Vademecum dei tools

Passare file a stdin:
$> cat file_with_input | ./program_to_run
(gdb) run < file_with_input
valgrind ... ./program_to_run < file_with_input

Reindirizzare stdout su file:
$> cat file_with_input | ./program_to_run > file_to_write_into

GDB
GDB permette di guardare all'interno del vostro codice per capire cosa sta succendendo nel programma mano a mano che ogni istruzione eseguita.
Per utilizzare al meglio gdb è necessario compilare il vostro programma con i seguenti flag:

1. -g: produce informazioni utili per il debug del codice
2. -ggdb: produce informazioni specifiche per gdb

E' inoltre disponibile un altro flag (-ggdb3 al posto di -ggdb) che produce ancora più informazione.

GDB fa uso dei **breakpoint** per permettere all'utente di guradare lo stato del programma a un certo istante dell'esecuzione.
I breakpoint vengono inseriti in delle righe del codice specificate dall'utente e, quando vengono raggiunti durante l'esecuzione, questa viene messa in pausa dando la possibilità all'utente di controllare lo stato delle variabili.

Uso: gdb ./program_to_run

Comandi all'interno di gdb:

- **list**: stampa a schermo il codice del programma
- **run**: comincia l'esecuzione (si ferma al primo breakpoint incontrato
- **break *function_name/line_number* [if ...]**: inserisce un breakpoint alla riga specificata. E' possibile inoltre specificare delle condizioni tali per cui il programma si ferma solo in certi casi alla riga specificata.
  Esempio: Vogliamo controllare cosa succede nella quarta iterazione del ciclo

```
1. int a = 0;
2.
3. for int(a = 0; a < 10; a++){
4.     a++;
5. }
```

  comando: break 4 if a == 4

In questo modo GDB si arresterà nel momento in cui la riga 4 verrà eseguita nella quarta iterazione del ciclo for.

- **print *nome_variabile***: stampa il contenuto della variabile specificata
- **next**: esegue la riga di codice successiva
- **continue**: riprende l'esecuzione normale del codice dopo un breakpoint
- **info *opzione***: permette di stampare informazioni di vari elementi del programma (dopo aver scritto info premere il tasto TAB per vedere la lista di opzioni possibili.)
- **help**
- **q**: uscita da GDB

Links: https://medium.com/@amit.kulkarni/gdb-basics-bf3407593285

Valgrind

Valgrind permette di analizzare più o meno tutto quello che vi serve sapere del vostro programma (memory leak, stack delle chiamate, memoria allocata su stack e heap.
Valgrind ha un particolare flag **--tool** che permette di scegliere cosa andare ad analizzare nel programma.
Tools:

1. memcheck: rileva errori di memoria
2. massif: heap profiling
3. callgrind: stack trace delle chiamate a funzione

Lettura output:

1. memcheck: terminale/file di log specificato da riga di comando
2. massif:
    - ms_print *file_generato_da_massif*
    - massif-visualizer *file_generato_da_massif*
3. callgrind: kcachegrind *file_generato_da_callgrind*

Links: https://medium.com/@sukhbeerdhillon305/using-valgrind-8c0f394339ac

Uso: valgrind --tool=*toolname parametri programma*

Memcheck

E' il tool di default di valgrind, permette di trovare eventuali memory leaks e altri errori legati alla gestione della memoria dinamica.

Uso: valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./program_to_run

Paramentri:

1. --leak-check=full
2. --show-leak-kinds=all
3. --track-origins=yes
4. --verbose
5. --log-file=valgrind-out.txt
6. --help

Links
https://valgrind.org/docs/manual/mc-manual.html#mc-manual.options
https://stackoverflow.com/questions/5134891/how-do-i-use-valgrind-to-find-memory-leaks
https://medium.com/@sukhbeerdhillon305/using-valgrind-8c0f394339ac

Callgrind
Esegue una profilazione del codice permettendo di vedere quali sono le funzioni che impiegano la maggior parte del tempo di esecuzione

Uso:
valgrind --tool=callgrind --dump-instr=yes --callgrind-out-file=callgrind.out ./program_to_run
Poi per controllare il risultato:
kcachegrind callgrind.out

Links:
https://valgrind.org/docs/manual/cl-manual.html#cl-manual.options
https://developer.mantidproject.org/ProfilingWithValgrind.html

Massif
links:

- https://access.redhat.com/documentation/it-it/red_hat_enterprise_linux/6/html/performance_tuning_guide/ch05s03s03
- https://valgrind.org/docs/manual/ms-manual.html

Tool Opzionali

Address Sanitizer

Links: https://github.com/google/sanitizers/wiki/AddressSanitizer

Usage:
gcc … -fsanitize=address …

Perf

links: https://dev.to/etcwilde/perf---perfect-profiling-of-cc-on-linux-of

Verificatore

Possibili messaggi di errore e relative soluzioni

Output is not correct

Possibili cause:

- L'implementazione è errata
- Parsing errato dell'input
- L'output è stampato nel formato sbagliato

Soluzioni:

- Testing in locale con subtask pubblico
- Prestare attenzione ad eventuali "\n" o stampate di debugging

Execution timed out - Execution killed, violating memory limits

Cause:

- L'implementazione non soddisfa i vincoli di tempo o di memoria
    - NB: Il valore di tempo e memoria eventualmente riportato non è indicativo

Soluzioni:

- Debugging locale
    - Valgrind, callgrind, kcachegrind per individuare
        - Funzioni dispendiose di tempo / memoria
        - Memory leaks

Execution killed with signal 11

Cause:

- L'implementazione non soddisfa i vincoli di memoria
- Potrebbe essersi verificato un crash dovuto a un errore di segmentazione

Soluzioni:

- Debugging locale
    - Valgrind, callgrind, kcachegrind per individuare
        - Funzioni dispendiose di tempo / memoria
        - Memory leaks
        - Segfault

Execution failed because the return code was nonzero

Cause:

- Il main non ritorna 0
- Comportamento inaspettato della propria implementazione

Soluzioni:

- Inserire return 0
- Debugging locale