Debug di errori nella gestione della memoria in C

(G.Cabodi)

Questo documento ha come obiettivo fornire una semplice e concisa panoramica (non esaustiva) di problemi ed errori legati all'uso di puntatori e allocazione dinamica.

Premessa
La "correttezza" di un programma è un concetto *"relativo",* nel senso che un programma va considerato corretto in riferimento a un insieme di requisiti, obiettivi, formulati in modo più o meno rigoroso/formale. In altri termini, un programma è corretto quando realizza ciò che ci si aspetta e lo fa senza errori: la descrizione di modi/metodi per formulare requisiti/specifiche di un programma, così come prassi e approcci per testarne la correttezza *NON sono obiettivo di questo documento*.

Obiettivi
Gli errori considerati in questo documento sono quelli che di solito generano ***"crash"*** del programma (il programma si "pianta", si interrompe), a causa di uso errato di puntatori e/o memoria (allocata in modo sia statico che dinamico). Ci si propone quindi semplicemente di rimuovere le ragioni di un crash, non necessariamente di far si che un programma realizzi effettivamente ciò per cui è stato scritto.

Attenzione: *nonostante quanto appena detto, è sempre bene aver chiaro che cosa si vuole realizzare nel programma. Difficilmente su può pensare di rimuovere gli errori senza aver la consapevolezza (almeno in parte) di ciò che si vuol realizzare nel programma. Si noti poi che gli strumenti che aiutano ad analizzare gli errori di memoria richiedono una certa consapevolezza nell'utilizzo: possono quindi rappresentare un'arma a doppio taglio, se utilizzati con imperizia.*

Il crash di un programma

Con il termine crash si intende una chiusura (terminazione) anomala (il programma si pianta/blocca/chiude improvvisamente) di un programma (il crash, più in generale, può essere anche a livello hardware, di sistema operativo, rete o altro), solitamente a causa di un problema hardware o software non gestito correttamente. I problemi legati alla gestione della memoria sono una delle possibili cause di crash, altre cause potrebbero essere: errori aritmetici (ad esempio divisione per 0), istruzioni illegali, violazioni di privilegio/sicurezza, problemi sul file system (ad esempio il tentativo di scrivere su un disco pieno), ecc.

Si veda ad esempio

https://en.wikipedia.org/wiki/Crash_(computing)

contenente una rapida e concisa panoramica sul concetto di crash.

Rimuovere una causa di crash implica di solito un ragionevole compromesso tra analizzare il programma e adottare opportune tecniche di debug: scoprire la vera causa di un crash non è infatti cosa banale, in quanto spesso un errore si manifesta non immediatamente, ma in modo indiretto e a valle dell'esecuzione di successive istruzioni.

La pagina web

https://www.eventhelix.com/embedded/debugging-software-crashes/

contiene una descrizione di diverse tipologie di software crash e di tecniche di debug utilizzabili per analizzare ed eventualmente correggere i problemi. Si noti che una parte significativa di tale pagina tratta di problemi legati alla memoria, descritti *successivamente* in questo documento.

Nei semplici programmi C affrontati nell'ambito del corso di Algoritmi e Strutture Dati, sono frequenti i crash causati da uso errato di puntatori e accessi a memoria.

Tipi di errori legati alla memoria.

I problemi di crash (software) legati all'uso della memoria, possono essere classificati in vari modi, ad esempio in base al tipo di memoria:

- globale
- heap
- stack

oppure in base al problema che scatena il crash

- memory corruption (scrittura in memoria e successivo utilizzo di dati non validi)
- storage violation (accesso illegale a memoria protetta e/o non accessibile)

Descrizioni delle diverse tipologie di errori sono reperibili, per memory corruption, su

https://en.wikipedia.org/wiki/Memory_corruption
https://www.proggen.org/doku.php?id=security:memory-corruption:start

mentre per la storage violation (e il cosiddetto "segmentation fault", uno dei tipi di storage violation) , su

https://en.wikipedia.org/wiki/Storage_violation
https://it.wikipedia.org/wiki/Errore_di_segmentazione

I "memory leak"

Si tratta di un caso molto particolare di errore, che tuttavia ***non genera solitamente un crash***, ma che, potenzialmente, innesca un sottoutilizzo della risorsa memoria, e quindi un potenziale degrado di prestazioni. In generale, i "leak" (perdite) sono legati all'uso dell'heap e

dell'allocazione dinamica, e quindi sono connessi a una o più mancate chiamate alla funzione "free". Attenzione, i **crash** legati alle chiamate di "free" non sono a rigore dei leak/leakage, ma problemi di "memory corruption" o "illegal storage", in quanto causati da un tentativo di liberare una parte di memoria non allocata o già liberata in precedenza.

Per una descrizione del problema dei memory leak, si vedano ad esempio le pagine web

https://it.wikipedia.org/wiki/Memory_leak
https://en.wikipedia.org/wiki/Memory_leak

Fare debug di errori di memoria

Obiettivo di ognuna delle descrizioni sopra citate dovrebbe essere quello di capire le tipologie dei potenziali problemi, per cercare di evitarli, tuttavia non sempre questo è possibile. Ciò fa si che, in molti casi, possa essere estremamente utile, oltre alle buone norme di programmazione e di revisione del codice scritto, l'utilizzo di opportuni strumenti di debug, orientati a individuare e rimuovere errori di memoria. Un debugger orientato a problemi di accesso a memoria è purtroppo uno strumento "intrusivo", nel senso che richiede di attivare, durante l'esecuzione di un programma, moduli software in grado di monitorare e verificare il corretto utilizzo delle varie aree di memoria su cui lavora un programma. In pratica il programma "gira" in un contesto diverso da quello privo di debugger e ciò talvolta può produrre tracce di esecuzione diverse, tra le modalità con e senza debug.

Una panoramica di debugger per problemi di memoria è reperibile su

https://en.wikipedia.org/wiki/Memory_debugger#List_of_memory_debugging_tools

Descrizioni di singoli strumenti sono consultabili su

https://valgrind.org/docs/manual/mc-manual.html

https://en.wikipedia.org/wiki/AddressSanitizer

https://www.cprogramming.com/tutorial/memory_debugging_parallel_inspector.html

Quasi tutti gli strumenti di debug della memoria includono una parte orientata ai leak. Una discussione sul problema specifico dei leak può ad esempio essere letta su

https://www.bogotobogo.com/cplusplus/CppCrashDebuggingMemoryLeak.php

Consigli pratici

Tra i vari strumenti di debug ci si limita a suggerire, ad un programmatore non ancora esperto, due degli strumenti (non commerciali e ampiamente disponibili) citati nel paragrafo precedente:

- valgrind
- address sanitizer

Valgrind

Si tratta di uno strumento open source, sviluppato in ambito Linux, ma disponibile su piattaforme Windows e Mac. Non è parte di un debugger, ma uno strumento esterno, un "profiler", che ha come scopo raccogliere statistiche di esecuzione di un programma, al fine di caratterizzarne le prestazioni.

Ad esempio, un programma eseguibile es2, avente due argomenti al main (ad esempio "a b") potrebbe essere eseguito, mediante valgrind, con il comando:

```
valgrind es2 a b
```

Valgrind include molteplici strumenti, quello adibito a verificare problemi di memoria si chiama "memcheck". Lo si può attivare, ad esempio (attenzione al doppio trattino --), con

```
valgrind –tool=memcheck es2 a b
```

Problema non trascurabile, per il programmatore inesperto, può essere l'interpretazione dei dati visualizzati da valgrind, nonché il fatto che l'individuazione di errori non è garantita al 100%.

Valgrind può essere integrato all'interno di altri strumenti di debug (es. Clion), a patto di effettuarne correttamente l'installazione.

AddressSanitizer

Si tratta di uno strumento sviluppato da Google, integrato nei compilatori Clang, GCC e Xcode, in modo tale che un programma, a patto di attivare le opportune opzioni di compilazione/link, venga eseguito in una forma in cui vengono controllati gli accessi alla memoria. Si noti che sono presenti più di un sanitizer (esiste ad esempio anche il leak sanitizer, ma non tutti sono presenti su tutte le piattaforme e i sistemi operativi)

AddressSanitizer può essere attivato ed utilizzato ad esempio, in CodeBlocks, Clion e Xcode, come descritto nel seguito

Compilazione in linea con GCC

Si può abilitare il sanitizer direttamente in compilazione sulla linea di comando gcc (quindi in ambito Linux, Mac, oppure Windows con ambiente Cygwin o WSL), aggiungendo le opzioni di compilazione:

```
-g –fsanitize=address
```

Se ad esempio si volesse generare l'eseguibile es1 a partire dai file es1.c e ts.c, si potrebbero usare i comandi:

```
gcc –c -g –fsanitize=address ts.c
```
(compila ts.c)

```
gcc –c -g –fsanitize=address es1.c
```
(compila es1.c)

```
gcc –o es1 -g –fsanitize=address ts.o es1.o
```
(link -> eseguibile es1)

oppure l'unico comando (per compilazione e link insieme)

```
gcc –o es1 -g –fsanitize=address ts.c es1.c
```

Come conseguenza, al termine dell'esecuzione vengono visualizzate statistiche sull'uso della memoria ed eventuali anomalie/errori

CodeBlocks

Occorre configurare il compilatore in modo da attivare il tool e fornire suficienti informazioni al debugger. Supponendo di utilizzare come compilatore GCC e come debugger GDB, occorre

- modificare le opzioni di compilazione in "`project->build options…->compiler settings`" (oppure "`settings->compiler`", qualora si voglia rendere l'opzione globale per tutti i progetti), selezionato il compilatore "`GNU GCC compiler`", nella sezione "`other compiler options`", inserire le opzioni:
  `-ggdb –fsanitize=address`
- modificare le opzioni di link ("`project->build options…->compiler settings`", oppure "`settings->compiler`", quindi selezionare "`linker settings`"): nella sezione "`other linker options`", inserire:
  `-lasan`
- In caso di crash, le informazioni sull'eventuale problema legato alla memoria vengono visualizzate sulla finestra di output (la console). Per evitare che, al termine dell'esecuzione, la finestra scompaia immediatamente, si consiglia di abilitare l'opzione "`Pause when execution ends`" nella finestra "`project->properties->Build targets`", sotto la selezione di "`Type: Console application`"

Clion

Le istruzioni su come abilitare i Google Sanitizers sono visibili su
https://www.jetbrains.com/help/clion/google-sanitizers.html

In breve:

- In windows occorre utilizzare `WSL-Ubuntu` (vedere https://www.jetbrains.com/help/clion/how-to-use-wsl-development-environment-in-product.html)
- in `CMakeLists.txt` occorre aggiungere una riga:
  `set(CMAKE_C_FLAGS="${CMAKE_C_FLAGS} -fsanitize=address -g")`
  Attenzione a usare eventualmente CMAKE_CXX_FLAGS nel caso si stia utilizzando il compilatore C++
  - in alternativa, in `Settings -> Build, Execution, Deployment -> CMake -> CMake options`, aggiungere
    `-DCMAKE_C_FLAGS="${CMAKE_C_FLAGS} -fsanitize=address"`
- Nelle preferenze (`Settings -> Build, Execution, Deployment -> Dynamic Analysis Tools -> Sanitizers`) abilitare `"Use visual representation for Sanitizer's output"`

ATTENZIONE: *evitare il copia/incolla diretto dal testo sopra, in quanto i doppi apici vengono spesso codificati in modo non compatibile con quanto richiesto da CMAKE!!! Meglio riscrivere oppure fare un copia incolla passando da un editor di puro testo.*

Xcode

In Xcode è sufficiente abilitare address sanitizer in `"Product->Scheme->Edit scheme"`, dove, per la configurazione Run oppure Test, tra le opzioni `"Diagnostics"`, è possibile configurare `"Runtime Sanitization"`