegna un tipo di memoria diversa chiamato **heap**, cioè mucchio. Nello **stack** quindi compariranno dei puntatori, che non saranno altro che gli indirizzi della memoria allocata dinamicamente. Quest'ultima invece è posizionata nell'**heap**. La mappa della memoria usata nel nostro programma quindi ora prende questa forma.

(L'esempio rappresenta lo **stack** con i vari indirizzi che puntano alla memoria dinamica posizionata nell'**heap**, che a sua volta può contenere un puntatore a altre zone allocate dinamicamente)



Come si vede anche dalle foto, mentre lo **stack** è molto ordinato, l'**heap** non è così uniforme perché contiene dei blocchi di memoria di grandezza diversa e generalmente i blocchi a disposizione gestiti dall'allocazione dinamica *non* sono in generale contigui.

La gestione della memoria nell'heap è molto complicata, perché bisogna tener conto in un registro che zone di memoria sono occupate e quali sono libere, oltre che segnalare il contenuto in quelle che sono piene. Questo è importante perché il compilatore ha bisogno di sapere quali zone sono libere o occupate in un certo momento per potere gestire il rilascio della memoria.

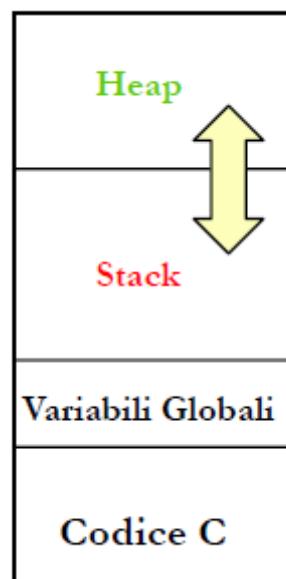
Siccome la gestione dello spazio disponibile e quello occupato nell'heap è molto complicato, si utilizza una struttura di dati chiamata *lista*. All'interno di questa lista sono presenti tutte le informazioni che sono: le dimensioni dello spazio disponibile, le informazioni necessarie ad aggiornare il registro delle zone libere e di quelle occupate e un puntatore alla prossima zona libera.

I blocchi, cioè gli elementi della lista, sono tenuti in ordine crescente di indirizzo e l'ultimo blocco punta al primo, crea quindi una *lista circolare*.

Per capire il funzionamento e il motivo di questa implementazione per la gestione della memoria dell'heap, si deve spiegare il funzionamento della `malloc`. Quando questa funzione viene lanciata, il compilatore inizia a cercare una zona libera di memoria a partire dal puntatore che rappresenta l'inizio degli indirizzi a disposizione dell'heap (elemento 1). Da questo si scandaglia la lista finché non si trova uno spazio disponibile, e quando lo si trova si aggiorna il registro. Se nel corso di questa ricerca il puntatore torna uguale a quello iniziale, cosa possibile siccome la lista è circolare, vuol dire che non sono presenti zone di memoria libere nel range, e quindi si tornerà un errore.



In generale quindi la memoria necessaria al funzionamento del programma è concettualmente semplificabile in questo modo:



Dove il *codice C* contiene le istruzioni in parte alle *variabili Globali*, che sono visibili a tutto il programma. Poi nello *stack*, si posiziona al gradino più basso sempre la funzione *main*, e seguono poi tutti le altre contenenti anche tutti i record di attivazione necessari a quella determinata funzione. Infine la zona dell'*heap*. Questo stack può aumentare l'altezza nel programma, analogamente l'*heap*, che può anche lui cambiare dimensione (questo è il motivo per cui c'è la freccia gialla).

Si ricorda poi che è possibile dichiarare la sezione di memoria dinamica allocata come variabile *static* e quindi evitare che questa determinata zona di memoria venga creata tutte le volte che si entra nella funzione che la richiede, ma al contrario, sia creata alla partenza del programma e poi utilizzata per tutto il suo svolgimento. Questo può essere utile se si vuole evitare di dichiarare molte volte zone di memoria molto grandi.

Rischi nella gestione dinamica della memoria

Tra i maggiori rischi dell'uso di programmazione con uso della memoria dinamica è la produzione di *garbage*. La parola *garbage* è esemplificativa del fatto che si produce della memoria che non viene utilizzata, ma che è rimasta allocata. Questo avviene spesso quando si perde il puntatore alla zona di memoria dinamicamente allocata. Il *garbage* quindi viene creato semplicemente per opera del programmatore, che compie degli sbagli al momento dell'assegnazione e del rilascio di memoria dinamica.

Un altro errore piuttosto comune è il *dangling reference*, cioè un *assegnazione fluttuante*. Succede ad esempio quando un luogo di memoria è indicato da due puntatori e si esegue una `free` per liberare la zona con uno dei due. Il problema che eseguendo la funzione *solo* un puntatore verrà eliminato, mentre il suo gemello si mantiene e adesso indica una zona di memoria vuota, perché è stata de-allocata.

Questo può avere lati negativi sulla correttezza del risultato aspettato, e alcuni linguaggi hanno tolto questa funzionalità.

Garbage Collection

Il problema dei *dangling reference* è facilmente risolvibile attraverso una tecnica chiamata **garbage collection**, che consiste nella raccolta di zone di memoria non referenziate. Per il C/C++ esistono librerie ad-hoc.

Altri linguaggi di alto livello come LISP o JAVA hanno un sistema di garbage collection nativamente.

Ci sono anche altre tecniche per risolvere questo problema, come l'*Automatic Reference Counting* o ARC. Questo modo di fare è utilizzato ad esempio nell'Objective-C e nel Swift (linguaggi ecosistema Apple). L'ARC funziona in modo che tiene conto degli oggetti di memoria referenziati, e quando navigando la lista ne trovano uno non referenziato cambiano l'indicazione da occupato a libero.

Allocazione dinamica nel C++

L'operatore che crea un blocco di memoria dinamica nel C++ è la funzione `new`, che generalmente una variabile di un certo tipo assegna un blocco di memoria della dimensione opportuna e restituisce il puntatore che contiene l'indirizzo al blocco appena creato.

La sintassi completa è:

```
1 tipoPuntatore *ptr = new tipoPuntatore;
2 tipoPuntatore *ptr = new tipoPuntatore[3];
3
4 // ad esempio per creare lo spazio dinamico di 100 caratteri:
5
6 char* p = new char[100];
7
8 // Per liberare la memoria nel C++ si utilizza (singolo pointer o array di
9 // elementi)
10 delete ptr;
11 delete [] ptr;
```

Tipi di Dato Astratto

Come abbiamo visto i tipi di dato sono l'insieme di valori di una variabile.

L'*Abstract Data Type* è un modello matematico che include le operazioni definite nel modello assieme al tipo di dato. Il concetto di *ADT* esula dal linguaggio di programmazione perché sono cose che possono essere implementate in ogni linguaggio. Gli *ADT* invece hanno la necessità di usare strutture proprie di un linguaggio di programmazione per essere creati, di conseguenza la loro implementazione è differente.

Liste concatenate

Se bisognava gestire una sequenza di valori in ingresso fin ad ora l'unico modo per rappresentarli era l'utilizzo di un array. Nel caso però che ci si chieda un insieme dinamico di elementi, però, il concetto di array non funziona perché non riesce a gestire un insieme dinamico di elementi.

Ma adesso possiamo anche usare le **liste concatenate** come soluzione alternativa per creare un insieme di elementi *quando l'accesso casuale non è un requisito necessario*.

Nella lista concatenata i blocchi che la compongono possono anche essere in zone di memoria distanti, perché sono allocati nell'heap. E già qui ci si distanzia dal concetto di base dell'array, che consiste in una serie di spazi tutti consecutivi.

Per implementare una coda è necessario che ogni elemento contenga almeno le informazioni per accedere all'elemento successivo, ma nel caso di una lista doppiamente concatenata, l'oggetto contiene anche il link all'elemento precedente.

Vantaggi e svantaggi rispetto all'array

VANTAGGI	SVANTAGGI
Flessibilità di modifica	Onerosità dell'accesso agli elementi, cioè ho un unico modo per accedere a un determinato elemento e questo modo è di scorrere tutta la lista
Riduzione e aumento dinamico delle dimensioni	Non ho un accesso casuale ai dati (come invece l'array)
Memorizzo solo quello che serve, non ho bisogno di sovradimensionare e non rischio di sottodimensionare	

Modello matematico dietro le *liste concatenate*

La lista è una sequenza di zero o più elementi di un determinato tipo di dato, anche strutturato se necessario.

Il numero di elementi (n) di tale lista è detta *lunghezza*. Se $n \geq 1$, a_1 è il primo elemento dell'insieme. Se $n=0$ la lista è vuota. La lista procede in ordine lineare, quindi a_1 viene prima di a_2 e così via. Inoltre l'accesso è sequenziale quindi da un elemento posso passare solo al suo successivo o precedente, non posso "saltare". Essendo un insieme dinamico posso partire con zero elementi al suo interno e poi farla crescere o decrescere.

E' utile definire l'esistenza di un blocco *FINE*. Creandolo, la *distanza* è il numero di elementi tra il primo blocco e il blocco *FINE*, e varierà nel tempo.

Implementazione

L'implementazione in C necessita dell'utilizzo di una variabile strutturata:

```

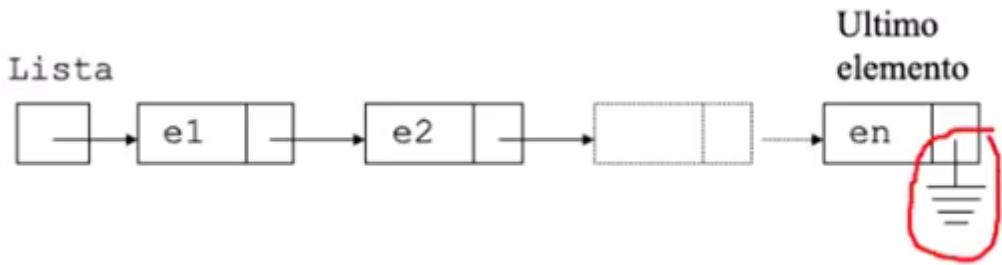
1 | typedef struct Telemento{
2 |   Tipoelemento info;
3 |   struct EL *Prossimo;
4 | }Telemento;

```

All'interno di questa struttura dobbiamo notare la struttura *autoreferenziata* `struct EL *` `Prossimo` che ci permette di raggiungere il prossimo elemento della lista. Questa definizione è una forma di ricorsione. Quando si alloca l'oggetto il calcolo dello spazio necessario però non cade in un loop infinito perché gli elementi sono concatenati attraverso un puntatore a un elemento, non l'intero elemento e quindi possiede un valore di memoria fisso.

Il campo `struct EL *Prossimo` è chiamato link e viene usato per legare una struttura di `EL` a una struttura dello stesso tipo. Inoltre questo puntatore è necessario a contenere le informazioni necessarie per passare al prossimo elemento.

La rappresentazione di una lista quindi può essere una cosa del tipo:



Nella rappresentazione il simbolo della messa a terra segnala l'ultimo elemento della lista, mentre il primo è chiamato *testa*. L'ultimo puntatore viene inizializzato a `NULL` e serve a indicare che siamo arrivati alla fine della lista. La *testa* della lista, che può essere anche `NULL` se la lista è vuota, di solito contiene il puntatore al primo elemento della lista. Per trovare l'ultimo elemento quindi bisognerà scorrere la lista finché non si raggiunge un elemento con `prox == NULL`.

Dopo aver definito il tipo di dato necessario a creare un blocco della lista, attraverso i `typedef` creiamo degli alias che potranno tornarci utili nella scrittura del programma. (Uno che indica il singolo blocco della lista e l'altro che indica l'elemento *testa* della lista)

```

1 | typedef struct EL ELEMLista;
2 | typedef ELEMLista *ListadiElem;
```

Per dichiarare la lista poi si può fare:

```

1 | ListadiElem Lista1, Lista2, Lista3; // DichiaraZione standard
2 |
3 | ELEMLista *Lista1; // DichiaraZione abbreviata
4 |
5 | struct EL *Lista1; // DichiaraZione con NomeTipo completo
```

La dichiarazione standard mette in evidenza che si sta dichiarando una lista, la dichiarazione abbreviata mette in evidenza il tipo di blocchi che conterrà la lista, mentre invece la dichiarazione con il nome del tipo di dato completo permette di indicare esplicitamente il tipo dei suoi elementi, senza ricorrere a un `typedef` (che però è molto comodo nell'uso).

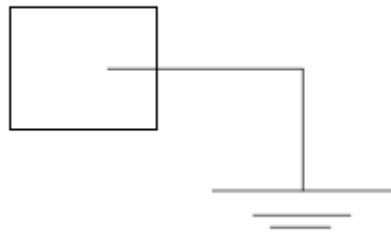
Per inizializzare una lista si deve creare il primo elemento, ovvero la *testa della lista* e assegnarci il valore `NULL`.

```

1 | ListadiElem Lista;
2 | Lista = NULL;
```

Facendo questo si viene a creare un oggetto rappresentabile in questo modo:

Lista



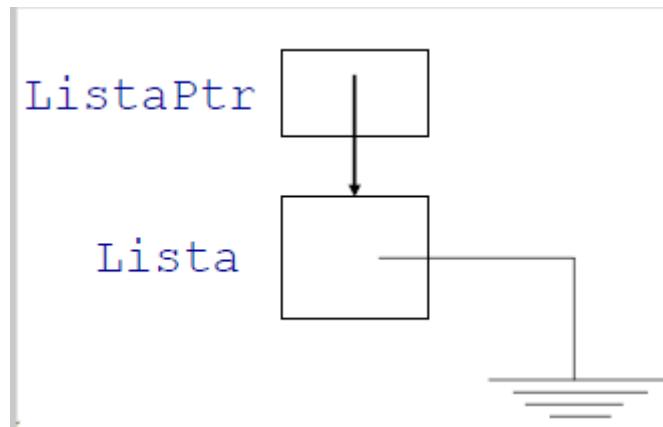
Un altro modo di inizializzare la lista è dichiarare una testa in modo globale e creare una funzione apposita per crearla. *L'OPERAZIONE DI ACCESSO ATTRAVERSO UNA VARIABILE GLOBALE è SCONSIGLIATO:*

```
1 #include <stdlib.h>
2 ListaDiElm Lista; // dichiaro globalmente
3 void Inizializza(void){
4     Lista = NULL;
5 }
```

Un metodo migliore per eseguire questa operazione, è attraverso un *doppio puntatore*, ovvero un puntatore all'indirizzo della testa della lista, che devo inizializzare a puntare a *NULL*.

```
1 void initializza(ListaDiElm *ListaPtr){
2     *ListaPtr = NULL;
3 }
```

Il concetto di come lavora questa funzione è rappresentabile con:



Dopodichè, ho bisogno di una serie di funzioni per la corretta implementazione di questo *ADT*. Queste funzioni sono:

- la funzione per il controllo se la lista è vuota

```

1 | bool IsListaVuota(ListaDiElem Lista){
2 |
3 |     if (Lista == NULL) return true;
4 |     else return false;
5 |
6 |
7 | // o anche
8 |
9 | bool IsListaVuota(ListaDiElem Lista){
10 |     return (Lista == NULL);
11 |

```

- La funzione passa l'elemento per copia, siccome è necessario controllare solo il primo elemento della lista, che è un valore finito
- Il secondo metodo implica il fatto che la funzione abbia bisogno di far tornare un valore di tipo `bool` e quindi esegue un casting dopo aver eseguito il controllo.
- Funzione di ricerca di un elemento

```

1 | bool Ricerca(ListaDiElem Lista, TipoElemento ElelCercato){
2 |
3 |     ElelLista *Cursore;
4 |     if (Lista != NULL){
5 |         cursore = Lista;
6 |
7 |         while (cursore != NULL){
8 |
9 |             if (cursore->info == ElelCercato) return True;
10 |             cursore = cursore->prox;
11 |
12 |         }
13 |     }
14 |     return false;
15 |

```

- dichiaro un elemento `cursore` che è un oggetto dello stesso tipo del blocco della lista e servirà per spazzolare tutti gli elementi di una lista
- con il controllo `Lista != NULL` controllo di non essere arrivato all'ultimo elemento
- nel ciclo `while` controllo se il mio elemento è quello cercato, se non lo è passo all'elemento successivo. Se trovo l'elemento cercato ritorno `true`. (quindi questa funzione permette di trovare solo il primo elemento, non calcola la possibilità di avere una ripetizione)
- se `return false` vuol dire che ho passato tutta la lista fino all'elemento che punta a `NULL` e non ho trovato nulla
- *Questo algoritmo è molto influenzabile dalla quantità di elementi che sono presenti nella lista*
- Funzione di ricerca attraverso la ricorsione:

```

1 | bool RicercaRicorsiva(ListaDiElem Lista, TipoElemento ElementoCercato){
2 |
3 |     if (Lista == Null) return false;
4 |     else
5 |         if (Lista->info == ELEMERCATO)
6 |             return true;
7 |         else
8 |             return RicercaRicorsiva(Lista->Prox, ELEMERCATO);
9 |

```

- `List == NULL` è per controllare di non essere arrivati all'ultimo elemento della lista, e questo passaggio è anche il *passo base* della funzione ricorsiva
- Cerco l'elemento che voglio trovare con `List->info == ELEMERCATO`, se si trova torna `true`, mentre se non si trova si rilancia la funzione in maniera ricorsiva sul prossimo elemento
- Il numero di confronti con la funzione di ricerca non ricorsiva è lo stesso, ma il record di attivazione è molto maggiore, e aumenta di molto quando l'elemento cercato è molto distante dalla testa.
- si può fare anche un'implementazione con una funzione che torni l'elemento cercato quando viene trovato
- Due funzioni minori sono quella che ritorna il valore del primo elemento, e quella che torna il valore dell'ultimo elemento:

```

1 | TipoElemento TestaLista(ListaDiElem Lista){
2 |     if IsListVuota(Lista) return TipoElemento();
3 |     return Lista->prossimo->info;
4 |
5 |
6 | ListaDiElem CodaLista(ListaDiElem Lista){
7 |     if IsListVuota(Lista) return NULL;
8 |     return Lista->prossimo->prossimo;
9 |

```

- In `TestaLista` controllo se la lista non è vuota, in caso torno un elemento vuoto (usando un costruttore), invece se non è vuota, torno il campo info del primo valore
- `CodaLista` ritorna il valore `NULL` se la lista è vuota, mentre torna il valore `prossimo` del primo elemento della lista, cioè il secondo elemento della lista.
- La funzione inserimento in testa è particolare delle liste, perché impiega sempre lo stesso tempo a essere eseguita:

```

1 | void InserimentoInTesta(ListaDiElem *Lista, TipoElemento ELEM){
2 |
3 |     ELEMLista *Punt;
4 |     Punt = malloc(sizeof(ELEMLista));
5 |     Punt->info = ELEM;
6 |
7 |     Punt->prox = *Lista;
8 |     *Lista = Punt;
9 |
10|

```

- il funzionamento di base è che si crea un elemento nuovo, si impone il campo `prossimo` come il primo elemento della lista e si impone il puntatore a testa al primo

- elemento della lista.
- o L'operazione ha un *costo unitario e costante* per cui è un azione da prediligere
- Funzione inserimento in coda:

```

1 void InserisciInCoda(ListaDiElem *Lista, TipoElemento Elel){
2
3     ElelLista *Ptr;
4
5     if (Listavuota (*Lista)){
6         Ptr = malloc(sizeof(ElelLista));
7         Ptr->Prox = NULL;
8         Ptr->Info = Elel;
9         *Lista = Ptr;
10    }
11    else InserisciInCoda(&(*Lista)->Prox), Elel);
12 }
```

- o la funzione lavora in modo ricorsivo finché non arrivo alla funzione di base, cioè finché non arrivo all'ultimo elemento
- o a questo punto recupero l'elemento precedente, e impongo il suo campo `prossimo` come indirizzo del nuovo elemento, e il campo `prossimo` del nuovo elemento come puntatore a `NULL`
- o L'inserimento in coda ha un costo di risorse e di tempo variabile in base al numero di elementi della lista, che si contrappone al costo unitario dell'inserimento in testa
- Funzione inserisci in lista ordinata (dove sto inserendo i valori in modo che rispettino una relazione d'ordine tra di loro):

```

1 void InserisciInOrdine(ListaDiElem * Lista, TipoElemento Elel){
2
3     ElelLista *Punt, *PuntCorrente, *PuntPrecedente;
4     PuntPrecedente = NULL;
5     PuntCorrente= *Lista;
6
7     while ((PuntoCorrente != NULL) && (Elel > PuntoCorrente->Info)){
8         PuntoPrecedente = PuntoCorrente;
9         PuntoCorrente = PuntoCorrente->Prox;
10    }
11
12     Punt = malloc(sizeof(ElelLista));
13     Punt->Prox = PuntCorrente;
14     Punt->Info = Elel;
15
16     if (PuntoPrecedente != NULL) PuntoPrecedente->Prox = Punt;
17     else *Lista=Punt;
18 }
```

- o attraverso due variabili di servizio tengo conto dell'elemento corrente e del successivo. Controllo che il mio valore non sia compreso tra questi due rispetto alla mia relazione d'ordine: se questo è vero allora si inserisce l'elemento, se invece risulta falso allora si scorre ancora la lista
- o la cosa più difficile di questa funzione è mantenere l'ordine, senza perdere link tra gli elementi e senza scombinare un ordine che preesisteva
- Altre funzioni per l'implementazione delle liste che possono tornare utili sono ad esempio:

- `StampaDiTuttaLaLista`, che si occupa di stampare tutti i valori `info` degli elementi che compongono la lista
- `EliminaElemLista`, che si occupa di eliminare un elemento all'interno della lista, che può essere all'inizio, in coda o in mezzo. Per implementare questa funzione:
 - si trova l'elemento da eliminare
 - si stacca il puntatore precedente e lo si ricollega al blocco successivo a quello trovato
 - si esegue un operazione di `free(Elemento)`
- funzione per il calcolo della lunghezza della lista
- funzione per rovesciare l'ordine della lista
- funzione per concatenare due liste

Estensione al concetto di lista: *Liste doppiamente concatenate*

Le *liste doppiamente concatenate* hanno due puntatori, uno che punta al `prossimo` elemento e uno che punta al `precedente`.

```

1 struct EL{
2     tipoelemento Info;
3     struct EL *Prox;
4     struct EL *Prec;
5 };
6
7 typedef struct EL Elemlista;
8 typedef Elemlista *Lista;
```

Questa implementazione ha il costo di essere più grande, perché occupa più spazio in memoria, però ha il lato positivo di facilitare l'inserimento e la cancellazione degli elementi.

Nelle liste doppiamente concatenate si possono eseguire tutte le funzioni delle liste semplicemente concatenate, però bisogna ricordare sempre dell'aggiornamento e la gestione del secondo puntatore.

Estensione al concetto di lista: *Liste circolari*

Questa struttura è molto importante, tanto che viene usata per gestire il *registro dell'heap*. Questa struttura ha come testa il primo elemento, e nessun elemento punta a `NULL`, perché l'ultimo elemento punta di nuovo alla testa.

```

1 struct EL{
2     tipoelemento Info;
3     struct EL *prox;
4 }
5
6 typedef struct EL Elemlista;
```

Sommario: LE LISTE

Le liste sono un importante esempio di struttura dinamica che nel C viene implementata attraverso puntatori (quindi un implementazione di basso livello). Possono essere singolarmente o doppiamente concatenate, ordinate o meno, circolari. Hanno una serie di aspetti positivi e negativi:

VANTAGGI	SVANTAGGI
efficienza della struttura dinamica della lista rispetto all'array: si evita lo spreco di memoria e anche l' <i>overflow</i> , ovvero l'eccessivo riempimento	Nella struttura c'è un <i>overhead</i> ovvero un ingombro maggiore dell'elemento per via dei puntatori che lo compongono
Tempo necessario all'esecuzione delle funzioni per l'inserimento in coda sempre costante	La funzione di ricerca e di inserimento in coda è molto penalizzata siccome si deve scorrere tutta la lista

Stack

L'ADT chiamata **stack** può essere anche chiamata *lista LIFO*, cioè *Last In First Out*, o *pushdown list*.

L'ADT *stack* si usa ad esempio per i record di attivazione delle funzioni.

Modello matematico dietro lo stack

Il modello matematico che permette di caratterizzare questo tipo di dato è una lista L dove l'inserimento e l'estrazione degli elementi avviene esclusivamente da un solo lato, ovvero l'alto (*top/cima*).

Le operazioni consentite sono quindi quelle di `push()` e di `pull()`, che servono rispettivamente a aggiungere in cima alla pila e a togliere l'elemento in cima alla pila.

Questo è molto importante perché consente di avere l'unicità sull'elemento da modificare al momento dell'estrazione dallo stack e anche sulla posizione di inserimento di un prossimo elemento.

Implementazione

Le operazioni standard consentite nella gestione dello stack, come predetto nel modello matematico sono:

```
1 | Push(PILA, elemento);  
2 | Pop(PILA);
```

Dove:

- `Push()` si occupa di inserire nella pila
- `Pop()` si occupa dell'operazione di *pull*, ovvero di estrarre dalla pila. Si nota che questa funzione ha bisogno solo del parametro che indica la pila, perché l'elemento da estrarre è univoco.

Per la natura con cui è creato lo stack si riempie sempre dal basso e poi lo si svuota estraendo gli elementi dall'alto.

Altre operazioni semplificano la gestione dello stack, come ad esempio:

- `top(PILA)` : funzione che restituisce la testa della pila
- `StackIsEmpty(PILA)` : funzione che controlla se lo stack è vuoto
- `StackIsFull(PILA)` : funzione che controlla se lo stack è pieno (*questa funzione è necessaria solo se si è implementato lo stack in maniera statica).

Un problema si può creare nel caso ci sia la necessità di recuperare un elemento che non è posto il cima. Si crea quindi un array o uno stack temporaneo che servirà a contenere tutti i valori estratti in ordine finché non trovo il valore cercato. Quando trovo l'elemento cercato, riporto tutti gli elementi all'interno dello stack attraverso operazioni di `push()` e posso eliminare la struttura temporanea.

Ricorda che le funzioni di `push()` e di `pop()` hanno un costo temporale e di calcolo unico, cioè costano sempre la stessa quantità di risorse e di tempo, quindi si possono usare ad oltranza senza il rischio che il tempo di esecuzione cresca troppo.

Principalmente si possono fare due tipi di implementazione: con l'array e con le liste. Entrambi hanno lati positivi e lati negativi.

Implementazione con Array

L'implementazione attraverso un array dello stack ha la principale caratteristica di aver uno spazio di memoria fissa. Questo per lo più è uno svantaggio, perché lo stack è una struttura che tende a crescere e decrescere nel corso del programma, quindi avere la dimensione fissata ha un certo peso sul programma. Oltre a questo va gestito il caso in cui lo stack sia pieno.

Le funzioni di base per un implementazione di una struttura stack con array sono:

```
1 #define StackMAX 20 // dichiarazione della massima dimensione dello stack
2
3 struct Tipostack {
4     int N;          // numero di elementi presenti nello stack al momento
5     int s[StackMAX]; // elementi presenti nello stack
6 }Tipostack;
7
8 typedef struct Tipostack STACK; // Alias per le variabili strutturate
9 typedef struct Tipostack *StackPtr; // Puntatore a tipostack
10
11 bool StackIsEmpty(StackPtr p){
12     return (p->N == 0);
13 }
14
15 bool StackIsFull(StackPtr p){
16     return (p->N == MAXP);
17 }
18
19 void Push(StackPtr, Tipoelemento ELEM){
20     if (StackIsFull(p)){
21         exit();
22     }
```

```

23     p->s[p->N] = Elemt;
24     p->N=p->N+1;
25 }
26
27 int Pop(StackPtr p){
28     if (StackIsEmpty(p)){
29         exit();
30     }
31     p->N = p->N-1;
32     return p->s[p->N];
33 }

```

All'interno di questa implementazione dobbiamo notare:

- la creazione di un `Tipostack` che serve per avere il controllo sul numero di elementi presenti all'interno dello stack e logicamente sul valore di quegli argomenti
- le funzioni `StackIsFull()` e `StackIsEmpty()` che sono necessarie al controllo sulla quantità di elementi all'interno dello stack, siccome questo è di dimensione fissata
- la funzione `Push()` che inserisce un elemento nello stack e aumenta il contatore di spazi, logicamente controllando che lo stack non sia pieno
- la funzione `Pop()` che estrae un elemento dalla cima dello stack, cioè dal valore del contatore, ne diminuisce il valore e torna il valore estratto

Implementazione con le *Liste*

Implementare uno stack con le liste permette alla struttura di crescere e decrescere in modo dinamico e indefinitamente grande (logicamente si ricordano le limitazioni fisiche). Il problema di questa implementazione è che la gestione può diventare molto complicata dal punto di vista logico.

Un implementazione tipo della struttura stack con le liste ha queste particolari funzioni:

```

1 struct stackNode{      // Dichiaro l'elemento base della lista
2     TipoElemento info;
3     struct stackNode *prox;
4 };
5
6 typedef struct stackNode StackNode; // Dichiaro gli alias
7 typedef stackNode *StackNodePtr;
8
9 bool StackIsEmpty(StackNodePtr Node){
10     if (Node == NULL) return true;
11     else return false;
12 }
13 void PrintStack(StackNodePtr CurrentPtr){
14
15     if (StackIsEmpty){
16         cout << "Stack vuoto!" << endl;
17         exit();
18     } else {
19         while (CurrentPrt != NULL){
20             cout << CurrentPrt->info;
21             CurrentPrt = CurrentPrt->prox;
22         }
23     }
24 }

```

```

23     }
24
25 }
26
27 void Push(StackNodePtr *TopPrt, TipoElemento Eelem){
28
29     StackNode *Punt;
30     Punt = new StackNode;
31
32     Punt->Info=Eelem;
33     Punt->prox=*TopPrt;
34     *TopPrt = Punt;
35
36 }
37
38 TipoElemento Pop(StackNodePtr *TopPrt){
39
40     StackNodePtr tmpPtr=*TopPrt;
41     TipoElemento topvalue = (*TopPrt)->info;
42
43     *TopPrt = (*TopPrt)->Prox;
44     free(tmpPtr);
45
46     return topvalue;
47 }
```

In questa implementazione dobbiamo notare: (L'implementazione è fatta con dei doppi puntatori perché Riccardi ha fatto così. Si può fare anche molto più semplicemente con un puntatore singolo sfruttando la struttura della lista e le sue funzioni)

- si definisce una variabile struttura che contiene la struttura di base di un blocco della lista e si dichiarano
- le funzioni `StackIsEmpty()` e `Printstack()` non sono necessarie ma facilitano l'implementazione
- la funzione `Push()` prende il nuovo elemento, crea un blocco di dimensione dinamica nello stack e gli inserisce l'elemento. Poi esegue le operazioni di inserimento in testa della lista
- la funzione `Pop()` si occupa di eliminare l'elemento di testa (esattamente le stesse operazioni che si farebbero per una lista concatenata) e tornare il valore dell'elemento eliminato

Un utilizzo particolare dello stack: *la notazione postfissa*

Nel nostro uso quotidiano usiamo la notazione *infissa*, cioè quella che usa le parentesi, ma che ha di negativo il fatto che ci può essere un ambiguità di precedenza nell'esecuzione delle operazioni che viene risolta parzialmente con le parentesi, ovvero con la creazione di una tabella di priorità.

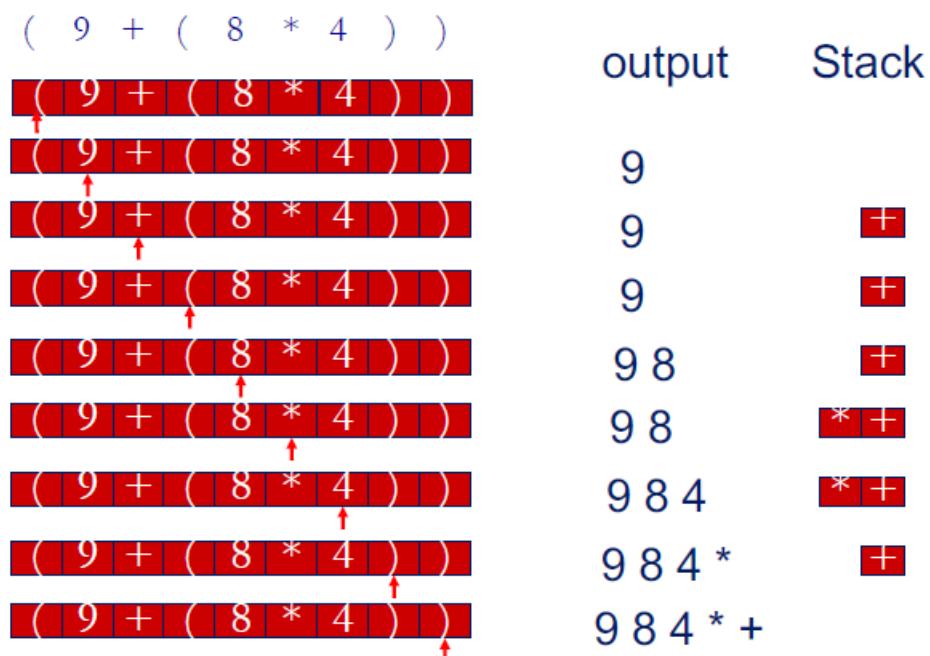
Per evitare l'utilizzo di questa tabella si può usare la *notazione postfissa*, chiamata anche *notazione polacca*. I principali lati positivi di questa notazione per la scrittura è che:

- è utile per il calcolatore infatti è un tipo di scrittura che la macchina la preferisce e con cui lavora. Usandola si evita un *layer* di traduzione eseguito dal linguaggio di programmazione.
- è apprezzata perché non ha ambiguità, quindi non ha bisogno di usare tabelle di precedenza e associatività, non lascia quindi spazio alla soggettività
- ha come lato negativo che utilizza un implementazione attraverso *stack*, che può risultare complicata

La traduzione da notazione infissa a postfissa è semplificabile in questo algoritmo:

- leggo una stringa in notazione infissa, che può essere `(9 + (8 * 4))`
- eseguo un continuo `push()` di tutti gli elementi della stringa:
 - se trovo un operatore lo inserisco nello *stack*
 - se trovo un numero lo stampo
 - se trovo una parentesi aperta o uno spazio ignoro
 - se trovo una parentesi chiusa eseguo un `pull()` dallo stack e lo stampo

Otterrò quindi un comportamento uguale a questo:



Questa funzione può essere scritta in codice:

```

1 void infissa2postfissa(const char* e){
2     // Alloca lo stack
3     for (int i=0; e[i] != '\0'; ++i){ // se la stringa non è finita continua
4         if ((e[i] == '(') || (e[i] == ' ')) continue;
5         // se la stringa contiene una parentesi, ignora e vai avanti
6         if (e[i] == ')')
7             cout << pop(s);
8         // se incontri la parentesi tonda estrai e esegui, ritorna il valore
9         finale
10        if ((e[i] == '-') || (e[i] == '*') || (e[i] == '+')
11            || (e[i] == '/'))
12            Push(s, e[i]);
13        // se contiene un operando inserisci in cima allo stack
14        if ((e[i] >= '0') && (e[i] <= '9'))
15            cout << e[i];
16    }
17    // Free s
18 }
```

Coda o Queue

L'ADT coda o queue è chiamata anche lista FIFO, cioè First In First Out. Il concetto di coda nella programmazione non è accomunabile a quello della vita reale, infatti ci sono un sacco di modi per fare delle code. Nella programmazione invece ne esiste uno solo.

La struttura della coda ha per definizione due accessi con funzioni diverse: la *coda* e la *testa*, che hanno il compito rispettivamente di uscita e di ingresso. È proprio questa caratteristica che differenzia lo stack dalla coda. Di conseguenza questa struttura funziona in modo molto lineare:

- l'inserimento avviene solamente dal lato della coda
- l'estrazione avviene solamente da lato della testa

Implementazione

La struttura della coda ha principalmente due operazioni formali che permettono l'inserimento e la rimozione.

```
1 void Enqueue(TipoCoda Coda, TipoDato Dato);
2
3 TipoDato Dequeue(TipoCoda Coda);
4
5 void CheckIfEmpty(TipoCoda Coda);
6 void PrintQueue(TipoCoda Coda);
```

- la funzione `Enqueue()` che prenda la funzione (e in alcune implementazioni è proprio chiamata) `Put()` e si occupa dell'inserimento dal lato della coda
- la funzione `Dequeue()` che prende la funzione (e può anche prendere il nome di) `Get()` e si occupa dell'estrazione dell'elemento in testa alla coda e di ritornarlo
- altre due funzioni molto meno importanti ma che semplificano l'implementazione sono `CheckIfEmpty()` e `PrintQueue()`, che con il loro nome spiegano bene la loro funzione

Come lo stack si può implementare questa struttura con due modi differenti: attraverso l'uso delle liste e con l'uso degli array.

Implementazione con LISTE

```
1 typedef struct Tnodo{
2     TipoElemento info;
3     Tnodo* next;
4 }Tnodo;
5
6 struct queue{
7     EL *Head;
8     EL *Tail;
9 }
10
11
12 typedef struct EL Elemlista;
13 typedef EL* Elemptr;
14 typedef struct queue Queue;
```

```

15 typedef Queue *QueuePtr;
16
17 void Enqueue(QueuePtr qP, TipoElemento El){
18
19     Elemptr = new Elemlista;
20     Elemptr->info = El;
21
22     qP->Tail = Elemptr;
23     Elemptr->next = NULL;
24
25 }
26
27 TipoElemento Dequeue(QueuePtr qP){
28
29     Elemlista el = qP->Head->info;
30     QueuePtr tmp = qP->Head;
31     qP->Head = qP->Head->Prox;
32     delete [] tmp;
33
34     return el;
35
36 }
37 bool checkIfEmpty(QueuePtr qP);
38
39 /* primo inserimento, devo modificarli entrambi
40 * Head-Tail -> Tnodo inserito
41 * aggiungo un altro
42 * head -> Tnodo -> Tnodo2 -> NULL, con Tail -> Tnodo2
43 *
44 */

```

In questa implementazione bisogna notare:

- la variabile strutturata rappresenta il blocco di base della lista, che contiene al suo interno il valore dell'elemento e il puntatore al prossimo blocco, come da definizione per le liste
- la struttura `queue` fornisce la testa e la coda della lista, inizializzati nel *main* o in una funzione a parte
- definisco degli alias per rendere il codice più leggibile
- la funzione `Enqueue()` è il corrispettivo per inserimento in coda della lista, con l'unica differenza che in questo caso la struttura `queue` fornisce già il puntatore alla coda, quindi evita di fare un ciclo su tutta la struttura
- la funzione `Dequeue()` si occupa di eliminare l'elemento in testa, impostare il nuovo elemento di testa nella struttura `queue` e poi liberare lo spazio del blocco. Questa funzione torna l'elemento eliminato

Implementazione con Array

L'implementazione del concetto di coda con la struttura array avviene con l'importante concetto per cui la struttura è allocata staticamente.

A lato di questo tipo di implementazione invece esiste il concetto di *buffer* o *array circolare*, che è semplicemente un array con una diversa gestione degli indici. In quest'ultima struttura si deve fare attenzione a come viene calcolato l'elemento successivo che è leggermente diverso da un normale array:

```

1 | i = (i+1);           // Incremento normale in una coda di array
2 | i = (i+1) % Codasize; // Incremento in un array circolare
3 |
4 | // i = Codasize-1 sarà l'ultimo elemento dell'array
5 | // i = Codasize +1 sarà il primo elemento dell'array

```

L'array circolare quindi impone che:

- attraverso la relazione `(i+1) % codasize` si verifica se incrementare o ritornare a zero. Infatti la relazione con l'operatore di modulo può anche essere chiamata verifica della condizione di pienezza, infatti se `i == Codasize` vuol dire che il resto è uguale a zero e che quindi il buffer torna al valore zero
- date queste condizioni si può dedurre che `i = Codasize - 1` è l'ultimo elemento dell'array e `i = Codasize + 1` è il primo elemento dell'array
- il `front` descrive un indice di valore uguale a zero
- `rear` indica il prossimo elemento.
- la coda è vuota quando `front == rear`
- la coda è piena quando `(rear+1) % Codasize == front`

Un implementazione possibile del buffer circolare è con un valore usato come *sentinella* ovvero come indice per indicare che la coda è piena. Questa implementazione ha bisogno quindi di una colonna con il ruolo di sentinella, motivo per cui serve inizializzare un array con dimensione `Codasize+1` per usare questa tecnica.

Un tipo di implementazione possibile per la coda con gli array può essere:

```

1 | TipoElemento info;
2 | struct queue{
3 |     int Codasize; // contiene al max Codasize-1
4 |     int Front;
5 |     int Rear;
6 |     TipoElemento *s;
7 | };
8 | typedef struct queue Queue;
9 | typedef Queue *QueuePtr;
10 | QueuePtr Q;
11 |
12 | // Togliere dalla coda, incrementa di uno l'indice Front.
13 |
14 | // Quando aggiungo in coda sta crescendo l'indice Rear.
15 |
16 | bool QueueIsEmpty(Q);
17 | bool QueueIsFull(Q);
18 |
19 | void Enqueue(QueuePtr pt, TipoElemento El);
20 | TipoElemento Dequeue(QueuePtr pt);
21 |
22 | void QueuePrint(QueuePtr Q); // ricordarsi che è circolare, quindi devo
23 | usare il modulo

```

- le funzioni sono praticamente uguali a quelle che faresti per gestire una lista, ma più facili dal punto di vista logico. Se si vuole usare il buffer circolare bisogna ricordare il metodo per incrementarlo correttamente e implementarlo nel modo giusto.

L'operatore XOR (OR esclusivo)

L'operatore *XOR*, rappresentabile anche con il simbolo \oplus o il simbolo \wedge , rappresenta quello che nel linguaggio naturale è "o uno o l'altro".

A	B	X
0	0	0
1	1	0
1	0	1
0	1	1

Questo simbolo ha anche proprietà associative:

- $A \wedge B \wedge C = (A \wedge B) \wedge C = A \wedge (B \wedge C)$
- $A \wedge 0 = A : A \wedge A = 0$

Questo simbolo può essere usato ad esempio per scambiare due variabili senza la variabile temporanea:

```
1 tmp = x;
2 x = y;
3 y = tmp;
4
5 // usando l'operatore XOR
6 // x == A; y == B;
7 x = x ^ y; // x == A ^ B, y == B
8 y = x ^ y; // y == (A ^ B) ^ B
9      // == A ^ (B ^ B)
10     // == A ^ 0 == A
11
12 x = x ^ y; // x == (A ^ B) ^ A
13      // == (A ^ A) ^ B
14
15 // Alla fine delle 3 assegnazioni si ottiene lo scambio di variabili senza
variabile tmp. Queste sono operazioni a livello di bit.
```

Algoritmi di Ricerca e Ordinamento

Lo scopo di un algoritmo di ricerca o di ordinamento è che esegano l'operazione nel miglior modo possibile, nel minor tempo possibile e senza compiere errori. Questi tre punti sono molto difficili da mantenere assieme, soprattutto quando si inizia a parlare di grandi moli di dati.

Per capire se un algoritmo è efficiente o meno però dobbiamo trovare un modo per valutarne l'esecuzione. Infatti se si esegue su due computer diversi, oppure se si calcolano moli di dati diverse, il funzionamento e il tempo di esecuzione possono cambiare notevolmente.

Per questo adesso facciamo un analisi statica sul solo tempo di esecuzione di un algoritmo, senza contare l'hardware su cui lavora. Possiamo rappresentare quindi due algoritmi che hanno un *numero di istruzioni* che devono eseguire per ogni dato. Rappresenteremo con N il numero dei dati. Come si vede dall'esempio sotto il calcolo avviene più velocemente su un architettura che esegue meno istruzioni al secondo perché ha ricevuto un implementazione migliore.

	A	B
Numero di Istruzioni al Secondo	10^9	10^7
Numero Istruzioni Algoritmo Richieste dal Programmatore per il Computer	$2N^2$	$50 N \log_2 N$
Tempo Impiegato per ordinare 10^6 numeri	$\frac{2 (10^6)^2}{10^9} = 2 \cdot 10^3 \text{ secondi}$	$\frac{50 \cdot 10^6 \cdot \log_2 \cdot 10^6}{10^7} \approx 100 \text{ secondi}$

Inoltre questo ci fa capire che con due algoritmi diversi, se uno è più efficiente lo è sempre tralasciando l'architettura.

Ci sono classi di algoritmi che sono studiati da anni per essere i migliori. Ormai infatti esistono algoritmi ottimizzati al massimo per qualsiasi cosa.

Algoritmi di ricerca

L'algoritmo di ricerca è un algoritmo che permette di trovare un elemento avente determinate caratteristiche tra un insieme di elementi. Gli elementi dell'insieme sono caratterizzati da una *chiave* e da un gruppo di *dati satellite*. Negli algoritmi di ricerca i dati satellite di solito si ignorano perché non sono utilizzati nella ricerca. Di conseguenza la chiave dell'elemento acquista una certa importanza, siccome è quella cosa che deve eguagliare l'elemento ricercato. Inoltre la chiave può essere *univoca* o *multipla*, cioè se esiste una sola o più volte nel gruppo di elementi. Di conseguenza nella ricerca bisogna tener conto se si sta cercando un singolo elemento, l'ultimo, uno in particolare o se invece deve restituire tutti i risultati. Per questo bisogna stare attenti ad usare l'algoritmo corretto adeguato al comportamento che ci si aspetta.

Analisi Complessità della *ricerca sequenziale*

Dato un determinato insieme di input e un algoritmo, può accadere che due machine astratte diano luogo a tempi di esecuzione diversi. Di conseguenza si parla di complessità dell'input e se ne distinguono 3 casi:

1. Caso medio
2. Caso peggiore
3. Caso migliore

	Numero cercato		Numero Passi
<u>Caso peggio</u>	18	21 3 15 6 7 11 5 18	N
<u>Caso medio</u>	6	21 3 15 6 7 11 5 18	N/2
<u>Caso migliore</u>	21	21 3 15 6 7 11 5 18	1

Nella ricerca sequenziale accade che:

1. Nel *caso peggiore* l'elemento cercato è l'ultimo rispetto all'elemento di partenza, quindi il numero di passi necessari a raggiungerlo è uguale al numero degli elementi.
2. Nel *caso migliore* l'elemento cercato è il primo rispetto all'elemento di partenza, quindi il numero di passi necessari al suo raggiungimento è 1.
3. Nel *caso medio* compaiono tutti gli altri casi, che possono essere semplificati con la metà dei valori di input.

E questa descrizione vale solo se io sto cercando la *prima occorrenza*, infatti se devo trovare tutte le volte che compare un certo valore la complessità sarà sempre uguale al numero di elementi, perché non avrò mai la garanzia che tra quelli non controllati non ci sia un'altra occorrenza. In questo caso quindi la *complessità media*, migliore o peggiore è N .

Un implementazione di questo algoritmo potrebbe essere:

```

1 int SequentialSearch(char* item, int cout, char key){
2     register int t;
3     for (t=0; t<count; ++t){
4         // questo algoritmo esce alla PRIMA occorrenza dell'elemento cercato
5         if (key == item[t]) return t;
6     }
7     return -1; // nessuna occorrenza!
8 }
9
10 /*
11 * VARIANTE CON SENTINELLA (se arrivo alla sentinella arrivo sono alla fine)
12 */
13 /* faccio un array con N elementi in cui cercare e (N+1) = elemento cercato,
14 facendo così posso fare la ricerca con un solo elemento.
15 */
16 int SequentialSearchGuard(char* item, int count, char key){
17     register int t;
18     // Aggiorno il valore sentinella
19     item[count] = key;
20     // con una sola condizione verifico se sono alla fine o se ho trovato
21     // l'elemento
22     // fa avanzare t finchè è uguale alla cella o se è vuoto

```

```

22     for (t = 0; key != item[t]; ++t){
23         if (t<count) return t; // found
24         else return -1;    // not found
25     }
26 }
```

E va notato che è indifferente in questi archivi decidere se scorrere da destra a sinistra o da sinistra a destra. La keyword `register` indica di salvare il valore nella memoria più veloce possibile.

Inoltre l'implementazione con la sentinella permette di ridurre le condizioni che vengono controllate all'interno del ciclo, ottimizzando il numero di confronti. La complessità dell'algoritmo però non cambia anche se abbiamo ridotto le condizioni da valutare.

Analisi Complessità della *ricerca sequenziale in una sequenza ordinata*

Nella ricerca sequenziale di prima non avevamo dei punti di riferimento rispetto allo stato dell'input, quindi non sapevamo caratterizzarlo e neanche in che luogo poterlo andare a prenderlo.

Se invece usassimo un archivio ordinato rispetto a un elemento ad esempio dal più grande al più piccolo, avremmo potuto sapere qualcosa dello stato dell'input e quindi avere un punto di partenza per dove iniziare a cercare.

Per questo motivo un altro algoritmo è quindi la *ricerca in una sequenza ordinata*. Gli algoritmi che ricercano in un insieme di dati ordinati sono avvantaggiati perché permettono innanzitutto di fermare la ricerca nel caso in cui il valore cercato è fuori dal range e in secondo luogo permettono di trovare subito se ci sono ricorrenze.

Ad esempio in un insieme di numeri, se dovessi trovare il valore 7 e avessi già passato il valore 8 saprei che mi posso fermare. E allo stesso modo, se sto cercando una ricorrenza del valore 6, ma dopo il singolo elemento con questo valore ne trovo uno maggiore a lui, allora so che il valore 6 compare solo una volta nell'insieme.

Un implementazione tipo di questo algoritmo può essere

```

1 int SequentialSearchSorted(char* item, int count, char key){
2
3     register int t;
4     t=0;
5     while( (t<count) && (key>= item[t])){
6         if (key == item[t]) return t;
7         t++;
8     }
9     return -1;
10}
11}
```

Questo algoritmo permette quindi di eliminare il dovere di scorrere sempre tutto l'insieme dei valori nel caso in cui voglio trovare tutte le ricorrenze, ma non permette però di diminuire la complessità. Infatti, se il cercassi un valore che è l'ultimo rispetto al mio valore di partenza io dovrei sempre scorrere tutto l'insieme (descrizione del caso peggiore).

Algoritmo di ricerca binaria:

La ricerca binaria, che si può chiamare anche dicotomica, necessita di un accesso casuale ai dati ordinati, di conseguenza non vanno bene strutture astratte come la lista. Con questi requisiti si può cercare l'elemento centrale di un insieme ordinato e in base al valore di questo elemento si cerca avanti o dietro questo elemento. Usando questa tecnica si dimezza l'insieme in cui si sta cercando ad ogni operazione.



L'implementazione tipica di questo algoritmo è:

```
1 mid = (TOP + BOTTOM)/2;
2
3 // confronto su questo elemento:
4 if (mid == ElCercato) return OK;
5 if (mid < ElCercato){
6     // si sposta la ricerca nella metà superiore dell'insieme
7     BOTTOM = (mid+1);
8     TOP = TOP;
9     mid = (TOP + BOTTOM)/2;
10    if (mid < mid){
11        ... // si continua in questo modo all'infinito (sotto scritta meglio
12        questa funzione)
13    }
14}
15 // logicamente esiste anche il caso mid>ElCercato
16 if (mid > ElCercato){
17    ...;
18    TOP = (mid-1);
19    ...; // tutto uguale il resto
20} // che esegue esattamente le stesse cose, prendendo però solo il
21 // sottoinsieme superiore.
22 /* cosa succede se non esiste l'elemento
23 * Arrivo a un caso in cui BOTTOM > TOP, in questo caso posso fermarmi
24 perché viene meno la definizione di BOTTOM e di TOP
25
26 // La chiave di questa ricerca quindi sono gli indici e l'ordinamento
27 // dell'insieme.
```

Un implementazione tipo dell'algoritmo di ricerca binaria:

```
1 int BinarySearch(char* item, int count, char key){  
2     int bottom, top, mid;  
3     bottom = 0;  
4     top = count -1;
```

```

5 // creo un ciclo finché BOTTOM e TOP non si invertono
6 while (bottom <= top){
7     mid = (bottom+top)/2; // Mid è sempre l'elemento centrale per ogni
ciclo
8     if (key<item[mid]) top = mid-1; // se la cosa cercata è minore
dell'elemento in mezzo, allora si prende il sottoinsieme sotto
9     else if(key>item[mid]) bottom = mid+1;
10    // se la cosa cercata è maggiore del MID, allora si prende il
sottoinsieme superiore
11    else return mid; // FOUND
12 }
13 return -1; // Non trovato
14 }
```

Bisogna notare che:

- le keyword `TOP`, `BOTTOM` e `mid` servono ad indicare rispettivamente l'elemento finale, iniziale e centrale dell'insieme.
- la funzione è facilmente implementabile anche in modo recursivo

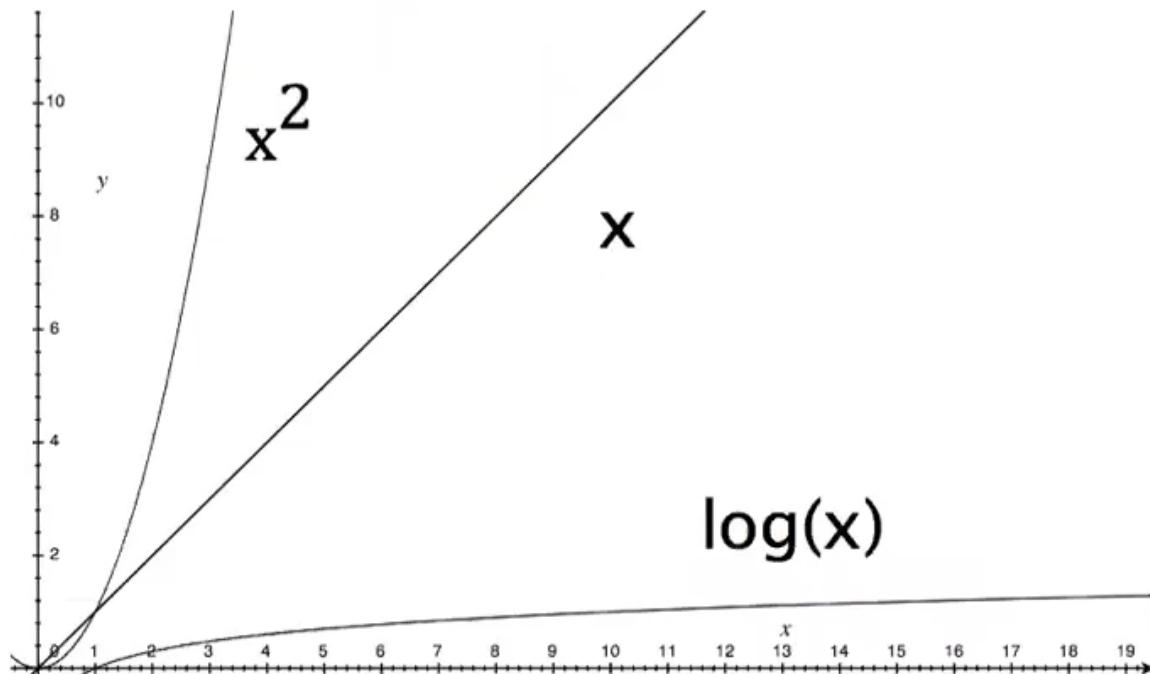
Qual è la complessità della ricerca binaria/dicotomica:

$$O(\log_2(n))$$

Nel caso peggiore dobbiamo fare questo numero di operazioni per trovare l'elemento.

Questa inoltre è la prova che il logaritmo di ricerca dicotomica va meglio della ricerca sequenziale.

Si ricorda che la crescita logaritmica permette di avere un tempo di esecuzione che cresce molto lentamente rispetto al numero di dati inseriti rispetto ad altri algoritmi.



Analisi dei Tempi di Esecuzione:

$$Tempo\ di\ Esecuzione = T(N)$$

Dove si prende N la dimensione dei dati di ingresso, e assumiamo che si prenda il numero di istruzioni come dipendenza per il tempo di esecuzione. (Ovvero non consideriamo aspetti dinamici dell'esecuzione dell'algoritmo).

Queste funzioni, che appunto indicano il tempo di esecuzione, possono avere molti elementi che le compongono, che possono contare di più o di meno.

Esempio:

$$T(N) = 1 + N + N^2 + \log(N)$$

In cui:

- 1 = Elemento costante
- N = Componente lineare
- N² = parte quadratica, nonché quella che cresce più velocemente

Spesso però non servono queste funzioni dirette, ma alcune funzioni che permettano di delimitare la crescita di una funzione. Per questo ci viene in aiuto la notazione O (anche chiamata BigO).

Questa notazione fornisce il limite superiore ad una funzione, anche delimitatamente in un intervallo.

$$f(n) = O(g(n))$$

Mentre un'altra funzione ci permette di ingabbiare questa funzione dal basso.

$$f(n) = \Omega(g(n))$$

Unendo queste due funzioni, si ottiene una terza funzione che, dato che riesce a indicare i massimi punti che toccherà con O, e i minimi punti con Ω, allora la funzione che ne risulta sarà una stima della funzione stessa (ovvero f(n) si comporterà più o meno come g(n)).

$$f(n) = \Theta(g(n))$$

Tutti questi ragionamenti, logicamente valgono in intervalli limitati.

Ricapitolando

- abbiamo il **caso peggiore/minore/medio** di T(n) a seconda dell'input.
- La notazione **O/Omega/Theta** che indica invece come la funzione si approssima.

Esempi:

- la T(n) di un'operazione *push()* su uno stack:

$$\Theta(1)$$

(Per quando male, andrà sempre uguale).

- Inserimento di un elemento in una coda FIFO in un array circolare:

$$\Theta(1)$$

- Ricerca sequenziale in un array non ordinato, nel caso peggiore

$$\Theta(N)$$

- Ricerca sequenziale in un array non ordinato, caso medio:

Algoritmi di ordinamento

L'ordinamento degli array o degli insieme è spesso richiesto affinché siano efficienti i calcoli su questi insiemi o per l'esecuzione di alcuni algoritmi che necessitano di un insieme ordinato, come ad esempio la ricerca dicotomica. Alcuni problemi invece necessitano di questi algoritmi per essere risolti, come il problema della ricerca della mediana.

In questa sezione vedremo gli esempi più importanti di algoritmi di ordinamento con rispettiva analisi della complessità. Questa analisi sarà fatta sia per il calcolo del $T(n)$ e del O .

In particolare vedremo l'**insertion-sort** e il **merge-sort**, che rappresentano anche due macroclassi di complessità per gli algoritmi.

Algoritmo *insertion-sort*:

Questo algoritmo è efficiente per array di piccole dimensioni, quindi non è sempre una via da escludere rispetto agli altri. Concettualmente è come prendere una carta alla volta e posizionarla correttamente in un altro mazzo, quindi il problema più grande è quello di non creare disordine nel mazzo, in modo che poi le carte che verranno inserite dopo siano al posto giusto.

Le operazioni che eseguirò sono:

- estraggo il primo valore (A) e lo posiziono all'inizio
- estraggo il secondo valore: se questo è maggiore lo metto dopo l'elemento precedentemente posizionato e se questo è minore lo metto prima. Questa operazione si può anche descrivere come un confronto tra l'ultimo elemento ordinato e il primo elemento non ordinato.



E la complessità asintotica dell'algoritmo è:

- nel caso *medio*:

$$\Theta(N^2)$$

- nel caso *peggiore* (ovvero quando devo completamente girare l'array):

$$\Theta(N^2)$$

Questo impone all'algoritmo di essere efficiente solo con N sufficientemente piccoli.

La complessità prende questo valore perché ho due cicli annidati, come si vede anche nel codice: uno che scorre l'array di partenza e uno che scorre l'array di arrivo.

```

1 void Insertionsort(A){
2     n = DimensioneArray(A);
3     for (int j=1; j<n; ++j){
4         // Prendo la carta da inserire
5         x = A[j];
6         // Secondo indice, quello del ciclo annidato. è anche l'indice
7         // superiore, che indica il sottoArray tra 0->(j-1).
8         i = j-1;
9         /* controllo di non essere alla fine dell'array.
10          * Quando trovo il caso in cui x < ElementoArray[i], allora
11          * assegno la carta a quel posto
12          */
13         while( (i >= 0) && (x < A[i])){
14             A[i+1] = A[i];
15             i = i-1;
16         }
17         // Prendo la prossima carta.
18         A[i+1] = x;
19     }
}

```

Analisi di complessità del *insertion-sort*:

Posso eseguire un calcolo sulla complessità dell'algoritmo in modo statico, attraverso la logica dei cicli e assegnando a ogni linea il contributo sulla complessità:

```

1 void Insertionsort(A{
2     n = DimensioneArray(A);
3     for (int j=1; j<n; ++j){      // c1*n
4         x = A[j];                // c2*(n-1)
5         i = j-1;                  // c3*(n-1)
6         while( (i >= 0) && (x < A[i])){ // c4 * (sommatoria da j=2 a n di tj)
7             A[i+1] = A[i];        // c5 * (Sommatoria da j=2 a n di (tj-1))
8             i = i-1;            // c6 * (Sommatoria da j=2 a n di (tj-1))
9         }
10         A[i+1] = x;            // c7 * (n-1)
11     }                         // -----
12 }                         // La somma di queste complessità porta alla funzione T(n),
13 // dove n è il numero di operazioni.

```

Da questa ne esce un insieme di equazioni contenente c con un valore numerico che sono costanti, e una serie con t che segna il numero delle operazioni per j volte.

Per arrivare a delle equazioni calcolabili, introduciamo, per logica la complessità nel *caso migliore* che è:

$$t_j = 1$$

perché il primo confronto è già falso, quindi non eseguo mai il corpo istruzioni del `while`. Questa situazione si può anche descrivere dicendo che in questo caso *l'array è ordinato in partenza*.

Sapendo questo dato posso tradurre le equazioni di complessità come:

```

1 void Insertionsort(A){
2     n = DimensioneArray(A);
3     for (int j=1; j<n; ++j){      // c1*n
4         x = A[j];                // c2*(n-1)
5         i = j-1;                  // c3*(n-1)
6         while( (i >= 0) && (x<A[i])){ // c4*(n-1)
7             A[i+1] = A[i];        // 0
8             i = i-1;              // 0
9         }
10        A[i+1] = x;            // c7 * (n-1)
11    }                         // -----
12 }                           // quindi posso calcolare e trovare omega.

```

Che quindi è rappresentabile: (Dove T rappresenta il numero di operazioni per N elementi)

$$T(n) = (c1 + c2 + c3 + c4 + c7) * n = bn + a = \Theta(n)$$

Da questo posso dedurre che la complessità del caso migliore è lineare.

Eseguiamo adesso lo stesso calcolo nel caso sia preso in considerazione il *caso peggiore*, cioè:

$$t_j = n$$

Questa operazione significa che io devo completamente girare il mio array di valori, e di conseguenza il numero di operazioni è uguale al numero dei valori che questo array contiene.

Da questa relazione possiamo quindi tradurre il *caso generale* e trovare la complessità del caso peggiore:

```

1 void Insertionsort(A){
2     n = DimensioneArray(A);
3     for (int j=1; j<n; ++j){      // c1*n
4         x = A[j];                // c2*(n-1)
5         i = j-1;                  // c3*(n-1)
6         while( (i >= 0) && (x<A[i])){ // c4*(sommatoria da j=2 fino a n)
7             A[i+1] = A[i];        // 0
8             i = i-1;              // 0
9         }
10        A[i+1] = x;            // c7 * (n-1)
11    }                         // -----
12 }                           // Posso calcolare
13
14

```

Che si traduce in:

$$T(n) = c_1 n + c_2 n + c_3(n-1) + c_4 \sum_{j=2}^n j + c_5 \sum_{j=2}^n (j-1) + c_6 \sum_{j=2}^n (j-1) + c_7(n-1)$$

Poi si sa che:

$$\begin{aligned}\sum_{j=2}^n j &= \frac{n(n+1)}{2} - 1 \\ \sum_{j=2}^n (j-1) &= \frac{n(n+1)}{2} - n = \frac{n(n-1)}{2}\end{aligned}$$

Possiamo quindi tornare e sostituire i contributi:

$$\begin{aligned}T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1) \\ T(n) &= \frac{1}{2}(c_4 * c_5 + c_6)n^2 + (c_1 + c_2 + c_3 + \frac{1}{2}c_4 - \frac{1}{2}c_5 - \frac{1}{2}c_6 + c_7)n - c_2 - c_3 - c_4 - c_7 \\ &= c'n^2 + b'n + a' \rightarrow \Theta(n^2)\end{aligned}$$

Con queste ultime operazioni siamo quindi riusciti a definire il comportamento dell'*insertion-sort*, trovando il suo valore asintotico.

Categorie di algoritmi che si comportano come l'*insertion-sort*:

Ricapitolando, l'*insertion-sort* è un algoritmo che:

- fa l'ordinamento *sul posto*, usa quindi un solo array senza necessità di array d'appoggio.
- confronta e scambia tra loro elementi consecutivi dell'array.

Da queste informazioni possiamo dedurre che:

Teorema: La complessità nel caso peggiore per ogni algoritmo di ordinamento che si comporta come l'**insertion-sort** è al massimo:

$$\Theta(n^2)$$

Algoritmi più efficienti richiedono scambi tra elementi distanti dall'array.

Algoritmo di *merge-sort*:

L'algoritmo di *merge-sort* si comporta nel caso peggiore:

$$\Theta(n * \log(n))$$

e ha la particolarità di unire due concetti:

1. la ricorsione
2. Divide et Impera (operazione di fusione per unire i sotto-array)

Come funziona:

- si prende l'array e si divide in due metà, si creano quindi due sotto-problemi
- continuo a dividere ancora in due sotto-array finché non arrivo alla soluzione base, ovvero un sotto-array di dimensione 1.

- Arrivato all'array di 1 elemento, allora restituisco l'array ordinato.
- Da questi due array diversi unisco (*merge*) e genero un array che è la somma nell'ordine corretto degli array in ingresso.

```

1 // Array = A, indice destro = p, indice sx = r
2 MergeSort(A, p, r){
3     if(p<r){
4         // trovo il punto centrale -> DIVIDE
5         q = Floor((p+r)/2);
6         // Sottoinsieme sx -> Chiamata ricorsiva (IMPERA)
7         MergeSort(A, p, q);
8         // Sottoinsieme dx -> Chiamata ricorsiva (IMPERA)
9         MergeSort(A, q+1, r);
10        // Unisce i due sottoinsieme -> Funzione principale
11        Merge(A, p, q, r);
12    }
13    // caso base
14 }
15
16 // Vettore A, indici p, w, r = dx, centrale, sx
17 Merge(A, p, q, r){
18     // Sottoarray sinistro
19     n1 = q-p+1;
20     // Sottoarray destro
21     n2 = r-q;
22     // tutti questi elementi tra p-q-r sono ordinati!
23     int j, i;
24     // Riempio gli array
25     for (i=0; i<n1; ++i){ L[i] = A[p+i]; }
26     for (j=0; j<n2; ++j){ R[j] = A[q+1+j]; }
27     // Valore sentinella
28     L[n1] = R[n2] = MAXINT;
29     // aggiorno intanto che faccio i confronti
30     i=j=0;
31     // r - p+1 passi
32     for(int k=p, k<=r; ++k){
33         if ( L[i] <= R[j] ){
34             A[k] = L[i];
35             i++;
36         } else{
37             A[k] = R[j];
38             j++;
39         }
40     }
41 }
```

Analisi dei tempi di esecuzione del *merge-sort*:

Principalmente si prende in considerazione il tempo della funzione `Merge()`:

$$T_{\text{merge}}(n) = \Theta(n)$$

$$\Theta(n) = \Theta(r - p + 1)$$

Quindi nel caso peggiore, l'algoritmo di *merge-sort* ha complessità

$$\Theta(n \log(n))$$

Questo valore arriva per il calcolo delle complessità in funzioni ricorrenti, dove si mette a sistema il caso base e il caso ricorsivo:

$$T(n) = \begin{cases} \Theta(1) & \text{Se } n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

Dove:

- c : costante piccola
- a : numero sotto-problemi generato dal passo *divide*
- $1/b$: dimensione dei sotto-problemi rispetto a quello originale
- $D(n)$: Tempo/Risorse impiegato per dividere i problemi in sotto-problemi (operazione di `Floor`)
- $C(n)$: Tempo impiegato per ricombinare le sotto-soluzioni

Nel caso di *merge-sort* l'equazione di ricorrenza ha i seguenti parametri:

$$a = 2$$

$$b = 2$$

$$T_{MergeSort}(n) = \begin{cases} \Theta(1) & \text{Se } n \leq c \\ 2T\left(\frac{n}{2}\right) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

Il termine *divide* ovvero $D(n)$, corrisponde al calcolo della metà (cioè quello che restituisce `Floor`)

$$D(n) = \Theta(1) \text{ (è un valore costante)}$$

$$\text{Impera: } = 2T\left(\frac{n}{2}\right)$$

$$C(n) : \text{Operazione combinazione/fusione}$$

$$C(n) = \Theta(n)$$

E a questo punto siccome $a=b=2$ e ricordando che dall'operazione sopra ci risulta questo:

$$\Theta(n) \leftarrow C(n) = \Theta(n)$$

$$\Theta(n) \leftarrow D(n) = \Theta(1)$$

possiamo scrivere che:

$$T_{MergeSort}(n) = \begin{cases} \Theta(1) & \text{Se } n \leq c \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

Ma nel caso dell'*insertion-sort* ottengo un polinomio, quindi non riesco a trovare la complessità statica totale.

Le equazioni di ricorrenza si possono essere risolte in vari modi ad esempio la sostituzione o l'albero di ricorsione.

Risoluzione della complessità della ricorsione con il *metodo dell'albero di ricorsione*

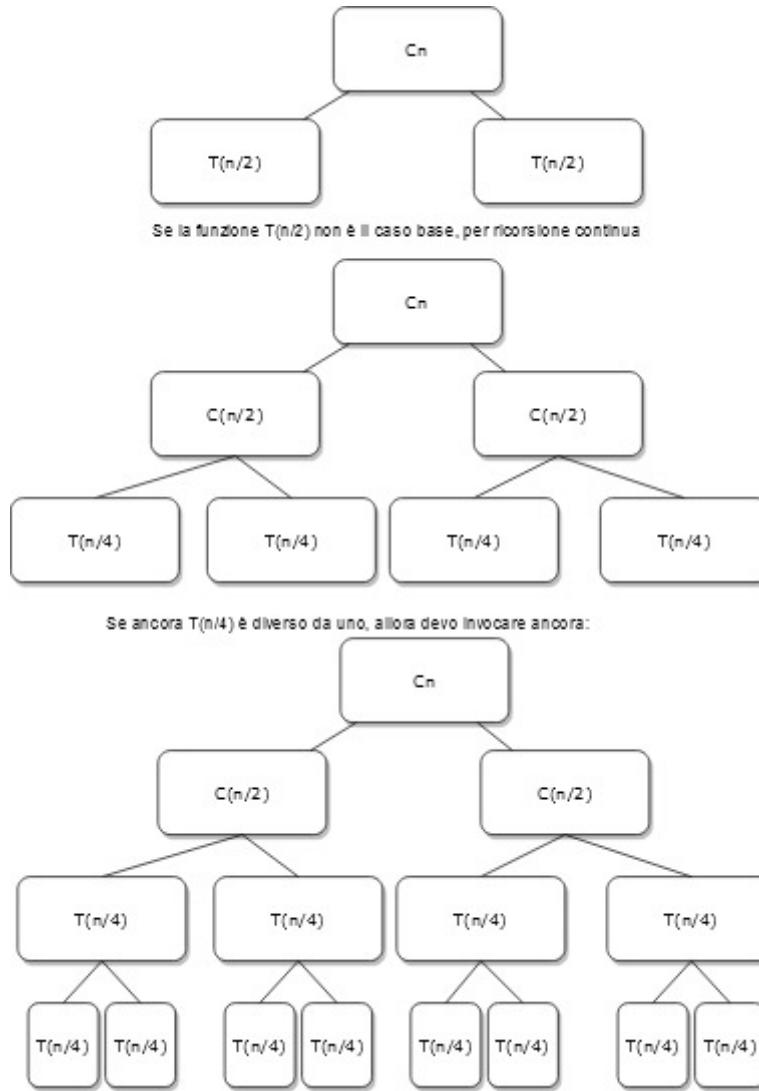
Dal sistema di prima possiamo fare alcune deduzioni:

- il significato di $\Theta(1)$ è che si approssima a una funzione lineare, che vuol dire una funzione costante
- quindi si può sostituire $\Theta(1)$ con C e $\Theta(n)$ con Cn

Da queste deduzioni otteniamo dei termini espliciti:

$$T_{MergeSort}(n) = \begin{cases} C \text{ Se } n = 1 \\ 2T\left(\frac{n}{2}\right) + Cn \text{ Se } n > 1 \end{cases}$$

Che si può risolvere attraverso la creazione di un *albero di ricorsione*:



E si continua in questo modo finché non si è trovato il caso di base, che costa c , di conseguenza l'ultimo "piano" di rami dell'albero sono n foglie che costano ognuna C . (quindi $n*C$)

Poi dovremmo calcolare il numero di livelli che impiego ad arrivare al caso base e il rispettivo costo.

Dovremmo risolvere 3 problemi quindi:

1. Calcolare il numero di livelli
2. Calcolare il numero delle foglie
3. calcolare il costo di C_i (costi di ciascun livello)

Se risolviamo 1, 2, 3 troviamo $T(n)$ sommando i costi di tutti i livelli, inclusi i costi delle foglie.

Il costo di ciascun livello si può scrivere anche in questo modo:

$$\begin{aligned} C_1 &\rightarrow Cn \\ C_2 &\rightarrow C(n/2) + C(n/2) = Cn \\ C_3 &\rightarrow C(n/4) + C(n/4) + C(n/4) + C(n/4) = C(n) \end{aligned}$$

Che in generale:

$$C_i = 2^i * c \left(\frac{n}{2^i} \right) = cn$$

$$C_{NumLevel.} := cn$$

Quindi il costo di ciascun livello del *merge-sort* è Cn . Di conseguenza:

$$T(n) = \sum_i^{NumLevel} C_i$$

Il calcolo dei numero dei livelli viene fatto per *induzione*: (si conosce la soluzione per un caso base e si risolve per un caso generico). Cioè:

IPOTESI:

$$NumLivelli = \log_2(n + 1)$$

da questo, il CASO BASE:

$$\begin{aligned} n &= 1 \\ NumLivelli &= 1 = 0 + 1 \rightarrow OK! \end{aligned}$$

CASO INDUTTIVO:

$$\begin{aligned} n &= 2^i \\ NumLivelli &= \log(2^i + 1) = i + 1 \end{aligned}$$

Noi sappiamo che ogni volta che passiamo di livello, moltiplichiamo per due nel livello successivo, quindi:

$$\begin{aligned} ProssimoN : n &= 2^{i+2} \\ &n = 2^{i+1} \\ &(i + 1) + 1 = \log_2(2^{i+1}) + 1 \end{aligned}$$

Che quindi dimostra la formula per il caso bersaglio che stavamo cercando.

Con questo possiamo descrivere il caso induttivo. Da questo ragionamento arriviamo nell'ultimo caso, nel quale dobbiamo sommare tutti le somme parziali dei livelli:

$$\begin{aligned} \sum_i^{NulLivelli} C_i \\ T(n) &= cn * NumLivelli = cn * (\log n + 1) \\ &= cn\log(n) + cn = -(n\log(n)) \end{aligned}$$

E con questa si dimostra come il *merge-sort* ha complessità costante, sia nel caso peggiore che nel caso migliore e vale:

$$\Theta(n\log(n))$$

Risoluzione della complessità della ricorsione con il *metodo della sostituzione*

Come prima partiamo dalla complessità della funzione che è:

$$T_{MergeSort}(n) = \begin{cases} \Theta(1) & \text{Se } n \leq c \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

Voglio dimostrare che $T(n) = \Theta(n\log(n))$.

Dimostrare per induzione che:

$$T(n) = cn * \log(n) \text{ per } c < 0$$

Assumo che il CASO BASE:

$$T(1) = \Theta(1)$$

e il CASO INDUTTIVO:

Assumo che la diseguaglianza valga per $n < k$. Che quindi

$$T(n) \leq cn * \log(n)$$

Dimostrare che questa sopra vale per $n = k$, in particolare $n = K/2$.

$$T\left(\frac{k}{2}\right) \leq c \frac{k}{2} \log\left(\frac{k}{2}\right)$$

La ricorsione ci dice che:

$$\begin{aligned} T(k) &= 2T\left(\frac{k}{2}\right) + k \\ T(k) &\leq 2c\left(\frac{k}{2}\right)\log\left(\frac{k}{2}\right) + k \\ T(k) &\leq ck(\log(k) - \log(2)) + k \\ T(k) &\leq ck\log(k) - ck + k \\ T(k) &\leq ck\log(k) + k(1 - c) \\ T(k) &\leq ck\log(k) + (1 - c)k \\ T(k) &\leq ck\log(k) \\ \text{per } c &\geq 1 \end{aligned}$$

Questo vuol dire che l'assunto vale per $k > n$, rispetto all'ipotesi che era più ristretta.

Ricapitolando algoritmi di ordinamento

Gli algoritmi di ordinamento sono dipendenti:

1. dal tipo di elementi da ordinare
2. dal verso di ordinamento: crescente o decrescente

Inoltre il *teorema dell'insertion-sort* afferma che i metodi basati su confronti e scambi sul posto che ordinano una sequenza di N numeri non può fare meglio che $\Theta(N^2)$ [Esempio: InsertionSort, BubbleSort].

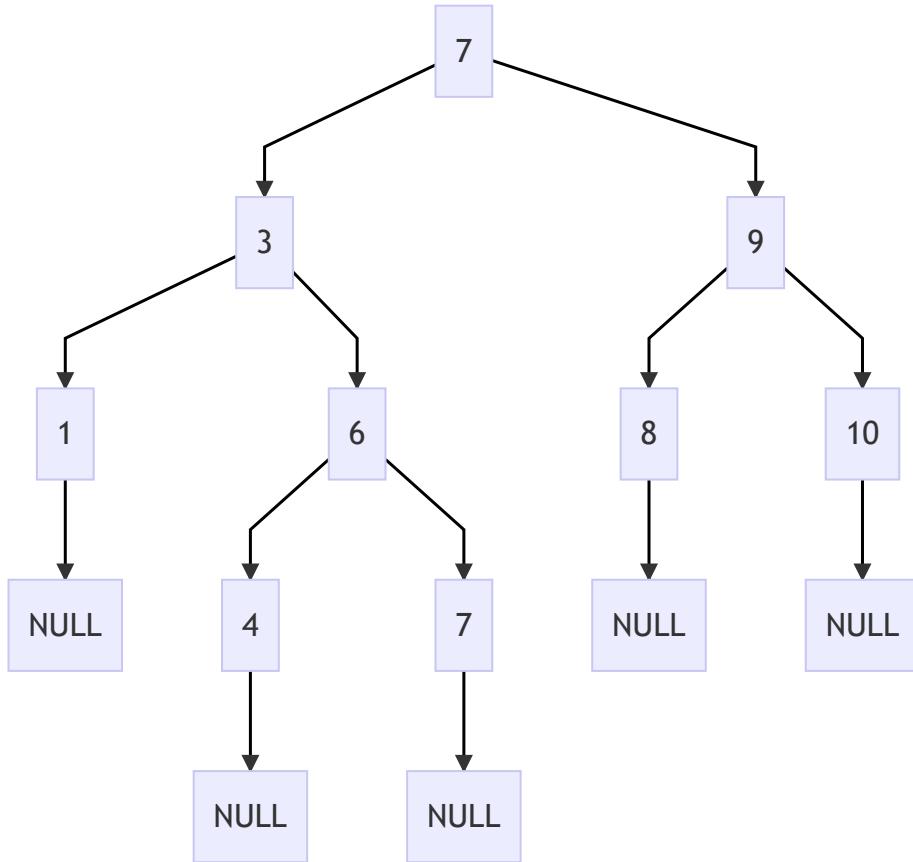
Mentre con gli algoritmi basati sui confronti si può arrivare nel caso *peggiore* a complessità $\Theta(n\log(n))$. [Esempio MergeSort, QuickSort].

Inoltre si può anche dimostrare che **tutti i metodi per confronto hanno metodo ottimale al massimo come $n\log(n)$** .

Per migliorare ancora la complessità di algoritmi di ordinamento quindi bisogna abbandonare il metodo dei confronti e si può arrivare a complessità inferiore a $n\log(n)$. [Esempio CountingSort, RadixSort, BucketSort].

Alberi Binari di Ricerca

La struttura chiamata *albero binario di ricerca* è stata creata per semplificare la ricerca in un insieme di moltissimi elementi, attraverso un particolare tipo di indicizzazione. Queste tecniche usate assieme permettono di avere una struttura che ha un tempo medio di ricerca di un elemento di $\Theta(\log n)$. Allo stesso modo, sono possibili anche le operazioni di inserimento e cancellazione, anche se più complicate.



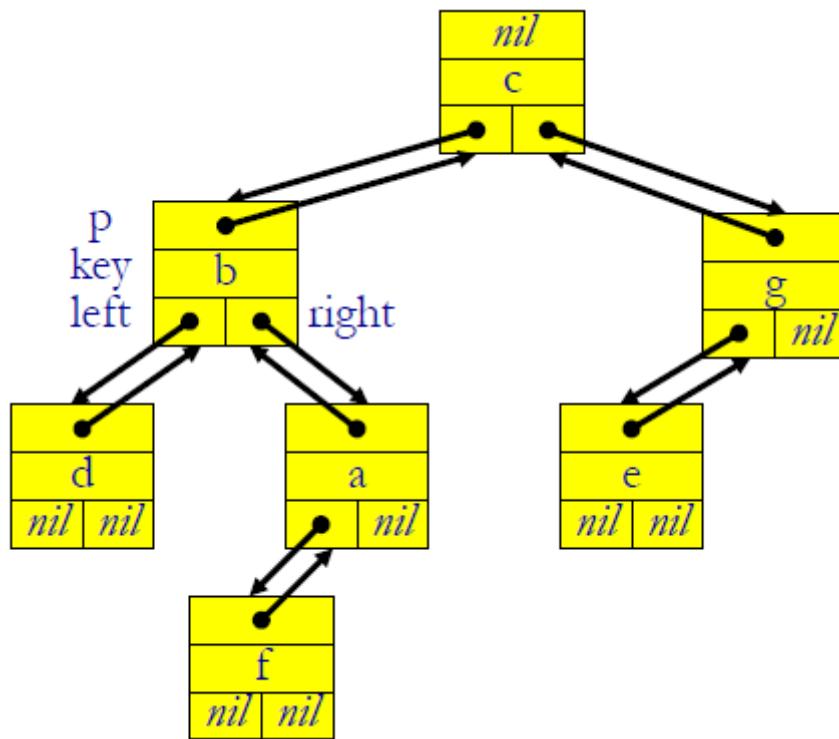
Questo in alto è la rappresentazione di un albero binario di ricerca, in cui si può notare che ogni valore che è maggiore di un determinato nodo finisce nel suo sotto-albero sinistro, mentre chi è minore del valore del nodo finisce nel sottoalbero destro (anche se questa è una convenzione, perché può benissimo venir implementato all'inverso). L'aggettivo binario invece sta a indicare come ogni nodo abbia due sotto-alberi.

I rapporti di minore, maggiore e uguale vanno mantenuti sempre, dal momento dell'inserimento a quello della cancellazione, infatti solo questo modo di agire permette all'albero di continuare a funzionare correttamente.

Tutti i rami di questo albero finiscono poi con un valore nullo, una casella vuota, che sta ad indicare che è terminato.

La rappresentazione su un albero binario non è univoca, cioè ho più modi di rappresentare lo stesso insieme di elementi.

Questa struttura al momento dell'implementazione prende questa forma in memoria:



Logicamente la forza dell'albero binario è che non è necessario passare attraverso tutti i valori, quindi nel caso in cui questo albero viene bilanciato male si dice che *l'albero degenera*, ovvero prende più la forma di una lista che di un albero.

Visita o stampa delle chiavi

Presi come implementazione tipo quella di una lista doppiamente concatenata, possiamo definire ogni blocco come:

```

1  typedef struct Blocco{
2
3      TipoDato dato;
4      Blocco *right;
5      Blocco *left;
6
7 }Blocco;
```

Altre implementazioni includono più puntatori, ad esempio quelli al padre, al nodo o ad altri elementi. Con questo tipo di codice invece abbiamo tutto il necessario per un corretto funzionamento senza eccedere nell'utilizzo della memoria.

Con questo tipo di organizzazione ci sono vari metodi per stampare gli elementi dell'albero binario:

Stampa **in ordine**, anche definito come attraversamento simmetrico.

- Consiste nel nodo di destra, nel nodo al centro e poi nel nodo di sinistra.
- L'implementazione tipo, usando la ricorrenza è:

```

1 void Inorder(Blocco* x){
2     if (x != NULL){
3         Inorder(x->left);
4         cout >> x->dato;
5         Inorder(x->right);
6     }
7 }
```

Dove il caso di base è `if (x == NULL)`

- La complessità di questo algoritmo è descritta da:

$$\text{Caso Base: } T(0) = c$$

$$\text{Caso Induttivo: } T(n) = (c + d)n + c$$

Dove `c` e `d` sono costanti e `n` è il numero di nodi. Infatti questo si dimostra con il metodo della sostituzione dicendo che il caso base:

$$T(0) = c = (c + d) * 0 + c$$

mentre il caso induttivo:

$$\begin{aligned} T(n) &= T(k) + d + T(n - k - 1) = \\ &= (c + d)k + c + d + (c + d)(n - k - 1) + c \\ &= (c + d)n + c \end{aligned}$$

Questo significa che il caso induttivo è la somma del tempo di esecuzione della costante ??

Stampa in post-ordine.

- Si stampa prima il sotto-albero sinistro, quello destro e infine la chiave
- L'implementazione tipo, usando la ricorsione:

```

1 void PostOrder(Blocco* x){
2     if (x != NULL){
3         PostOrder(x->left);
4         PostOrder(x->right);
5         cout >> x->dato;
6     }
7 }
```

Infine la stampa in pre-ordine.

- si stampa la prima la chiave, poi il sotto-albero sinistro, infine quello destro
- L'implementazione tipo, usando la ricorsione è:

```

1 void PostOrder(Blocco* x){
2     if (x != NULL){
3         cout >> x->dato;
4         PostOrder(x->left);
5         PostOrder(x->right);
6     }
7 }
```

La ricerca di una chiave

Nell'algoritmo di ricerca si deve trovare un elemento, o più di uno, che corrisponda alle caratteristiche cercate. Questa idea si deve sviluppare anche tenendo conto delle proprietà dell'albero binario.

Un implementazione tipo di questo algoritmo potrebbe essere:

```
1 Blocco Search(Blocco* x, TipoDato k){  
2     if (x == NULL || k == x->dato){ return x; }  
3     if (k < x){ return Search(x->left, k); }  
4     if (k > x){ return Search(x->right, k); }  
5 }
```

Questa implementazione restituisce `NULL` se non trova nessun elemento, mentre invece stampa la prima ricorrenza trovata.

La complessità di questo algoritmo è, *nel caso peggiore*, l'altezza dell'albero, cioè $\Theta(h)$. Questo significa che nel caso peggiore l'elemento cercato è o l'ultimo elemento a destra o l'ultimo elemento a sinistra. Di conseguenza la complessità aumenta linearmente con l'aumento dell'altezza dell'albero, da questo ne risulta che il buon bilanciamento dell'albero è molto importante.

Ricerca del Massimo e del Minimo

Questi algoritmi per la ricerca del massimo o del minimo consentono di trovare l'elemento con il valore più piccolo o più grande possibile all'interno dell'albero. Questi algoritmi poi tornano utili anche per altre funzioni più complesse.

Anche in questi casi la complessità del *caso peggiore* è l'altezza dell'albero, cioè anche in questi casi $\Theta(h)$.

Un implementazione tipo può essere:

```
1 TipoDato Minimo(Blocco* x){  
2     while(x->left != NULL){  
3         Minimo(x->left);  
4     }  
5     return x;  
6 }  
7  
8 TipoDato Massimo(Blocco* x){  
9     while(x->right != NULL){  
10        Minimo(x->right);  
11    }  
12    return x;  
13 }
```

Ricerca del Successore e del Predecessore

Questo algoritmo deve tornare l'elemento il cui valore è subito successivo o subito precedente al valore. Questo significa che si deve trovare:

- se si sta cercando il *successore*: il valore più piccolo nel sotto-albero dei maggioranti
- se si sta cercando il *predecessore*: il valore più grande nel sotto-albero dei minoranti

Da queste definizioni si deduce anche che per questo algoritmo saranno utili quelli di ricerca del minimo e massimo.

Un implementazione tipo può essere:

```
1 Blocco Successore(Blocco* x){  
2     if (x->right != NULL){  
3         return Minimo(x->right);  
4     }  
5  
6     Blocco* y = Padre(x);  
7     while( y != NULL && x == y->right){  
8         x = y;  
9         y = Padre(x);  
10    }  
11  
12    return y;  
13}  
14  
15 Blocco* Padre(Blocco* Cursore, Blocco* x, Blocco* precedente){  
16     if (Cursore == x){  
17         return precedente;  
18     }  
19     else{  
20         if (x<Cursore)  
21             Padre(Radice->right, x, Cursore);  
22         if (x>Cursore)  
23             Padre(Cursore->left, x, Cursore);  
24     }  
25 }
```

- il primo blocco di codice con `if (x->right != NULL)` indica che se ci sono elementi sulla destra, cioè se esiste il blocco dei maggioranti allora si cerca il suo minimo.
- Se non esiste il blocco dei maggioranti, allora si deve cercare un nodo più vicino alla radice che sia il più grande possibile. Da qui la necessità di inserire la funzione `Padre`, che in base alla struttura che si è decisa per l'albero può avere implementazioni diverse (ad esempio se ogni elemento dell'albero contiene anche un puntatore al suo padre, l'implementazione è molto basilare). In questo caso invece l'implementazione non è né semplice, né senza spreco di risorse.
 - La funzione continua ad avanzare di nodi, seguendo la regola dell'albero, finché non trova l'elemento cercato. A quel punto restituisce l'elemento precedente, che viene passato alle varie funzioni attraverso l'intestazione della funzione.
 - Ci sono altri modi per implementare questa cosa, più efficienti o meno.
- Non esiste la possibilità che questa funzione non trovi nulla, eccetto che l'albero sia completamente sbilanciato verso i minoranti e manchi completamente il sotto-albero di sinistra.

E l'algoritmo per trovare il predecessore:

```
1 Blocco Successore(Blocco* x){  
2     if (x->left != NULL){  
3         return Minimo(x->left);  
4     }  
5  
6     Blocco* y = Padre(x);  
7     while( y != NULL && x == y->left){  
8         x = y;  
9         y = Padre(x);  
10    }  
11  
12    return y;  
13 }
```

che funziona allo stesso modo, ma nel sotto-albero opposto.

Entrambi questi algoritmi hanno una complessità di $\Theta(h)$, dove h è l'altezza dell'albero.

Ricerca dell'altezza dell'albero binario

Si definisce l'altezza dell'albero come la distanza massima tra la radice e ciascuna foglia, di conseguenza si devono calcolare tutte le distanze e prendere quella massima.

L'idea generale che sta alla base di questo algoritmo è che si deve attraversare a sinistra e a destra, calcolare le due altezze e poi prendere il massimo tra i due.

```
1 int AltezzaAlbero(Blocco* x){  
2     if(x == NULL) {  
3         return -1;  
4     } else {  
5         AltezzaSx = AltezzaAlbero(x->left);  
6         AltezzaDx = AltezzaAlbero(x->right);  
7         return max(AltezzaSx, AltezzaDx);  
8     }  
9 }  
10  
11 int max(int val1, int val2){  
12     if(val1 > val2)  
13         return val1;  
14     if (val2 > val1)  
15         return val2;  
16 }  
17 }
```

Della funzione per calcolare l'altezza dell'albero si può notare una somiglianza alla funzione di post-ordine.

Inserimento di un nuovo elemento

La funzione per inserire il nuovo elemento è importante per lasciare intatta la struttura dell'albero e quindi continuare ad avere una struttura con una complessità di $\Theta(\log n)$ per la ricerca.

L'implementazione è:

```
1 void InserisciElemento(Blocco* T, Blocco* z){
2     // se l'albero è vuoto inizializzo
3     if(root(T) == NULL){
4         root(T) = z;
5     }else{
6         InserisciElementoRic(root(T), z);
7     }
8 }
9
10 void InserisciElementoRic(Blocco* x, Blocco* z){
11     if (z->dato < x->dato){
12         if (x->left == NULL){
13             z = x->left;
14         } else {
15             InserisciElementoRic(x->left, z);
16         }
17     }else{
18         if (x->right == NULL){
19             z = x->right;
20         } else {
21             InserisciElementoRic(x->right, z);
22         }
23     }
24 }
```

Nella funzione abbiamo nell'intestazione l'albero `T` e il nuovo blocco `z`, che verrà inizializzato come un nuovo elemento, con entrambi i puntatori a `NULL`. E' importante ricordare che in questo caso, se l'elemento dell'albero richiede anche il padre, va aggiunto al momento.

Anche la complessità di questa funzione è legata all'altezza infatti è definibile come $\Theta(h)$.

Cancellazione di un elemento

La cancellazione di un elemento da un albero binario è forse l'operazione più difficile in assoluto in questa struttura di dati. Infatti bisogna predisporre la funzione per eseguire tre diverse operazioni in quattro situazioni ben distinte:

1. si vuole eliminare un elemento che non ha figli (caso base e anche il più semplice)
2. si vuole eliminare un elemento che ha un figlio
3. si vuole eliminare un elemento con due figli

Il lato positivo di queste operazioni però sta nell'operatività con i puntatori, questa cosa infatti permette di effettuare un cambio di link tra due elementi con la semplicità di una modifica ai puntatori giusti. Inoltre dovendo toccare solo i link che collegano i padri all'elemento, non ci si deve particolarmente preoccupare dei figli di II generazione.

Questo è un esempio di funzione in cui il puntatore a padre può semplificare molto le operazioni.

(Tutte le implementazioni possono essere fatte in vari modi, qui sto riportando alcuni modelli per dare un'idea)

Eliminare un elemento senza figli

Se un elemento è senza figli vuol dire che è uno degli ultimi elementi dell'albero. Questo significa anche che non bisogna prestare particolare attenzione al comportamento dei figli, che non esistono.

Le uniche operazioni necessarie quindi sono quelle di eliminare l'elemento e impostare il link del padre a questo elemento al valore `NULL`.

Un'implementazione tipo può essere:

```
1 void EliminoElementoNoFigli(Blocco* ToBeDel){  
2     if (ToBeDel->dato > (Padre(ToBeDel))->dato){  
3         (Padre(ToBeDel))->right = NULL;  
4     } else {  
5         (Padre(ToBeDel))->left = NULL;  
6     }  
7     delete [] ToBeDel;  
8 }
```

Eliminare un elemento con un solo figlio

Se un elemento ha un solo figlio vuol dire che quest'ultimo prenderà il posto del padre.

Un'implementazione tipo può essere, quindi:

```
1 void EliminoElementoUNFigli(Blocco* ToBeDel){  
2     if (ToBeDel->dato > (Padre(ToBeDel))->dato){  
3         (Padre(ToBeDel))->right = (ToBeDel->right==NULL)? ToBeDel->left :  
4             ToBeDel->right;  
5     } else {  
6         (Padre(ToBeDel))->left = (ToBeDel->right == NULL)? ToBeDel->left :  
7             ToBeDel->right;  
8     }  
9     delete [] ToBeDel;  
10 }
```

Eliminare un elemento con entrambi i figli

Il problema in questo caso è che bisogna trovare un elemento che mantenga inalterate le relazioni dell'albero, ovvero un elemento maggiore o uguale rispetto a tutti gli elementi sulla sinistra e minore di tutti gli elementi sulla destra.

Questo processo diventa molto facile se si utilizzano le funzioni di minimo e massimo:

- posso trovare il massimo tra i minimi e sostituirlo con il valore da eliminare
- posso trovare il minimo tra i massimi e sostituirlo con il valore da cancellare

Un'implementazione di questa funzione può essere (in questo caso uso il minimo dei massimi):

```

1 void EliminoElementoDUEFigli(Blocco* ToBeDel){
2     if (ToBeDel->dato > (Padre(ToBeDel))->dato){
3         (Padre(ToBeDel))->right = Minimo(ToBeDel->left);
4     } else {
5         (Padre(ToBeDel))->left = Minimo(ToBeDel->right);
6     }
7     delete [] ToBeDel;
8 }
```

Unisco le funzioni di eliminazione

L'implementazione di un'unica funzione per l'eliminazione permette la cancellazione di un elemento in qualsiasi caso questo sia posizionato. Per fare ciò abbiamo bisogno di una formulazione ricorsiva.

Un'implementazione di questa funzione può essere:

```

1 bool Cancella(Blocco* Canc){
2     if(Canc->left == NULL && Canc->right == NULL){
3
4         if (ToBeDel->dato > (Padre(ToBeDel))->dato){
5             (Padre(ToBeDel))->right = NULL;
6         } else {
7             (Padre(ToBeDel))->left = NULL;
8         }
9         delete [] Canc;
10        return true;
11    }
12    else if (Canc->left != NULL || Canc->right != NULL){
13
14        if(Canc->left != NULL){
15            if (ToBeDel->dato > (Padre(ToBeDel))->dato){
16                (Padre(ToBeDel))->right = Canc->left;
17            } else {
18                (Padre(ToBeDel))->left = Canc->left;
19            }
20        }else{
21            if (ToBeDel->dato > (Padre(ToBeDel))->dato){
22                (Padre(ToBeDel))->right = Canc->right;
23            } else {
24                (Padre(ToBeDel))->left = Canc->right;
25            }
26        }
27
28        delete [] Canc;
29        return true;
30    }
31    else if (Canc->left != NULL && Canc->right != NULL){
32
33        if (ToBeDel->dato > (Padre(ToBeDel))->dato){
34            (Padre(ToBeDel))->right = Minimo(ToBeDel->left);
35        } else {
36            (Padre(ToBeDel))->left = Minimo(ToBeDel->right);
37        }
38        delete [] ToBeDel;
```

```
39     return true;
40
41 }
42 }
```

Altri tipi di Alberi

Il limite dell'albero binario di ricerca è lo sbilanciamento. Infatti questo può fare degenerare le prestazioni delle operazioni da $\Theta(h)$ a $\Theta(n)$ molto velocemente.

Un modo per risolvere questo problema è l'uso di *alberi rosso-neri* che rappresentano uno dei modi in cui gli alberi binari di ricerca vengono bilanciati per ottenere operazioni di *insert* o *delete* su insiemi dinamici che chiedano tempo di $\Theta(\log n)$ nel caso peggiore.

Un altro modo sono i B-alberi (ovvero che ogni elemento ha *tot* elementi, sempre) che sono alberi bilanciati particolarmente adatti per memorizzare grandi quantità di dati in memoria secondaria. Questi sono prediletti perché gli alberi *n-ari* non hanno particolari caratteristiche che permettono di diminuire il numero di passaggi nel normale uso, ma solo invece importanti in caso di specifici usi.

Questi alberi *n-ari* hanno la funzione di ricerca che ha complessità di $\Theta(\log_m(n))$, dove *m* è il numero di rami che ha ogni elemento.