Appunti pre-esame orale da ricordare

Capitolo V - Funzioni

Quando passiamo un **argomento ad una funzione questo viene sempre passato per valore**, quindi se il valore dell'elemento viene cambiato dentro la funzione, il **cambiamento non si ripercuote fuori dalla funzione**, a meno che l'elemento passato non sia un **array o un puntatore** alla variabile nel main, **queste 2 cose vengono passate per riferimento** perché ne sto passando l'indirizzo e non una copia locale che vale solo per la funzione.
Quindi:
Se devo passare per valore → `funzione(variabile);`
Se devo passare per riferimento → `funzione(&variabile);` (dentro la funzione poi devo usare * per modificare il valore della variabile)

Capitolo VI - Array

Un **array è una struttura di dati contigua** in memoria **tutte dello stesso tipo**. Un **array non è altro che un puntatore al primo elemento di se stesso**, **il tipo di un array dipende dal tipo di valore che devo mettere dentro** (int, float, char, struct ecc...)
Sugli array si possono fare degli ordinamenti in base ad un criterio specifico, i più famosi (anche se poco efficienti ma semplici) sono il **bubble sort il selection sort e l'insertion sort**

BUBBLE SORT

```
for(int i = 0; i < dim-1; i++){

  for(int j = 0; j < dim-1; j++){
if(a[j] > a[j+1]){


int temp = a[j];
a[j] = a[j+1];
a[j+1] = temp;
}
}
}
```

Ordina l'array in maniera crescente (appunti più dettagliati → Bubble Sort Array)

SELECTION SORT

```
for(int i = 0; i < dim-1; i++){
int minimo = i;

  for(int j = i+1; j < dim; j++){

    if(a[j] < a[minimo]){
minimo = j;
}
}
int temp = a[i];
a[i] = a[minimo];
a[minimo] = temp;
}
```

Ordina l'array in maniera crescente (appunti più dettagliati → Selection Sort Array)

INSERTION SORT

```
for (int i = 1; i < n; i++) {
int elemento = arr[i];
int j = i - 1;


// Sposta gli elementi maggiori di "elemento" a destra
while (j >= 0 && arr[j] > elemento) {
arr[j + 1] = arr[j];
j--;
}


arr[j + 1] = elemento;  // Inserisce l'elemento nella posizione corretta
}
```

Ordina l'array confrontando l'elemento temporaneo nel primo for con l'elemento in posizione j che sarà l'elemento precedente

---

Sugli array si possono anche **fare delle ricerche di determinati elementi**, le più famose sono: la **ricerca binaria e ricerca tramite key**

RICERCA BINARIA

```
int ricerca_binaria_ricorsiva(int a[],int key, int low,int high){
```

```
  if(low <= high){
if(a[(low+high)/2] == key){
return (low+high)/2;
}else if(a[(low+high)/2] < key){
return ricerca_binaria_ricorsiva(a,key,(low+high)/2 + 1,high);
}else{
return ricerca_binaria_ricorsiva(a,key,low,(low+high)/2 - 1);
}
}
return -1;
}
```

Ricerca un determinato elemento e restituisce la posizione alla funzione chiamante (Questa è una verisone ricorsiva) (appunti più dettagliati → Ricerca Binaria Array)

RICERCA TRAMITE CHIAVE

```
int linear_search(int a[],int key, size_t n){
for(int i = 0; i < n; i++){
if( a[i] == key){
return i;
}
}
return -1;
}
```

Ricerca un determinato elemento e restituisce la posizione alla funzione chiamante (appunti più dettagliati → Ricerca Tramite Key Array)

Per passare l'array ad una funzione non serve usare un puntatore o altro basta passarlo normalmente come fosse una variabile normale, verrà in automatico passato per riferimento (int array[10] = {0}; funzione(array,10);) solitamente si passa l'array e la sua dimensione.
Se invece dobbiamo passare un elemento allora dobbiamo usare un puntatore, perché l'elemento viene passato per valore (int array[10] = {0}; funzione(&array[5]);)
(possiamo vedere un array come una sorta di puntatore e il singolo elemento come una variabile normale)

Possiamo anche creare array multidimensionali che sarebbero delle matrici o più semplicemente sono array che contengono altri array al loro interno
es. `int array[3][2] = {{1,2},{3,4},{5,6}};` → un array che contiene altri 3 array dentro con ognuno 2 elementi (array o matrice 2D), possiamo fare tutte le dimensioni che vogliamo.

Capitolo VII - Puntatori

I puntatori sono variabili i cui valori sono indirizzi di memoria. Quindi il puntatore contiene l'indirizzo di una variabile che contiene un valore. Il puntatore va dichiarato così: `tipo *nomePuntatore` → `int *ptr` e per assegnare un valore bisogna usare la "&" proprio perché abbiamo bisogno dell'indirizzo della variabile a cui farlo puntare (si dice che un puntatore "punta" alla variabile che gli è stata assegnata)

L' " * " si dice operatore di dereferenziazione e restituisce il valore che sta venendo puntato da un puntatore
`printf("%d", *yPtr);` → stamperà il valore della variabile a cui punta (es. 5)

Esiste l'**aritmetica dei puntatori, ovvero posso effettuare delle operazioni su di loro**, infatti **posso aggiungere un unità e farli spostare nella memoria puntata** (es. se il puntatore punta alla locazione AAA0000FFF e aggiungo 4 byte adesso il puntatore punterà a: AAA0004FFF, quindi mi sono spostato di 4 byte nella memoria che sta puntando).
Utile ad esempio con un array, infatti un array è un puntatore che punta al primo elemento dell'array, quindi se aggiungo dei byte (4 nel caso degli interi perché ogni int occupa 4 byte in memoria) mi posso spostare al prossimo elemento (posso anche sottrarli)

es. `aptr = aptr +1;` cosi incremento di 4 byte (4 per int, 8 per double ecc...)

Con i puntatori posso passare per riferimento delle variabili ad una funzione passandone l'indirizzo e lavorando (dentro la funzione) con un puntatore che punta all'indirizzo della variabile passata
Quindi : Quando l'indirizzo di una variabile viene passato a una funzione (&), si può usare nella funzione l'operatore di indirezione (*) per modificare il valore in quella locazione nella memoria della funzione chiamante.

Capitolo VII+ - Allocazione Dinamica

Se non sappiamo quanto dovrà essere lungo un array ad esempio, oppure vogliamo che finito il suo lavoro venga tolto dalla memoria per risparmiare spazio dobbiamo **allocarlo dinamicamente attraverso l'uso di malloc calloc o realloc e infine l'uso della free**, normalmente nella dichiarazione statica o automatica le variabili vengono caricate nello stack (un area di memoria che viene gestita dal sistema operativo) mentre quando allochiamo dinamicamente le variabili vengono poste nello heap (un area di memoria gestita dal programmatore) che quindi va poi opportunamente liberata.
Malloc → `malloc(sizeof(tipo) * dimByte)` → `int *y = malloc(sizeof(int) * 14);`
sintassi malloc Esempio
Malloc alloca memoria in base alla dimensione del tipo e la dimensione fornita dal programmatore

Calloc → `calloc(dimByte * sizeof(int))` → `int *x = calloc(14,sizeof(int));`
sintassi calloc Esempio
Calloc alloca memoria in base alla dimensione fornita dal programmatore e alla dimensione del tipo, in più inizializza a 0 tutta la memoria allocata

Realloc → `realloc(*ptr, dimByte)` → `x = realloc(x,20 * sizeof(int));`
sintassi realloc Esempio
Realloc ridimensiona una memoria già allocata precedentemente

Free → `free(*ptr)` → `free(x);`
sintassi free Esempio
Free libera la memoria allocata precedentemente

Capitolo X - Strutture

Una struttura è un insieme di dati (memorizzati in maniera NON contigua in memoria) **che possono avere tipo diverso**, le variabili al suo interno si chiamano membri della struttura e per fare riferimento alla struttura possiamo dichiarare della variabili del tipo della struttura stessa

Per accedere ai membri della struttura esistono 2 operatori: 1) . 2) →
chiamati **operatore punto e operatore freccia**.

l'**operatore punto** può accedere ad un membro utilizzando una variabile di tipo struttura es. `printf("%d",dati.codice);` → dati è la variabile di tipo struttura e codice è un membro di quella struttura.

l'**operatore freccia** accede ad un membro della struttura attraverso l'uso di un puntatore di tipo struttura

es. `printf("%d",datiPtr->codice);` → datiPtr è il puntatore di tipo struttura e codice è un membro di quella struttura.

La parola chiave **typedef** fornisce un meccanismo per creare sinonimi (o alias) per tipi di dati precedentemente definiti.
`typedef struct nomeStruttura nomeAliasStruttura;`

Le **unioni** sono simili alle strutture, solo che queste condividono lo stesso spazio di memoria:

```
union Dati {
int intero;
float decimale;
char carattere;
};
union Dati valore;
valore.intero = 10; // Ora la memoria contiene un intero
valore.decimale = 3.14; // Ora la memoria contiene un float (e sovrascrive l'intero)
```

- Nelle **strutture**, ogni membro ha il proprio spazio di memoria.
- Nelle **unioni**, tutti i membri condividono lo stesso spazio, quindi modificare un membro altera il valore degli altri.

Capitolo XI - Elaborazione di file

Esistono 3 stream:

1. standard input (stdin)
2. standard output (stdout)
3. standard error (stderr)

Per creare un file abbiamo bisogno di un puntatore alla struttura FILE, ogni file aperto contemporaneamente deve avere il suo puntatore per fare riferimento a quel file.
Per aprire un file si ha bisogno della funzione fopen("nomeFile","lettera") dove appunto si inserisce il nome del file o il suo percorso e una lettera che indica l'operazione che si deve fare:

Modalità	Descrizione
`r`	Apre un file esistente per la lettura.
`w`	Crea un file per la scrittura. Se il file esiste già, *elimina* i contenuti correnti.
`a`	Apre o crea un file per scrivere alla fine del file – cioè, le operazioni di scrittura aggiungono dati al file.
`r+`	Apre un file esistente per l'aggiornamento (lettura e scrittura).
`w+`	Crea un file per l'aggiornamento. Se il file esiste già, *elimina* i contenuti correnti.
`a+`	Append: apre o crea un file per l'aggiornamento; tutta la scrittura è effettuata alla fine del file – cioè, le operazioni di scrittura aggiungono dati al file.
`rb`	Apre un file esistente per la lettura in forma binaria.
`wb`	Crea un file per la scrittura in forma binaria. Se il file esiste già, elimina i contenuti correnti.
`ab`	Append: apre o crea un file per la scrittura alla fine del file in forma binaria.
`rb+`	Apre un file esistente per l'aggiornamento (lettura e scrittura) in forma binaria.
`wb+`	Crea un file per l'aggiornamento in forma binaria. Se il file esiste già, elimina i contenuti correnti.
`ab+`	Append: apre o crea un file per l'aggiornamento in forma binaria; la scrittura è effettuata alla fine del file.

es. `filePtr = fopen("persone.txt","r")` → apre il file e filePtr è il riferimento, il file è persone.txt aperto in modalità lettura (r)

Ogni file finisce con un carattere di **end-of-file**, in un certo senso è come se fosse il fine stringa delle stringhe, anche se l'end-of-file non è un vero e proprio carattere ma un segnale che indica che il file è finito

Per spostarsi all'interno di un file si utilizza un cursore, esistono quindi diverse funzioni che permettono di spostarlo come rewind o fseek o anche in un certo senso fscanf.

fscanf è una funzione uguale allo scanf solo che si usa per prendere dati da un file, analogamente **fprintf** scrive dati su un file.

Capitolo XII - Strutture Di Dati

Per le liste collegate le code e le pile abbiamo prima bisogno di sapere cosa sono le strutture autoreferenziali: sono strutture con un membro puntatore che puntano ad un'altra struttura dello stesso tipo.
Una lista collegata è una sequenza lineare di strutture autoreferenziali dove ogni membro puntatore è collegato ad un altra struttura.
Si accede a questa lista tramite un puntatore che identifica la testa della lista e a tutti gli altri nodi tramite il membro puntatore che li collega, una cosa molto utile delle liste è che può modificare la sua lunghezza durante il corso dell'esecuzione, aggiungendo o eliminando nodi.

Sulle liste possiamo fare operazioni di vario tipo, possiamo aggiungere/eliminare in coda o intesta o in mezzo, quindi si è più liberi, cosa che invece non vale per pile e code.

Lista collegata → inserimenti/eliminazioni in qualsiasi punto
Pila → LIFO → Last In First Out → cioè l'ultimo elemento inserito è il primo a essere rimosso (si utilizzano spesso funzioni di **push** per aggiungere un elemento in cima e **pop** per toglierlo)
Coda → FIFO → First In First Out → cioè il primo elemento inserito è il primo a essere rimosso