Testing dei programmi

… e uso dei file

Alcune note sul testing …

- ***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
- In generale è impossibile dimostrare la correttezza di un qualunque programma …
  - ***Obiettivo del testing: individuare malfunzionamenti***

… alcune note sul testing

- Testare il programma con tutti i possibili dati di test è impraticabile …
- Obiettivo: individuare classi di dati di test, selezionare un caso di test da ogni classe ed evitare casi di test ridondanti
- In questo corso non vedremo tecniche per scegliere i casi di test in maniera sistematica (obiettivo dei corsi di Ingegneria del Software), ma useremo il buon senso …
- Documentare i casi di test e i risultati del testing …

- ***Test suite***: un insieme di casi di test per un programma

Nel nostro esempio di ordinamento di un array…

- Nel programma per l'ordinamento dell'array, nella scelta dei casi di test dobbiamo tener in conto diversi aspetti
  - Il numero `n` di elementi dell'array: considerare array con diversi elementi, ma anche il caso particolare dell'array con un solo elemento e anche il caso di `n` = 0
  - la disposizione degli elementi nell'array `a` di input: considerare il caso in cui l'array è già ordinato, il caso in cui è ordinato in senso decrescente e il caso in cui non è ordinato

Una test suite per il nostro esempio

- 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

Testing e debugging

- Un malfunzionamento di un programma è causato da un difetto (errore, bug) nel codice
  - L'errore può essere introdotto in fase di analisi e specifica, di progettazione o di codifica
- ***Debugging***: Individuazione e correzione del 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
- NB: una volta corretto il difetto, rieseguire tutti i casi di test …

Come automatizzare il test

- Per automatizzare il test si possono usare i file per leggere dati di input e scrivere i dati di output
- ***Rivediamo prima alcuni alcuni concetti legati ai file …***

Flussi (stream)

- In C, il termine ***stream*** 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 spesso rappresentano file memorizzati da qualche parte (hard disk o altri tipi di memoria a lungo termine); in altri casi sono associati a periferiche (schede di rete, stampanti, etc)

Il tipo FILE *

- Memorizzare dati in maniera non volatile
- In C (libreria <stdio>) è definito un tipo FILE che astrae il concetto di sequenza di dati contenuti nella memoria di massa

```
FILE *infile
FILE *outfile
```

  - Le variabili infile e outfile sono puntatori a file
- Portabilità delle operazioni su file

Flussi standard

- `<stdio.h>` fornisce 3 flussi standard:

Nome (*FILE)	Flusso	Periferica di default
`stdin`	Standard input	Tastiera
`stdout`	Standard output	Video
`stderr`	Standard error	Video

- Questi flussi sono pronti per essere usati—non c'è bisogno di dichiararli e non c'è bisogno di aprirli o chiuderli

File di testo e file binari

- In generale un file è una sequenza di byte
- I byte in un ***file di testo*** rappresentano caratteri (le lettere, i numeri, la punteggiatura, etc) e quindi possono essere stampati, letti e modificati con un "text editor"
  - Es.: Il codice sorgente di un programma C
- In un ***file binario,*** i byte possono assumere qualsiasi valore
  - Gruppi di byte potrebbero rappresentare altri tipi di dati, come interi, numeri in virgola mobile, strutture
  - Un programma compilato è memorizzato in un file binario
- ***Noi vedremo solo l'uso di file di testo…***

Apertura di un file

- La dichiarazione di un file **non** ne comporta la creazione
- Utilizzo della funzione *fopen*

  file_pointer_variable = fopen(*file_name, mode*);
  - *file_name*: stringa che specifica il nome del file
  - *mode*: stringa che specifica la modalità di trasferimento dati (dettagli prossima slide)
- `fopen` restituisce un puntatore al file che di solito viene memorizzato in una variabile per poter accedere al file:
  - fp = fopen("in.dat", "r");

    /* opens in.dat for reading */
- Se non riesce ad aprire il file, allora fopen restituisce il puntatore nullo

Modalità di apertura

- Modalità per i file di testo:

Stringa	*Significato*
`"r"`	Lettura
`"w"`	Scrittura
`"a"`	Scrittura in modalità “append”
`"r+"`	Lettura e scrittura
`"w+"`	Lettura e scrittura (azzera il file se esiste)
`"a+"`	Lettura e scrittura (accoda la scrittura se il file esiste)

Le operazioni di scrittura creano il file se il file non esiste.

Chiusura di un file

- La funzione `fclose` permette di chiudere un file quando non dobbiamo più fare operazioni
- L’argomento per la chiamata di `fclose` è un puntatore ad un file precedentemente aperto
- `fclose` restituisce 0 se la chiusura avviene con successo
- Altrimenti restituisce il codice di errore `EOF` (una macro definita in `<stdio.h>`)

File: Scrittura e Lettura orientata ai caratteri

- Esistono diverse funzioni in C per leggere/scrivere su fle di testo
  - Es. Lettura/scrittura di singoli caratteri, lettura/scrittura per linee
- Negli esempi che vedremo useremo le funzioni di I/O formattato `fscanf` e `fprintf`
  - Simili alle funzioni `scanf` e `printf` definite per lo standard input e standard output

Le funzioni del gruppo “printf”

- `printf` scrive sempre su `stdout`, mentre `fprintf` scrive sullo stream indicato come primo argomento:

```
printf("Total: %d\n", total);
  /* writes to stdout */
fprintf(fp, "Total: %d\n", total);
  /* writes to fp */
```

- Una chiamata a `printf` è equivalente ad una chiamata a `fprintf` con `stdout` come primo argomento
- Entrambe le funzioni restituiscono il numero di caratteri stampati; un valore negativo indica che si è verificato un errore

Le funzioni del gruppo “scanf”

- `scanf` legge i dati da `stdin`, mentre `fscanf` legge i dati dal flusso indicato come primo argomento

```
scanf("%d%d", &i, &j); // reads from stdin
fscanf(fp, "%d%d", &i, &j); //reads from fp
```

- Una chiamata a `scanf` è equivalente ad una chiamata a `fscanf` con `stdin` usato come primo argomento
- Entrambe le funzioni restituiscono un intero pari al numero di dati in input che è stato possibile assegnare ad altrettante variabili
- Restituiscono EOF se c’è un errore di input prima che si possa leggere un byte

EOF e condizioni di errori

- Se chiediamo alle funzioni …scanf di leggere un certo numero *n* di dati, è intuitivo aspettarsi un valore di ritorno pari a *n*
- Il valore di ritorno può essere minore di *n* se la funzioni non è riuscita ad assegnare tutti i valori a causa di:
  - ***EOF (End-of-file).*** La funzione ha trovato il segnalatore di fine file nell’input
  - ***Errore di lettura.*** La funzione non riesce a leggere altri caratteri dal flusso di input
  - ***Manca corrispondenza di formato.*** L’input non corrisponde al formato

EOF e condizioni di errori

- Ogni flusso ha due segnalatori ad esso associati: un ***segnalatore di errore*** ed un ***segnalatore di fine file***
- Questi indicatori vengono azzerati (FALSE) quando il file viene aperto
- Se si arriva alla fine del fine viene attivato (TRUE) il segnalatore di fine file
- Se si verifica un errore di input (o di output) viene attivato il segnalatore di errore
- Una mancata corrispondenza con il formato non attiva nessun segnalatore

EOF e condizioni di errori

- Le funzioni `feof` e `ferror` possono essere usate per controllare se si è verificata la condizione di errore
- La chiamata `feof(fp)` restituisce un valore nonzero se il segnalatore `EOF` è stato attivato dalla precedente operazione sullo stream `fp`.
- La chiamata `ferror(fp)` restituisce un valore nonzero se il segnalatore di errore di input è attivato

Esempi

- Realizziamo una funzione per caricare dei numeri interi da un file in un array
  - Assumiamo di conoscere la dimensione n (numero di elementi) del file
  - Assumiamo che gli interi nel file siano separati da un carattere newline (un intero per ogni riga)
- Realizziamo una funzione per scaricare il contenuto di un array in un file
  - Scriviamo un intero per riga (interi separati dal carattere newline)

La funzione finput_array

```
// file vettore.c

/* NB: la lettura degli elementi del file va normalmente fatta usando
   un ciclo while e deve terminare quando si verifica la condizione
   di EOF.
   In questo caso usiamo un for perché assumiamo che il file
   contenga gli n elementi richiesti in input per l'array a      */

void finput_array(char *file_name, int a[], int n)
{
    FILE *fd = fopen(file_name, "r");
    if( fd==NULL )
          printf("Errore in apertura del file %s \n", file_name);
    else {
          for(int i=0; i<n; i++)
                fscanf(fd, "%d", &a[i]);
          fclose(fd);
          }
}
```

Versione più robusta

```
/* In questa versione il numero di elementi da leggere n è un parametro di I/O.
   Se il file ha almeno n elementi, allora verranno letti i primi n elementi del file.
   Altrimenti n verrà modificato con il numero di elementi presenti nel file*/

void finput_array(char *file_name, int a[], int *n)
{
    FILE *fd = fopen(file_name, "r");
    if( fd==NULL )
        printf("Errore in apertura del file %s \n", file_name);
    else {
        int i = 0;
        fscanf(fd, "%d", &a[i]);
        while(i < *n && !feof(fd)) {
            i++;
            fscanf(fd, "%d", &a[i]);
        }
        fclose(fd);
        if(i < *n)        // abbiamo letto meno di n elementi
            *n = i;
        }
}
```

Schema di lettura sequenziale di un file

- Visita totale (leggiamo tutti gli elementi del file)
  -
```
int elem;
fscanf(fd, "%d", &elem);
while(!feof(fd)) {
    elabora elem;
    fscanf(fd, "%d", &elem); }
```
- Visita finalizzata (ci fermiamo quando si verifica una condizione)
  -
```
int elem
int trovato = 0;
fscanf(fd, "%d", &elem);
while(!feof(fd) && !trovato) {
    elabora elem;
    if(condizione(elem))
        trovato = 1;
    else fscanf(fd, "%d", &elem);
}
```

La funzione foutput_array

```
// file vettore.c

void foutput_array(char  *file_name, int  a[], int n)
{
    int i;
    FILE *fd;

    fd=fopen(file_name,  "w");
    if( fd==NULL )
        printf("Errore  in apertura  del file %s \n", file_name);
    else {
        for(i=0;  i<n; i++)
            fprintf(fd,  "%d\n", a[i]);
        fclose(fd);
        }
}
```

Come automatizzare il test

- Per automatizzare il test si possono usare i file per leggere dati di input e scrivere i dati di output
- Nel nostro esempio dell'ordinamento dell'array, per ogni test case, avremo in input (ad esempio per TC4):
  - Un file "TC4_input.txt" contenente gli elementi dell'array di input (uno per riga)
  - Un file "TC4_oracle.txt" contenente gli elementi dell'array ordinato (uno per riga) che ci si aspetta di ottenere (oracolo)
  - … oltre al numero di elementi da ordinare
- … e in output:
  - Un file "TC4_output.txt" risultante dall'esecuzione del programma (output effettivo)
  - … oltre ad una indicazione dell'esito del test (PASS / FAIL)

Dati di test: esempio per TC4 (con PASS)

TC4_input.txt	TC4_oracle.txt	TC4_output.txt
5	1	1
8	2	2
2	3	3
9	4	4
10	5	5
1	6	6
4	7	7
7	8	8
3	9	9
6	10	10
12	11	11
11	12	12

Dati di test: esempio per TC4 (con FAIL)

TC4_input.txt	TC4_oracle.txt	TC4_output.txt
5	1	5
8	2	8
2	3	2
9	4	9
10	5	10
1	6	1
4	7	4
7	8	7
3	9	3
6	10	6
12	11	12
11	12	11

Chiamata con argomenti su linea di comando

./test_ordina_array 12 TC4_input.txt TC4_oracle.txt TC4_output.txt

- test_ordina_array è il nome del programma eseguibile
- Gli argomenti TC4_input.txt e TC4_oracle.txt sono i nomi del file di input e dell'oracolo (e devono esistere all'atto dell'esecuzione)
- L'argomento TC4_outptut.txt è il nome del file di output che verrà creato dal programma di test
- L'argomento 12 è il numero di elementi contenuti nei file con nome TC4_input.txt e TC4_oracle.txt

Il nostro esempio con argomenti sulla linea di comando

```
// file test_ordina_array.c

# include <stdio.h>
# include <stdlib.h>
# include "vettore.h"

int main(int argc, char *argv[])
{
    if(argc != 5)
        printf("Numero parametri non corretto \n");
    else {
        int n = atoi(argv[1]);
        int *a = (int*) calloc(n, sizeof(int));
        if(a == NULL)
            printf("Memoria insufficiente \n");
        else {
            finput_array(argv[2], a, n);
            ordina_array(a, n);
            foutput_array(argv[4], a, n);

// continua sulla prossima slide confronto con oracolo …
```

continua

```
// continua  da slide precedente  …

                    int *oracle =  (int*) calloc(n, sizeof(int));
                    finput_array(argv[3], oracle, n);
                    if(confronta_array(a, oracle, n))
                              printf("PASS \n");
                    else printf("FAIL \n);
                    }
          }
}
```

La funzione confronta_array

```
// file vettore.c

// restituisce  1 se i due vettori di ingresso sono uguali, 0 altrimenti

int confronta_array(int  a[], int b[], int  n)
{
    int i =0;
    int trovato = 0;

    while (i < n && !trovato)
          if(a[i] != b[i])
                    trovato = 1
          else i++;

    return  (i == n) ? 1 : 0;
}
```

La funzione confronta_array

```
/* restituisce 1 se i due vettori di ingresso sono uguali, 0 altrimenti

    versione con short circuit evaluation       */


int confronta_array(int a[], int b[], int n)
{
    int i =0;

    while (i < n && a[i] == b[i])        // short circuit evaluation
        i++;

    return (i == n) ? 1 : 0;
}
```

… ricapitolando, nel nostro piccolo progetto di esempio abbiamo i files: (1 di 2)

- **File di interfaccia (header files)**
  - **utile.h** interfaccia del modulo utile, contiene la funzione scambia()
  - **vettore.h** interfaccia del modulo vettore, contiene tutte le funzioni realizzate per manipolare gli array
- **File sorgenti .c**
  - **utile.c** realizzazione del modulo utile
  - **vettore.c** realizzazione del modulo vettore
  - **test_ordina_array.c** contiene la funzione main()

… ricapitolando, nel nostro piccolo progetto di esempio abbiamo i files: (2 di 2)

- **Casi di test (creati a mano)**
  - **TC1_input.txt** contengono i valori da inserire nel vettore nei
  - **…….** 4 casi di test
  - **TC4_input.txt**

- **Oracoli (creati a mano)**
  - **TC1_oracle.txt** contengono i corretti valori attesi nei
  - **…….** 4 casi di test
  - **TC4_oracle.txt**

- **Output (creati dal programma)**
  - **TC1_output.txt** contengono i valori prodotti dalla esecuzione
  - **…….** del programma nei 4 casi di test
  - **TC4_output.txt**

Il Makefile del nostro esempio

```
test_ordina_array.exe:     utile.o  vettore.o  test_ordina_array.o
        gcc utile.o   vettore.o  test_ordina_array.o  -o test_ordina_array.exe

utile.o:            utile.c
        gcc -c utile.c

vettore.o:          vettore.c  utile.h
        gcc -c vettore.c

test_ordina_array.o:        vettore.h  test_ordina_array.c
        gcc -c test_ordina_array.c

clean:
        rm -f utile.o   vettore.o  test_ordina_array.o test_ordina_array.exe
```

Per compilare il programma

- Se ho creato il Makefile
  - make test_ordina_array.exe
- Se non ho creato il Makefile
  - Compilazione dei moduli e produzione dei file oggetto .o
    - gcc -c utile.c vettore.c test_ordina_array.c
  - Collegamento dei moduli oggetto e creazione dell'eseguibile
    - gcc utile.o vettore.o test_ordina_array.o –o test_ordina_array.exe

Per eseguire il programma

- Esecuzione del programma sui 4 casi di test e creazione dei file di output
  - ./test_ordina_array.exe 1 TC1_input.txt TC1_oracle.txt TC1_output.txt
    - Crea TC1_output.txt
  - ./test_ordina_array.exe 9 TC2_input.txt TC2_oracle.txt TC2_output.txt
    - Crea TC2_output.txt
  - ./test_ordina_array.exe 10 TC3_input.txt TC3_oracle.txt TC3_output.txt
    - Crea TC3_output.txt
  - ./test_ordina_array.exe 12 TC4_input.txt TC4_oracle.txt TC4_output.txt
    - Crea TC4_output.txt

Come testare il programma sull'intera test suite (in un sol colpo)

- Usiamo due file aggiuntivi
  - un file di input test_suite.txt che indica per ogni test case, il nome del test case e il numero di elementi dell'array
    - L'insieme dei test case per un programma viene chiamato ***test suite***
  - un file di output result.txt che memorizza l'esito di ogni test case (PASS / FAIL)
- Nel nostro esempio …

test_suite.txt	result.txt
TC1 1	TC1 PASS
TC2 9	TC2 PASS
TC3 10	TC3 FAIL
TC4 12	TC4 PASS

Come scrivere il programma di test

- Aprire il file test_suite.txt in lettura e leggere le varie righe del file finché non si raggiunge la fine del file (EOF)
- Per ogni riga del file test_suite.txt
  - Usare il primo elemento della riga (l'identificativo del caso di test, es. per la prima riga TC1) per costruire le stringhe per i nomi dei file di input, oracolo e output (es. per la prima riga TC1_input.txt, TC1_oracle.txt, TC1_outptut.txt)
  - Eseguire il codice del programma di test visto in precedenza usando come numero di elementi da ordinare il secondo elemento della riga del file test_suite.txt e come nomi dei file di input, oracolo e output quelli costruiti in precedenza
  - Scrivere una riga nel file result.txt in base al confronto tra l'array di outptut e l'array contenente l'oracolo (es. per la prima riga "TC1 PASS" se il programma ha funzionato e "TC1 FAIL" se non ha funzionato)
- Farlo come esercizio …

Esercizi

- Realizzare il programma che testa l'intera test suite in maniera automatica
  - Passare come argomenti su linea di comando i nomi dei file contenenti la test suite (es. test_suite.txt) e i risultati del test (es. result.txt)
- Realizzare i programmi di test delle funzioni del modulo vettore utilizzando l'allocazione dinamica della memoria per gli array e il caricamento dei dati da file

Esercizi

- Aggiungere al modulo vettore (sia al file vettore.h che al file vettore.c) le seguenti funzioni
  - Funzione che prende in input un array di interi e restituisce la somma degli elementi dell'array
  - Funzione che prende in ingresso due array di interi e restituisce in uscita l'array che contiene come elemento di posizione i la somma degli elementi di posizione i degli array di input
  - Funzione che prende in ingresso due array di interi e e restituisce il prodotto scalare dei due array. Il prodotto scalare di due array a e b è definito come:

$$\sum_i a[i]*b[i]$$

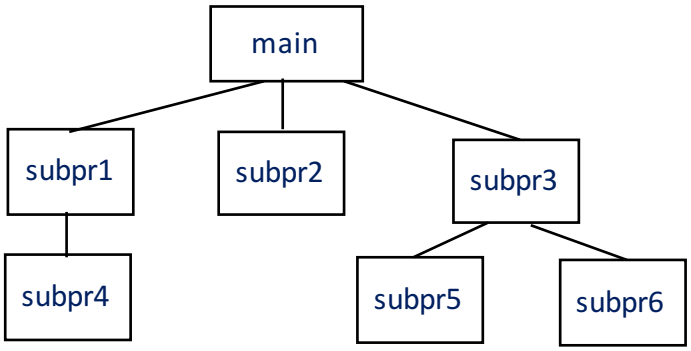
Esercizi su file

- Creare un modulo (file .c e file .h) contenente le seguenti funzioni sui file (di interi)
  - Funzione che prende in input un file e restituisce il numero di elementi del file
  - Funzione che prende in input un file e un intero x e restituisce 1 se il file contiene x, 0 altrimenti
  - Funzione che prende in input un file di interi e restituisce in output un file che solo i numeri positivi pari
  - Funzione che prende in input due file di interi f1 e f2 e restituisce in output un file contenente la sequenza degli elementi contenuti in f1 e f2
  - Funzione che prende in input un file di interi e restituisce in output un file contente gli opposti degli elementi del file di input
  - ...

E se il mio programma ha più funzioni ?

- ***Strategia big-bang***: integra il programma con tutti i sottoprogrammi e lo verifica nel suo insieme
  - Pessima strategia per programmi grandi: difficile localizzare la funzione contenente il difetto in caso di malfunzionamento …
- ***Strategie incrementali***: testare e integrare un sottoprogramma alla volta, considerando la struttura delle chiamate tra sottoprogrammi (architettura del programma)
  - **Bottom-up**
  - **Top-down**
  - **Sandwich**
  - **…**

Strategia bottom-up



- verificare prima i sottoprogrammi terminali (più in basso) e poi via via quelli di livello superiore …
  - **un sottoprogramma può essere verificato se tutti i sottoprogrammi che usa (chiama) sono stati verificati**

Strategia bottom-up e driver

- Per ogni sottoprogramma da verificare è necessario costruire un programma main (detto ***driver***) che:
  - acquisisce i dati di ingresso necessari al sottoprogramma;
  - invoca il sottoprogramma passandogli i dati di ingresso e ottenendo i dati di uscita;
  - visualizza i dati di uscita del sottoprogramma
- … usare la specifica del sottoprogramma per individuare i casi di test …