Introduzione

martedì 19 settembre 2017 16.25

INFORMATICA: Scienza della rappresentazione e dell'elaborazione dell'informazione

ARCHITETTURA DI UN ELABORATORE ispirata al modello di Von Neumann

- CPU central process unit
- RAM random access memory - volatile
- ROM read only memory - persistente
- I/O

TECNOLOGIA DIGITALE:

- tensione alta (Vh) => 1
- tensione bassa (Vl)=> 0

SOFTWARE: parte intangibile, programmi che vengono eseguiti dal sistema: Si distinguono

- Sotware di base (OS e software di comunicazione)
  - **FIRMWARE**
    È uno strato di micro-programmi scritti dai costruttori che agiscono direttamente al di sopra dello strato HW
  - **SISTEMA OPERATIVO**
    È lo strato di programmi sopra HW e firmware e che gestisce l'HW, costituisce una macchina virtuale alla quale l'utente può interfacciarsi
    N.B. le operazioni sono gestite in maniera differente dai diversi sistemi operativi pertanto diversi OS realizzano diverse macchine virtuali sullo stesso elaboratore fisisco
- Software applicativo
  - IDE
    Consentono la scrittura, la verifica e l'esecuzione di nuovi programmi.
  - PROGRAMMI APPLICATIVI
    Risolvono problemi specifici degli utente (es. word Processor, fogli elettronici, DBMS)

Elaboratore e Macchina di Turing

martedì 19 settembre 2017 16.48

ELABORATORE ELETTRONICO: strumento in grado di eseguire azioni su dati per produrre risultati. Le istruzioni per far eseguire azioni all'elaboratore devono essere scritte in un linguaggio.

TESI DI CHURCH-TURING: Non esiste alcun formalismo capace di risolvere una classe di problemi più ampia della Macchina di Turing.
Dunque se neanche la macchina di Turing riesce a risolvere quel problema, esso non è risolubile.
Da qui la definizione della funzione caratteristica del problema P è
Fp: A -> B ,la quale se $x \in A$, $F(x) \in B$ è detta computabile.
Un **problema** è **RISOLUBILE** se la **funzione** è **COMPUTABILE**.

La MACCHINA DI TURING **(TM**) si compone di:

- Nastro (memoria)
- Testina (per leggere / scrivere dalle celle di memoria)
- Dispositivo di controllo (per regolare la testina)

Il suo comportamento dipende:

- Simbolo letto sul nastro mediante la testina
- Stato in cui si trova il dispositivo di controllo

NB: La macchina di Turing è generale ma, una volta definita la parte di controllo, essa diventa specifica.

Nel caso in cui però il dispositivo di controllo vogliamo che possa variare, abbiamo bisogno di poter descrivere l'algoritmo richiesto in un dato linguaggio direttamente in memoria e di una macchina che lo sappia interpretare.
La MACCHINA DI TURING UNIVERSALE (**UTM**) è l'interprete del linguaggio scelto ed esegue le operazioni di:

- Fetch andare a cercare le istruzioni da compiere
- Decode interpretare le istruzioni
- Execute eseguire l'algoritmo

Algoritmi e Programmi

giovedì 21 settembre 2017 15.54

ALGORITMO: Una sequenza finita di mosse che risolve in un tempo finito una classe di problemi.
Un algoritmo deve rispettare le caratteristiche di:

- Eseguibilità
- Non-ambiguità
- Finitezza

Due **algoritmi** si dicono **equivalenti** quando hanno gli stessi domini di ingresso e di uscita e, con gli stessi valori di ingresso producono gli stessi valori in uscita. Possono però avere diversa efficienza.

PROGRAMMA: La formalizzazione testuale di un algoritmo in un linguaggio di programmazione

Linguaggio di programmazione

venerdì 12 gennaio 2018 12.19

LINGUAGGIO DI ALTO LIVELLO: linguaggio che si basa su una macchina virtuale (non direttamente sull'hardware) la quale provvederà a tradurre le istruzioni in linguaggio macchina.

TIPI DI LINGUAGGI:

- Imperativi: C , Pascal , Basic , ...
- A Oggetti: Java , Python, C++ , ...
- Dichiarativi: Prolog , ...
- Funzionali: Lisp , ...
- ...

LINGUAGGIO DI PROGRAMMAZIONE: è una notazione formale usata per descrivere algoritmi e tiene conto di due aspetti:

- **Sintassi** L'insieme delle regole formali che dettano le modalità per costruire frasi corrette
  Tali regole vengono espresse attraverso altri linguaggi che, siccome operano per altri linguaggi, sono detti **metalinguaggi** come:
  - BNF
  - EBNF
  - Diagrammi Sintattici

  Es. EBNF:

```
<naturale> ::=
  0 | <cifra-non-nulla>{<cifra>}
<cifra-non-nulla> ::=
  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |9
<cifra> ::=
  0 | <cifra-non-nulla>
```

  <naturale> è un elemento non terminale che mi descrive la sintassi di un preciso oggetto che voglio descrivere in un linguaggio

  1 è un elemento terminale
  | è un simbolo che indica un'alternativa
  [ ] è un simbolo che rappresenta un elemento che può essere omesso
  { } è un simbolo che rappresenta un elemento che può essere ripetuto più volte
  $\{\}^n$ è un simbolo che rappresenta un elemento che può essere ripetuto n volte

  Es. struttura di un numero naturale:

  Possiamo dunque definire una **grammatica formale** con la quadrupla **(VT, VN, P, S)** dove:
  - VT è l'insieme finito di simboli terminali
  - VN è l'insieme finito di simboli non terminali
  - P è l'insieme finito delle regole di scrittura (produzioni)
  - S è un simbolo non terminale detto simbolo iniziale (o scopo)

  Data una grammatica G si dirà L_G il linguaggio generato da G.

  Mediante il **meccanismo di derivazione** si potrà stabilire se un testo scritto in un dato linguaggio è grammaticalmente corretto rispetto a G. Un esempio di meccanismo di derivazione è il LEFT-MOST il quale implica che, partendo dal simbolo iniziale o scopo S , si riscrivi sempre il simbolo non terminale più a sinistra associando gli elementi terminali della frase.
  Del controllo della sintassi se ne occupa il **PARSER** altrimenti detto **COMPILATORE**

- **Semantica** L'insieme dei significati da attribuire alle frasi

Compilatore

giovedì 28 settembre 2017 18.15

Per eseguire sulla macchina hardware un programma scritto in un linguaggio di alto livello è necessario tradurre il programma in sequenze di istruzioni di basso livello mediante:

1. INTERPRETAZIONE | Il programma viene tradotto 'frase per frase' durante l'esecuzione | Es: Python
2. COMPILAZIONE | Il programma viene completamente tradotto ed è pronto per l'esecuzione | Es: C

NB: Linguaggi come il Java sono eseguibili mediante compilazione (creazione dell'oggetto) e poi interpretazione (dell'oggetto da parte della JVM)

EDITOR	File sorgente	COMPILATORE	File oggetto	LINKER (+ librerie)	File eseguibile	LOADER	File caricato in RAM

Il **COMPILATORE** è dunque colui che traduce il file sorgente in file oggetto e ha i compiti di:

- Analisi del file sorgente: lessicale (simboli validi?), sintattica (grammatica corretta?), semantica (vincoli sui dati?)
- Sintesi del file oggetto: generazione del codice (allocazione di memoria e registri) e ottimizzazione

Linguaggio C

giovedì 28 settembre 2017 18.36

Caratteristiche:

- Sequenziale
- Imperativo
- Strutturato a blocchi

Si basa su:

- Dati
- Espressioni
- Dichiarazioni
- Funzioni
- Blocchi di istruzioni

Istruzioni e Regole di visibilità per gli identificatori

lunedì 16 ottobre 2017 09.37

Le **istruzioni** esprimono azioni che, una volta eseguite, comportano una modifica permanente nello stato interno del programma.

Le istruzioni possono essere:

- **Semplici** (e terminano con il punto e virgola)
- **Di controllo**
  - Composta (blocco "{ }")
  - Condizionale (selezione)
  - Iterazione (ciclo)

Esistono delle **regole di visibilità** per gli identificatori che definiscono in quali parti del programma essi possono essere usati:

- Non sono visibili prima della loro dichiarazione
- Sono visibili in tutti gli ambienti contenuti in quello della dichiarazione
- Se in un ambiente sono visibili due definizioni dello stesso identificatore, quella valida è quella dell'ambiente più vicino al punto di utilizzo

Principi di programmazione

lunedì 16 ottobre 2017 09.51

EFFICIENZA: algoritmo che consumi poche risorse

MODULARITA': Dividere il problema in sottoproblemi costruendo sottoparti di soluzione in modo che possano essere riutilizzate in altri contesti

ORDINE: Leggibile e di facile comprensione. Uso dei commenti

REGOLE DI NAMING: Le variabili devono avere nomi autoesplicativi

REGOLE DI INDENTAZIONE

Prepocessore C

lunedì 16 ottobre 2017 09.56

Il **preprocessore** agisce prima del compilatore sul file sorgente.
Agisce esclusivamente sul testo per manipolarlo.

Cosa può fare?

- Includere altre porzioni di testo prese da altri file
  #include nomefile

- Effettuare ricerche e sostituzioni
  # define testo1 testo2 (**MACRO**: sotituisce semplicemente il testo1 con il testo2 senza controlli)

- Inserire o sopprimere parti del testo al verificarsi di certe condizioni
  #ifdef / #ifndef condizione
  …testo…
  #endif

Funzioni

martedì 17 ottobre 2017 12.41

Una funzione è una serie di istruzioni che risolvono parti specifiche di un problema, permette di associare loro un nome e rende un'espressione parametrica.

Si compone di:
- Valori in ingresso variabili
- Corpo
- Valore di ritorno unico

L' interfaccia (o firma)di una funzione è composta da tipo di valore di ritorno, nome, lista dei parametri con tipi associati.

I **parametri** dichiarati nella firma della funzione si dicono **formali**.
I **parametri** passati alla funzione all'atto della chiamata si dicono **attuali**

All'atto dell'invocazione di una funzione:
- Si crea una nuovo **istanza**
- Si alloca la memoria per parametri e eventuali variabili interne
- Si trasferiscono i parametri attuali all'istanza della funzione
- Si trasferisce il controllo all'istanza della funzione
- Si esegue il corpo della funzione

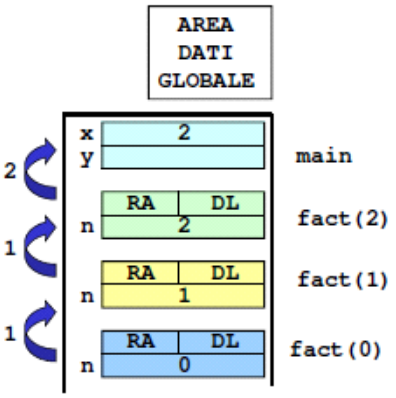
Questo processo che si svolge a **run-time** introduce un nuovo **binding** nell'ambiente in cui la funzione è definita. La struttura che contiene i binging dei parametri e degli identificatori locali è detta **record di attivazione**.

I parametri possono essere trasferiti
- Per valore (una copia del valore)
- Per riferimento (in C si tratta sempre una copia, ma questa volta si può sfruttare il passaggio di un indirizzo di memoria contenente il dato valore permettendo di alterarlo)

Si dicono **funzioni ricorsive** quelle definite in termini di loro stesse.

Record di attivazione

giovedì 19 ottobre 2017 15.16



E' l'ambiente che viene a crearsi a run-time al momento
in cui una funzione viene chiamata e contiene tutto ciò che la caratterizza:

- Parametri ricevuti
- Variabili locali
- Indirizzo di ritorno (**Return Address** - **RA**)
  [indica in che punto del codice del chiamante
  tornare al termine della funzione]
- Collegamento al record di attivazione del chiamante (**Dynamic Link** - **DL**)
  [indica l'indirizzo di memoria del record di attivazione del chiamante
  da cui proseguire]

Osservazione: Il record di ativazione è lo stesso per una data funzione ma varia da una funzione all'altra.

Osservazione: Ogni nuova chiamata a funzione genera un nuovo record di attivazione che viene allocato in una pila detta **stack** di tipo **LIFO** (last-in first-out).

Il record di attivazione nasce e muore con la funzione a cui fa riferimento:

- Al momento del return viene semplicemente assegnato il valore di return all'indirizzo puntato dal Return address
- Al momento della deallocazione viene semplicemente rimandato il puntatore di esecuzione all'indirizzo di memoria del precedente record di attivazione puntato dal Dynamic Link

Procedure

giovedì 19 ottobre 2017 15.06

Una procedura è sostanzialmente una funzione senza valori di ritorno.
Il return serve soltanto per restituire il controllo al chiamante.
Poiché la procedura non prevede dati di ritorno, essa comunica con il chiamante sostanzialmente solo tramite:

- Parametri (passaggio per riferimento)
- Aree di dati globali

Il C mette a disposizione due operatori di basso livello per permettere all'utente di farsi carico di gestire il passaggio per riferimento:

- **Operatore di estrazione di indirizzo** &
- **Operatore di dereferenziamento** *

E si ha: &x = a <==> x = *a

Variabili globali - parametri condivisi e el. con stato

martedì 24 ottobre 2017 13.04

CONDIVISIONE DEI PARAMETRI

Un altro metodo mediante il quale una procedura può attuare modifiche sui dati effettivi del chiamante sono le **variabili globali**, le quali:

- Sono allocate nell'area dati globale (fuori da ogni funzione)
- Esistono prima della chiamata del main
- Sono visibili a tutti i file se precedute dalla clausola **extern**
- Sono inizializzate automaticamente a 0 se non specificato
- Il tempo di vita è quello dell'intero programma

Es:

```
int quoziente, int resto;

void dividi(int x, int y) {
  resto = x % y; quoziente = x/y;
}

int main(){
  dividi(33, 6);
  printf("%d%d", quoziente,resto);
}
```

ELEMENTI CON STATO

Esse sono utilizzate anche per costruire specifici elementi dotati di stato .
(Ad esempio una funzione che deve restituire un numero dispari partendo dall'ultimo generato. In questo caso infatti il programma dev
e tenere conto dell'ultimo numero fornito)

Es:

```
int ultimoValore = 0;

int prossimoDispari(void){
  return 1 + 2 * ultimoValore++; }
```

OSS: E' importante considerare il fatto che le variabili globali sono potenzialmente visibili in tutti i file dell'applicazione. Questo ci pone davanti **problemi di protezione**.

Il C mette a disposizione un'apposita clausola, **static**, per mantenere la visibilità della variabile solo all'interno del file in cui è definita e quindi non può essere acceduta mediante extern da altri file.
Come accedere dunque alla variabile? Attraverso un'appositá funzione definita nel file in cui la variabile è dichiarata.

Es 1:
La variabile globale, già resa permanente, viene resa protetta.

```
static int ultimoValore = 0;

int prossimoDispari(void){
  return 1 + 2 * ultimoValore++;
}
```

Es 2:
La variabile definita in una funzione, viene resa permanente.

```
int prossimoDispari(void){
   static int ultimoValore = 0;
 return 1 + 2 * ultimoValore++;
}
```

Progetti su più file

martedì 24 ottobre 2017 13.14

La suddivisione di collezioni di funzioni in file distinti è una buona pratica per aumentare la leggibilità e la modularità di un progetto. In C è possibile definire delle vere e proprie librerie di funzioni richiamabili ogni volta che ne abbiamo bisogno senza bisogno di riscrivere il codice per intero ogni volta.

Sarà poi il linker a collegare tutti i singoli file .c in un unico progetto fino a generare un eseguibile

Quello che dobbiamo tenere presente però è che, per la fase di compilazione, è necessario che il compilatore sia a conoscenza di una funzione che si sta chiamando e per questo è necessario che almeno la firma della funzione sia specificata prima della sua invocazione.

Una firma del tipo:

```
int funzioneParametrica ( int a, int b, char c);
```

è detta **prototipo** di funzione e i nomi dei parametri formali possono essere omessi.
Il prototipo di una funzione realizza quella che si dice la **dichiarazione** di una funzione e cioè specifica le modalità in cui essa dovrà essere invocata.
Con il termine **definizione** di una funzione si fa invece riferimento alla realizzazione di un corpo per la funzione.

Per rendere tutto ancora più modulare, oltre a dividere i file contententi le definizioni delle funzioni e lasciare le dichiarazioni nei rispettivi file che ne fanno uso, si cerca di rendere anche questa operazione più automatica scrivendo le dichiarazioni in un **file header** (con estenzione .h) che andremo a incorporare mediante clausola **#include** dove necessario. Attenzione! Nel file header vanno specificati solo i prototipi delle funzioni a cui si vuole fare riferimento.

L'inclusione di un file header va fatta con parentesi (< >) se si fa riferimento ad una libreria standard in un path preciso e conosciuto dal sistema o mediante doppi apici (" ") se si fa riferimento a una nostra libreria definita localmente

Es:
File main.c

```
#include <stdio.h>
#include "f2c.h"

int main() {
    float c = fahrToCelsius(86);
}
```

File f2c.h

```
float fahrToCelsius(float);
```

File f2c.c

```
float fahrToCelsius(float fahr){
    .
    .
    .
    return celsius;
}
```

Bisezione ed es. di suddivisione progetto in più file pt.1

giovedì 26 ottobre 2017 12.19

Dato un intervallo [a, b] in cui la funzione f(x) nei punti a,b restituisce due valori di segno opposto, troviamo la c t.c. f(c) = 0

Come gestire la definizione della funzione f(x)?
Per rendere la funzione generica di può pensare di dichiararla in un file header e definirla in un file Funzione.c

Es:

_____
File main.c

_____

```
#include <stdio.h>
#include <math.h>
#include “Common.h"
#include "Zeri.h"

int main() {
     double zero;
     CODICEUSCITA code;
     code = bisezione(0, 2, 30, 0.0001, &zero);
     if (code == OK) {
          printf("Zero: %.10f\n\n", zero);
     } else {
          printCodiceUscita(code);
          printf("\n\n");
     }
     return (0);
}
```

_____
File Funzione.h

_____

```
double funzione (double x);
```

_____
File Funzione.c

_____

```
double funzione (double x){
     return x*x - 2
}
```

_____
File Common.h

_____

```
#define BOOLEAN int
#define TRUE 1
#define FALSE 0
```

****
Come gestire i codici di uscita per la gestione degli errori?
****

_____
File Zeri.h

_____

```
#define CODICEUSCITA int
#define OK 0
#define TROPPEITERAZIONI 1
#define INTERVALLONONVALIDO 2

#include "Common.h"
#include "Funzione.h"

void printCodiceUscita( CODICEUSCITA code);

CODICEUSCITA bisezione(double a, double b, int maxIterazioni, double
epsilon, double *xZero);
```

_____
File Zeri.c

_____

```
#include <stdio.h>
#include "Zeri.h"

void printCodiceUscita(CODICEUSCITA code){
     switch(code){
          case OK: printf("Ok.");
                   break;
          case TROPPEITERAZIONI: printf("Troppe iterazioni.");
                   break;
          case INTERVALLONONVALIDO: printf("Intervallo non valido.");
                   break;
          default: printf("Codice sconosciuto.");
                   break;
     }
}
```

Bisezione ed es. di suddivisione progetto in più file pt.2

venerdì 12 gennaio 2018 12.03

```
CODICEUSCITA bisezione(double a, double b, int maxIterazioni, double epsilon, double
*xZero)
{
     CODICEUSCITA cod_toReturn;
     int i;
     double xa, xb;
     double fa, fb;
     double xm, fm;
     BOOLEAN stop = FALSE;
     if (a > b) {
          xb = a;
          xa = b;
     } else {
     xa = a;
     xb = b;
     }
     if (funzione(xa) * funzione(xb) >= 0) {
           cod_toReturn = INTERVALLONONVALIDO;
     }
     for (i = 0; i < maxIterazioni && !stop; i++) {
          fa = funzione(xa);
          fb = funzione(xb);
          xm = (xa + xb) / 2;
          fm = funzione(xm);
          if (fm * fa < 0) {
               xb = xm;
          } else {
               xa = xm;
          }
          stop = ((fm == 0.0) || (doubleAbs(xb - xa) < epsilon));
     }
     if (stop) {
          *xZero = xm;
          cod_toReturn = OK;
     } else {
          cod_toReturn =  TROPPEITERAZIONI;
     }
     return cod_toReturn;
}
```

Vettori (array)

giovedì 26 ottobre 2017 13.01

Un vettore è una **collezione finita** di N variabili memorizzate in **maniera contigua** dello **stesso tipo** identificate da un **indice** compreso tra 0 e N-1

Es. definizione :

```
int v1 [ 4 ];                   OK
int N = 10;
int v2 [ N ];                   SBAGLIATO (posso ovviare usando il define)
int v3 = { 43, 12, 34, 0};      OK
```

Inserimento elemento, letto da tastiera, nell'array in prima posizione

```
scanf("%d", &v1[0]);
```

La **dimensione logica** di un array può essere inferiore alla **dimensione fisica** : quella utilizzata può essere minore di quella allocata.

Stringhe (char array)

martedì 31 ottobre 2017 09.12

Una **stringa** è un array di N caratteri terminato dal carattere **'\0'** (dunque può ospitare N-1 caratteri).

Es. definizione :

```
char s[] = "ape";              //char s[4] = { 'a', 'p', 'e', '\0' };
```

Inserimento caratteri letti da tastiera nella stringa:

```
#define N 4

//metodo 1
char s[N]; int i;
for(i = 0; i < N; i++){
    scanf("%c", &s[i]);
}s[N] = '\0'

//metodo 2
char s[N];
scanf("%s", s);
```

N.B. Utilizzando il secondo metodo la scanf interpreta la terminazione della stringa quando l'utente inserisce uno spazio ' ' o un carriage return '\n'

N.B. Gli array non possono essere copiati per valore

es:

```
char s2[100], s1[] = "stringa di prova";
s2 == s1;
```

N.B. Osservare che è possibile accedere al vettore anche con la sintassi classica dei puntatori

```
char v[20];
k=10;
v[k] == *(v + k);
```

N.B. Tenere ben presente l'aritmetica dei puntatori anche nel caso dei vettori

```
double a[2], *p, *q;

p=a;    // p = a[0];
q=p+1;  // q = a[1]; // Notare che secondo l'aritmetica dei puntatori q assume il valore
                     // della cella di memoria di p più il numero di byte
                     // della dimensione del tipo double

printf("%d\n", q-p);                // Per questo motivo qui viene stampato 1
printf("%d\n", (int) q - (int) p);  // Qui viene stampato 8 (la dimesione del
                                    // tipo double in byte) <==> sizeof
```

Libreria per funzioni sulle stringhe: **<string.h>**

Funzioni delle stringhe <string.h>

martedì 7 novembre 2017 09.22

strlen

```
int strlen(char* str);
```

Restituisce la lunghezza della stringa str

strcmp

```
int strcmp(char* str1, char* str2);
```

Compara la stringa str1 e la stringa str2 e restituisce:

- <0 se str1 precede lessicograficamente str2
- >0 se str1 segue lessicograficamente str2
- =0 se str1 e str2 sono identiche

strcat

```
char* strcat(char* dest, char* src)
```

Appende la stringa src in coda alla stringa dest

N.B. La stringa dest deve essere abbastanza grande da poter contenere anche src (\0 incluso)

strchr

```
char* strchr(char* str, char car);
```

Restituisce la posizione della prima occorrenza del carattere car nella stringa oppure NULL se il carattere non viene trovato.

- **NULL** è una variabile puntatore a void che troviamo nella libreria stdlib.h

strstr

```
char* strstr(char* str, char* sub);
```

Restituisce la posizione dell'inizio della prima occorrenza della stringa sub nella stringa str oppure NULL se la sottostringa non viene trovata

Struttura

martedì 7 novembre 2017 09.44

Si definisce struttura una collezione finita di variabili non necessariamente dello stesso tipo, ognuna identificata da un nome.

Es:

```
struct persona {
     char nome[20];
     int eta;
     float stipendio;
} pers1;
struct persona pers2, pers3, … ;
```

stampare il nome di pers1:

```
printf("%s\n", pers1.nome);
```

Typedef

martedì 7 novembre 2017 11.04

Il typedef definisce un nuovo identificatore di tipo

```
typedef int MioIntero;
MioIntero X, Y, Z;
int W


typedef enum {lu, ma, me, gi, ve, sa, dom} Giorni;
typedef enum {true, false} Boolean;
```

File

martedì 14 novembre 2017 10.57

Il **file** è un'astrazione fornita dal S.O. per consentire la memorizzazione di informazioni su memoria di massa (in maniera persistente)

Il concetto di scrittura su file è definito da una testina di lettura/scrittura (ideale) che indica in ogni istante il record corrente e, dopo ogni operazione essa si sposta sulla registrazione successiva.

A livello di S.O. esso è denotato univocamente dal suo **nome assoluto**, che comprende il **percorso** e il **nome relativo**.
In certi S.O. il percorso può comprendere anche il nome dell'unità.
Es.

- in DOS o Windows:
  `C:\temp\prova1.c`
- in UNIX e Linux:
  `/usr/temp/prova1.c`

In linguaggio C esso è gestito mediante una variabile di tipo **FILE** (struttura definita in <stdio.h>).
Tale struttura può essere gestita soltanto le funzioni della libreria standard.

fopen

```
FILE* fopen(char filename[], char modalita[])
```

La funzione serve ad aprire un dato file in una certa modalità e restituisce un puntatore alla struttura creata (NULL in caso l'apertura sia fallita).

Modalità:
- r apertura in lettura (read)
- w apertura in scrittura (write)
- a apertura in aggiunta (append)

seguita opzionalmente da:
- t apertura in modalità testo (default)
- b apertura in modalità binaria

ed eventualmente da:
- + apertura con possibilità di modifica

fclose

```
int fclose(FILE*)
```

La funzione chiude il file e restituisce 0 se tutto è andato bene, EOF (valore intero negativo se si è verificato un errore)

fgetc

```
int fgetc(FILE* f);
```

La funzione legge un carattere e restituisce la sua codifica ASCII

fputc

```
int fputc(int c, FILE* f);
```

La funzione stampa un carattere data la sua codifica ASCII su un file e restituisce il carattere stesso se la funzione ha successo, altrimenti EOF in caso di errore

fgets

```
char* fgets(char* s, int size, FILE* f);
```

La funzione legge una stringa di lunghezza size-1 oppure fino al carattere \n o EOF se essi si presentano prima. Se la funzione ha successo ritorna la stringa altrimenti NULL

fputs

```
int fputs(char* s, FILE* f);
```

La funzione stampa lastringa s fino al carattere terminatore su file. Se la funzione ha successo ritorna un valore non negativo altrimenti il carattere EOF.

fprintf

```
int fprintf(FILE* f, char* format, ...);
```

La funzione stampa su file i valori riportati seguendo la formattazione indicata. Restituisce il numero di caratteri scritti correttamente.

fscanf

```
int fscanf(FILE* f, char* format, ... );
```

La funzione legge da file i valori riportati seguendo la formattazione indicata. Restituisce il numero di valori letti correttamente.

FUNZIONI PER ESERCITARSI

Funzione per leggere stringhe dal file fino ad un dato carattere separatore e le restituisce mediante il parametro buffer. Infine ritorno il numero di caratteri letti.

```
int readField(char buffer[], char sep, FILE *f) {
    int i = 0;
    char ch = fgetc(f);
    while (ch != sep && ch != '\n' && ch != EOF) {
        buffer[i++] = ch;
        ch = fgetc(f);
    }
    buffer[i] = '\0';
    return i;
}
```

File Binari

martedì 21 novembre 2017 10.20

Un **file binario** è una pura sequenza di byte, senza alcuna strutturazione particolare.

N.B. Input e output avvengono sotto forma di una sequenza di byte, infatti, la fine del file è SEMPRE rilevata in base all'esito delle operazioni di lettura (non ci può essere EOF, perché un file binario non è una sequenza di caratteri e qualsiasi byte si scegliesse come marcatore potrebbe sempre capitare nella sequenza)

fwrite

```
int fwrite(TYPE* addr, int dim, int n, FILE *f);
```

Funzione che scrive sul file binario *f*, *n* elementi di dimensione *dim* prelevati a partire dall'indirizzo di memoria *addr*.
La funzione restituisce il numero degli elementi effettivamenti scritti.

Es:

```
int main(){
     FILE *fp;
     int vet[10] = {1,2,3,4,5,6,7,8,9,10};
     if ((fp = fopen("numeri.dat","wb")) != NULL) {
          fwrite(vet, sizeof(int), 10, fp);
          fclose(fp);
     }
}
```

fread

```
int fread(TYPE* addr, int dim, int n, FILE *f);
```

Funzione che legge da file binario *f*, *n* elementi di dimensione *dim* e li colloca in memoria a partire dall'indirizzo *addr*.
La funzione restituisce il numero di elementi effettivamente letti.

Es:

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
     FILE *fp;
     int vet[40], i, n;
     if ((fp = fopen("numeri.dat","rb")) != NULL)
          n = fread(vet, sizeof(int), 40, fp);
          for (i=0; i<n; i++){
               printf("%d ",vet[i]);
          }
          fclose(fp);
     }
}
```

Algoritmi di ordinamento pt.1

giovedì 16 novembre 2017 12.17

https://brilliant.org/wiki/sorting-algorithms/
https://www.toptal.com/developers/sorting-algorithms

```
void scambia(int *a, int *b){
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
int trovaPosMax(int v[], int n){
    int i, posMax=0;
    for (i=1; i<n; i++){
        if (v[posMax]<v[i]){
            posMax=i;
        }
    }
    return posMax;
}
```

NAÏVE SORT

(semplice, poco efficiente)

Trovare il massimo, verificare che sia in ultima posizione, se non lo è allora scambiare l'elemento corrente con l'ultimo.

```
void naiveSort(int v[], int n){
    int p;
    while (n>1) {
        p = trovaPosMax(v,n);
        if (p<n-1){
            scambia(&v[p],&v[n-1]);
        }
        n--;
    }
}
```

Complessità: (n-1) + (n-2) + … + 1 -> O-grande($N^2/2$) [SEMPRE!]
OSS: Nel caso in cui l'array è già ordinato l'algoritmo controlla lo stesso che tutti elementi più grandi si trovino in ultima posizione

BUBBLE SORT

(semplice, un po' più efficiente)

Scorro tutto l'array confrontando gli elementi adiacenti e scambiando questi quando il primo è maggiore del secondo fino a portare il magggiore in ultima posizione. Nel caso in cui non venga effettuato nessuno scambio l'array è ordinato e l'algoritmo può interrompersi.

```
void bubbleSort(int v[], int n){
    int i, ordinato = 0;
    while (n > 1 && !ordinato){
        ordinato = 1;
        for (i=0; i < n-1; i++){
            if (v[i] > v[i+1]) {
                scambia(&v[i], &v[i+1]);
                ordinato = 0;
            }
        }
        n--;
    }
}
```

Complessità: (n-1) + (n-2) + … + 1 -> O-grande($N^2/2$) [CASO PEGGIORE!]
n -> O-grande(N) [CASO MIGLIORE!]

INSERT SORT

(intuitivo, abbastanza efficiente)

Partendo dalla posizione 1 del vettore procediamo con il memorizzare il valore in una variabile temporanea per poi far scorrere indietro gli elementi nelle posizioni precedenti finchè non trovo che il valore nella variabile temporanea non si trova nella corretta posizione.

```
void insOrd(int v[], int pos){
    int i = pos-1, x = v[pos];
    while (i >= 0 && x < v[i]) {
        v[i+1] = v[i];
        i--;
    }
    v[i+1] = x;
}

void insertSort(int v[], int n){
    int k;
    for (k=1; k < n; k++){
        insOrd(v, k);
    }
}
```

Complessità: 1 + 2 + 3+ … + (n-1) -> O-grande($N^2/2$) [CASO PEGGIORE!]
n-1 [CASO MIGLIORE!]
1/2 + 2/2 + … + (n-1)/2 -> O-grande($N^2/4$) [CASO MEDIO!]

MERGE SORT

(non intuitivo, molto efficiente)

L'idea di base è di suddividere array a metà per renderlo più piccolo e minimizzare la complessità data dal valore al quadrato. Si parte suddividendo l'array a metà, si comincia ad ordinare l'array secondo il merge sort che quindi riporterà a dividere l'array a metà finchè non arriva ad array di 2 elementi; ordina questi due elementi e risale fino a ricostruire l'array iniziale ordinato.

```
void merge(int v[], int i1, int i2, int fine, int vout[]) {
    int i = i1, j = i2, k = i1;
    while ( i <= (i2 - 1) && j <= fine ) {
        if (v[i] < v[j]){
            vout[k] = v[i++];
         } else {
            vout[k] = v[j++];
        }
        k++;
    }
    while (i<=i2-1) { vout[k] = v[i++]; k++; }
    while (j<=fine) { vout[k] = v[j++]; k++; }
    for (i=i1; i<=fine; i++) { v[i] = vout[i]; }
}

void mergeSort(int v[], int first, int last, int vout[]) {
    int mid;
    if ( first < last ) {
        mid = (last + first) / 2;
        mergeSort(v, first, mid, vout);
        mergeSort(v, mid+1, last, vout);
        merge(v, first, mid+1, last, vout);
    }
}
```

Complessità: O(N*log2 N) [SEMPRE!] ma c'è il problema del raddoppiamento della memoria utilizzata per ordinare l'array

Algoritmi di ordinamento pt.2

venerdì 12 gennaio 2018 12.13

QUICK SORT

(non intuitivo, alquanto efficiente)

Si determina arbitrariamente un pivot. Si comincia a scorrere l'array da sinistra finchè non si trova un valore maggiore del pivot o si scavalca l'indice di destra, poi si scorre l'array da destra finchè non si trova un valore minore del pivot o si scavalca l'indice di sinistra. Se i due indici non si sono accavallati e dunque sono stati trovati due numeri nell'array rispettiamente uno maggiore del pivot e uno minore del pivot, essi vengono scambiati, e si continua a scorrere l'array da destra e da sinistra finchè i due indici non si trovano nella stessa posizione, a quel punto, se la posizione in cui si trovano è diversa dalla posizione del pivot o il valore in quella cella è diverso dal valore del pivot abbiamo trovato la posizione centrale dell'array per il pivot e scambieremo il valore in quella cella con quello del pivot. A questo punto viene diviso l'array e chiamata la funzione quick sort in modo ricorsivo sui due array generati.

```
void quickSortR(int a[], int iniz, int fine)
{
    int i, j, iPivot, pivot;
    if (iniz < fine)
    {
        i = iniz;j = fine;
        iPivot = fine;
        pivot = a[iPivot];
        do /* trova la posizione del pivot */
        {
            while (i < j && a[i] <= pivot) i++;
            while (j > i && a[j] >= pivot) j--;
            if (i < j) scambia(&a[i], &a[j]);
        }while (i < j);

        /* determinati i due sottoinsiemi */
        /* posiziona il pivot */
        if (i != iPivot && a[i] != a[iPivot])
        {
            scambia(&a[i], &a[iPivot]);
            iPivot = i;
        }

        /* ricorsione sulle sottoparti, se necessario */
        if (iniz < iPivot - 1)
            quickSortR(a, iniz, iPivot - 1);
        if (iPivot + 1 < fine)
            quickSortR(a, iPivot + 1, fine);
    }
}

void quickSort(int a[], int dim)
{
    quickSortR(a, 0, dim - 1);
}
```

Complessità: O-grande(N^2) [CASO PEGGIORE!]
O(N*log2 N) [CASO MIGLIORE!]

Ambiente globale

giovedì 23 novembre 2017 12.32

L'**ambiente globale** è quello in cui tutte le funzioni sono definite.
Qui si possono definire anche variabili dette globali.
Tale ambiente non coincide con quello di nessuna funzione (neppure il main).

(vedi Variabili globali - parametri condivisi e el. con stato)

Allocazione dinamica

giovedì 23 novembre 2017 13.13

malloc

```
void * malloc (size_t dim);
```

Funzione che richiede l'allocazione di un'area di memoria grande *dim* byte al sistema operativo e restituisce un puntatore all'indirizzo dell'area di memoria allocata (NULL se per qualche motivo qualcosa è andato storto).
L'allocazione viene effettuata nell'area di memoria heap.

Es:

```
// Definisco un array di DIM elementi
#define DIM 10
int * p = (int *) malloc(DIM * sizeof(int));

// Libero l'area di memoria allocata in p ()
free(p);
```

Se non si dealloca la memoria si incorre nel cosiddetto memory leaking, cioè un consumo non voluto della memoria da parte di variabili non più utilizzate ma ancora valide in memoria.

ADT

martedì 28 novembre 2017 09.33

ADT (Abstract Data Type) definisce una categoria concettuale (classe) con le sue proprietà.
Implica la definizione di un dominio e delle funzioni e dei predicati inerenti allo stesso dominio.

typedef

```
typedef int counter;
```

Funzione che ci permette di definire una nuova entità partendo da un tipo di dato già esistente o una struttura.

Operazioni da definire per un ADT

- Costruttori
- Selettori
- Predicati (verificano la presenza di una proprietà)
- Funzioni
- Trasformatori (cambiare lo stato dell'oggetto)

La gestione dell' ADT è a discapito del programmatore, il C non fornisce alcuna protezione contro un uso scorretto dello stesso.

Liste

martedì 28 novembre 2017 10.10

Una **lista** è una struttura dati astratta che denota una collezione omogenea di dati. E' una struttura dati dinamica poiché può cambiare di dimensione.

N.B. L'accesso a un elemento della struttura avviene direttamente solo al primo elemento della sequenza; per accedere a un qualunque elemento, occorre scandire sequenzialmente tutti gli elementi che lo precedono.

```
#include <stdlib.h>
#include "element.h"

typedef struct list_element{
    Element value;
    list_element *next;
} node;
typedef node *list;

list cons(Element d, list l);
list emptyList();
BOOLEAN empty(list);
Element head(list);
list tail(list);
```

Rappresentazione binaria e complemento a due

giovedì 7 dicembre 2017 12.36

La **rappresentazione binaria** è basata su notazione posizionale con base di rappresentazione B = 2.

$1010 = 1 * B^3 + 0 * B^2 + 1 * B^1 + 0 * B^0$, B = 2 => 8 + 2 = 10 < da binario a decimale >

10 = 0 + B * (1+ B * (0) , B = 10 (**Metodo di Horner)** < da binario a decimale >

$$v = \sum_{k=0}^{n-1} d_k B^k$$

10 => 10 / 2 = 5 , **0**
5 / 2 = 2 , **1**
2 / 2 = 1 , **0**
1 / 2 = 0 , **1** < da decimale a binario >

Con N bit, si possono ottenere 2^n combinazioni, rappresentando i numeri da 0 a $2^n - 1$

Il **complemento a due** è la convenzione adottata dai sistemi informatici per eseguire le operazioni tra numeri in base 2.
L'idea è quella di cambiare il valore del bit più significativo da $+2^{n-1}$ a -2^{n-1}.

Ad esempio in complemento a due (8-bit) avremo
00000101 = -0 + 5 = 5
10000101 = -128 + 5 = -123

Con N bit, si possono ottene
re 2^n combinazioni, rappresentando i numeri da -2^n a $+2^n - 1$

$$v = -d_{n-1} B^{n-1} + \sum_{k=0}^{n-2} d_k B^k$$

Come rappresentare un numero decimale in base 2 con complemento a due?
$V' = (2^n - 1) - |V| + 1$
Ad esempio in complemento a due (8-bit) avremo
12 = 00001100
-12 = (11111111 - 00001100) + 00000001 = 11110011 + 00000001 = 11110100

$(2^n - 1) - |V|$ è sostanzialmente l'inversione della rappresentazione in base 2 di |V|

L'algoritmo funziona anche al contrario per riportare il numero negativo rappresentato in complemento a 2 al corrispondente numero positivo
Ad esempio in complemento a due (8-bit) avremo
-12 = 11110100
12 = (11111111 - 11110100) + 00000001 = 00001011 + 00000001 = 00001100

Se si stanno sommando due numeri positivi o due numeri negativi di modulo molto grande può succedere che il risultato vada in overflow
Per verificare se il risultato è errato basta controllare:

- se i due numeri sono positivi, che il bit più significativo sia 0
- se i due numeri sono negativi, che il bit più significativo sia 1