INFORMATICA A

Giacomo Kirn

Luglio 2021

Contents

1 Introduzione

1.1 Librerie utili

- `math.h`: contiene le funzioni matematiche più utili, valore assoluto (`abs`), `cos`, `tan`, `exp`, `pow`.
- `time.h`: accede all'orologio di sistema, `srand(time(0)); r=rand();` (genera un numero a caso tra 0 a un massimo), con `time(0)` ho i millisecondi esatti di distanza da un certo tempo 0 ed è completamente casuale (se voglio un numero a caso tra 0 a 10 faccio `c=r%10;`).
- `limits.h`: si ha a disposizione `INT_MIN` e `INT_MAX` che mi dice il numero più grosso che posso scrivere.
- `string.h`: per acquisire frasi per le stringhe usando `gets(stdin)`.
- `stdlib.h`: per la memoria dinamica.

1.2 Struttura calcolatore

I sistemi informatici, ispirati dalla macchina di Von Neumann, sono formati da hardware e software. In generale i sistemi informatici sono costituiti da:

- **Unità di elaborazione (CPU)** che esegue le istruzioni e permette ai programmi di essere eseguiti, inoltre gestisce l'intero funzionamento della macchina con il sistema operativo;
- **Memoria centrale (MM, RAM)** che contiene i dati e le istruzioni che devono essere eseguite;
- **Bus di sistema** (insieme di connessioni fisiche) che collega la CPU e la MM tra di loro e con qualsiasi altra componente accessoria rispetto al funzionamento essenziale (la tastiera ecc.) che sono le interfacce delle periferiche (tastiera, schermo mouse, hard disk ecc.).

Memoria centrale: una tabella con un numero di colonne fissato, che sono i bit che stanno su una riga (possono essere h bit: 16, 32, 64...), si legge una riga alla volta; il numero della riga è l'indice di riga, che è l'indirizzo di una riga (è una sequenza di k bit, posso avere $2\hat{k}$ indirizzi). Ci sono diversi tipi di memoria centrale:

- **RAM** (random access memory), memoria volatile, celle che perdono il loro valore quando si toglie tensione, quando spengo il computer tolgo

tensione alla RAM e i dati dentro vengono copiati su un file sul disco per poi essere ripresi quando viene ridata tensione;

- **ROM**, i dati fondamentali per avviare il computer, è una memoria permanente, carica in memoria il sistema operativo quando viene acceso, non si perde quando il computer è spento;
- **EPROM** che permette di programmare la ROM.

l'HDD (hard disk) è memoria permanente, non centrale poiché non è in esecuzione, non si interfaccia direttamente con la CPU, non ci sarà mai un programma che viene eseguito dentro lì. **CPU**: il cuore del sistema informatico, quello che esegue e coordina la vita del calcolatore; ha tre elementi principali:

- **CU (control unit)** che esegue le istruzioni, le prende, le decodifica e le esegue;
- **ALU (unità aritmetico logica)** che esegue le operazioni aritmetiche;
- **CLOCK** che scandisce le esecuzioni e sincronizzazioni.

c'è però bisogno di tanti **REGISTRI** (memorie interne alla CPU) dove io posso scrivere i dati, (risultati delle operazioni ecc.), questi registri sono divisi in:

- **DR (data register)** che contiene una parola e ha dimensione della parola (h bit);
- **AR (address register)** che contiene gli indirizzi (k bit);
- **Registro istruzione corrente (CIR)**, una riga di memoria e contiene un'istruzione (h bit);
- **Registro contatore di programma (PC)**, tiene traccia dell'indirizzo che corrisponde all'istruzione appena eseguita, fondamentale per la sequenzialità (k bit);
- **Registro interruzione (INTR)**, su stato periferiche;
- **Registro di stato (SR)**, contiene il bit di overflow o di riporto, informazioni sui risultati ALU.

Per risolvere problemi di temperatura e di surriscaldamento si è optato per le installazioni di più CPU. **Bus di sistema**: connessioni fisiche per il transito dei dati. Esistono diversi tipi di bus:

- **Bus dati**, trasmette dati (h bit);
- **Bus indirizzi** (k bit);
- **Bus controlli** che trasmette istruzioni.

Nei bus esistono dei ruoli diversi, la CPU e la RAM o periferica e in queste connessioni c'è chi comanda, **master** (CPU) e chi obbedisce, **slave** (periferica). **Periferiche**: hanno tanti registri con le informazioni:

- **PDR (peripheral data register)**, dati;
- **PCR (peripheral command register)**, comandi;
- **PSR (peripheral state register)**, stato.

Le istruzioni (h=16 bit) devono essere codificate in binario e inserite nella MM, e sono costituite da: codice operativo (primi 4 bit) '00' e indirizzo operando (ultimi 10 bit); il codice operativo (obbligatorio) ci dice l'operazione da seguire e gli operandi (facoltativo) indica i dati da utilizzare o l'indirizzo (k=10 bit). Le variabili sono codificate in binario e salvate in parole nella MM, lo spazio a esse riservato dev'essere ben definito e allocato 'sotto' le istruzioni del programma, il programma deve conoscere gli indirizzi delle variabili. Un programma in memoria è diviso in una prima parte di istruzioni e una seconda di variabili, il programma parte dalla prima istruzione: per prima cosa avviene il **FETCH**, acquisizione dell'istruzione da parte della MM: trasferimento da PC a AR dell'indirizzo della cella contenente l'istruzione, lettura dalla MM della cella dell'indirizzo dato dalla AR (contenuto trasferito sul DR perchè l'istruzione è un dato), sposta da DR a CIR, incrementa PC (definisce la prossima istruzione); poi avviene la **DECODIFICA**, la CPU capisce cosa deve fare e infine l'**ESECUZIONE**.

1.3 Memoria

La memoria di un PC è divisa in celle e ogni cella contiene un dato che può essere numerico, un carattere o una stringa (sequenze di caratteri in celle adiacenti). Approssimando non c'è limite al numero delle celle o al valore dei numeri che esse contengono. Quando dico `int 'nomevariabile'` una fetta della memoria viene chiamata `'nomevariabile'`. Quando scrivo `'nomevariabile'`, intendo l'indirizzo di quell'area di memoria.

1.4 Variabili

Le variabili corrispondono a locazioni di memoria; ogni variabile ha un nome (identificatore), un tipo (insieme di operazioni e valori ammissibili), una dimensione, un indirizzo (individua la cella) e un valore. La lettura non modifica i valori delle variabili, inserendo un nuovo valore in una variabile si distrugge il valore vecchio. Le variabili rappresentano i dati su cui lavora il programma, sono denotate mediante identificatori, ogni variabile ha il suo e durante l'esecuzione hanno un valore ben definito, perciò devono sempre essere inizializzate. La variabile allocata segue il complemento a 2 (se ho una CPU a 8 bit, con il complemento si va da -$2\hat{7}$ a ($2\hat{7}$)-1, quindi da -128 a 127, perchè il complemento a 2 ha sia i numeri negativi che positivi). Se davanti a `int` metto `unsigned` memorizza solo i valori positivi.

1.5 Identificatori

Nomi delle variabili, sequenza di cifre, lettere o underscore, sono sensibili al maiuscolo e il primo carattere dev'essere una lettera. Ci sono parole chiave come `int` o `return` che non possono essere usate come identificatori.

1.6 Codice binario

Si parla di overflow quando il risultato corretto dell'addizione eccede il potere di rappresentazione dei bit a disposizione. Si può avere overflow senza riporto perduto: da due addendi positivi si riceve un risultato negativo. Si può avere un riporto perduto senza overflow: un riporto perduto senza effetto collaterale (due addendi discordi generano un risultato positivo). Per la rappresentazione in virgola fissa devo ricordarmi solo la mantissa e l'esponente (29,16=0,2916*$10\hat{2}$; 2916 è la mantissa e 2 è l'esponente). La virgola fissa è poco intelligente perché ti da un numero fisso di cifre per la parte intera e per la parte dopo la virgola centrando tutto intorno all'unità indipendentemente dalla grandezza del numero, cosa che non ha senso per i numeri grandi; sono più importanti le cifre significative, per questo si usa la virgola mobile: immaginare tutti i numeri come 0,'prima cifra diversa da zero' per base elevato a esponente (234,4=0,2344*$10\hat{3}$), mantissa è ciò che c'è dopo la virgola, ed esponente, così ho un tot di bit per la mantissa e un tot per l'esponente. L'effetto è che riesco a rappresentare con una densità maggiore i numeri vicino allo zero, in questo modo riesco a rappresentare numeri molto più grandi e piccoli rispetto a quelli che potrei rappresentare con la virgola fissa (ne concentro tanti intorno allo zero e ne prendo tanti lontani dallo zero) (con la virgola mobile ho il vantaggio che con gli stessi bit codifico un

intervallo più vasto, più sparso lontano dallo zero e più concentrato e preciso vicino allo zero). L'unico effetto negativo è che alcune operazioni hanno dei rischi: dato che tengo solo n cifre significative e sommo un numero che non rientra in queste quella somma non ha nessun effetto, e se ripeto l'operazione un infinito numero di volte l'errore diventa grande e considerevole (sommo 1 un miliardo di volte, ho un errore grande). Perciò non valgono una serie di proprietà della matematica continua, il calcolatore segue la matematica discreta.

1.7 Codifica dei caratteri

I caratteri (lettere maiuscole, minuscole, simboli ecc.) vengono codificati con i bit: codifica ASCII (american standard computer interchange interface), si utilizzano 7 bit per rappresentare 128 caratteri (log in base 2 di 128). Ma perchè dato che le memorie dei primi calcolatori avevano 8 bit ne usano solo 7? perché i segnali devono essere trasmessi e quindi sporcati, allora si usa l'ottavo bit per il controllo: in ogni codificato il numero di 1 è pari o dispari, metto l'ultimo bit a 1 o 0 per rendere il numero di 1 pari (bit di parità), se c'è un errore di trasmissione e 1 bit arriva sbagliato mi accorgo che il numero di 1 è dispari e capisco che c'è un errore (l'errore è basso per cui è improbabile che ci siano 2 errori). La codifica ASCII codifica l'inglese, la lingua della Danimarca ha molti più caratteri e si usa ISO 8859, per le lingue neo latine.

1.8 Codifica suoni e immagini

Esistono vari modi per codificare il suono: wav (CD), mid, mp3, ra e altri, il formato mp3 ad esempio è un formato compresso che perde precisione. Ne esistono anche per le immagini: jpeg, gif, pcx, tiff...; il formato classico è RGB (red, blue, green), per il monitor, che emette la luce, i colori primari sono rosso blu e verde; tutti i monitor emettono luce rossa verde e blu e ottengono tutti gli altri colori mischiando questi, la codifica RGB fa una matrice di tutti i pixel del monitor e per ognuno regola la quantità dei 3 colori (intensità da 0-255). E' conveniente rappresentare l'immagine secondo codifiche che permettono di ridurre le dimensioni: codifiche **lossless** (permettono senza perdita di informazioni di comprimere l'immagine, per esempio non ripetendo il valore nelle aree costanti ma registrando la variazioni: gif, png); o codifiche **lossy**: perdita di informazione e qualità (jpeg, compresse a blocchi e ogni blocco ha meno dettagli dell'originale). I video invece combinano la codifica del suono e dell'immagine.

In generale che siano caratteri numeri... i dati stanno in memoria che occupano spezzoni (parole) da 64 bit, una parola può essere usata per contenere

una parola un numero o altro.

1.9 Assegnamento

`'variabile'='espressione'`, dove l'espressione può essere una costante, una variabile o una combinazione di espressione costruita mediante operatori aritmetici (`+-*/%`, `%` è il modulo che restituisce il resto della divisione intera). Se scrivo `w='a'` vuol dire che alla variabile w è stato assegnato il carattere a, se scrivo `w=a` vuol dire che a w è stato assegnato il valore della variabile a. Per gli assegnamenti posso anche scrivere `x=x+1` (se `x=5`, `x=x+1=6`). `x=x+1=++x`, `x=x*5` si può dire `x*=5`, `x=x-1=- -x`, `x=x+7` si può dire `x+=7`. Quando faccio le divisioni il risultato è troncato (`13/5=2`). Si parla di pre-incremento quando faccio `++i`, prima incremento poi guardo cosa vale. Il post-incremento prima guardo cosa vale poi incremento.

1.10 Tipi di variabili

Ci sono diversi tipi di variabili: `int` (numeri interi), `char` (caratteri-ASCII), `float` (numeri con la virgola). Le variabili `float` hanno tot bit per mantissa e tot esponente, esistono i `double float` (doppio bit mantissa e doppio esponente), si parla di inizializzazione quando do un valore iniziale a una variabile (quando la dichiaro do subito un valore) posso usare il `const` che blocca il valore della variabile. Il C permette di fare cose del tipo: `int b, b='a'`, che è controintuitivo. Nella `printf` si usa `%d` (interi), `%c` (caratteri), `%f` (float), `%lf` (double float), `%s` (stringa) `\n` (a capo), `\t` (tabulazione, incolonnare), `\a` (allarme), `\` (escape, annulla il significato del successivo). Nella `scanf` devo mettere `&`. Riassunto: `int`, utilizzano 1 parola, esistono gli `short int` e `long int`, oppure anche `signed` o `unsigned` (per `unsigned` gli interi diventano da 0 a il doppio, non più da meno n a più n ma solo o-2n); `float` o `double float` (i `float` 8 byte, i `double` 16 byte), lo spazio occupato in memoria però dipende dal compilatore; `char` rappresentati con ASCII, vale il minore (prima c'è l'alfabeto maiuscolo poi minuscolo). Quando sommo `int` e `float` si seguono le regole dei `float`. In generale quando si sommano due tipi diversi quello più piccolo in memoria subisce una conversione automatica (cast implicito); se io sommo `int` e `float` l'`int` viene convertito automaticamente in `float` (a volte queste conversioni generano un warning). Al posto di mettere `const int...` si può usare `#define N 10` si mette in alto dopo le librerie, è meglio definire le costanti così perchè non si spreca la lettura di una cella della RAM (non va in giro a cercare variabili) [per tradizioni `const` maiuscole e variabili minuscole].

1.11 Huffman

Esiste l'Huffman coding tree che permette di sfruttare la sequenza relativa dei singoli caratteri per risparmiare spazio, attraverso il suo albero la codifica dei caratteri più utilizzati occupa uno spazio molto più ristretto rispetto ai caratteri meno utilizzati. Il problema di questa codifica è che se si perde una parte di codice di una parola ad esempio, si ha perso tutta la parola. [10 bit: $2\hat{1}0$ possibili numeri (1000), 20 bit: $2\hat{2}0$ (1 mln), 30 bit: $2\hat{3}0$ (1 mld)].

2 Algoritmi e Strutture Dati

2.0.1 Algoritmo

L'algoritmo è una sequenza precisa di operazioni, definite con precisione, che portano alla realizzazione di un compito (in un tempo finito). Le operazioni devono essere comprensibili ed eseguibili (da un esecutore). Il linguaggio con il quale scrivo l'algoritmo dev'essere caratterizzato dalla sequenzialità delle istruzioni, deve avere un costrutto condizionale e un costrutto iterativo, e inoltre dei foglietti in memoria dove posso scrivere dei valori.

- **Sequenzialità**: ogni istruzione comincia quando quella prima termina, si eseguono tutte le istruzioni.
- **Condizionale**: c'è un bivio, si arriva a un'istruzione in cui ci sono 2 opzioni 'altrimenti', eseguita una delle due si continua con le istruzioni dopo; c'è una condizione che viene valutata. per cui: `if (condizione con istruzioni a seguito se l'istruzione dell'if è vero) else (se l'if è falso continuo con l'istruzione dell'else)`.
- **Iterativo**: serve per ripetere più volte la stessa istruzione, il flusso prosegue e se un'istruzione vera ripeto le operazioni. si parla di cicli (`while qualcosa è vero faccio una cosa, se l'istruzione nel while è falsa vado avanti: while (qualcosa) {istruzioni}`).

Si parla di correttezza quando l'algoritmo viene eseguito senza errori, mentre si parla di efficienza quando l'algoritmo usa le risorse a disposizione in modo minimale (o ragionevole). Un algoritmo si dice ricorsivo se continua a richiamare se stesso finchè non si arriva al caso base che interrompe l'algoritmo (ho un archivio e devo cercare una scheda, ho N schede, parto dalla scheda N/2, dal punto di vista alfabetico guardo se la scheda cercata è prima o dopo e così elimino metà archivio e continuo così finché non trovo la scheda desiderata). Esiste il ciclo `do-while`, scrivo il `do` con nelle graffe le istr e all'est il `while` con la condizione. `Switch`: si scrive `switch (var (solo int o char)) { case v1: istr1; break;...case vn:...}` questo costrutto guarda il valore della variabile se è uno dei v salta a eseguire il codice subito dopo i `:`, appena trova `break` salta fuori dallo `switch` (per questo il `break` è importante. Il `break` si può anche usare per uscire da un ciclo (il ciclo corrente non da tutti insieme); c'è anche il `continue` che salta direttamente a verificare la condizione del ciclo (non sta a guardare quello che viene sotto ma fa ricominciare il ciclo).

2.1 Diagrammi di flusso

I diagrammi di flusso sono linguaggi semi-formali intelligibili solo all'essere umano, i linguaggi che può leggere anche la macchina sono i linguaggi di programmazione. Programmare è tradurre gli algoritmi da diagrammi di flusso a istruzioni che la macchina può capire e dettagliare in quanto noi umani possiamo ragionare a un livello molto alto, la macchina ha bisogno di istruzione molto compatte (in binario ad esempio). Per cui il programmatore deve essere in grado di ideare l'algoritmo e codificarlo in un programma (tradurre in linguaggio macchina). Il `FOR` si può usare in questo modo: `for (exp iniz; cond; exp incr) { }` (per esempio nel `incr` posso mettere `i++`).

2.2 Algebra di Boole

E' l'algebra della logica, si basa su operazioni logiche, le operazioni sono applicabili a operandi logici, operandi che possono assumere solo valore vero o falso (vero 1 bit, falso 0) e danno vita a espressioni booleane. Ci sono operatori relazionali e logici:

- **Operatori relazionali**: `==, !=, <, >, <=, >=`;
- **Operatori logici**: `&&, !, ||` (esiste anche `XOR`, che è l'or esclusivo: o uno o l'altro ma non entrambi); quelli logici permettono di costruire operazioni composte. `||` e `&&` sono commutativi; le doppie negazioni si elidono (`!!`).

Ogni espressione può assumere solo due valori, posso quindi considerare tutti i possibili valori degli ingressi ad un'espressione booleana e calcolare i valori di output corrispondente; si può creare una tabella di verità che rappresenta tutti i valori che un'espressione composta può assumere al variare delle espressioni che la compongono:

- `!` (not): se A falsa `!A` è vera.
- `&&` (and): `A&&B` è vera se e solo se sia A che B sono vere.
- `||` (or): `A||B` è vera se e solo se almeno una delle due è vera.

Le leggi di De Morgan illustrano come distribuire la negazione rispetto a `||` e `&&`: `!(a && b) == !a || !b !(a || b) == !a && !b`. Esistono regole di precedenza: prima `!` poi `&&` poi `||`. Se ho una condizione, e metto un insieme il programma considera vero sempre tranne quando inserisco uno 0.

3 Programmazione in C

3.1 Struttura programma

C'è una prima parte dichiarativa, nella quale dichiaro gli elementi del programma (librerie, variabili ecc.) e una parte esecutiva, che contiene le istruzioni da eseguire: istruzioni che possono essere di assegnamento o strutture di controllo (condizionali, iterative o cicli) o istruzioni di input, output (`printf`, `scanf`). ci sono istruzioni semplici, che terminano con il `;` e istruzioni che si possono raggruppare e vengono raccolte in un blocco chiuso tra {}. Teorema di Bohm e Jacopini: tutti i programmi possono essere scritti in termini di 3 strutture di controllo: sequenza, istruzioni eseguite in un certo ordine; selezione, istruzioni che permettono di prendere strade diverse in base a una condizione; iterazione, istruzioni che permettono di ripetere una cosa fintanto che una condizione booleana rimane vera.

3.2 Stringhe o Array

Sequenza di caratteri, si indicano rinchiuse tra doppi apici `\ "`. Sono una sequenza di foglietti della memoria, per cui sono un insieme di valori con un insieme di operazioni ad essi applicabili, è un tipo di variabile strutturata (quelle semplici sono `int`), e sono un insieme di variabili semplici (in C è possibile creare un tipo di variabile built-in, gli array sono user-defined).

I qualificatori di variabili sono `signed` (-2$\hat{3}$1 – 2$\hat{3}$1-1, scrive in CP2) e `unsigned` (0 – 2$\hat{3}$2-1) (si usano per `int` e `char`) e allocano lo stesso spazio; mentre i quantificatori sono `short` e `long` (si usano per `int` lo `short` e per `int` e `double` il `long`).

Le variabili strutturate possono essere omogenee o eterogenee, gli array sono omogenei, è una sequenza di variabili omogenee: posso dare un indice (un elemento precede un altro nella sequenza), le variabili sono tutte dello stesso tipo (omogenei), e si alloca in celle consecutive di memoria. Sintassi: `tipo nomeArray [dimensione];` (dimensione è un numero fisso noto a compile-time, non può essere una variabile e non può essere modificato durante l'esecuzione). Si può accedere ai singoli elementi dell'array mettendo nelle quadre il numero (`vet[0]`); ogni elemento è una variabile del tipo dell'array, per cui posso fare assegnamenti, operazioni logiche, aritmetiche e di I/O. Se scrivessi `int vet [300]; vet` (da solo) vuol dire: `vet == &vet[0]` (indirizzo prima cella array) (si stampa l'indirizzo con `%p`). Non posso mai dichiarare l'array senza dichiarare la dimensione (con la `#define` si può mettere una variabile nella dimensione degli array dato che la dimensione

dell'array è nota a priori). Ci sono dei vantaggi di sintassi compatta: `int n [5]=0` (vuol dire che tutti gli elementi sono inizializzati a 0); `int n [5]={13}` (alla posizione 5 c'è il 13, le altre sono 0). Si possono usare espressioni come indici. Acquisizione: bisogna procedere elemento per elemento con la `scanf`.

3.3 Stringhe

Array di caratteri (alfanumerici, simboli, di comando), si codificano con l'ASCII esteso a 8 bit (256 caratteri). si dichiarano come gli array (`char luogo[dim]`). Per far si che durante l'acquisizione non prenda l'invio bisogna far pulire il buffering in ingresso attraverso `fflush(stdin)` oppure attraverso `scanf (\%*c")`. Attraverso la `scanf` o la `gets` si possono acquisire stringhe direttamente: `char str[10]; scanf(\%s", str); gets (str);` la `scanf` non ha bisogno di `&` poichè `str` è già l'indirizzo di `str[0]`, la `scanf %s` acquisisce solo fino al primo invio o spazio, se voglio acquisire una frase uso la libreria `string.h` (usando `gets(nomestringa)`, la `gets` termina con il primo invio, la `scanf` con il primo spazio o invio). Per le stringhe c'è un carattere speciale: `\0`, si mette al termine della parola, viene inserito automaticamente in memoria e ci dice le dimensioni effettive di una parola, è presente in modo che si stampino tutte le lettere fino al terminatore di stringhe. Esiste `strlen` per la lunghezza delle stringhe (sempre in `string.h`): `len1=strlen(str1)`. Esiste `strcmp` (`string.h`), restituisce 0 quando le due stringhe hanno la stessa lunghezza e stessi elementi nelle stesse posizione, altrimenti minore di 0 se S1 precede S2 (in ordine alfabetico) o maggiore se è vero il contrario: `cmpr=strcmp(S1, S2)`.

3.4 Matrici

Gli array a 1D realizzano i vettori, gli array a 2D realizzano le matrici, la dichiaro scrivendo ad esempio: `int a [20][30]` (alloco 20*30 elementi interi, 600 variabili distinti). Le matrici possono essere sommate solo se hanno la stessa dimensione (stesso numero righe e colonne); esiste il prodotto per uno scalare; esiste il prodotto riga per colonna: possibile solo se la prima matrice ha lo stesso numero di colonne del numero di righe della seconda. La trasposizione consiste nello scambiare gli elementi `a[i,j]` con gli elementi `a[j,i]`.

3.5 Struct

Tipi di dati strutturati (aggregazione di variabili), sono una collezione di variabili eterogenee (non ordinate e variabili diverse). si possono creare tipi

user-defined (generate dall'utente). Sono una sorta di contenitore di variabili disomogenee più semplici, le variabili aggregate sono dette campi della struct. Sintassi: `struct { tipo1 Nomecampo1; ... } NomeStruct;` (si possono dichiarare più variabili dalla stessa struttura: più di un NomeStruct). Va nella parte dichiarativa del `main`. Per accedere ai campi si usa l'operatore dot (il punto): `Nomestruct.NomeCampo` (a questo punto tutta sta roba diventa una variabile del tipo del NomeCampo). Se ho due struct identiche si può fare `Nomestruct1=NomeStruct2` (i valori si assegnano da una struttura all'altra).

3.6 Typedef

Serve a definire un nuovo tipo. Sintassi: `typedef NomeTipo NomeNuovoTipo`; Va messo al di fuori del `main`, così che all'interno del `main` posso usare il nuovo tipo; questa cosa ci permette di dare un significato al nuovo tipo. Posso fare il `typedef struct` anche (la `struct` è un tipo di dato).

3.7 Puntatori

Le variabili hanno un indirizzo grazie al quale si può accedere ad esse (in memoria una variabile può occupare una o più celle), l'indirizzo è l'indirizzo della locazione in memoria della cella in cui inizia una variabile (ogni cella al suo interno ha un valore); l'indirizzo di una variabile non muta durante l'esecuzione. Quando una variabile è a sinistra di un assegnamento si usa il suo indirizzo per modificare il suo valore, quando a destra si usa il valore al quale si accede tramite l'indirizzo. In C si può accedere agli indirizzi tramite `&`. Si possono creare delle variabili, chiamate puntatori, che contengono l'indirizzo di un'altra variabile. In C i puntatori si distinguono in base al tipo della cella puntata, la loro sintassi infatti è: `int *punt;` (se non mettessi l'asterisco creerei una variabile normale). Se voglio assegnare a un puntatore l'indirizzo di una variabile dello stesso tipo: `punt=&x;` se voglio sapere il valore delle variabili puntate tramite il puntatore si usa la dereferenziazione facendo: `*punt;` (`&` è l'operatore di referenziazione). Questo sistema sarà utile quando non si sanno le variabili ma solo il loro indirizzo. I puntatori possono essere dichiarati con le altre variabili. Si possono fare i doppi puntatori (o tripli) che puntano a puntatori che puntano a un tipo dato. Il `NULL`: è una costante simbolica che rappresenta un valore speciale che può essere assegnato a un puntatore, significa che la variabile non punti a niente (di solito si impone che `NULL` rappresenti il valore 0). Se inizializzo a `NULL` un puntatore vuol dire che non punta a nessuna cella di memoria. Per il C si possono creare puntatori che non si sa ancora a cosa puntino usando il `void *` (in questo modo tutti i puntatori sono compatibili con puntatori

di altri tipi). Al posto di scrivere `(*p).primoCampo=12;` si può scrivere `p->primoCampo=12;` (freccia con il meno e il maggiore). Esiste la `sizeof` che ci dice quanti byte in memoria occupa una cella (`sizeof()`). Sono permesse delle aritmetica dei puntatori: somma tra puntatori: se faccio `p=p+3` gli dico vai avanti di 3 (dimensione del tipo del puntatore), per cui se `p` è un `int p+3` avanza di 4 byte 3 volte (se è variabile indirizzo avanza di 8 byte 3 volte), infine quindi è come se facessi puntare al puntatore de `int` (o `char` o `float` o...) avanti. E' utile per array e vettori: `v[n]; &v[0]` è `v, &v[3]` è `v+3`. Se ho due `punt` che puntano a due caselle dell'array e faccio la differenza ho la distanza in caselle tra le due.

3.8 Funzioni

Le funzioni servono per poter fare particolari diagrammi di flusso. Permettono di definire blocchi di istruzione che risolvono compiti di alto livello per poter scrivere programmi più leggibili. Le funzioni restituiscono qualcosa. Quindi in un diagramma di flusso è possibile sostituire dei blocchi che svolgono funzioni di basso livello con blocchi che ne svolgono di alto o viceversa (i blocchi ad alto livello rendono l'algoritmo più comprensibile). Praticamente sono sequenze di istruzioni che risolvono sottoproblemi (la sequenza può essere riutilizzata anche in altri programmi), alla fine ci troveremo senza dover occuparci di ogni singola istruzione, ma solo di blocchi concettuali che portano a compimento compiti intermedi (e scrivere istruzioni che le risolvono in un secondo blocco). In C una funzione è una sequenza di comandi che ha un nome, può essere invocata, può ricevere dei parametri che ne influenzano l'esecuzione e produce un valore risultato. Con le funzioni si può scrivere molto meno codice, possono essere riutilizzate, permettono di esprimere in modo sintetico operazioni complesse e si possono definire operazioni specifiche. Le funzioni stanno all'esterno del `main`: Sintassi: `void NomeFunzione () {...}`. Nel `main` metterei `NomeFunzione ();`. `Void` è una keyword e in questo caso si usa poiché la funzione non restituisce niente ma stampa e basta (se volessi fare altro non potrei usare il `void`). Se volessi includere anche l'acquisizione di un `char` (ad esempio) non posso più mettere `void` ma `char NomeFunzione () {... return NomeChar}` (la funzione deve restituire un `char`); a questo punto nel `main` devo mettere `char1(variabile)=NomeFunzione();`. Nelle parentesi prima delle graffe posso far tenere presente alla funzione dei valori: `int NomeF (int min, int max) {...}`. Sintassi: `TipoRestituito NomeFunzione (lista argomenti) {VarLocali; CorpoFunzione; return TipoRestituito;}` (consentono di restituire solo 1 valore) (lista argomenti è una sequenza di coppie (tipo nomeargomento)). Per invocare nel `main` la funzione faccio `NomeFunzione`

`(lista argomenti)`, la lista argomenti non è la stessa della funzione quando va creata, ma una lista di espressioni (variabili, costanti, invocazione di funzione) di tipo corrispondente a quelle della definizione (i valori scritti vengono usati per la funzione, con l'esempio di `int min` e `max` dovrei mettere nel `main NomeFunzione (2, 10)`, per cui c'è una porzione di memoria chiamato record di attivazione, riferito alla funzione, creato una volta invocata la funzione, in questo caso il 2 sarà copiato in `min` e il 10 in `max`). Prima del `main` devo mettere `TipoDato NomeFunzione (lista argomenti)` (in questo caso la lista argomenti non deve essere dettagliata al massimo, posso anche mettere solo il tipo). Quando termina l'invocazione di una funzione il suo record di attivazione viene distrutto. Nelle funzioni ci sono parametri formali (si usano per definire la funzione e sono gli argomenti, gli argomenti sono parametri formali in input, dopo il `return` nella funzione c'è il parametro formale in output) e parametri attuali (usati quando invoco la funzione nel `main`) che danno valore ai parametri formali, l'invocazione copia il valore da attuale a formale. [Nel `main` e nella funzione le variabili occupano porzioni di memoria completamente diverse, c'è uno spazio per il `main` e uno per la funzione]. Per cui ogni sottoprogramma può usare solo le variabili dichiarate al suo interno che sono variabili dette locali che nascono e muoiono con la vita del sottoprogramma (le variabili non dichiarate all'interno del sottoprogramma si possono usare se passate come argomenti).

3.9 Gestione della memoria di una funzione

Una variabile nel record di attivazione di una funzione non può modificare una variabile in un altro record di attivazione (per esempio quello del `main`), il record di attivazione è chiamato anche ambiente di una funzione. Ci sono procedure che richiamano loro stesse (ricorsione), possono esistere più istanze di una funzione addormentate in attesa della terminazione della gemella per riprendere l'esecuzione (in questo caso il compilatore non può sapere quanto spazio allocare per le variabili del programma). Per la ricorsione non esiste un solo record di attivazione ma si usa il sistema dello stack, modalità LIFO (last in first out). Le variabili delle funzioni sono chiamate variabili automatiche, dichiarate nelle funzioni e nei blocchi di istruzione, create quando il flusso di esecuzione entra nel campo di visibilità e distrutte una volta uscite da tale ambito; di volta in volta sono allocate in celle differenti e non conservano i valori prodotti da precedenti esecuzioni della funzione o del blocco. I parametri possono essere built-in o user defined; i valori dei parametri attuali non sono modificabili tra le istruzioni del chiamante, i sottoprogrammi lavorano su copie dei parametri attuali (tranne gli array apparentemente). I `return` possono essere built-in o user defined ma non possono essere array

(ma può essere una `struct` contenente un array) e può essere un puntatore a qualsiasi tipo. Passaggio per riferimento: il puntatore ci da la possibilità di passare da un record di attivazione a un altro. Funzioni ed array: gli array si possono passare anche in ingresso a una funzione.

3.10 Punto sulla situazione

Non si possono restituire array (non è assegnabile), il modo per farlo è sfruttare il fatto che il passaggio a parametri (nella chiamata) viene passato l'indirizzo, in realtà scrivo direttamente nel vettore del `main` attraverso puntatori. Quando passo vettori di interi devo passare anche la lunghezza, altrimenti non posso sapere quanti elementi sono significativi. Con le `struct` non ci sono questi problemi, le `struct` vengono passate in input e restituite dalle funzioni come i normali tipi base; questo perché è possibile fare assegnamenti tra `struct` (vale anche se la `struct` ha un array tra i propri campi), ed è sempre possibile utilizzare il passaggio per riferimento anche con i tipi strutturati.

3.11 Ricorsione

La ricorsione è la versione informatica dell'induzione matematica (si parte dal caso base e si dimostra il generale). Nella ricorsione, a differenza dell'iterazione in cui ho un ciclo, divido il caso in due: ho un caso base semplice del quale conosco già la risposta, e un passo induttivo che tende al caso base e definisco la soluzione del problema in termini di operazioni semplici e della soluzione dello stesso caso base su dati "più piccoli" (per tali dati e per ipotesi il problema si considera risolto). Versione ricorsiva del fattoriale: (con una funzione). `int fatt (int n) {if (n==0) return 1; else return n*fatt(n-1);}`. Un programma ricorsivo è un programma che continua a richiamare se stessa. (l'ultima funzione chiamata è la prima che termina, LIFO). In ogni chiamata ricorsiva si è in ambienti distinti. La ricorsione non va avanti all'infinito se mi avvicino sempre di più al caso base, per cui una ricorsione è giusta se avviene questo.

3.12 Memoria dinamica

Le variabili sono dichiarate nelle funzioni e create e distrutte con la creazione e distruzione della funzione che le contiene. Ci sono anche variabili statiche: variabili globali (dichiarate fuori dal `main` e viste da tutte le funzioni con lo stesso nome, per cui se in una funzione ne aumento il valore lo aumento per tutte le funzioni) o locali al `main` dichiarate con `static`. Le variabili,

qualsiasi tipo, possono essere allocate dal compilatore immediatamente in memoria, a prescindere dal programma o la funzione. Esistono le variabili dinamiche, allocate e deallocate nell'atto di creazione esplicita, vengono create a runtime tramite un comando, queste non possono avere un nome ma conosco il loro indirizzo, per cui ci accedo tramite puntatori; queste variabili non vengono create nello stack (delle altre variabili e ordinato), ma nello heap (mucchio) riservato esclusivamente per queste variabili, nello heap le variabili possono essere raggiunte solo tramite puntatori (o catene di essi che partono da variabili nello stack). Sono definite nella libreria `stdlib.h`, queste variabili vanno create e distrutte esplicitamente tramite le funzioni: `malloc` (allocarle) e `free` (rilasciarle), non c'è nessun vincolo per il quale queste funzioni devono stare all'interno della stessa funzione. La `malloc`: prototipo: `void *malloc (int);`. Restituisce un puntatore generico, non si può sapere a priori cosa si vuole puntare). Come parametro riceve il numero di byte da allocare (di solito si usa la `sizeof` per la dimensione dei dati da allocare). Se non c'è più memoria disponibile restituisce `NULL`. Allocazione: `typedef ... TipoDato; typedef TipoDato *PTD; PTD ref; ... ref=(PTD) malloc(sizeof(TipoDato));` (`PTD` davanti alla `malloc` non dovrebbe essere obbligatorio); l'unico modo per accedere alla variabile presente nello heap è con la dereferenziazione (`*ref` in questo caso). La creazione di variabili nello heap avviene dove si trova posto, ogni volta che rieseguo il programma cambia. Bisogna cancellare le variabili dello heap, che se no si satura, lo si fa attraverso `free` (libera la memoria allocata dalla `malloc`): Prototipo: `void free (void*); come argomento ha un puntatore (nel caso dell'esempio avrei: free (ref); poiché ref è già un puntatore), non serve specificare la dimensione`. La `malloc` alloca spazio, non tipi. Quando faccio `free` non elimino il puntatore ma solo il valore a cui puntava, bisogna stare attenti perchè si continua lo stesso a puntare a una casella nello heap, ma senza avere valori, per cui è bene assegnare a ciò che libero il valore `NULL`.

3.13 Liste

E' una struttura dati dinamica la cui logica è non creare strutture sovradimensionate, ma creare elementi divisi in due parti: una parte è l'informazione utile (e ne contengono solo una) e l'altra è il puntatore all'elemento successivo, altro elemento sempre diviso in due (o punta a un altro o a nulla); è tutta nello Heap, e perchè si possano usare deve esistere una variabile nello stack che punta a uno di questi elementi. Mi permettono di inserire un elemento in mezzo a due che già esistono senza tanti problemi. Non c'è più l'aritmetica dei puntatori, gli indirizzi non sono contigui. Come si costruis-

cono: sono fatte da cose che contengono un puntatore (il primo è un puntatore a parte). Sintassi: `typedef struct EL {TipoElemento info; struct EL *prox; } ElemLista; typedef ElemLista *ListaDiElem;` (è il puntatore iniziale). A questo punto posso creare una lista concatenata di oggetti; la lista dev'essere mantenuta ordinata. Le liste possono essere di tanti tipi, noi cominciamo con le liste concatenate: cominciano con un puntatore al primo e terminano con un puntatore nullo e si attraversano solo in un verso. Esistono le concatenate circolari (l'ultimo elemento punta al primo), le doppiamente concatenate (ogni elemento punta al successivo e al precedente), e le doppiamente concatenate circolari. Liste concatenate: Sintassi: `typedef struct EL {TipoElemento info; struct EL *prox; } ElemLista; typedef ElemLista *ListaDiElem;` (è il puntatore iniziale) `ListaDiElem ptr; ptr=malloc(sizeof(ElemLista)); (*ptr).info=10; (*ptr).prox=NULL;` Tutte le operazioni che sono possibili fare su una lista possono essere messe in funzioni che hanno come parametro un puntatore al primo elemento della lista, le funzioni che modificano la lista sono scritte in modo che restituiscano un puntatore al primo elemento della lista.

3.14 Alberi

E' una struttura dati dinamica composta da nodi che si puntano tra di loro, non si ha solo un puntatore all'elemento successivo, ma due, è un nodo da cui spuntano due rami. Per gli alberi la ricorsione è necessaria. Nomenclatura "rubata" dall'albero genealogico. Sintassi: `typedef struct Nodo {int dato; struct Nodo *left; struct Nodo *right;} nodo; typedef nodo *tree;` Foglia: nodo che non ha neanche un successore (entrambi i nodi puntano a NULL).

4 Basi di Dati

4.1 Introduzione

I linguaggi visti finora solo linguaggi interattivi, si dice alla macchina che fare; esistono linguaggi dichiarativi: descrivono il risultato senza dire nei dettagli alla macchina che fare, sono permesse meno cose del C ma con istruzioni molto più semplici. Per questi programmi esiste una sezione di dati unica e tanti programmi che possono usarli, esiste quindi un DBMS (Data Base Management System), che permette ai vari programmi di accedere ai dati senza creare problemi, bisogna consentire l'accesso ai dati in maniera concorrente e corretta; bisogna essere in grado di condividere i dati (senza replicazione), devono essere robusti (senza falle nel sistema, operazioni memorizzate e fisse), i dati devono mantenere la qualità (integrità e correttezza), l'efficienza e ovviamente la privacy; bisogna avere un meccanismo che riesce ad avere più copie fisiche che sembrino una sola copia senza replicazione. Nel corso ci limitiamo a estrarre le informazioni dai dati, non a gestirli. Come sono fatti i Data Base: c'è una tabella, le colonne sono gli attributi e le righe sono la istanza; per costruire questo modello si usa il modello relazionale: le colonne e le righe si vedono in linea con la teoria degli insiemi.

Grado : numero di domini;

Cardinalità : numero di n-uple (t-uple);

Attributo : nome dato al dominio in una relazione.

Una t-upla sarà la riga e l'attributo la colonna. Se non si conosce un dato per svariate ragioni si mette `NULL`, che appartiene a tutti i domini e indica che non si sa l'informazione (sconosciuto, inesistente, senza informazione). Con il `NULL` non si può fare alcun tipo di confronto. Esistono dei vincoli nei dati, ma ci pensa il Data Base, noi ci aspettiamo che i dati siano giusti a priori. Per cui i dati sono corretti e sappiamo che esiste un sottoinsieme degli attributi che ha le proprietà di unicità e minimalità:

- **Unicità**: non esistono due t-uple con due chiavi uguali;
- **Minimalità**: sottraendo un qualsiasi attributo alla chiave si perde unicità;

Per cui il set di attributi che garantisce unicità e minimalità si chiama chiave, noi lavoreremo pensando che la chiave è affidabile (se il sottoinsieme non è minimale si parla di super chiave). Ci sono dei vincoli di chiave esterna: vincolo gli elementi di una tabella con una chiave di un'altra tabella (se voglio

i nomi e cognome dei 30 in un esame devo prenderli da un'altra tabella con un'altra chiave).

4.2 SQL

E' un linguaggio dichiarativo, in questo caso serve per estrarre dati da queste tabelle: Prima operazione: PROIEZIONE (da una tabella prende delle certe colonne): prima parola chiave. from (ci dice la tabella da cui prendere), select (la colonna da prendere); per cui la sintassi sarà: `select NOME, CCS (in questo caso) from TABELLA STUDENTE.` (se scrivo `select *`, mi fa vedere tutte le colonne). Per rimuovere i duplicati uso `distinct`: `select distinct CCS`. Si parla di SELEZIONE quando voglio selezionare righe: `SELECT * from STUDENTE WHERE Nome='alex'` (oppure 6 se fosse stato un intero senza apici) (gli diciamo quale t-upla selezionare). Le convenzioni sono: stringhe con apici, numeri come numeri, esistono stringhe con caratteri numerici: `'1234'` (in questi casi si guardano le cifre quando sono più grandi: `'1234'<'56'`); esistono operazioni booleane: `and` (`p1 and p2`), `or` (`p1 or p2`), `not` (`p1`); ovviamente ci sono anche i comparatori. (`=, <>, <, <=, >, >=`) Mentre valuta una certa tupla non valuta le altre, nella clausola WHERE si scrivono clausole che vedono una riga per volta per decidere se tenerla o no. Ogni confronto con `NULL` finisce male, per cui si usa `is` (`WHERE CCS is (not) NULL`) (`is` si usa solo per i controlli con il `NULL`). Se nel `FROM` metto due tabelle con la virgola in mezzo il sistema fa una grande tabella con i dati di tutte e due con il prodotto cartesiano; nel `WHERE` se ho una tabella mischiata devo mettere il punto tra nome tabella e nome attributo. Si può usare `JOIN` e `ON` al posto di `where` (`from studente JOIN esame ON studente.matr=esame.matr`), con il `JOIN` si possono accoppiare tuple di tabelle diverse e poterci scrivere duplicati sopra. (il `join` mette insieme più colonne sulla stessa riga). Variabili in SQL: posso scrivere `SELECT S1.nome, S1. matr FROM studente S1, studente S2` (`S1` e `S2` sono delle denominazioni che do io) `WHERE S1.nome=S2.nome AND S1.matr <> S2.matr` (cerca gli omonimi). ORDINAMENTO: `order by`: ci mostra il risultato in ordine di quella cosa (di solito crescente se metto `DES` è decrescente). Funzioni aggregate: non si mostra tutto il risultato della query (tutte le righe che soddisfano una condizione), ma mostriamo qualcosa che viene calcolato su queste righe tramite le funzioni aggregate: `count` (cardinalità), `sum` (sommatoria), `max`, `min`, `avg`; Farei: `select count (distinct (oppure all) CodCli) from Ordine` (estrarre il numero di valori distinti di `CodCli` per tutte le righe di ordine). Se faccio: `select max (importo) as MaxImp from ordine` (dico al programma di chiamare la tabella derivata con il massimo valore MaxImp); la funzione `max` ci da solo il valore del massimo, ma non

sappiamo niente sulla riga dal quale viene. Consegue il divieto assoluto di mettere nella stessa query un aggregato e un qualcosa che non lo è. Query con raggruppamento, `group by`, il raggruppamento funziona che creo dei gruppi in base a una caratteristica (se non c'è un gruppo con una caratteristica quello non esiste, non è che esiste ma è vuoto). c'è anche la clausola `having` che seleziona i gruppi, permette di esprimere predicati sul gruppetto, predicati su aggregati; le tuple vengono viste divise a gruppetti. Se c'è la clausola `group by` posso mettere aggregati e attributo nel `select` (gli attributi solo per cui è raggruppato). posso mettere nella `select` solo attributi 'vittime' di raggruppamento. Esiste il doppio raggruppamento. Non esiste l'aggregato di aggregato!!! E' possibile confrontare il valore di una riga con il risultato di una query: query nidificate: `select distinct codord from dettaglio where codprod = any (select codprod from prodotto where prezzo > 100)` (`= any` è almeno 1) in questo caso se nell'altra query c'è almeno un risultato uguale a `codprod` è vera la condizione (si può usare anche `all`, `not in` è equivalente a `<> all`). Esiste anche l'`in` che è equivalente di `=any`. Costruttore di tupla: `select * from Persona P where (Nome,Cognome) in (select Nome, Cognome from Persona P1 where P1.CodFisc <> P.CodFisc).` Viste: offrono la visione di celle "virtuali" per fare query complessi, posso scrivere qualunque query con davanti `create view` che crea una tabella virtuale con il risultato della query e posso usarla come fosse una tabella: `create view OrdiniPrincipali as select * from Ordine where Importo > 10000` Query: `select CodCli from OrdiniPrincipali` OPPURE: `select CodCli from (select * from Ordine where Importo > 10000)`

5 File

5.1 File

Sono strutture dati persistenti e solitamente memorizzate su dischi (si usano all'interno dei programmi); realizzano la persistenza dei dati (del contenuto delle variabili). Tramite i file, i dati possono sopravvivere al termine dell'esecuzione del programma (i file sono usati anche per memorizzare i programmi!) Quando se ne chiede l'esecuzione, il sistema operativo copia il programma (eseguibile, conservato in un file) in memoria centrale e inizia a eseguirlo. I file sono strutture di dati sequenziali (si leggono e si scrivono gli elementi al loro interno in sequenza) e possono essere binari o di testo: binari significa che sono una sequenza di byte senza alcuna interpretazioni; mentre di testo significa che sono una sequenza interpretata. Le periferiche sono tutte viste come file (mouse, tastiera...). Ogni file aperto da un prog. ha un descrittore, risiede nella tabella dei file aperti, una delle strutture dati che il S.O. associa ai programmi in esecuzione; Il descrittore memorizza: la modalità d'uso (read, write), la posizione corrente all'interno del file, l'indicatore di eventuale errore, l'indicatore di eof(end-of-file). L'apertura del file restituisce un descrittore, per la precisione, un puntatore a un descrittore. Come si dichiara e si apre un file: puntatore al descrittore: `FILE *fp;` (`FILE` è un nuovo tipo). Apertura del file: `FILE *fopen (char *NomeFile, char *modalità);` (crea nello heap la struttura dati che parla con il sistema operativo e crea un canale di comunicazione con il file). NomeFile da il percorso e apre o crea il file, mentre la modalità ci dice la modalità di apertura: `\r"`, lettura in modalità testo, posizionamento inizio file (read); `\w"`, scrittura modalità testo (write), posizionamento inizio file; `\a"`, scrittura modalità testo (append) posizionamento fine file. La scrittura di file è permessa solo su file aperti in `\w"` o `\a"`, `\r"` solo lettura. Modalità binarie: `\rb"` o `\wb"`, `\ab"`. Non è possibile aprire più volte lo stesso file nella stessa modalità. Modalità di accesso (anche insieme): `\r+"` apre un file in lettura/scrittura in modalità testo posizionandosi all'inizio; `\w+"` scrittura/lettura modalità testo posizionamento inizio; `\a+"` scrittura/lettura modalità testo posizionamento fine. Per accorgersi dell'errore di apertura di un file si può fare `fp=fopen(...); if(fp==NULL) { printf ("errore"); exit(1); }` Per scrivere nei file si usa la `fprintf`, identica alla `printf`, ma con un parametro in più: `FILE *`: `fprintf (FILE *fp, "stringa",...);` Esempio: `fprintf (fp, "Nome: %s \n Età: %d", Nome, Età);` (scrive su fp). E' presente la funzione `fputc`, prototipo: `int fputc (char, FILE *fp);`, restituisce un intero che sarà uguale al `char` scritto se scrittura avviene correttamente, mentre `EOF` in caso contrario; si scrive `int fputc ('a',`

fp); (corrisponde a putc; putchar è equivalente di fputc, stampa stdout). Ci sono le fputs che scrivono stringhe, prototipo: int fputs (char *, FILE *);, restituisce un intero minore di 0 se non scrive correttamente, scrittura su file se non c'è nessun errore; int fputs ("ciao a tutti\n", fp);, corrispondente puts che stampa su stdout. La scrittura di file binari usa la fwrite, per qualsiasi tipo: size_t fwrite (const void *, size_t size, size_t count, FILE *fp); (void è puntatore generico) (scrive in binario, size è il numero di byte di un elemento, count è numero di elementi da scrivere, prototipo fwrite (Buffer, sizeof (int), 1, fp)). Chiudere un file: fclose (FILE *fp);, importante per il S.O. (salva le scritture che il S.O. ha tenuto in buffering) (fclose elimina la struttura dati nello heap). Per leggere un file di testo uso la fscanf, uguale alla scanf, la sintassi è la stessa ma con un puntatore FILE: fscanf (FILE *fp, "stringa" ,...); (uguale alla scanf): fscanf (fp, "%d %c %f", &a, &b, &c);. Può anche leggere da file binari tramite la fread: size_t fread (void *, size_t size, size_t count, FILE *fp);. Per leggere il carattere del file di testo uso la fgetc, sintassi: int fgetc (FILE *fp); (rest. il carattere letto o EOF); si scrive c=fgetc (fp); (equivalente di getc). Per leggere le stringhe si usa fgets, sintassi: char *fgets (char *, int n, FILE *); (n è max n-1 caratteri, fgets legge e termina col primo \n); fgets (Buffer, 80, fp);. E' possibile verificare se la fscanf ha letto correttamente i caratteri o se ha letto un EOF: if (fscanf (...)!=EOF) {...}. Si può accorgere del EOF attraverso la funzione feof, prototipo: int feof (FILE *); (feof restituisce 1 se fp ha letto il EOF, 0 altrimenti). Si può accorgere di eventuali errori nella lettura con la ferror, prototipo: int ferror (FILE *); (ferror restituisce 1 se c'è un errore durante la lettura/scrittura di fp, 0 altrimenti). La funzione fseek permette di spostare il punto corrente di lettura/scrittura di fp (fseek è come un indicatore): Prototipo: int fseek (FILE *fp, long int offset, int whence); (fp è il file, offset è la posizione dalla quale partire e whence è la posizione da cui partire), (whence può essere SEEK_SET, SEEK_CUR, SEEK_END). Restituisce 0 se correttamente eseguita, diverso da 0 altrimenti. L'indicatore di posizione corrente può essere letto tramite ftell, prototipo: long int ftell (FILE *);, restituisce la distanza in byte dal whence.