Fondamenti di Informatica e Laboratorio T-AB
Ingegneria Elettronica e Telecomunicazioni

Lab 15
Debugging

In teoria

Produrre un programma software consiste in

1. Analizzare il problema
2. Progettare l'algoritmo che lo risolve
3. Scegliere un linguaggio di programmazione
4. Codificare l'algoritmo nel linguaggio scelto

Al termine della fase di codifica il programma in teoria è pronto per l'utilizzo.

In pratica

In pratica dopo la prima stesura il programma presenta nella quasi totalità dei casi degli errori.

Questi devono essere risolti nella successiva fase di debugging che è parte integrante della produzione del software.

Tipologie di errore

Possiamo classificare gli errori di programmazione in 4 gruppi sufficientemente distinti:

- Errori di compilazione (compile time errors)
  - In pratica sono gli errori di sintassi
- Errori di collegamento (build time errors)
  - La sintassi dei singoli file .c e .h è corretta ma non lo sono i riferimenti fra di loro
- Errori di esecuzione (run time errors)
  - Il programma tenta di eseguire una operazione illecita
- Errori di logica applicativa :(very sad errors):
  - Il programma non si comporta come previsto

Contromisure generali

1. Studio
   - Es: qual è la differenza fra char c; / char* c; / char c[];
2. Stile
   - Indentazione / nomenclatura / commenti
3. Compiler/Linker Output
   - Warning / Errori / Warning / Warning
4. Debugger
   - Breakpoints / Call stack / Watches / …
5. Esperienza

Errori (quasi) di compilazione

In compilazione l'errore più grave e frequente è quello di trascurare i warning. Salvo casi rarissimi tutti i warning anticipano errori di runtime o logici.

Quindi è fondamentale analizzare TUTTI i warning, capirli e risolverli.

Il miglior modo di procedere è quello di analizzare errori e warning a partire dal primo (quello più in alto nel Build output) e dopo ogni soluzione effettuare una nuova compilazione.

Errori e warning di compilazione

error: '...' undeclared (first use in this function)

```
int main(...){
i=3+2; //Chi è i?
```

warning: implicit declaration of function '...'

```
#include <stdio.h>
int main(){
float x = miafunzione(1,2); //catastrofe!!!
```

Errori e warning di compilazione

warning: initialization makes integer from pointer without a cast

```
char c = "\n"; //forse ‘\n’?
```

warning: control reaches end of non-void function

```
int main(...){
    printf("hello");
} //dov’è il return?
```

Errori e warning di compilazione

warning: unused variable '...'

warning: variable '...' set but not used

O la variabile non serve o abbiamo dimenticato qualcosa

error: expected declaration or statement at end of input

error: expected ';' before `...`

error: parse error before `...'

Manca qualche `{` o `}` o `;`?

Errori e warning di compilazione

error: conflicting types for '...'

Quante volte abbiamo dichiarato la funzione ‘...’? È sempre uguale?

fatal error: '___.h': No such file or directory

nella #include verificare di aver usato doppi apici, verificare il nome del file, verificare da file system che il file sia realmente nella cartella dove deve essere

Errori di collegamento

Errori che avvengono dopo la compilazione del singolo file nel tentativo individuare nei diversi file inclusi le funzioni che abbiamo usato nel nostro programma

`error: undefined reference to '...'`

`C'è una funzione che stiamo usando e non abbiamo` **`definito`**

`Spesso è anticipato in compilazione da:`

`warning: implicit declaration of function '...'`

Errori di esecuzione

Una volta che la build è andata a buon fine, se siamo ‘fortunati’, provando ad eseguire il nostro programma incapperemo in situazioni che lo terminano inaspettatamente.

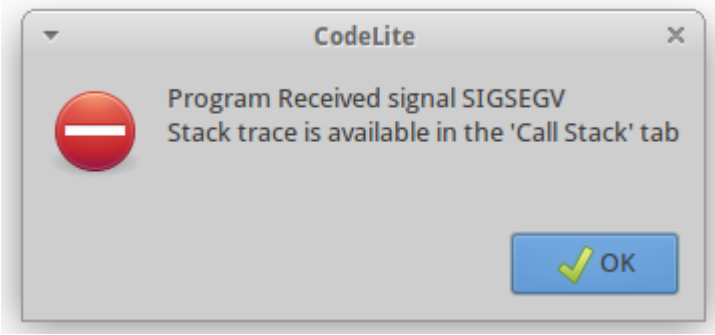
Prima della terminazione, in console, può essere stampato un messaggio. Ad es.:

- floating point exception
- segmentation fault
- out of memory (oppure killed)

Errori di esecuzione

Se lanciamo il programma in Debug da codelite, invece del messaggio di errore in console, si apre una maschera in codelite (di solito).

Dopo l'OK codelite ci mostra la posizione del codice in cui si è verificato l'errore e ci permette di analizzare call-stack e valore delle variabili



Errori di esecuzione

- floating point exception

È un errore che in pratica corrisponde sempre ad una divisione intera per 0

- Divisione: int i = 5/0;
- Modulo: int i = 5%0;

- Attenzione: se l'operazione non è intera in C non si ha errore ma esistono i valori **inf** e **nan.** L'esecuzione continua!!!

Errori di esecuzione

- segmentation fault

Sintetizza i casi di tentativo di accesso illecito alla memoria

- Accesso ad un vettore oltre i suoi indici minimo e massimo
- Utilizzo di un puntatore non inizializzato (anche puntatore a FILE)
- Utilizzo di un puntatore per cui è già stata chiamata la free()
- Incremento ed utilizzo di un puntatore oltre la memoria allocata
- Eccessivo numero di chiamate a funzione nello stack
- […]

Errori di esecuzione

- out of memory (killed)

Una malloc non riesce ad allocare la memoria richiesta perché insufficiente

- Raramente capita se il programma ha davvero bisogno di più memoria di quella disponibile (fisica e virtuale)
- Più spesso è dovuto:
  - ad un errore nel calcolo dei byte da allocare
  - alla omissione della free quando dovuto

Errori logici

Quando il programma continua la sua esecuzione senza errori bloccanti ma si comporta in modo imprevisto siamo in presenza di errori logici.

Gli errori logici sono specifici di ciascun programma e dipendono strettamente dal problema che si vuole risolvere.

Gli errori logici sono i più difficili da individuare e correggere.

Errori logici

Alcuni errori logici frequenti:

- Errata formattazione dell'input (uso %f per leggere un intero o %d per leggere un float)
  - GUARDARE TUTTE LE _printf e _scanf 5 VOLTE!
  - Riguardarle ogni volta che si cambia qualcosa nei loro dintorni
- Mancato consumo di caratteri dal buffer di input
  - Sintomo: in genere il programma non non attende l'immissione di input da parte dell'utente
  - Controllare bene le _scanf("%c",…)

Errori logici

Alcuni errori logici frequenti:

- Applicazione di una divisione intera quando ne sarebbe servita una non intera
  - GUARDARE TUTTE LE DIVISIONI 10 VOLTE!
  - Ricordate che è la divisione ad essere intera e non l'eventuale successivo assegnamento
- Arithmetic overflow
  - Ciascun tipo numerico ha i suoi limiti minimo e massimo
  - Un int ad esempio va da -2147483648 a 2147483647
  - Supere i limiti è come fare un giro di orologio con la lancetta dei minuti, ma nessuno ci avvisa che è cambiata l'ora

Errori logici

Alcuni errori logici frequenti:

- Generazione di un loop infinito
  - Durante il debug, se il programma sembra fermo a non fare niente, mettere un breakpoint in ogni ciclo sospetto (e non sospetto)
  - Verificare bene le condizioni di ciclo, e nel corpo del ciclo se ci sono delle istruzioni condizionali, assicurarsi che vengano opportunamente aggiornati gli indici in tutti i casi possibili

Esercizio 0

Scrivere un programma che legga, tramite un'apposita funzione *read(..)*, da input una serie di interi (al più 10) e li inserisca in un array. La lettura può terminare prima se l'utente inserisce un valore minore ad 1.

Il programma poi (nel main) dovrà stampare a video tutti i numeri diversi da 5 memorizzati nell'array.

Esercizio 0

- Scaricate l’esercizio 0
- Il programma dovrebbe leggere dati da un file e stamparli, ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 1

Scrivere un programma che richieda all'utente di inserire degli interi (massimo 5) e li inserisca in un vettore.

La lettura può terminare in anticipo e l'utente inserisce uno zero: in tal caso gli elementi rimanenti del vettore devono risultare inizializzati a "-1".

Il programma stampi poi a video gli elementi inseriti dall'utente nel vettore (fino alla dimensione logica: si escludano gli eventuali "-1").

Si verifichi il funzionamento del programma con l'input: 1, 2, 3, 0

Esercizio 1

- Scaricate l’esercizio 1
- Il programma dovrebbe leggere dati da un file e stamparli, ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 1

L’output corretto è:

```
Inserire un numero (0 per terminare): 1
Inserire un numero (0 per terminare): 2
Inserire un numero (0 per terminare): 3
Inserire un numero (0 per terminare): 0
I numeri inseriti sono: 1 2 3
```

Esercizio 2

Scrivere un programma che, dato un array v di dimensione 6, stampi a video i soli numeri pari presenti nell'array. Per fare ciò il programma utilizza una funzione definita in un modulo separato *functions.h/functions.c* che si occupa di scrivere in un secondo vettore v2 i soli numeri pari presenti in v. Tale funzione dovrà anche restituire il numero di elementi presenti in v2. La dichiarazione della funzione è:

```
int filtra_pari(int* v, int dim, int* v2);
```

Esercizio 2

- Scaricate l’esercizio 2
- Il programma dovrebbe leggere dati da un file e stamparli, ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 2

L’output corretto è:

```
Esercizio 2
v2: -2 4 6
```

Esercizio 3

Si scriva un programma che legga da input il contenuto del file “data.txt” (fornito insieme all’esercizio) e lo stampi a video.

Ogni riga del file è strutturata come segue:

`codice tipo valore`

Dove “codice” è una stringa di 4 caratteri, “tipo” è una stringa di 5 caratteri e “valore” è un intero.

I dati devono essere memorizzati in una opportuna struttura “Item” (definita mediante typedef).

Esercizio 3

- Scaricate l’esercizio 3
- Il programma dovrebbe leggere dati da un file e stamparli, ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 3

L'output corretto è:

```
LIST OF ITEMS
code: S001, type: TYPE0, val: 2
code: S002, type: TYPE2, val: 4
code: S003, type: TYPE1, val: 9
code: S004, type: TYPE1, val: 2
code: S005, type: TYPE2, val: 9
```

Esercizio 4

Si scriva un programma che legga da input (due volte) il contenuto del file “data.txt” (fornito insieme all’esercizio) e lo stampi a video (due volte).

Ogni riga del file è strutturata come segue:

```
codice tipo valore
```

Dove “codice” è una stringa di 4 caratteri, “tipo” è una stringa di 5 caratteri e “valore” è un intero.

I dati devono essere memorizzati in una opportuna struttura “Item” (definita mediante typedef).

Esercizio 4

La lettura deve essere effettuata dalla funzione

```
Item* readItems(char* fileName, int* dim)
```

Che deve allocare dinamicamente un vettore di dimensione adeguata a memorizzare tutti i dati e restituire l'indirizzo della sua prima cella. L'intero "dim", passato per riferimento, dovrà contenere al termine della funzione il numeri di dati letti.

La funzione deve essere definita nel modulo "functions.h/functions.c"

Esercizio 4

La stampa dovrà essere effettuata dalla funzione:

```
void printItem(Item o)
```

Che riceve in ingresso un “Item”, passato per valore.

La funzione deve essere definita nel modulo “functions.h/functions.c”

Esercizio 4

- Scaricate l’esercizio 3
- Il programma dovrebbe leggere dati da un file (due volte) e stamparli (due volte), ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 4

L'output corretto è:

```
ITEMS1:
S001 TYPE0 2
S002 TYPE2 4
S003 TYPE1 9
S004 TYPE1 2

ITEMS2:
S001 TYPE0 2
S002 TYPE2 4
S003 TYPE1 9
S004 TYPE1 2
```

Esercizio 5

Si scriva un programma che sommi i numeri complessi “1 + 1*i”, “2 + 2*i” e “3 + 3*i”.

Si definisca mediante typedef una struttura “Complex” atta a memorizzare un numero complesso. La somma dovrà essere effettuata mediate la funzione:

```
Complex sum(Complex v[], int dim)
```

Definita nel modulo “functions2.h/functions2.c”

Esercizio 5

La funzione:

```
Complex sum(Complex v[], int dim)
```

Dovrà essere implementata mediante una chiamata alla funzione ricorsiva tail:

```
void sum_aux(Complex v[], int dim, Complex* s)
```

Definita nel modulo “functions1.h/functions1.c”. La funzione esegue la somma mediante chiamate ricorsive che incrementano via via il valore del parametro “s”, passato per riferimento.

Esercizio 5

- Scaricate l’esercizio 5
- Il programma dovrebbe sommare un vettore di numeri complessi utilizzando una funzione ricorsiva tail, ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 5

L’output corretto è:

```
Exercise 5
sum: 6.000000 + 6.000000 * i
```

Esercizio 6

Scrivere un programma che (tramite funzione *read*) legga da input un certo numero di valori e li memorizzi in un array *v*. Il numero di valori non è noto a priori e va chiesto all'utente.

Si definisca poi una funzione

```
float transform(int*v, int dim)
```

che, preso in ingresso il vettore e la sua dimensione, ne calcoli la media e la restituisca. Tale funzione dovrà anche memorizzare in ogni (tranne l'ultima) cella la somma della cella corrente e della successiva. La media va calcolata sui valori originali del vettore.

Infine il programma dovrà stampare a video la media ed il vettore calcolato.

Esercizio 6

- Scaricate l’esercizio 6
- Il programma dovrebbe leggere e poi modificare un vettore di interi di dimensione non nota a priori, ma contiene errori
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori

Esercizio 6

L’output corretto è:

```
Quanti numeri vuoi inserire?:4
1
2
3
4
la media è 2.500000
3
5
7
4
```

Esercizio 7

- Scaricate l’esercizio 7
- Capite dal codice cosa il programma dovrebbe fare
- Utilizzate il debugger (e la testa) per individuare e correggere gli errori