# Prova Finale Algoritmi e Strutture Dati

## Introduzione agli strumenti

Alessandro Barenghi

Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB)
Politecnico di Milano

alessandro -dot- barenghi - at - polimi -dot- it

# Scaletta

## Logistica

- Come accedere al verificatore e raggiungere i tutor

## Strumenti per lo sviluppo

- Editor e compilatore
- Strumenti di debugging

## Strumenti per la valutazione delle prestazioni

- Valgrind
  - callgrind
  - massif

# Memorandum scadenze

## Per laureandi a luglio

- 3 luglio, ore 23.59 CEST. Segnalate (email al docente) la necessità di valutazione
  - Il verificatore verrà temporaneamente chiuso (la riapertura è prevista nella giornata del 6 luglio) per consentire l'estrazione delle sottoposizioni.

## Per tutti gli altri

- 4 settembre, ore 23:59 CEST

## Per laureandi a gennaio (superato 145 CFU+ iscrizione ad appello di laurea)

- la piattaforma sarà riaperta per 10 giorni nella sessione d'esame invernale

# Accesso al verificatore

## https://dum-e.deib.polimi.it

### Utile da sapere

- Il verificatore termina automaticamente il vostro programma non appena eccede i limiti di tempo/memoria impostati
  - test "Open" volontariamente lasciato con margini molto più alti per valutare quanto più lento/grande del necessario è un programma
- Specifica, archivio di casi di test e generatore negli allegati del test "Open"
- Spiegazione dei messaggi di feedback dati dal verificatore: `https://dum-e.deib.polimi.it/documentation`
- Il progetto sarà annullato in caso di plagio, distribuzione dei sorgenti, tentativi manomissione della piattaforma (e.g., override opzioni di compilazione)

# Contatti tutor

- Sezione Barenghi:
  - Giuseppe Boccia giuseppe.boccia@mail.polimi.it
  - Matteo Cenzato matteo.cenzato@mail.polimi.it
- Sezione Martinenghi:
  - Cristian Lo Muto cristian.lomuto@mail.polimi.it
  - Giorgio Miani giorgio.miani@mail.polimi.it
- Sezione Pradella:
  - Giorgio Pristia giorgio.pristia@mail.polimi.it
  - Federico Toschi federico.toschi@mail.polimi.it
- Canale telegram: `https://t.me/+oaOqymJIMP0zMmU0`

# Ambiente integrato vs. strumenti separati

## Scelta dell'ambiente di sviluppo

- Premessa: la base di codice che svilupperete sarà piccola ($\leqslant$ 1000 SLoC)
- Per chi avesse già esperienza di programmazione robusta: usate pure l'ambiente che vi è più congegnale

## Due alternative

- Applicazioni separate: usare un compilatore, un editor di testo, un debugger
  - Meno integrazione, tutti i passi sono visibili
- Ambiente integrato di sviluppo (IDE)
  - Elevata integrazione, più difficile separare visivamente gli effetti dei vari passi

# Ambiente di sviluppo consigliato

## Sistema operativo

- una distribuzione Linux a piacere: Debian, Ubuntu, Fedora, Arch, Gentoo

## Ambiente di sviluppo

- Editor di testo: uno con con evidenziatore di sintassi; e.g., Kate, Vim, Emacs
- Compilatore: `gcc` è quello usato dal verificatore
  - le opzioni di compilazione sono `-Wall -Werror -std=gnu11 -O2 -lm`
  - il verificatore ne ha anche altre per consentire l'isolamento del processo, non servono
  - aggiungere l'opzione `-g3` aggiunge informazioni di debug al binario
- Un emulatore di terminale: quello di default della distribuzione va benissimo

# Flusso di sviluppo

## Flusso di sviluppo consigliato

1. Progettate la vostra soluzione su carta (o tablet, per quel che vale)
   - Pensate a quali strutture dati sono necessarie, come usarle, quali soluzioni algoritmiche sono le migliori
2. Sviluppate lo pseudocodice delle parti più impegnative
   - scritto anche solo in un file di testo, può essere trasformato in commenti nel sorgente
3. Implementate la vostra soluzione, effettuando test periodici di correttezza
4. Misurate le prestazioni concrete, analizzate colli di bottiglia, migliorate la complessità computazionale "alle costanti"

# Compilazione

## Opzioni di compilazione

- È conveniente usare opzioni di compilazione che mimano quelle del verificatore
- Mimare il verificatore: `gcc -Wall -Werror -std=gnu11 -O2 -lm test.c -o test`
  - il verificatore ne ha anche altre per favorire l'isolamento del processo, non servono
  - aggiungere l'opzione `-g3` aggiunge (utili) informazioni di debug al binario
- Opzionale: potete "ridurre" il comando di compilazione al minimo creando un file di testo chiamato `Makefile` che contiene i seguenti due righi:

  ```
  CFLAGS += -Wall -Werror -std=gnu11 -O2
  LDFLAGS += -lm
  ```

  e compilare con il comando `make programma` il vostro sorgente `programma.c`

# Esecuzione

## Meccanizzare input e output

- Il verificatore fornirà al vostro programma file i dati in ingresso via `stdin`: è utile evitare di doverli scrivere a mano quando fate prove in locale
- Fornire contenuto del file `file_ingresso` in input al programma `programma`
  → `./programma < file_ingresso`
- Fornire contenuto del file `file_ingresso` in ingresso al programma `programma` e salvarne l'uscita su file
  → `./programma < file_ingresso > file_uscita`
- Confrontare il contenuto di due file di testo
  → `diff ./public_output ./program_output`
  → vengono stampate solo le differenze: se identici non stampa nulla
  → Alternative grafiche: Meld e Kdiff

# Debugging - 1 - GDB

## Ispezionare lo stato a runtime

- Ispezionare lo stato di un programma durante la sua esecuzione può essere fatto
  - A colpi di `printf`: funzionale... fino ad un certo punto
  - Con un debugger: `gdb`, lo GNU Project Debugger

## Come usarlo

- Lanciate il programma desiderato con `gdb ./programma`
- Live demo
- Sommario dei comandi comuni:
  `http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf`

# Debugging - 2 - Address SANitizer (ASAN)

## Cos'è?

- Combinazione di passi aggiuntivi di `gcc` + libreria runtime
- Individua accessi a variabili fuori dai limiti *con precisione al singolo byte*
- Usa, se disponibili, le informazioni di debug per stampare il rapporto

## Come usarlo

- Aggiungete alle opzioni di compilazione `-fsanitize=address`
- Lanciate il programma come sempre: in caso di errore verrà interrotto
- Live demo

# Valgrind

## Valgrind

- Suite di strumenti per l'ispezione del comportamento di un programma
- Include sia strumenti per il debugging (memcheck), sia strumenti di misura delle prestazioni (cachegrind/callgrind, massif/dhat)
- Funziona istrumentando i programmi (= aggiungendo codice al loro interno prima di eseguirli), l'esecuzione viene rallentata (circa 2.5×)
  - L'istrumentazione è incompatibile con ASAN (fanno, in parte, lo stesso mestiere)
- Manuale di riferimento disponibile al `https://valgrind.org`

# Debugging - 3 - Memcheck

## Cos'è?

- È lo strumento della suite Valgrind che controlla a runtime se avvengono:
  - *Memory leaks* (memoria non usata e non deallocata
  - *Use-after-free* (accessi in lettura/scrittura a mem deallocata)
  - *Double-free* (doppie invocazioni di `free` sullo stesso ptr)
  - Letture da variabili non inizializzate

## Come usarlo

- Rimuovete alle opzioni di compilazione `-fsanitize=address` se c'è
- Lanciate il programma con: `valgrind ./programma`
- Per analisi più approfondita `--leak-check=full --show-leak-kinds=all`
- Per tracciare dove è stata allocata la mem. con errori `--track-origins=yes`
- Live demo

# Misurare le prestazioni di un programma

## Tempo

- Istrumentazione manuale del codice con `clock_gettime`/`rtdscp`
  - Fattibile, ma non necessaria in questo progetto
- Istrumentazione automatica con Valgrind: callgrind e cachegrind

## Spazio

- Utilizzo totale: `time` (BSD) dà una visione sintetica
  - per evitare conflitti con l'omonimo builtin di Bash, usare `/usr/bin/time`
- Massif: fornisce una visione dettagliata nel tempo

# Callgrind

## Cos'è?

- Callgrind istrumenta il codice aggiungendo punti di misura del tempo trascorso
- Produce un resoconto testuale (inteso per lettura meccanizzata)
- Se presenti, utilizza le informazioni di debug

## Come usarlo

- Rimuovete alle opzioni di compilazione `-fsanitize=address` se c'è
- Lanciate il programma con: `valgrind --tool=callgrind ./programma`
- Esaminate l'output: `kcachegrind callgrind.out.PID` dove `PID` è il Process ID
- Live demo

# Massif

## Cos'è?

- Stessa filosofia di callgrind, ma, ad ogni punto di misura, registra la memoria dinamica occupata
- Produce un resoconto testuale (inteso per lettura meccanizzata)
- Se presenti, utilizza le informazioni di debug

## Come usarlo

- Rimuovete alle opzioni di compilazione `-fsanitize=address` se c'è
- Lanciate il programma con: `valgrind --tool=massif ./programma`
- Esaminate l'output: `massif-visualizer massif.out.PID`
- Live demo

# Installazione strumenti

- Mono-comando per installare tutti i tool su Debian GNU/Linux:
  ```
  apt install gdb hotspot valgrind build-essential kcachegrind massif-visualizer
  ```
  - Ricordarsi di usare `su -` o `sudo` per acquisire diritti di amministrazione
- Lo stesso mono-comando installa i tool su Ubuntu