L02.1. TESTING DEI PROGRAMMI

Il testing e il debugging sono una parte dell'implementazione del software. **Definizione di Testing**: esercitare il programma con dati di test per verificare che il suo comportamento sia conforme a quello atteso (definito nella *specifica*).

- **Oracolo**: è l'output atteso, quello che ci si aspetta il programma produca;
- **Malfunzionamento**: comportamento del programma diverso da quello atteso.

L'obiettivo principale del testing è quello di individuare malfunzionamenti, che è causato da un difetto (errore o bug) nel codice. L'errore può essere introdotto in fase di analisi, specifica, progettazione o codifica.

Il **Debugging** serve per individuare e correggere quel difetto che ha causato il malfunzionamento, più alta è la fase in cui si introduce il difetto, maggiore è la difficoltà nel rimuoverlo. La ricerca di un difetto può essere fatta inserendo nel codice sorgente punti di ispezione dello stato delle variabili, una volta corretto il difetto, bisogna rieseguire tutti i casi di test.

Testare il programma con tutti i possibili dati di test è impraticabile, l'obiettivo è individuare **classi di dati di test**, selezionare un caso di test da ogni classe ed evitare casi di test ridondanti. Può essere utile creare una **test suite**, ovvero un insieme di casi di test per un programma.

Esempio:

In un ordinamento di un array bisogna tener conto di alcuni ***aspetti*** nella scelta dei casi di test, come ad esempio:

- Il numero n di elementi dell'array:
  - caso generale un array con n>1;
  - caso particolare n=1.
- La disposizione degli elementi:
  - caso generale un array non ordinato;
  - un array già ordinato in modo crescente/decrescente.

Considerati questi aspetti possiamo produrre dei ***test case***:

- **Test case 1 – TC1** (un solo elemento)
  - *Array di input*: 5
  - *Oracolo*: 5
- **Test case 2 – TC2** (input ordinato in maniera crescente)
  - *Array di input*: 1 2 3 4 5 6 7 8 9
  - *Oracolo*: 1 2 3 4 5 6 7 8 9
- **Test case 3 – TC3** (input ordinato in maniera decrescente)
  - *Array di input*: 10 9 8 7 6 5 4 3 2 1
  - *Oracolo*: 1 2 3 4 5 6 7 8 9 10
- **Test case 4 – TC4** (non ordinato)
  - *Array di input*: 5 8 2 9 10 1 4 7 3 6 12 11
  - *Oracolo*: 1 2 3 4 5 6 7 8 9 10 11 12

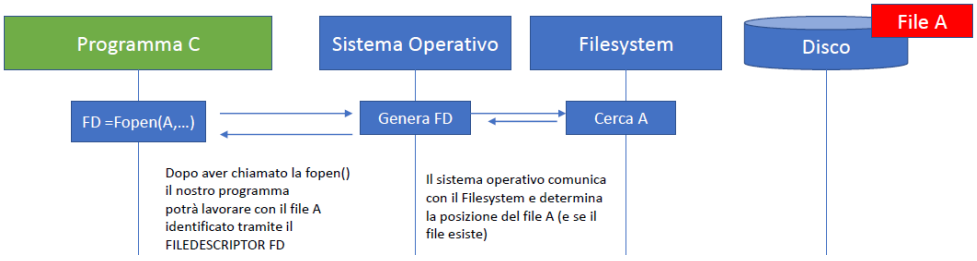
Per automatizzare il test si possono usare i ***file*** per leggere dati di input e scrivere i dati di output, semplificando così operazioni di test da ripetere.

L02.2. USO DEI FILE

In C, il termine **stream** (o flusso) indica una sorgente di input o una destinazione per l'output. Molti programmi (piccoli) ottengono il loro input da uno stream (ad es. la tastiera) e lo inviano ad un altro stream (ad esempio il video), programmi più grandi possono avere necessità di usare più stream.

Gli stream rappresentano file memorizzati da qualche parte (hard disk o altri tipi di memoria a lungo termine) e sono associati a periferiche (schede di rete, stampanti, ecc…).

Per accedere ad un file su disco è necessario chiedere al sistema operativo di restituirci un **FILE DESCRIPTOR,** ovvero un identificatore che, nel nostro programma, sarà collegato al file stesso.



Per richiedere un **FILE DESCRIPTOR**, si utilizza la funzione ***fopen*** che restituisce un puntatore ad una struttura FILE (il file descriptor).

Per utilizzare i file dobbiamo ***sempre*** effettuare due operazioni:

1. FILE *fopen(const char *filename, const char *mode), *filename* è il nome (drive o percorso) del file da aprire, *mode* è una "stringa di modalità" che specifica quali operazioni abbiamo intenzione di compiere sul file, il valore di ritorno è un puntatore a FILE.
2. Int fclose(FILE *stream), ritorna 0 in caso di successo ed *EOF* in caso di fallimento.

Per quanto riguarda la stringa ***mode***, può assumere diversi valori:

"r"	*Apre il file in lettura (il file deve esistere)*
"w"	*Apre il file in scrittura (non è necessario che il file esista)*
"a"	*Apre il file in accodamento (non è necessario che il file esista)*
"r+"	*Apre il file in lettura e scrittura (il file deve esistere)*
"w+"	*Apre il file in lettura e scrittura -tronca il file se esiste*
"a+"	*Apre il file in lettura e scrittura -accoda se il file esiste*

Per effettuare **input e output da stream** esistono diverse funzioni:

1. char *fgets(char *s, int size, FILE *stream), legge da stream fino al carattere newline (o finché size-1 caratteri sono stati letti) e li memorizza in s. Ritorna s in caso di successo, NULL in caso di errore o se si raggiunge la fine del file senza aver letto alcun carattere.
2. Int fscanf(FILE *stream, const char *format, ...), legge da stream fino ad un carattere di spazio e non lo memorizza. Restituisce il numero di dati letti e scritti con successo.
3. Int fprintf(FILE *stream, const char *format, ...), scrive su stream. Restituisce il numero di caratteri scritti, -1 in caso di errore.

Per effettuare **input e output su stringhe** utilizziamo due funzioni:

1. Int sprintf(char *str, const char *restrict format, ...), differisce dalla funzione printf() in quanto i caratteri vengono scritti nell'area puntata da str. Naturalmente str deve essere ampia a sufficienza per contenere i caratteri in output più il terminatore '\0'. Restituisce il numero di caratteri memorizzati.
2. Int sscanf(const char *str, const char *format, ...), legge i caratteri da una stringa, ha lo stesso comportamento della funzione scanf(), ma l'input avviene da str. Restituisce il numero di dati letti e scritti con successo.

Esempio uso file:

```
fgets(str, sizeof(str), stdin);        /* legge una riga dell'input */
sscanf(str, "%d%d", &i, &j);           /* estrae due interi */
```

La prima funzione permette di leggere un'intera riga da input (fino a newline), mentre la seconda permette di estrarre due interi dalla stringa.

Esempio testing ordinamento:

Usiamo 2 file per i dati di input:

1. Input.txt, contenente gli elementi dell'array di input;
2. Oracle.txt, contenente gli elementi dell'array ordinato.

Mentre per l'output:

1. Output.txt, risultante dall'esecuzione del programma (output effettivo), oltre ad una indicazione dell'esito del test (PASS / FAIL).

input.txt	oracle.txt
5 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1 5 8 2 9 10 1 4 7 3 6 12 11	5 1 2 3 4 5 6 7 8 9 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 11 12

Esempio file di testing somma:

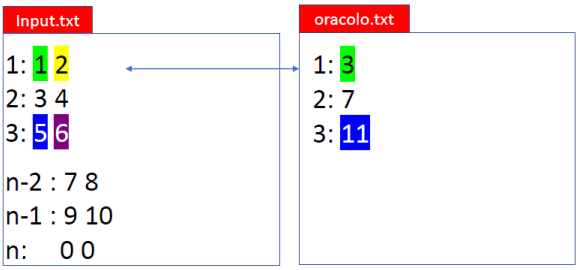
Per ottimizzare il testing, possiamo pensare di scrivere un programma **Driver** che prenda in input una o più coppie di interi da un file (una coppia per riga) e vi applichi, ad esempio, somma per registrarne l'output. Possiamo pensare di mantenere i valori attesi della somma in un file esterno (che chiameremo **Oracolo**) immettendo su ogni riga il valore corretto della somma corrispondente agli interi i1 e i2 presenti nel file di input alla riga i-esima.

Per quanto riguarda il driver (nel caso di somma) automatizza il processo di testing leggendo un file di input (input.txt), applicando sull'input la funzione che vogliamo (in questo caso la somma) ed infine confrontando il valore stampato dalla printf con il valore atteso.

Se in qualsiasi momento il risultato della funzione somma non è quella aspettata (quella definita in oracolo) il driver si arresta e restituisce fallito.

Se tutto va bene, il driver restituisce «superato».

```
#include <stdio.h>
Int somma( int a, int b) {
    return a+b
}
void main (){
    FILE *input;
    FILE *oracolo;
    input = fopen (" input.txt ","r");
    oracolo = fopen("oracolo.txt", "r");
    int i1, i2, esito=1;
    while(fscanf(input,"%d %d \n",&i1,&i2)!=EOF) {
        int r = somma(i1,i2), or;
        fscanf( oracolo,"%d \n",&or);
        if(or!=r){
            esito=0;
            break;
        }
    }
    fclose(input);
    fclose(oracolo);
    if(esito==0) printf("fallito");
    else printf("superato");
}
```



L02.3. PUNTATORI E ALLOCAZIONE DINAMICA DELLA MEMORIA

Gli array sono caratterizzati da una **cardinalità**, ovvero dalla dimensione massima, e da **riempimento**, ovvero la dimensione utilizzata in un certo momento che può variare in base alle operazioni di inserimento e rimozione.
Quando usiamo un array a dimensione *statica* dobbiamo prevedere però una cardinalità molto grande e sufficiente per tutte le esecuzioni del programma, questo però ne consegue uno spreco di memoria in quanto è possibile che non utilizziamo tutta la memoria allocata.
Con l'**allocazione dinamica** occupiamo solo la memoria necessaria, permettendo di creare strutture dati la cui dimensione varia durante l'esecuzione.
Per usare l'allocazione dinamica esistono funzioni specifiche fornite dalla libreria *stdlib*, in particolare 3 funzioni:

1. `void *malloc(size_t size);`
   Alloca un blocco di memoria di size bytes senza inizializzarlo, size_t è un tipo intero senza segno definito nella libreria del C.
   Restituisce il puntatore al blocco.
2. `void *calloc(size_t nelements, size_t elementSize);`
   Alloca un blocco di memoria di nelements* elementSize bytes e lo inizializza a 0 (clear) e restituisce il puntatore al blocco.
3. `void *realloc(void *pointer, size_t size);`
   Cambia la dimensione del blocco di memoria precedentemente allocato puntato da pointer.
   Restituisce il puntatore ad una zona di memoria di dimensione size, che contiene gli stessi dati della vecchia regione indirizzata da pointer, troncata alla fine nel caso la nuova dimensione sia minore di quella precedente.

Un **puntatore** è una variabile che contiene l'indirizzo di un'altra variabile e questi puntatori sono "*type bound*" cioè ad ogni puntatore è associato il tipo a cui il puntatore si riferisce.
Nella dichiarazione di un puntatore bisogna specificare un asterisco (*) prima del nome della variabile pointer: T *p. Esempio:

- `int *pointer;` // puntatore a intero
- `char *pun_car;` // puntatore a carattere
- `float *flt_pnt;` // puntatore a float

L'accesso all'oggetto puntatore avviene attraverso **l'operatore di deferenziazione** "*". Esempio:

- `*pointer = 5` //assegna all'oggetto puntato da pointer il valore 5
- `x = *flt_pnt` //assegna il valore dell'oggetto puntato da ftl_pnt alla variabile x

Prima di poter usare un pointer questo deve essere inizializzato, ovvero deve contenere l'indirizzo di un oggetto.
Per ottenere l'indirizzo di un oggetto si usa l'operatore unario **&**. Esempio:

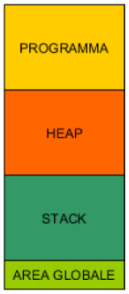
```
int i = 10, *p1;
p1 = &i;
printf("%d \n", *p1);
```

I **tipi di variabili** in C possono essere di vario tipo:

- **Globali**, dichiarate esternamente alle funzioni, sono visibili a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente e sono dette statiche, perché la loro allocazione in memoria avviene all'atto del caricamento del programma (e la loro deallocazione al termine del programma);
- **Locali**, dichiarate e visibili solo all'interno di una funzione;
- **Automatiche**, dichiarate in blocchi interni alle funzioni e vengono allocate in memoria a tempo di esecuzione (dell'istruzione dichiarativa) e deallocate al termine del blocco.

Quando si considera una variabile bisogna valutare tre aspetti importanti:

1. **Scope**, parte del programma in cui è attiva una dichiarazione (dice quando può essere usato un identificatore);
2. **Visibilità**, parte del programma in cui è accessibile una variabile (non sempre coincide con lo scope …);
3. **Durata**, periodo durante il quale una variabile è allocata in memoria.



Il Sistema Operativo riserva ad un processo (un programma in esecuzione), un segmento di memoria RAM. Questo, in generale è suddiviso in quattro distinte aree di memoria:

1. L'**area del programma**, che contiene le istruzioni macchina del programma;
2. L'**area globale**, che contiene le costanti e le variabili globali;
3. Lo **stack**, che contiene la pila dei record di attivazione creati durante ciascuna chiamata delle funzioni;
4. L'**heap**, che contiene le variabili allocate dinamicamente.

Variabili globali definite in un file F1 possono essere usate in un file F2 dichiarandole **extern** in F2: `extern int n;`
NB: con la dichiarazione extern non si definisce la variabile e non si alloca memoria
Dichiarazioni **static** di variabili globali le rendono private al file in cui sono dichiarate: `static int n;`
Dichiarazioni **static** di funzioni le rendono private al file in cui sono dichiarate, ossia ne modificano lo scope. Nel modulo vettore, la funzione minimo è usata solo localmente al modulo dalla funzione ordina_array.
Per renderla privata al modulo basta aggiungere la dichiarazione static: `static int minimo(int a[], int n);`

vettore.h

```
int ricerca_array(int a[], int n, int elem);
int minimo_array(int a[], int n);
…
```

vettore.c

```
#include <stdio.h>
#include "utile.h"                    // contiene funzione scambia
static int minimo(inta[], intn);      // dichiarazione locale
int ricerca_array(inta[], intn, intelem) { … }
int minimo_array(inta[], intn) { … }
```