Appunti informatica

Appunti informatica

Concetti fondamentali

Definiamo come **complessità computazionale** di un problema la complessità dell'algoritmo più efficiente in grado di risolvere tale problema; non è necessario esibire un simile algoritmo, è sufficiente dimostrare che un tale algoritmo esiste e che tale è la sua omplessità computazionale.

Informazioni generali sul C

La versione attuale di C è quella chiamata **C17** pubblicato nel 2018. In generale la storia del C parte negli anni 70 con lo sviluppo di *UNIX* con il piccolo linguaggio chiamato **B**. Nel 1973 venne chiamato C e poi 1989 divenne **C89** e poi con tutti i suoi successori chiamati **C90, C95, C99, C11 e C** ma in realtà queste informazioni sono superflue. Naturalmente il C è un linguaggio a basso livello perchè è molto vicino al linguaggio macchina. Inoltre il C è un linguaggio compilato che, a differenza di un linguaggio interpretato ha la necessità di avere un compilatore che trasforma il codice sorgente in azioni eseguibili per la macchina.
La particolarità del linguaggio C (ma anche di altri), è che sono **compilati**, ovvero eseguiti da un compilatore che traduce il codice sorgente in linguaggio macchina per ottenere un risultato. Inoltre, è un linguaggio **imperativo** (sequenziale), dove il concetto di oggetto ed istanza non esistono e dove tutti i problemi possono essere risolti con funzione **do, for if** ecc. Caratteristica importante del linguaggio sono i **puntatori**, ovvero variabili che contengono l'indirizzo di memoria di un'altra variabile permettendo di accedere direttamente a una licazione di memoria.
Il linguaggio presenta diversi pregi come:

- **Efficienza** data dai puntatori e dalla gestione della memoria;
- **Tipizzazione** dove ogni variabile ha il suo tipo ;
- **Basso livello** permette di produrre un codice finale più compatto ed efficiente;
- **Portabilità** permette lo sviluppo e il successivo passaggio da una macchina ll'altra,
  Le debolezze invece sono:
- Alta probabilitàdi errore nei programmi;
- Programmi difficili da capire a causa della flessibilità del linguaggio
- Difficoltà nella modifica perchè non è sempre considerata la manutenzione
  Per lo sviluppo ci si basa su un compilatore, delle **librerie run-time** e almeno un file sorgente. Ogni file sorgente viene compilato e reso file oggetto che, grazie a dei set di librerie eseguiti in contemporanea, servono ad avere un unico file eseguibile. Il file eseguibile è dato dal **Linker**,

ovvero un processo che unisce tutti i file oggetto (otetnuto dalla compilazione dei files), codice delle librerie esterne e **start-up code** (codice necessario al sistema operativo).

C sintassi

PER LE FUZINZIONI SCORRI FINO ALLA FINE DEL PDF

Nella header o, le prime righe vengono introdotte le librerie che si andranno ad usare durante il nostro programma. Per libreria definiamo un insieme di funzioni già stabilite che possono essere utilizzate semplicemnte richiamandole.
Sotto la dichiarazione delle variabili possiamo anche usare il comando **#define** che permette di definire sia delle costanti con ***DEFINE CONST 5*** oppure puoi dichiarare delle pseudofunzioni con il define. Queste funzioni si chiamano **Espressioni condizionali**, un esempio è: **DEFINE Max(a,b)(a > b ? a : b)**. Questa fuznione permette trovare il massimo valore tra due numeri casuali. Nella header di un codice sorgente trovaimo tutte istruzioni con carattere iniziale **#**, questo eprchè sono informazioni sviluppate dal **pre-processore**. Questo pre-processore controlla le prime righe di codice e, se trova una libreria, sostituisce la riga `#include <stdio.h>` con tutto il codice della libreria presente nelle directory. Se invece andassimo a mettere sotto le librerie questa definizione `#define Cifre 9` assegneremo alla stringa *Cifre* il valore 9. Un altro modo di dichiarare costanti è tramite il tipo **const** che, se usato nel main o anche come variabile globale, crea una variabile **read only** che non può essere modificata dall'utente finale.

Nella scrittura in C per la letterura da tastiera dobbiamo utilizzare la scnaf per prendere dati in input e, nel caso di caratteri si usa il getchar per il singolo carattere. Ecco alcune istruzioni base:

- **scanf**("%tipo", &nome_variabile).
- per la **printf** invece è: printf("testo %tipo", nome_variabile)
- simile alla printf abbiamo il **puts** che ha la stessa sintassi della printf ma manda a capo la stringa successiva.

Le operazioni di input, output ed errore sono gestite dai **canali standard (standard streams)**, ovvero tutti quei collegamenti tra il programma e l'ambiente operativo in cui viene eseguito. I cannali sono 3, ovvero

- **standard input stdin**
- **standard output stdout**
- **standard error stderr**

Un po' di comandi

- Il comando `system("PAUSE")` permette di avere l'output con **cmd** e non tramite il terminale dell'ide che si sta usando.
- `goto etichetta` permette di saltare pezzi di codice e di andare a una specifica etichetta (label) senza che vi sia una condizione effettiva.

- `return valore` permette di dare ocme ritorno un valore in base alla dichiarazione del tipo della funzione, queto perchè una funzione senza un return non può avere fine. Per esempio il main ha sempre bisogno di un return 0 perchè nella dichiarazione è di tipo int, quindi deve restituire un intero alla fine. Altre funzione, come quelle di tipo void, non hanno bisongo di un return.

Printf

Nella **printf** possiamo usare delle costanti aggiuntive per dare un formato alla stringa finale come ad esempio:
`printf("valore di n = %4d", n)` darà come risultato un valore allineato a destra di almeno 4 cifre. La stessa cosa ma con un - ci darà l'allineamento a sisnistra `printf("valore di n = %-4d", n)`.
Se invece andassimo a mettere questa notazione `printf("valore di n = %4.2d", n)` otterremo un output di almeno 4 caratteri e, nel caso di un intero, tanti zeri quanti ne servono per arrivare a 4 cifre. Con un float invece il valore dopo il punto indica il numero di cifre dopo la virgola.

Tipologie di dati e variabili

Ogni tipo può avere dei vincoli aggiuntivi per la rappresentazione del dato. Questi sono: **Long, short, signed e unsigned**.
Tutti i tipi di dati sono:

Tipo	Descrizione	Grandezza
%d	intero	2 Byte
%i	intero	2 Byte
short int %i	intero corto	2 Byte
long int %i	intero lungo	4 Byte
unsigned int %i	intero senza segno	2 Byte
unsigned long int %i	intero lungo senza segno	4 Byte
%c	char	1 Byte per carattere
unsigned char %c	carattere senza segno	1 Byte
%s	stringa	Dipende dalla grandezza
%f	float	4 Byte
%f	double	8 Byte
%lf	long double	10 Byte
%g	float senza 0 alla fine e in notazione scientifica	2 Byte
%o	intero ottale	2 Byte
%x	intero esadecimale	2 Byte

Tipo	Descrizione	Grandezza
%p	puntatore	
%[^\n]s	per prendere in input una stringa con gli spazi	

Nota bene che in C ogni float viene gestito come un double, per questo l'istruzione `float n = 4.5f` è un rafforzativo per la variabile stessa.

Una funzione utile per vedere quanti bit occorrono per i tipi oppure per una specifica variabile è **sizeof()**, che restituisce la grandezza in bit.

Abbiamo una divisione dei tipi di dato utilizzabili in C. I più importatnti sono quelli **scalari**, perchè i valori che li compongono sono distributi su una scala lineare, su cui si può stabilire una relazione di ordine totale. Questi tipo sono i puntatori, i tipi enumerativi e quelli aritmetici. Per definizione la **variabile** è l'individuazione di una certa area di memoria mediante un nome simbolico. Ogni variabile viene inserita nella RAM. Ogni variabile porta con se 2 valori, L-value e l'R-value. Il primo permette di vedere il valore dell'indirizzo di memoria ed è immutaible. Il secondo è il valore assegnato alla variabile che può cambiare

In informatica, **word (parola)** è un termine che identifica la dimensione nativa dei dati usati da un computer. Una word è un gruppo di bit di una determinata dimensione che sono gestiti come unità da un
microprocessore. La dimensione (o lunghezza) della word è un'importante caratteristica dell'architettura di un computer. Le architetture dei PC sono delle word, e possono essere a 32 o 64 bit. Queste avranno gli stessi registri ma con grandezze diverse, così come la grandezza delel variabili sarà diversa.

Per informazioni sull'ASCII visita le dispense di architetture degli elaboratori

In C non esiste il tipo boleano (boolean), per questo un semplice controllo su un flag va fatto con if(x) dove si controlla il valore di x e, in caso ci fosse un numero diverso da 0 la condizione si verifaca, altrimenti restituisce un valore falso.

Quando parliamo di **variabili** dobbiamo tenere presente la zona di codice dove queste vengono inizializzate e usate. Queste variabili lavorano tramite delle regole che permettono a ogni variabile di lavorare su una determinata porzione di codice e si chiamano **scope**. Lo scope permette di poter inizializzare più variabili con lo stesso nome in parti di codice diverso. Nel caso di variabili locali possiamo creare più variaibili con stesso nome in funzioni diverse. Attenzione che non è possibile fare la stessa cosa con variaibili globali. Ogni variabile viene memorizzata sullo stack di memoria che permette alla CPU di lavorare in fase di running. Questo avviene solo con variabili editabili, ma se usassimo una varibile di tipo static, ovvero che non può cambiae di valore, questa verrà inserita in una *locazione di memoria permanente* per tutta al durata del programma.
Una variaibile globale può essere utilizzata anche su altri file se si aggiunge la parola **extern** `extern int Valore`.

Operatori

Molti sono gli operatori in C che possono migiliorare la lettura e scrittura del ocdice. I più comuni sono:

Tipo	Descrizione
+=	addizione del valore a sinistra con quello a destra
-=	sottrazione del valore a sinistra con quello a destra
*=	moltiplicazione del valore a sinistra con quello a destra
/=	divisione del valore a sinistra con quello a destra
++a	incremento di 1 prefisso e restituisce il valore
a++	restituisce e incrementa di 1 postfisso
--a	decremento di 1 prefisso e restituisce il valore
a--	restituiesce e decrementa di 1 postfisso
^	Xor logico
~	complemento a 1
>>	shift a destra del primo operando per un numero di bit pari al valore del secondo `x >> 3`
<<	shift a sinistra del primo operando per un numero di but paru al valore del secondo `x << 4`

Se gli incrementi o decrementi sono l''unica operazione di una riga di comando lavorano allo stesso modo, ovvero che il valore sarà sempre incrementato o decrementato.

Casting

Il **casting** permette di passare una variabile di un tipo a un altro, ma questo non vuol dire che avrà lo stesso valore per l'utente finale. La funzione di casting inoltre permette di svolgere operazioni in modo più flessibile anche se non si hanno certi tipi di dato. Inoltre, il cacsting segue una gerarchia, ovvero che se in un'operazione andiamo a castare degli interi in float, naturalmente tutti gli interi diventeranno float. La gerarchia è ***int < float < double < long double***. `float avg = nHrs/(float)nDays;` In questo caso tutto diventa float.

Sistemi di calcolo

La macchina di Turing universale (un modello) definisce l'insieme dei problemi calcolabili. La **Turing equivalneza** è la proprietà dei modelli di calcolo che hanno lo stesso potere computazionale di una macchina di Turing.

La programmazione si basa su strutture di controllo, le 3 fondamentali sono la *concatenazione, la selezione o condizione e l'iterazione*. Queste strutture permettono la scrittura facilitata del codice, ma non cambia la potenza della macchina di Turing.

I cicli

Abbiamo diversi modi di fare dei cicli iterativi in C, ma in generale nei linguaggi di programmazione. Il primo è il costrutto `while(espressione)` che controlla che l'espressione al suo interno sia vera o falsa. In caso fosse vera il while permette di far eseguire il suo blocco di codice sottostante, altrimenti in caso fosse falsa farebbe saltare quel blocco di codice.
Il `Do{codice}while(espressione)` a differenza del while permette l'esecuzione del codice almeno una volta e, avendo il while alla fine del blocco di codice, si controllerà che l'espressione sia vera o falsa. Come prima in caso fosse vera torna a eseguire tutto il blocco di codice sovrastante e controlla nuovamente nel while, altrimenti esce dal ciclo e continua fino a terminare.
Il ciclo for è invece una rappresentazione di un do-while compatta, dove sono presenti la dichiarazione di un contatore, l'espressione di controllo e un operazione sulla sul contatore. Questo permette al for di essere un ciclo *finito*, ovvero dove si saprà da subito il numero di iterazioni massime, a differenza di un while che può avere infinite iterazioni. La sintassi del for è `(contatore, condizione, incremento)` `quindi (for int i = 0; i < n; i++)` dove la dichiarazione interna di i non è necessaria se già dichiarata fuori dal for.
Il ciclo for viene eseguito in un modo specifico, ovvero per prima si legge l'espresisone del for, poi il blocco di codice all'interno del for e poi si incrementa controlla che sia ancora vera la condizione.
Esistono anche for che lavorano con due variabili contemporaneamente, ad esempio `for(int i = 0, j = 0; i + j < 100; i++, j++)`.

Nei cicli possiamo trovare 2 istruzioni facoltative, ovvero la **break e la continue**. La prima se messa in un qualsiasi ciclo fermerà il ciclo stesso, saltando tutte le istruzioni successive. Con il continue invece torneremo all'inizio del ciclo, saltando sempre tutte le informazioni sottostanti al continue.

Vettori

Un vettore è una variabile aggregata in grado di contenere un certo numero di dati **tutti dello stesso tipo**. La dichiarazione di un vettore di interi è `int vettore[100]` dove viene dichiarata assieme al nome simbolico anche la grandezza massima di *celle* del vettore. Un array può anche essere dichiarato così : `int br[5] = {1,2,3,4,5}`Questo però vuol dire che quando andremo a lavroare con il nostro vettore partiremo dalla cella 0 e arriveremo fino all cella N - 1 perchè lo 0 è compreso. Questi sono vettori di tipo monodimensionale, a differenza dei multidimensionali che sono delle matrici come ad esempio `int M[10][5]` dove avremo una matrice di 10 righe e 5 colonne.

Matrici

Con **Matrice** o **Array multidimensionale** definiamo un array che ha 2 dimensioni, ovvero le righe e le colonne. La definizione di una matrice è: **int Matrice[10][20]** dove 10 è il numero di righe e 20 il numero di colonne. Dal punto di vista della memoria invece, una matrice viene trattata come una sequenza ordinata, quindi un unico grande vettore con le celle una di fianco all'altra. Anche qui parliamo di **Matrici quadrate** quando il numero di righe è uguale a quello delle colonne.

Dal punto di vista del codice, l'assegnazione di un valore in una matrici viene fatta con due cicli for annidati, quindi:

```
int M[10][15];
printf("Inserire i valori della matrice\n");
for(int i = 0; i < 10; i++){    //sfoglia le righe della matrice
    for(int j = 0; j < 15; j++){        //sfoglia el colonne della amtrice
        scanf("%d", &M[i][j]);
    }
}
```

Questo permette di inserire i valori riga per riga, ma se invece volessi inserire i valori per colonna? Ecco l'esempio

```
int M[10][15];
printf("Inserire i valori della matrice\n");
for(int i = 0; i < 10; i++){    //sfoglia le righe della matrice
    for(int j = 0; j < 15; j++){        //sfoglia el colonne della amtrice
        scanf("%d", &M[j][i]);
    }
}
```

Basta invertire gli indice nell'assegnamento della matrice!

MATRICI DA SPIEGARE PAG 38

Le stringhe

Nel linguaggio C le stringhe non sono un vero e proprio tipo di dato, ma bensì vengono rappresentate da un **vettore di caratteri** con un carattere che ne identifica la fine, ovvero il **\0**. Una stringa viene inizializzata così: `char string[100]` e indica un vettore di 100 caratteri, ma solo 99 sono utilizzabili perchè l'ultimo è il carattere terminatore **\0**. Solo in un caso la dichiarazione del vettore di caratteri non ha bisogno della grandezza, ovvero `char nome[] = "ape"`, questo perchè la lunghezza verrà fissata una volta che il programma sarà compilato. La peculiarità delle stringhe è che in realtà sono dei puntatori a carattere, ovvero delel *variabili* che andranno a puntare la prima cella del nostro vettore di caratteri. Questo perchè quando noi andiamo a creare una stringa di caratteri, questa stringa viene registrata in memoria e ogni carattere avrà una sua cella di momoria nello stack. Sapendo che ogni carattere ha grandezza 1 byte una stringa di N lettere avrà grandezza in memoria di N byte. Il puntatore a stringa non fa altro che puntare all'indirizzo di memoria della stringa e quindi alla cella del primo carattere registrato. I puntatori in C si identificano dal carattere * e un esempio di inizializzazione è: `char *ptr1;`. La parte importante però rimane quella di capire la differenza principale tra un puntatore e una stringa. La prima punta alla cella di memoria e non può esistere che un puntatore non sia stato inizializzato. Proprio per questo, a meno che la stringa non sia stata inizializzata staticamente,

dobbiamo prima prendere in input tutti i caratteri e poi lavorare con i puntatori.
ARRIVATI PAGINA 28 LUCIDI 8

Teoria

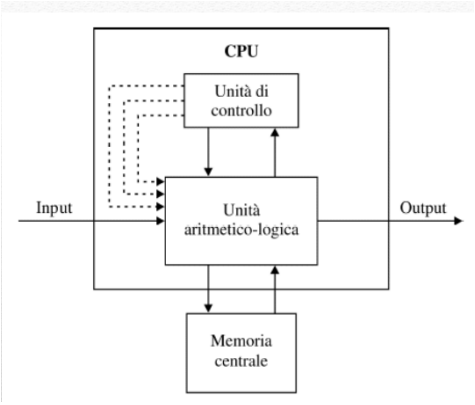
John von Neumann

John von Neumann, Americano di origini Ungheresi, è stata uno delle menti più brillanti e straordinarie del secolo scorso che ha contribuito significativamente allo sviluppo del calcolatore e del computer odierno. Fin da quando era un bambino possiamo già vedere delle caratteristiche più uniche che rare, come la memoria perfetta, la capacità aritmetico logiche impressionanti e il fatto che parlasse quattro lingue all'età di dieci anni. Il suo periodo di vita è fortemente segnato dall'arrivo dei nazisti e dalla seconda guerra mondiale. Proprio per evitare problemi si trasferisce in America dove continuerà i suoi studi e, grazie all'**IBM** completerà il progetto lasciato da Turing, ovvero una **macchina universale**, quindi programmabile.
Neumann riesce a trasformare l'**ENIAC** (grande mainframe che era d'aiuto per calcoli molto complessi) nell'**EDVAC** grazie all'aggiunta di un software che la rendesse programmabile. Questa è l'**architettura di Von Neumann** che permette a un calcolatore di essere programmato via software e non tramite la manipolazioni diretta delle componenti hardware. Successivamente Neumann è coinvolto nello sviluppo della bomba atomica ed è proprio lui che ha progettato la bomba su Hiroshima e Nagasaki introducendo la detonazione in aria, che permette alle onde di essere più distruttive. Ha preso parte anche nello sviluppo del missile Atlas che mandò nello spazio l'essere umano nel 1962, ma lui non vide quessto progetto fino alla fine perchè morì nel 1957

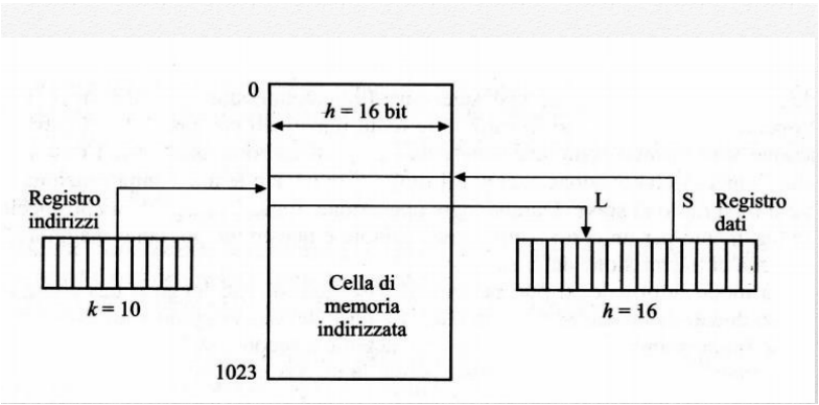
Alan Turing

Alan Turing (Londra, 1912 – Wilmslow, 1954) è stato un matematico, logico e crittografo britannico, considerato uno dei padri dell'informatica e uno dei più grandi matematici del XX secolo. Introduce il concetto di **macchina universale di Turing**, ovvero una macchina in grado di poter fare le stesse operazioni di tutte le macchine presenti e future e dimostrare che a livello matematico hanno dei limiti. Molto importante è stato il suo lavoro sulla crittografia durante la Seconda Guerra Mondiale dove ha lavorato con gli inglesi per decodificare i messaggi dei tedeschi. Per fare questo inventò **Colossus**, il primo vero calcolatore che permetteva di decodificare i caratteri dei tedeschi e intercettare le conversazioni. È stato anche il primo a parlare di **Intelligenza Artificiale** e infatti con il **Test di Turing** (Articolo da lui scritto) possiamo vedere che immagina una macchina senziente in grado di cammuffarsi con gli esseri umani. Purtroppo poi Turing verrà incarcerato per essere omosessuale ma, gli fu concessa al possibilità di salvarsi tramite un trattamento ormonale che l'avrebbe reso impotente. Questo trattamente lo distrusse sia fisicamente che mentalmente e per questo arriva al suicidio con del cianuro.

Architettura di Neumann oggi



In quseta imamgine possiamo vedere una rappresetnazione semplificata dell'architettura di Von Neumann e quindi il funzionamento della CPU di ogni calcolatore. La parte principale è l'**unità aritmentico logica** che permette di eseguire operazioni e di restituire un risultato. Questo è coordinato dall'**unità di controllo** che, come dice il nome, controlla che le operazioni eseguite dall'ALU siano giuste sulla base di ciò che c'è nella memoria centrale. Infatti, nella memoria centrale troviamo sia le istruzioni che devono essere eseguite sia il programma finale. L'unità di controlla vede anche se l'indicazione della cella di memoria nel registro delle istruzioni è giusta per una specfica operazione.
Affinchè la lettura e scrittura possa avvenire abbiamo bisogno di due registri e una cella di memoria. Il registro degli indirizzo invia l'indirizzo alla memoria e preleva il dato all'indirizzo nel registro dati, esegue l'operazione e inserisce il dato finale nel registro dati.



La memoria è gestita nel nostro calcolatore in modo gerarchico e quindi non tutte le memorie sono uguali per eseguire dei programmi. Più le memorie sono vicine alla CPU e più veloce sarà il passaggio dei dati. Infatti tutte le memorie vicine al processore sono volatili, quindi vengono *pulite* al riavvio della macchina. Il disco rigido (HDD) è una memoria permanente e quindi molto grande ma allo stesso tempo più lenta di una volatile. In questa gerarchia bisogna ricordare che nell'accedere all'indirizzo di una variabile, molto probabilmente le richieste successive si troveranno in celle molto vicine alla prima variabile.

LUCIDI 8 PAGINA 39

Le funzioni

Con **funzione** deifniamo l'unità elaborativa fondamentale di ogni programma in C (ma anche di altri linguaggi). Ogni funzione può essere vista come un blocco di codice isolato dal main che viene eseguito solo se viene chiamato nel main. Ogni funzione ha un particolare compito e molto spesso

servono per rendere generico un problema. Questa caratteristica è alla base della figura del programmatore, ovvero risolvere problemi che accomunano più entità e situazioni. Più in generale le funzioni permettono di dividere il main da tutte quelle operazioni che andrebbero a *sporcare* la lettura del codice.
Diversi sono i concetti fondamentali su cui si basano le funzione, come quella delle variabili **locali**, ovvero quando una variabile viene definita in una funzione e non ha valore al di fuori della funzione dove è stata inizializzata.
La funzione è fatta dal **tipo di ritorno**, un nome simbolico e dentro le parentesi troveremo i **parametri formali** necessari alla funzione per essere richiamta. **NOTA BENE** che possono esistere funzioni con nome uguali ma con numero o tipo di parametri necessari diversi. Un tipo di funzione è `int MAX (int a, int b)` .
Possiamo trovare anche funzioni dette **ricorsive**, ovvero che emulano un loop richiamando loro stesse. Queste funzioni oltre a essere di un certo spessore logico sono anche le più pericolose perchè necessitano **sempre** di una condizione di uscita altrimenti si arriva al loop infinito. Il concetto che sta dietro la ricorsione è quello che ogni funzione chiamata mette in stand-by quella chiamate, facendola aspettare fino alla fine del suo ciclo. Ecco un esempio

```
int fact(int);  //prototipo di funzione

main()  //main
{
int y = fact(3);
}

int fact(int n) //funzione
{
return (n<=1) ? 1 : n*fact(n-1);        //ricorsione
}
```

In questo esempio possiamo vedere la funzione fact che deve sviluppare il fattoriale di un numero preso in nput nel main. Quindi, seguendo la logica di prima posso affermare con cetezza che la funzione avrà una specie di effetto matrioska dove verranno create tante funzione fact fino ad arrivare a n - 1. Una volta che la condizione sarà falsa le funzioni torneranno indietro portante con loro un valore che verrà restituito a *y* come valore finale. Inoltre in questo esempio si può vedere un esempio di **prototipazione**, ovvero di dichiarazione di funzione messa in cima al main. I prototipi non sono strettamente necessari e vengono utilizzati per migliorare la lettura del codice e anche per avere un ordine logico e mentale migliore. Quando si dichiara uno o più prototipi in testa al main bisogno che alla coda del main vengono messe tante funzioni quanti sono i prototipi.

I Puntatori

Con **puntatore** definiamo una variabile contenete il nome dell'area di memoria dove è stata dichiarata la nostra variabile associata. Grazie ai puntatori possiamo conoscere gli indirzzi ma non ci possiamo

influire sopra. Per dichiarare un puntatore abbiamo bisogno del carattere * per la dichiarazione.
Esempio

```
int *iPtr;
int vettore[5]={10,20,30,40,50};
printf("indirizzo di vettore: %p\n", vettore); //stampiamo l'indirizzo del vettore con %p
iPtr = vettore; //assegnamo alla variabile puntatore l'indirizzo del vettore
printf("indirizzo di vettore : %X\n",iPtr); //stampiamo l'indirizzo con %X, ovvero in esadecimale
printf("primo elemento di vettore : %d\n",*vettore);
printf("secondo elemento di vettore : %d\n",*(vettore +1)); //lavoriamo sul puntatore del vettore e, aggiungendo 1 andremo a prendere il valore della cella successiva
++iPtr;
printf("secondo elemento di vettore : %d\n",*iPtr);
//Come output avremo
indirizzo di vettore: 0000003833fff760
indirizzo di vettore : 33FFF760
primo elemento di vettore : 10
secondo elemento di vettore : 20
secondo elemento di vettore : 20
```

Molte sono le caratteristiche principali dei puntatori che li rendono al contempo utili e confusionari da utilizzare. Definiamo tutte le procedure per usare nel modo giusto un puntatore di tipizzato.

```
int *ptr, n = 5;
*ptr = &n;
```

Molto importante è il momento dell'allocazione della variabile al puntatore, questo perchè un puntatore può puntare solo a una variabile statica.
Con la definizione nell'esempio ho appena creato un puntatore di tipo intero e una variaibile n uguale a 5. Poi ho assegnato al puntatore l'indirizzo di memoria di n; questo vuol dire che se io andassi a modificare il valore di ptr o di n in questo momento entrambi cambierebbero valore, avendo in comune l'indirizzo di memoria. Adesso possiamo lavorare con ptr preceduto da asterisco se vogliamo modificare il contenuto del puntatore, altrimenti possiamo vedere la cella di memoria ospitante facendo uscire in output ptr con operatore **%p**.
MOLTO IMPORTANTE è la distinzione tra l'operatore di referenza (&) e quello di deferenza (*). Il primo permette di indicare la cella di memoria, il secondo indica il valore contenuto nella cella di memoria.
Per rilasciar un puntatore da una variabile basta fare `ptr = null`, così da non far putnare il puntatore a nessuna variabile.

Con i puntatori possiamo lavorare con qualsiasi tipo di variabile (questo perchè un puntatore ha grandezza statica dettata dall'architettura del nostro PC. Se ad esempio abbiamo un'architettura a 32

bit, tutte le celle di memoria e puntatori saranno a 32 bit) e infatti vengono ampiamente utilizzati nel caso di stringhe e vettori. Infatti abbiamo definito una stringa come puntatore a carattere, il che vuol dire che data una stringa il puntatore punterà sempre al primo carattere della stringa, seguendo tutte le celle di memoria successive. Questo perchè una stringa verrà allocata in memoria in modo contiguo, e essendo un char grande 1 byte l'indirizzo di memoria cambierà di 1 valore. Inoltre gli indirizzi di memoria sono scritti in esadecimale.

L'unico modo per allocare dinamicamente la memoria a un puntatore è usare la funziona **malloc()**, ecco un esempio

```
int *p;
p = (int *) malloc(sizeof(int));
```

Possiamo utilizzare i puntatori per puntare a vettori, quindi essendo il vettore un insieme di valori omogenei allocate in indirizzi di memoria contigua, il puntatore permette di lavorare direttamente sul vettore andando a puntare il primo indirizzo, quindi la cella di indirizzo 0. Ecco un esempio di puntatore a un vettore

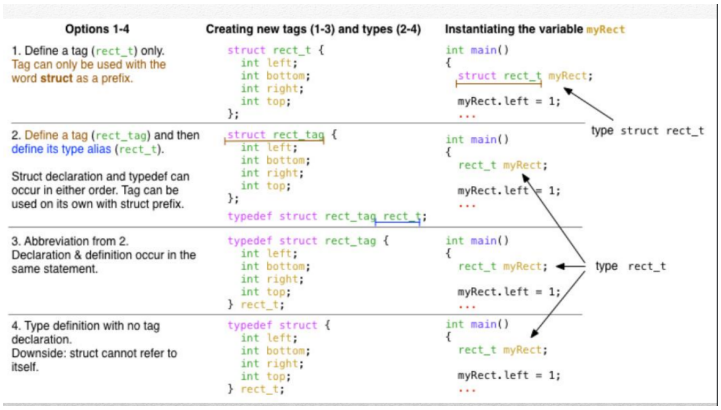
```
int a[10];
int *pa;
pa=&a[0];
for(int i = 0; i < 10; i++){
    printf("Valore in posizione %d: %d", i, *(pa + i))
}
```

Per sfogliare il puntatore basta addizionare o sottrarre l'indice al puntatore, così facendo andremo a scorrere le celle di memoria.

ARRIVATO A PAGINA 43

Le struct

Con **struct** definiamo un tipo di dato strutturato, formato da un insieme di dati eterogenei. Questo, a differenza dei vettori, permette di avere tanti tipi di variabili. Le strutture sono costituite da campi che sono identificati ad nomi e possono contenere informazioni di diverso tipo. Ci sono diversi modi di dichiarare le struct, ecco alcuni:



Inoltre, le struct possono essere annidate, quindi dichiarate all'interno di altre struct

```
typedef struct
{
    char p_name[20], p_fiscode[16];
    struct
    { short p_day;
        short p_month;
        short p_year;
    } p_birth_date;
} PERSONA;
```

Allo stesso modo possiamo concatenare più struct tra di loro nei campi, come ad esempio

```
typedef struct{
    int giorno;
    int mese;
    int anno;
} data;
typedef struct{
    char nome[20];
    char cognome[40];
    data data_nasc;
} persona;

persona p;
p.data_nasc.giorno=25;
p.data_nasc.mese=3;
p.data_nasc.anno=1992;
p.nome="Giulio";
```

Come sempre, i puntatori possono puntare alle struct, rendendo il tutto leggermente diverso dal normale. Infatti, quando utilizziamo un puntatore a struttura il carattere speciale **punto(.)** viene sostituito da una freccia **->**. Per esempio

```
struct info{

    char nome[30];
    char secNome[30];
    char cognome[30];
};

int main(){
```

```
 struct info in, *pinfo;

 pinfo = &in;

 printf("Nome = %s", pinfo->nome);
}
```

Le liste

Con liste definiamo una struttura dati dinamica, che quindi non ha una grandezza predefinita. Queste strutture possono essere create tramite le struct e il loro concetto si basa su un puntatore al loro interno che può avere valore uguale a NULL oppure può puntare a un'altra struct.

```
struct nodolista {
    int id;
    struct nodolista *next;
};
```

Affinchè questa struttura possa essere utilizzata nel modo migliore abbiamo bisogno di alcune variabili di appoggio del tipo della struct. Tramite la funzione malloc (memory allocation) andiamo ad allocare la memoria per una singola istanza della nostra struct e, definendo un puntatore temporanea in una funzione possiamo lavorarci sopra e poi assegnare il puntatore alla nostra testa. Diversi sono i modi di aggiungere degli **item** alla nostra lista, il più comodo e utilizzato è quello in testa, ovvero facendo diventare la nuova istanza l'elemento in cima alla lista. Esistono anche l'inserimento in coda (la coda si riconosce perchè il puntatore non punta a una struttura ma a NULL) e quello in una specifica posizione. Andare a fare l'inserimento in posizione richiede molto lavoro perchè dal punto di vista logico dobbiamo sfogliare tutta la lista e poi una volta trovato l'elemento bisogna eliminare il collegamento con il successivo e unirlo alla nuova cella.

Per elimianre una cella è necessario capire la logica dietro la rimozione di una cella da una lista. Andando ad eliminare una cella in realtà eliminiamo i link a quella cella e colleghiamo la cella precedente con al cella successiva. Per fare questo dobbiamo utilizzare almeno due variabili di appoggio in una funzione per rendere il tutto più facile. Una variabile porterà il puntatore della cella attuale e l'altra avrà il puntatore della cella precedente, questo perchè abbiamo la necessità di circondare la cella da eliminare e, allo stesso tempo, di avere tutti i link necessari ad eliminarla. Una volta trovata portiamo il puntatore a precedente due celle avanti, facendolo puntare al successivo del puntatore ad attuale e andiamo a fare la **free()** del valore del puntatore ad attuale.

Appunti di laboratorio

Funzioni

Qui elenco alcune delle funzioni di libreria più utilizzate negli esercizi

---

Nome	Libreria	Utilizzo
sizeof()	stdlib.h	Funzione che restituisce il numero di bit necessari richiesti dal tipo di variabile inserito al suo interno
pow(val1, val2)	math.h	Permette di fare l'elevamento a potenza dove val1 è la base e val2 è l'esponente
isalpha(char c)	ctype.h	Controlla che il valore appartenga all'alfabeto e, in caso fosse un numero restituisce un numero diverso da zero, altrimenti restituisce 0 con una lettera
tolower(char c)	ctype.h	Trasforma la stringa o carattere dato in input in una stringa equivalente ma con i caratteri minuscoli
toupper(char c)	ctype.h	Trasforma la stringa o carattere dato in input in una stringa equivalente ma con i caratteri maiuscoli
strlen(char c)	string.h	Funzione che restituisce la lunghezza della stringa che ha come argomento
strcmp(char stringa1, char stringa2)	string.h	Confronta le due stringhe carattere per carattere e restituisce 0 se le stringhe sono uguali, un valore minore di 0 se stringa1 è minore di stringa2 e un valore maggiore di 0 se stringa1 è maggiore di stringa2
isspace(char c)	ctype.h	Controlla che il carattere inserito sia un carattere di spazio, quindi ' ', '\n', '\t' e ha come ritorno 1 se il carattere è uno spazio, altrimenti 0
srand(time(NULL))	time.h	Permette di iniziare a contare dei numeri per la generazione pseudo-randomica
rand()	stdlib.h	Permette di restituire il numero random generato con la srand. Possiamo definire anche degli intervalli di numeri facendo `rand() % (b - a) + a`. Oppure semplicemnte se volessi i numeri fino a 10 dovrei fare `rand() % 11`
malloc(sizeof())	stdlib.h	Funzione che permette di allocare dello spazio in memoria ben definito. Si utilizza la funzione sizeof per trovare il numero di byte necessari, molto utilizzata per la dichiarazione di liste
realloc(sizeof())	stdlib.h	Funzione che permette di cambiare la quantità di memoria allocata per una variabile, nello specifico per aumentare il numero di celle di un vettore di liste
free()	stdlib.h	Funzione che permette di liberare la memoria dei puntatori al suo interno, utilizzata per liberare la memoria alla fine di un programma o per eliminare una cella di una lista.