



SAPIENZA  
UNIVERSITÀ DI ROMA

## Integrazione e testing di ChatGPT in LLDB debugger: verso un debugging automatico

Facoltà di Ingegneria dell'Informazione Informatica e Statistica  
Laurea in Ingegneria dell'Informazione

**Carlo Capodilupo**

Matricola 1851916

Relatore

Prof. Giuseppe Antonio Di Luna

Anno Accademico 2022/2023

Tesi discussa il 26/10/2023  
di fronte a una commissione esaminatrice composta da:

Prof. Giuseppe Scotti (presidente)  
Prof.ssa Giuseppa Conte  
Prof. Emanuele De Santis  
Prof. Paolo Di Giamberardino  
Prof. Paolo Di Lorenzo  
Prof. Giuseppe Antonio Di Luna  
Prof.ssa Lara Pajewski

---

**Integrazione e testing di ChatGPT in LLDB debugger: verso un debugging automatico**

Tesi di Laurea. Sapienza Università di Roma

© 2023 Carlo Capodilupo. Tutti i diritti riservati

Questa tesi è stata composta con L<sup>A</sup>T<sub>E</sub>X e la classe Sapthesis.

Sito web: [carlo.capodilupo.eu](http://carlo.capodilupo.eu)

Email dell'autore: [capodilupo@proton.me](mailto:capodilupo@proton.me)

*A Dioniso*



## Sommario

Con l'esponenziale incremento della complessità del codice, l'utilizzo di un debugger diviene essenziale. Che si scriva in C o C++ per lo sviluppo di sistemi operativi, real-time, embedded o di videogiochi in cui sono richieste alte prestazioni e tempi di risposta rapidi, il debugging può essere reso più semplice, veloce ed efficiente sfruttando le risorse messe a disposizione dall'intelligenza artificiale.

Il progetto che presento si propone di soddisfare queste esigenze integrando, con un plugin scritto ad hoc, le funzionalità di ChatGPT (OpenAI) in LLDB, uno dei debugger di nuova generazione più utilizzati e altamente performanti.



# Indice

<b>1 Introduzione</b>	<b>1</b>
1.1 Motivazioni . . . . .	1
1.2 Requisiti . . . . .	3
1.3 Struttura della tesi . . . . .	5
1.4 Descrizione ad alto livello . . . . .	8
<b>2 Progetto</b>	<b>11</b>
2.1 Schema architetturale . . . . .	11
2.2 Schema delle dipendenze . . . . .	13
2.3 Diagramma degli stati . . . . .	16
2.4 Comandi . . . . .	18
2.4.1 mode . . . . .	18
2.4.2 switch . . . . .	18
2.4.3 code . . . . .	18
2.4.4 send . . . . .	18
2.4.5 ask . . . . .	19
2.4.6 wait . . . . .	19
2.4.7 autodebug . . . . .	19
2.4.8 save . . . . .	22
2.4.9 load . . . . .	22
2.4.10 export . . . . .	22
2.4.11 import . . . . .	22
2.4.12 clear . . . . .	22
2.4.13 info . . . . .	23
2.5 Interazione con ChatGPT: Prompt . . . . .	23
<b>3 Commento del codice</b>	<b>27</b>
3.1 Struttura delle directory . . . . .	27
3.2 Directory 'chatgpt-lldb-plugin' . . . . .	29
3.2.1 Sottodirectory 'main' . . . . .	29
3.2.2 __init__.py . . . . .	29
3.2.3 api_key_config.py . . . . .	29
3.2.4 browser_config.py . . . . .	29
3.2.5 LICENSE . . . . .	30
3.2.6 README.md . . . . .	31
3.2.7 requirements.txt . . . . .	33

3.2.8	setup.py . . . . .	33
3.3	Sottodirectory 'main' . . . . .	36
3.3.1	Sottosottodirectory 'api_connection' . . . . .	36
3.3.2	Sottosottodirectory 'browser_connection' . . . . .	36
3.3.3	Sottosottodirectory 'saves' . . . . .	36
3.3.4	__init__.py . . . . .	36
3.3.5	commands.py . . . . .	36
3.3.6	main.py . . . . .	46
3.3.7	prompts.py . . . . .	46
3.3.8	stepping.py . . . . .	50
3.3.9	version.py . . . . .	57
3.4	Sottosottodirectory 'api_connection' . . . . .	58
3.4.1	__init__.py . . . . .	58
3.4.2	api.messaging.py . . . . .	58
3.5	Sottosottodirectory 'browser_connection' . . . . .	60
3.5.1	__init__.py . . . . .	60
3.5.2	browser.messaging.py . . . . .	61
3.5.3	cleaner.py . . . . .	68
3.5.4	previous_chrome_pids.yaml . . . . .	69
3.6	Sottosottodirectory 'saves' . . . . .	69
3.6.1	dbChat_test.c.yaml . . . . .	69
<b>4</b>	<b>Risultati sperimentali</b>	<b>73</b>
4.1	Insieme dei dati . . . . .	73
4.2	Descrizione degli esperimenti . . . . .	75
4.3	I Sperimentazione . . . . .	75
4.3.1	Errori di logica, sintassi e semantica . . . . .	76
4.3.2	Buffer overflow . . . . .	82
4.3.3	Divisione per zero . . . . .	85
4.3.4	Accesso a memoria non allocata o non valida . . . . .	86
4.3.5	Puntatori nulli . . . . .	90
4.3.6	Overflow aritmetico . . . . .	90
4.3.7	Underflow aritmetico . . . . .	93
4.3.8	Ricorsione infinita . . . . .	94
4.3.9	Accesso a variabili non inizializzate . . . . .	94
4.3.10	Accesso fuori dai limiti dell'array . . . . .	101
4.3.11	File non trovato o non accessibile . . . . .	105
4.3.12	Comportamento non definito . . . . .	105
4.3.13	Loop infiniti . . . . .	109
4.4	II Sperimentazione . . . . .	112
4.4.1	Test 1 . . . . .	114
4.4.2	Test 2 . . . . .	117
4.4.3	Test 3 . . . . .	120
4.4.4	Test 4 . . . . .	123
4.5	Analisi dei risultati . . . . .	126
<b>5</b>	<b>Conclusioni</b>	<b>131</b>





# Capitolo 1

## Introduzione

### 1.1 Motivazioni

Ogni anno vengono prodotte più di 100 miliardi di linee di codice. Si stima che ogni 1000 di queste vi siano 25 bug.[18, 24] Fino a qui si potrebbe circonscrivere la correzione degli errori del codice attorno all’etica del programmatore[12], l’ingegnere del software è bene che si preoccupi di scrivere un codice chiaro, sviluppando sistemi modulari, scalabili e senza bug. Eppure per quanto possa essere diligente nel suo lavoro di scrittura, non può esimersi dal commettere degli errori[31]. Alcuni di essi, i più subdoli, possono essere difficili da scovare tanto che anche attraverso l’utilizzo di un debugger la procedura risulta lunga e dispendiosa[9] e talvolta si può anche arrivare a classificare come erronee parti di codice corrette del programma[14]. Il discorso si complica quando subentrano le scadenze e la competitività aziendale, infatti molto del codice che viene prodotto presenta errori che sebbene non vengano rilevati subito portano a perdite consistenti[19]. Soprattutto in un settore come quello dell’Information Technology che è in costante crescita. Dei 20 trilioni di dollari di GDP, con riferimento agli USA nel 2020, 1.4 di questi, quindi ben il 7%, riguarda le attività legate all’IT. Una parte considerevole, ben 607 miliardi di dollari sono spesi soltanto per trovare e sistemare i bug presenti nel codice.[18] Tutto ciò avviene senza considerare il costo dei crimini informatici che proliferano lì dove i bug sono numerosi[27], mettendo così a rischio sistemi importanti non solo da un punto di vista economico ma anche strategico e per il quale ne va della sicurezza del cittadino. Il debugging non è però cosa semplice, e per quanto ci si possa impegnare la questione potrebbe andare oltre le nostre capacità. Il debugging è due volte più difficile rispetto alla stesura del codice. Segue che chi scrive il codice nella maniera più intelligente possibile non è, per definizione, abbastanza intelligente per eseguirne il debugging.[16]. Pongo qui il mio punto: se il debugging è due volte più difficile rispetto la stesura del codice e chi scrive il codice il più intelligentemente possibile non può essere abbastanza intelligente per eseguirne il debugging resta aperta la possibilità di assottigliare il gap tra intelligenza del programmatore e intelligenza necessaria per il debugging grazie all’utilizzo di un’intelligenza artificiale, possibilmente portandolo a zero. A livello concettuale è coerente, lo è almeno fintanto che non si utilizza l’IA per scrivere il codice. Nella remota ipotesi della creazione di un’intelligenza artificiale forte[26] resta pur sempre la possibilità di avvalersi

dell'IA odierna quale strumento di supporto al debugging[13]. Una possibilità concreta, basata su modelli generativi[15] e in grado aiutare l'ingegnere del software nella revisione del codice. Possibilità la cui concretizzazione è motivata da ragioni etiche, economiche, strategiche e di sicurezza sia privata che pubblica. Da qui l'idea di integrare ChatGPT[7] di OpenAI in un debugger. OpenAI[22] si presenta infatti come una delle realtà al momento più innovative e con ampie potenzialità di crescita, capace di garantire un'interazione con il ChatBot sia tramite un'interfaccia web che attraverso le API Python rilasciate dall'organizzazione stessa. Quanto al debugger, una scelta ottimale si è rivelata LLDB Debugger[29], uno dei debugger più performanti per il linguaggio C e derivati, che consente la creazione di comandi ad hoc scritti in Python grazie all'utilizzo delle API LLDB[1]. Il ruolo del debugger può risultare essenziale in quanto grazie ad esso si potrebbero ricavare tutte quelle informazioni che vengono fuori dall'esecuzione del programma e consentirebbero una circoscrizione mirata dell'errore da parte dell'IA rivelandosi cruciali[6] nella rilevazione dei bug, al contrario di come si avrebbe se fosse tenuto da conto solo il codice. Tali informazioni si ricaverebbero mettendo in esecuzione il programma e vedendo l'output generato da questo. In caso di esecuzione senza interruzioni o meno si avranno situazioni diverse. Nel secondo caso diviene molto più utile leggere i valori delle variabili e conoscere la linea corrente lungo la quale l'esecuzione si è bloccata, la ragione di tale blocco e il valore delle variabili in quel punto. Per quanto possa infatti essere capace l'IA, bisogna tenere presente che non si ha a che fare con un sistema fatto per porre in esecuzione un'applicazione, pertanto, affinché ChatGPT risulti efficace bisogna fornirle un contesto e delle informazioni di debugging tratte da una precedente esecuzione del programma così che possa poi "ragionarvi" sopra e trarre le sue conclusioni. Senza queste informazioni, maggiore il grado di complessità, maggiore il rischio di ricevere risposte incomplete o approssimative. Tra le informazioni utili, da formattare opportunamente[23], sicuramente c'è anche il sistema che si sta utilizzando, in modo da considerare gli aspetti di gestione del software e compilazione[30] che variano da un sistema operativo all'altro. Ci si può quindi avvalere di uno strumento quale un ChatBot, ma ciò risulta veramente utile se si agisce ingegneristicamente, ovvero ottenendo il massimo con ciò che si ha a disposizione. Un'IA generativa risulta poco utile se le si vuole far porre in esecuzione un programma al fine di conoscere l'output, tuttavia è in grado di fare osservazioni interessanti avendo a disposizione una mole significativa di informazioni e una capacità computazionale che diventa importante con la divisione del carico lavorativo da processare sui vari nodi del servizio. Per ottenere supporto nel debugging allora, fornire informazioni aggiuntive non disponibili se non a chi esegue il programma e lo fa passo dopo passo è sicuramente la soluzione ottimale per scovare i bug del codice, il che potrebbe aiutarci a risolvere anche i casi più complessi. Appare quindi come soddisfacentemente motivata l'idea di utilizzare un'IA quale supporto per il rilevamento di errori nel codice[5] se integrata in un debugger attraverso un'applicativo capace di ricavare le informazioni necessarie al debugging automatizzato.

## 1.2 Requisiti

L'applicativo deve poter essere integrato in LLDB[4] e garantire l'utilizzo di ChatGPT all'interno del debugger. Entrambe richieste soddisfabili attraverso l'utilizzo del linguaggio di programmazione Python[28] grazie alle API[1, 2] che consentono l'interazione con LLDB e ChatGPT, rilasciate dalle rispettive case madri. L'utilizzo di Python, inoltre, se abbinato all'utilizzo di metodi indipendenti rispetto al sistema operativo, garantisce l'indipendenza dell'applicativo rispetto al sistema operativo. L'applicativo deve poi poter recuperare, noto l'obiettivo (target) del debugger: il nome del file, il codice del file sorgente associato, la linea corrente, le variabili visibili, le info di sistema e la versione del debugger. Il plugin deve poter raggruppare e organizzare queste informazioni in un prompt efficace, queste possono poi essere inviate a ChatGPT la quale può poi adempiere alle nostre richieste. Bisogna quindi sviluppare quella parte di codice che gestirà la comunicazione con l'IA, può avvenire sia attraverso l'utilizzo delle API di OpenAI che attraverso un Browser Web. In entrambi i casi si hanno vantaggi e svantaggi, il consumatore potrà quindi valutare il giusto compromesso. Nel primo si tratta di una versione a pagamento, più rapida e che offre anche la possibilità di salvare localmente le conversazioni, ma con una necessaria ridondanza dei messaggi che fa da "memoria" al ChatBot; nel secondo si ha gratuità del servizio a scapito di un tempo di attesa generalmente maggiore, ma regolabile, e del salvataggio remoto delle chat che risolve l'invio ridondante delle informazioni. Bisogna inoltre tenere da conto che al di là dell'aspetto economico e di invio delle informazioni, l'utilizzo delle API consente di ricevere risposta non appena ChatGPT l'ha emessa, mentre nella modalità browser si dovrà aspettare un tempo stimato entro il quale ChatGPT riceve ed elabora la risposta. Un tempo di attesa troppo ridotto ci impedirebbe di ricevere risposte complete, ma uno troppo lungo ci darebbe sì risposte complete ma in un tempo non dà tutti accettabile. Diviene quindi fondamentale garantire la regolabilità di questo tempo per la versione browser. Dovrà anche essere utilizzato un browser noto per le sue prestazioni veloci e una libreria che possa legare le attività del browser al codice Python in cui è scritto il programma. C'è da riflettere anche sull'aspetto dei dati in locale e in remoto, discorso legato alle capacità di "memorizzazione" di ChatGPT. Il bot non può ricordare nulla quando si utilizzano le API, bisogna infatti salvare messaggio dopo messaggio la conversazione in una struttura dati quale può essere una lista di dizionari e inviare ad ogni interazione tutti i messaggi precedenti, il bot non ha infatti memoria. L'aspetto negativo è che tanto più cresceranno i messaggi nella comunicazione tanto più sarà la mole di dati da inviare, tuttavia non vi è altro modo per tenere traccia della conversazione attraverso l'utilizzo delle API OpenAI. Tale aspetto tuttavia è del tutto compensato dalla comunicazione via browser poiché la conversazione è salvata automaticamente nei server di OpenAI e questo fa sì che si possa inviare solo l'ultimo messaggio e non questo assieme a tutti i precedenti. In entrambe le modalità può essere utile salvare la conversazione, onde riprenderla in sessioni successive di debugging. La modalità browser quindi risulta più conveniente sia per gratuità che per ottimizzazione del carico inviato, ma nonostante tutto, è comunque più lenta dell'utilizzo della modalità con le API ufficiali. L'utilizzo di Chrome Browser consente di mitigare per quanto possibile questo effetto, essendo uno dei browser più veloci in circolazione. Con il Chromedriver corrispondente e il pacchetto Python Selenium si

può inoltre interagire con l’interfaccia web attraverso la scrittura di codice così da rendere possibile l’implementazione della modalità browser. Ricapitolando l’utilizzo di due modalità differenti garantisce versatilità e un’offerta capace di adattarsi alle richieste dei consumatori, che siano aziende, privati o studenti, magari coadiuvata una configurazione semplice che consenta di specificare la propria API key in un caso o le impostazioni browser nell’altro. Ma come formalizzare la comunicazione con ChatGPT? Quando si utilizza un debugger si imposta un breakpoint, magari nel main con “`b -name main`”, si lancia l’esecuzione e poi si procede eseguendo istruzioni dopo istruzione con il comando “`step`” o “`next`” a seconda del fatto che si voglia entrare o meno nelle chiamate di funzione all’interno della funzione corrente. Giunti quindi ad un punto di nostro gradimento, qui appare ragionevole poter inviare tutte le informazioni raccolte precedentemente senza ricevere alcun feedback dal ChatBot, poi successivamente facendo le nostre richieste personalizzate con un secondo comando. In seguito potremo ripetere il procedimento, magari riutilizzando il primo comando con l’accortezza di evitare l’invio delle informazioni generali di sistema, la versione del debugger oltre che il codice del file sorgente dell’obiettivo di debugging avendoli già mandati nella prima interazione. Si devono quindi poter inviare informazioni e fare delle richieste personalizzate fino alla rimozione dei bug. Risulta utile poter salvare la chat in locale, in una directory a scelta, o in remoto e anche caricare tale chat oltre che eliminarla se non più utile. Interessante è poter vedere quale sia l’attuale modalità di utilizzo, se API o browser, e poter passare dall’una all’altra. Infine, per chi non ha particolare dimestichezza del debugging si potrebbe implementare un comando unico e comodo in grado di automatizzare il debugging attraverso l’invio delle informazioni essenziali e la formulazione di richieste specifiche, il tutto nei punti critici del programma come può essere lo stop dello stesso a seguito di un segnale quale può essere un `SIGABRT` o un `SIGSEGV`. Un comando per controllare tutto, la strada di accesso verso il debugging automatico.



**Figura 1.1.** Struttura stratificata del sistema

## 1.3 Struttura della tesi

Il documento di tesi è così strutturato:

### 1. Introduzione

Qui si introduce la tesi del candidato. Il capitolo è articolato in 3 punti:

#### (a) Motivazioni

Si presenta la necessità di migliorare il debugging a seguito della presenza costante di bug nel codice scritto dai programmati analizzando le statistiche correnti. Indipendentemente da quanto intelligentemente si possa programmare il debugging risulta molto più difficile. L'ottimizzazione di questo processo dà innumerevoli vantaggi sul piano economico e della sicurezza pubblica e privata. L'intelligenza artificiale può risultare un valido strumento per l'efficientizzazione del debugging.

#### (b) Requisiti

Si definiscono i requisiti che l'applicazione intende soddisfare. L'intelligenza artificiale di tipo generativo offerta da OpenAI è quanto viene scelto come IA di supporto alla correzione del codice. Non basta l'invio del codice al ChatBot per un debugging efficiente. Questo è un requisito minimo. Fornendo linea corrente, variabili visibili, output del programma e le informazioni circa la regolare terminazione del programma o meno specificando in tal caso la ragione dell'interruzione, si può ottenere un responso mirato ed esaustivo delle problematiche del codice oggetto del debugging.

#### (c) Struttura della tesi

Qui si commenta brevemente l'indice del documento di tesi dando una visione d'insieme del lavoro svolto dal candidato.

#### (d) Descrizione ad alto livello

Si tratta di una parte discorsiva che intende descrivere in modo generale il plugin implementato per soddisfare i requisiti richiesti. Viene data una visione panoramica e di sintesi del lavoro di tesi.

### 2. Progetto

Questo capitolo presenta l'organizzazione logica antecedente lo sviluppo dell'applicativo che intende soddisfare i requisiti sopra proposti. Inoltre espone i comandi che provvedono al soddisfacimento dei requisiti e giustifica e approfondisce la gestione dei prompt inviati al ChatBot.

#### (a) Schema architetturale

Viene mostrato e descritto lo schema architetturale del progetto definendo a livello logico quali siano le funzioni e i moduli astratti che vengono implementati nell'applicazione attraverso il codice Python.

#### (b) Schema delle dipendenze

Viene mostrato e illustrato lo schema delle dipendenze differenziandole opportunamente. C'è una netta divisione tra le dipendenze da installare prima dell'installazione del plugin, quelle che vengono installate nel

processo di installazione del plugin e quelle interne ad esso. Si specifica inoltre come tutte queste siano relazionate.

(c) **Diagramma degli stati**

Viene mostrato e illustrato il diagramma degli stati afferente all'applicativo sviluppato, quali comandi si possono utilizzare in ciascun stato e come si entra o esce dagli stati presentati.

(d) **Comandi**

Vengono presentati tutti i comandi resi disponibili dall'applicativo fornendo una breve descrizione degli stessi e specificando le condizioni e le modalità secondo le quali possono essere utilizzati.

(e) **Interazione con ChatGPT: Prompt**

Vengono specificati i prompt utilizzati nell'applicativo per comunicare con ChatGPT andando ad analizzarne la struttura e le ragioni che hanno portato a differenziarli in questo modo. Si motiva il perché l'utilizzo di determinati termini o frasi è stato preferito in luogo di altri.

### 3. Commento del codice

Tale capitolo è suddiviso in numerose sezioni organizzate prendendo a modello la struttura delle directory dell'applicativo sviluppato. In ciascuna sezione viene presentata una directory e i file in essa contenuti. Ogni file è opportunamente commentato con lo scopo di giustificare e chiarire, riga per riga, il codice scritto e le scelte operate nella sua stesura.

### 4. Risultati sperimentali

Il capitolo è dedicato alla verifica del corretto funzionamento dell'applicativo in ragione dei requisiti da soddisfare. Vengono passati in rassegna gli esempi delle due sperimentazioni differenziati per tipologia di bug, prima per testare le capacità di ChatGPT di rilevare gli errori in relazione alle informazioni formattate fornitegli, poi per evidenziare i vantaggi di utilizzo dell'applicativo quantificando il miglioramento nella rilevazione degli errori rispetto al solo invio del codice. Si analizzano poi i risultati ottenuti.

(a) **Insieme dei dati**

Si descrive l'insieme dei dati. Per le sperimentazioni sono stati realizzati e documentati 27 programmi che istanziano 15 diverse tipologie di errori. Si tratta di un insieme limitato, ma ciò in ragione di un dataset documentato mancante che è stato poi realizzato ad hoc dal candidato e dell'impossibilità di automatizzare i controlli che sono quindi stati effettuati manualmente dallo stesso. Vengono poi descritte le tipologie di errore che sono state istanziate per le sperimentazioni.

(b) **Descrizione degli esperimenti**

Viene descritta la procedura esecutiva di ogni sperimentazione: utilizzo del comando “autodebug” per l'automatizzazione del debugging, l'analisi del responso e l'astrazione e categorizzazione dei dati in una tabella. Sono poi presentate le sperimentazioni. Mentre nella I si vuole vedere se ChatGPT sia in grado e in che misura di rilevare gli errori nei programmi grazie alle informazioni fornite e ricavate attraverso l'applicativo realizzato, nella II

si vogliono osservare e quantificare i vantaggi di utilizzo delle informazioni di debugging, e quindi dell'applicativo che le recupera, rispetto al solo invio del codice.

(c) **I Sperimentazione**

Risponde alla domanda “ChatGPT è in grado di rilevare gli errori con le informazioni di debugging fornite? In che misura?”. Si passa quindi in rassegna ogni esempio proposto, categoria per categoria, descrivendo l'errore, specificando se ChatGPT con le informazioni di debugging lo ha identificato o se ha commesso degli errori e in ultima analisi se ha proposto delle correzioni.

(d) **II Sperimentazione**

Risponde alla domanda “Qual è il vantaggio di inviare le informazioni di debugging? Quanto può risultare vantaggioso?”. Vengono quindi selezionati alcuni dei test analizzati in precedenza allo scopo di valutare quanto le informazioni di debugging influiscano nella rilevazione degli errori. Onde valutare ciò, per ciascun esempio viene prima inviato soltanto il codice al ChatBot e poi, in una sessione separata, viene inviato il codice assieme a tutte le altre informazioni di debugging. Si confrontano poi i risultati ottenuti per vedere quanti errori vengono rilevati solo grazie al codice e quanti errori vengono invece rilevati grazie alle informazioni di debugging recuperate attraverso l'applicativo implementato.

(e) **Analisi dei risultati**

Vengono raccolti i dati dalle due sperimentazioni precedenti: La I, atta a valutare la capacità di ChatGPT di rilevare, con le informazioni fornite dal plugin, gli errori nella molitudine di tipologie presentate. Ci dice quanto è utile l'integrazione di ChatGPT nel debugger. La II, atta a valutare il vantaggio di utilizzo delle informazioni di debugging recuperate attraverso il plugin per la rilevazione degli errori rispetto al solo invio del codice. Da qui derivano le condizioni sotto le quali l'applicativo risulta più performante.

## 5. Conclusioni

Si traggono le conclusioni rispetto al lavoro di tesi svolto. Partendo dall'obiettivo del candidato, migliorare e automatizzare il debugging, fino ad arrivare ai risultati delle sperimentazioni. ChatGPT si rivela uno strumento utile per l'identificazione dei problemi del codice trovando i bug nel 92,6% dei casi grazie alle informazioni fornite dall'applicativo, tali informazioni, inoltre, migliorano la sua efficienza nello scovare gli errori fino al 50% di quanto non si avrebbe se fosse inviato soltanto il codice.

## 1.4 Descrizione ad alto livello

Si è preso atto delle motivazioni che ci spingono alla necessità di migliorare il debugging: rendere sicuro il codice che gestisce i sistemi del mondo di oggi, il risparmio in termini economici nell'avere codice con meno errori possibile e l'irriducibilità del quantitativo di errori nel codice indipendentemente da quanto intelligentemente lo si possa scrivere.

Al fine di migliorare il debugging si prospetta la possibilità di integrare l'intelligenza artificiale offerta dall'organizzazione OpenAI all'interno di un debugger, strumento essenziale per la rilevazione di errori nel codice. L'IA può rivelarsi uno strumento utile se gli si forniscono le informazioni appropriate, in che misura potrà poi esserlo sarà nostro compito verificarlo. Anzitutto va implementato un applicativo capace di integrare l'IA da noi scelta, ChatGPT, all'interno di un debugger valido come LLDB. Il plugin offre diverse funzionalità raggruppate in programmi. Raccoglie le informazioni di debugging attraverso gli strumenti offerti dal debugger e ne garantisce l'invio a ChatGPT. L'invio avviene grazie al comando "send". Deve inoltre essere possibile fare richieste personalizzate al ChatBot, questo è possibile con il comando "ask". Queste ma anche altre funzioni minori sono offerte al programmatore che voglia sfruttare l'IA per la rilevazione degli errori nel codice. Si forniscono quindi degli strumenti validi per un debugging manuale. Ciò è sufficiente, ma si può essere più ambiziosi di così, favorendo l'utente con un comando in grado di automatizzare le procedure di debugging e interagire al contempo con l'IA. Ciò è garantito dall'utilizzo del comando "autodebug".

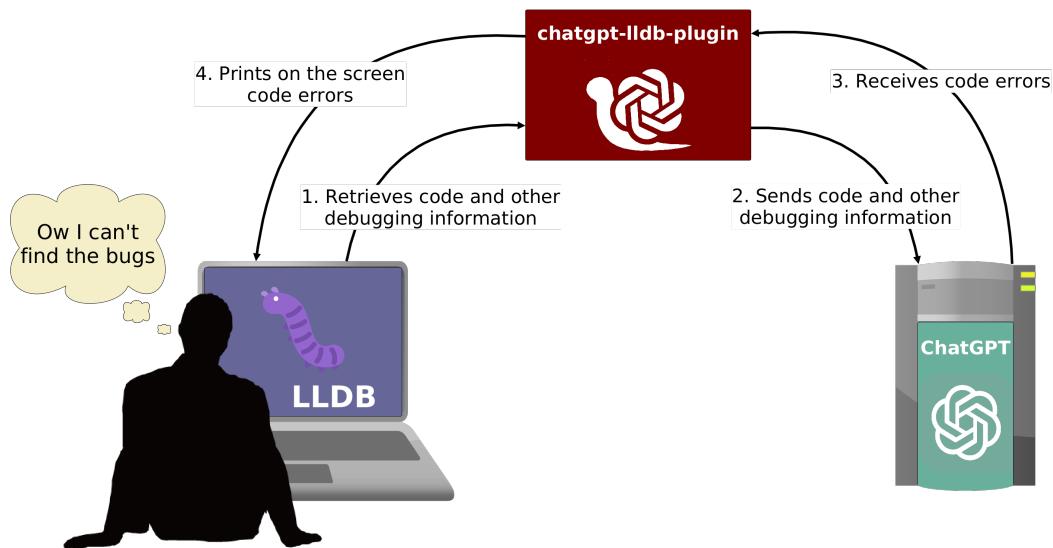
L'applicativo viene implementato e commentato. Per dare un'idea più astratta, organica e globale del plugin, viene realizzato:

- uno schema architetturale che raggruppa logicamente la struttura dell'applicativo;
- uno schema delle dipendenze che mostra le relazioni che intercorrono tra i file che compongono il plugin;
- uno diagramma degli stati che chiarifica gli stati entro i quali sono utilizzabili i comandi forniti.

Si dà inoltre una descrizione di tutti i comandi disponibili che vengono aggiunti a quelli standard già offerti dal debugger LLDB e si va a descrivere come è stata studiata l'interazione con ChatGPT e quindi i prompt che vi vengono inviati entro i quali vengono inserite le informazioni di debugging recuperate attraverso il debugger: sistema utilizzato, versione di LLDB utilizzata, nome del file sorgente oggetto del debugging, codice sorgente dello stesso, linea corrente al momento dell'invio delle informazioni, variabili visibili fino tale linea, output del programma che si sta debuggando e informazioni sulla rilevazione o meno di segnali che terminano l'esecuzione del programma e in caso contrario se l'esecuzione si è conclusa senza errori.

Non rimane che la parte sperimentale della tesi. Testare esempio dopo esempio le

capacità di ChatGPT in relazione alle informazioni fornitegli. Qui si vede, fornita una base di 27 esempi, istanze di 15 tipologie differenti di errori, quanto l'IA in questione sia capace di trovare i bug presenti nel codice. Nel 92,6% dei casi i bug vengono trovati, inoltre il problema viene rilevato senza ambiguità o evidenziando errori insussistenti nel 55,6% dei casi. Nella parte sperimentale della tesi si confronta anche il responso di ChatGPT sia se il prompt che inviamo contiene solo il codice, sia se questo contiene anche le altre informazioni di debugging. Ora che sappiamo che l'IA è in grado di rilevare gli errori, infatti, vogliamo constatare i vantaggi di utilizzo delle informazioni di debugging recuperate dall'applicativo e valutare quanto queste siano importanti per la rilevazione degli errori. In ultima analisi sapere quanto è vantaggioso utilizzare l'applicativo realizzato e sotto quali condizioni si ha un trade-off migliore. Risulta che negli esempi trattati, presi dalla sperimentazione precedente e modificati opportunamente per nascondere l'errore di cui si fanno istanza, sia attraverso calcoli che tramite l'importazione di dati e funzioni da altri file, le informazioni di debugging consentono un miglioramento dell'efficienza di ChatGPT nella rilevazione degli errori che può arrivare fino al 50%. Con un'efficienza maggiore lì dove gli errori causano terminazioni improvvise del programma o sono mascherati. L'integrazione di ChatGPT in LLDB quindi si rivela utile e l'applicativo che la implementa si presenta come uno strumento vantaggioso per il miglioramento del debugging dei programmi scritti in linguaggio C e derivati.



**Figura 1.2.** Schema esplicativo del funzionamento dell'applicativo



# Capitolo 2

## Progetto

### 2.1 Schema architetturale

ChatGPT LLDB Plugin viene effettivamente integrato all'apertura di LLDB attraverso il file “.lldbinit” dal quale LLDB prende i comandi da eseguire alla sua apertura. Con “command script import /home/user/chatgpt-lldb-plugin/main/commands.py” infatti carichiamo nel debugger il file con tutti i comandi che attinge poi dagli altri file del programma per eseguire le funzioni chiamate. L'integrazione avviene quindi in modo molto semplice e viene automatizzata attraverso “setuptools” (versione == 68.2.0) che installa anche le dipendenze Python. Il punto chiave resta l'importazione di “commands.py” per la quale sono necessari Python3 (versione == 3.9.2) e LLDB (versione == 11.0.1). Una volta aperto LLDB e caricato il target si hanno due modalità di base per l'utilizzo dell'applicativo, l' “api\_mode” e la “browser\_mode”, che prevedono rispettivamente la comunicazione con ChatGPT via le API di OpenAI e attraverso Chrome Browser con corrispondente Chromedriver per l'utilizzo del pacchetto Python Selenium che consente la gestione degli elementi delle pagine visualizzate dal browser. Si hanno poi i 13 comandi aggiuntivi rispetto a quelli base di LLDB che il plugin integra: “mode”, “switch”, “code”, “send”, “ask”, “wait”, “autodebug”, “save”, “load”, “export”, “import”, “clear”, “info” che verranno discussi ampiamente più avanti. Di questi, il comando “autodebug” è particolarmente interessante poiché integra in sé altri comandi, consentendo il recupero e l'invio delle informazioni necessarie al debugging, ricevendo e stampando poi il responso di ChatGPT. Si tratta quindi di un comando che consente l'automatizzazione del debugging forte dell'interazione con l'IA. Un'altra parte riguarda la gestione (salvataggio, caricamento, importazione ed esportazione) delle chat la quale consente di continuare il debugging in sessioni lavorative diverse. Il cuore del plugin è il recupero delle informazioni di debugging sull'obiettivo (target). Il target, una volta che è stato caricato, che è stato impostato un breakpoint ed stato messo in esecuzione, viene utilizzato come riferimento dal quale recuperare tutte le informazioni sull'esecuzione del programma. Questo avviene tramite le API LLDB per Python fornite dalla casa madre LLVM. Le informazioni di debugging recuperate differiscono a seconda del comando utilizzato e dell'esecuzione del programma. Nel caso si utilizzi il comando “send” vengono recuperate: piattaforma, versione di LLDB, nome del programma, codice numerato linea per linea, linea corrente e variabili visibili. Nel caso si utilizzi il comando “autodebug” si recuperano: piattaforma, versione di LLDB, nome del pro-

grammma, codice numerato linea per linea e output del programma. Se il programma termina in modo improvviso, allora vengono recuperate anche: linea dell'errore, variabili visibili e ragione della terminazione. Infine vi è la gestione dei prompt da inviare a ChatGPT. Ciò consente la formalizzazione accurata delle informazioni di debugging recuperate dall'applicativo nei prompt da inviare al ChatBot, costruiti affinché l'AI possa comprendere agevolmente le nostre richieste, anche preservando la numerazione delle righe del codice e separando opportunamente dati e richieste.

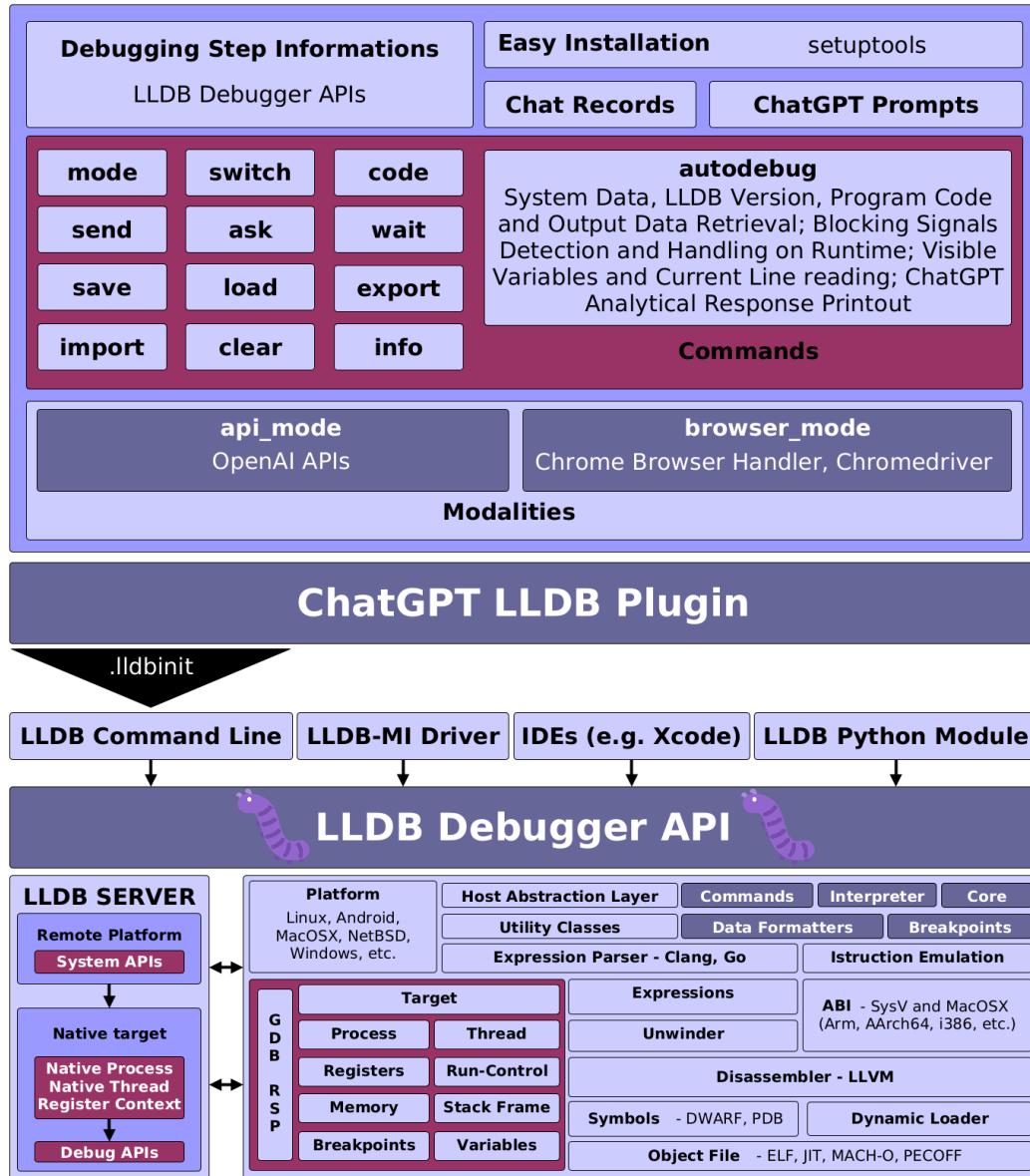


Figura 2.1. Schema architetturale dell'applicativo

## 2.2 Schema delle dipendenze

Vi sono tre grandi dipendenze per il corretto funzionamento del programma. Anzi-tutto l'interprete Python3[28] (versione == 3.9.2) e LLDB Debugger[4] (versione == 11.0.1). Questi sono applicativi strettamente necessari, il primo garantisce l'esecuzione del plugin, scritto in Python, mentre il secondo è l'applicativo cui il plugin deve essere integrato. Inoltre, affinché possa essere utilizzato l'installer file setup.py deve essere installato il pacchetto Python setuptools[32] (versione == 68.2.0) che consente da terminale di installare tutto il plugin digitando semplicemente “cd chatgpt-lldb-plugin && pip install -e .” e premendo invio. Tuttavia quest'ultima dipendenza può essere arginata procedendo con un'installazione manuale la quale come accennato precedentemente nello schema architerrurale necessita del comando “command script import PATH/TO/commands.py” da inserire in “.lldbinit” una sola volta oppure alternativamente in ogni sessione di LLDB quando si vogliono utilizzare i comandi introdotti dal plugin. Il mancato utilizzo del pacchetto “setuptools” è quindi accettabile, ma solo a condizione che prima di importare i comandi vengano installati manualmente gli altri pacchetti Python necessari al plugin, ovvero: “cpython-lldb”[1] per l'utilizzo delle API di LLDB, “openai”[2] per l'utilizzo delle API di LLDB, “psutil”[25] per la gestione del processo browser, “PyYAML”[10] per la gestione dei file YAML utilizzati per salvare i dati formattati e “selenium”[3, 21] per l'interazione con l'interfaccia web. In ogni caso, nell'utilizzo standard per il quale si è ideato l'applicativo, si ha l'idea di un'installazione semplice, la quale non può prescindere da “setuptools”, restano allora queste tre dipendenze da avere pre-installate: l'interprete Python 3, LLDB Debugger e il pacchetto Python setuptools. Oltre queste dipendenze che devono essere installate prima del plugin, ve ne sono di altre, tuttavia queste sono soddisfatte in parte dall'interprete Python 3 che fornisce la libreria standard python con i moduli:

- **atexit**, questo modulo fornisce un'interfaccia per registrare funzioni da eseguire alla fine del programma o al momento dell'uscita. È utile per la gestione delle operazioni di “pulizia” o “chiusura” prima che il programma termini.
- **io**, è utilizzata per la gestione di operazioni di input/output in memoria.
- **multiprocessing**, consente la creazione e gestione di processi multipli per eseguire operazioni in parallelo su sistemi multi-processore, di coordinare l'esecuzione di task concorrenti e gestire la sincronizzazione tra processi.
- **os**, fornisce una serie di funzioni per interagire con il sistema operativo sottostante. È utilizzato per operazioni come la gestione dei file, la navigazione nelle directory, la manipolazione delle variabili di ambiente.
- **platform**, permette di ottenere informazioni sulla piattaforma su cui il programma Python è in esecuzione, come il nome del sistema operativo, la versione e l'architettura.
- **re**, consente di utilizzare le espressioni regolari per cercare e manipolare testo in modo avanzato. È utilizzato per operazioni di ricerca e sostituzione di pattern all'interno delle stringhe.

- **socket**, fornisce funzionalità per la programmazione di socket che sono utilizzate per la comunicazione di rete. È possibile utilizzarlo per creare client e server che comunicano tramite TCP/IP o UDP.
- **subprocess**, consente di eseguire processi esterni da un programma Python. È utilizzato per eseguire comandi o programmi esterni e interagire con essi attraverso il proprio programma.
- **sys**, fornisce accesso a variabili e funzioni per gestire il comportamento del programma Python, nonché di comunicare con il sistema sottostante.
- **threading**, supporta la programmazione multi-threading in Python. Fornisce un'API per la gestione dei thread e la sincronizzazione tra di essi.
- **time**, offre funzionalità per la gestione del tempo, come la creazione di pause o ritardi all'interno di un programma.

In altra parte sono invece soddisfatte da pacchetti python elencati nel file “requirements.txt” che vengono installati grazie a “setuptools”, questi sono stati elencati prima. Il plugin è quindi un pacchetto Python che necessita di ben 7 librerie python compresa quella standard e “setuptools”. Come si può vedere dallo schema, il sistema è modulare: si ha il macro-pacchetto “chatgpt-lldb-plugin” contenente i file python “setup.py”, “api\_key\_config.py” e “browser\_config.py” e il pacchetto “main”. Quest’ultimo contiene il cuore del programma e quindi i file: “main.py”, “commands.py”, “prompts.py”, “stepping.py” e “version.py” oltre che i due micro-pacchetti “api\_connection” e “browser\_connection” che gestiscono la l’interazione con ChatGPT secondo le due modalità alternative. In linea generale, quando si installa il programma con il comando “cd chatgpt-lldb-plugin && pip install -e .”, viene eseguito “setup.py” che si avvale del pacchetto Python “setuptools” per installare le dipendenze “cpython-lldb”, “openai”, “psutil”, “PyYAML” e “selenium” (nelle versioni indicate nel file “requirements.txt”) e del file “main.py” per creare o modificare il file di caricamento iniziale dei comandi integrati “lldbinit”, questo è quanto accade nell’installazione e non ha necessità di essere ripetuto. A questo punto si hanno i comandi in “commands.py” che verranno d’ora in avanti caricati ad ogni sessione di LLDB. I comandi necessitano dei pacchetti “api\_connection” e “browser\_connection” per l’interazione con ChatGPT e del file “prompts.py” dal quale si ricavano i prompt per la corretta comunicazione dei dati e delle richieste al ChatBot. Il file “prompts.py” inoltre prende i dati che poi formatta da “stepping.py”, punto cardine del plugin che si occupa, tramite l’utilizzo delle API LLDB di recuperare le informazioni di base sulla piattaforma come il nome del sistema operativo e il tipo di processore, il nome dell’eseguibile, il codice del file sorgente, le variabili visibili, la linea corrente, l’output e le informazioni su eventuali segnali che bloccano l’esecuzione del target del debugging. Tornando all’interazione secondo le due modalità alternative, il pacchetto “api\_messaging” si avvaluta del modulo “openai” per l’utilizzo delle API di OpenAI e di api\_key\_config.py per recuperare la OpenAI API Key necessaria per l’utilizzo delle omonime API e collegata al proprio account sul sito web dell’organizzazione, questo solo in caso la variabile d’ambiente “OPENAI\_API\_KEY” non fosse stata precedentemente impostata. D’altro canto il pacchetto “browser\_connection” si avvale del modulo Selenium per la gestione

dell’interazione con l’interfaccia web di Chrome Brower e di psutil per la gestione del processo Chrome Brower che viene lanciato all’avvio della modalità. Il modulo “yaml” della dipendenza “PyYAML”, infine, consente di gestire il salvataggio e la lettura delle chat salvate in locale con i comandi “save”, “load”, “import” ed “export” nonché il salvataggio del PID del processo Chrome Brower aperto durante la “browser\_mode”, grazie al quale è possibile chiudere il browser alla chiusura di LLDB Debugger, operazione che avviene per via del monitoraggio e della gestione da parte di “cleaner.py”.

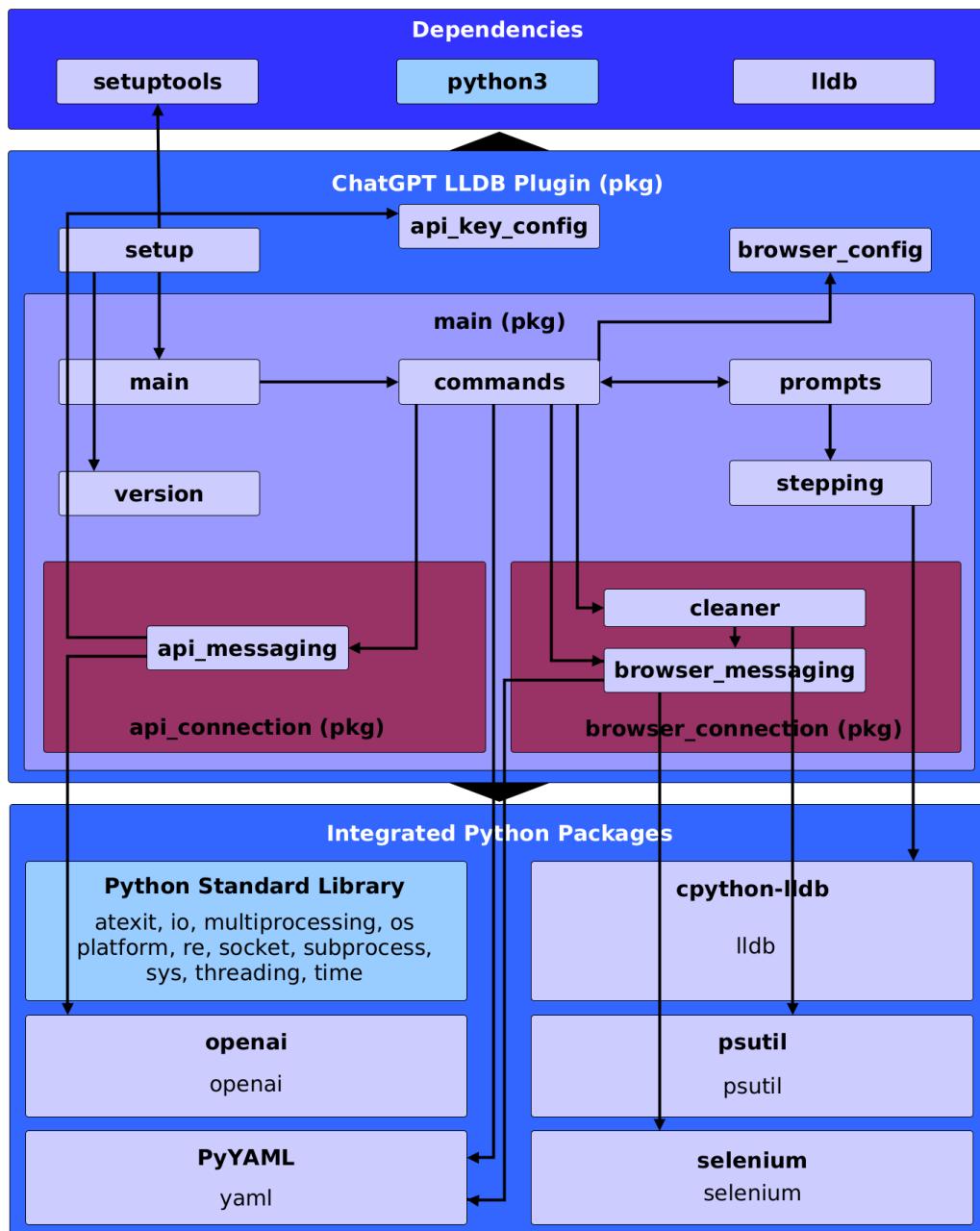


Figura 2.2. Schema delle dipendenze dell'applicativo

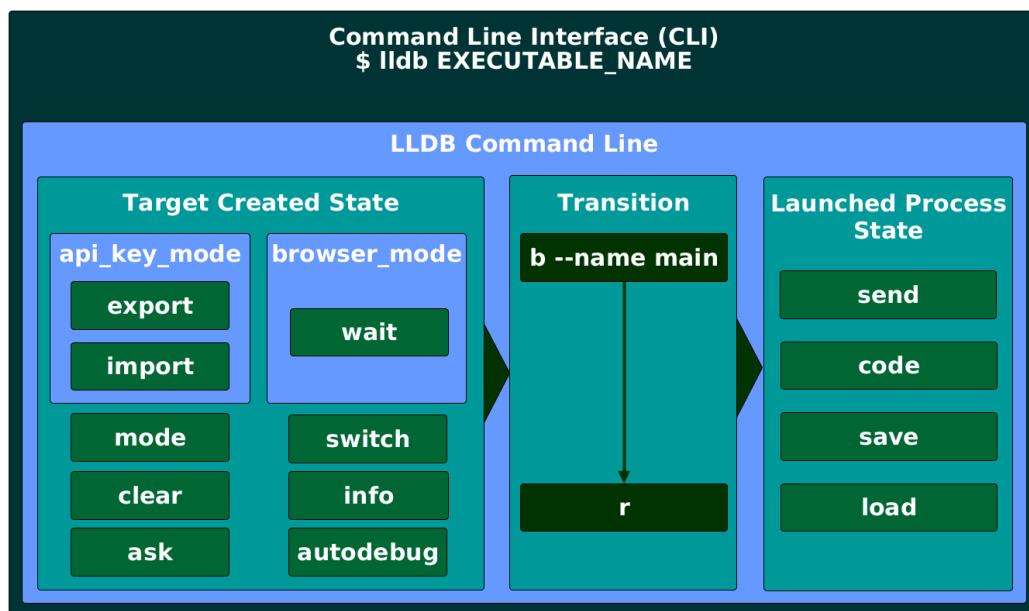
### 2.3 Diagramma degli stati

Si hanno due stati che caratterizzano l'esecuzione dell'applicativo “chatgpt-lldb-plugin”:

- **Target Created State.** In questo stato si è appena creato il target, generalmente questo avviene in modo diretto da linea di comando (CLI) con il comando “lldb EXECUTABLE\_NAME”, oppure attraverso l'ausilio di due comandi, prima di “lldb” da linea di comando, il quale lancia LLDB, e una volta in LLDB attraverso il comando “target create PATH/TO/EXECUTABLE\_NAME”. In questo stato inoltre si può entrare in caso di terminazione del processo associato all'obiettivo del debugging. Qui possono essere eseguiti solo i seguenti comandi: “mode” che mostra la modalità della connessione con ChatGPT ovvero “api\_key\_mode” oppure “browser\_mode”; “switch” il quale cambia la modalità passando dalla corrente all'alternativa; “clear” il quale elimina la storia della chat per la sessione corrente; “info” che elenca tutti i comandi del plugin fornendo una breve descrizione degli stessi ed un'esempio di utilizzo; “ask” nel quale argomento va posta la nostra richiesta per ChatGPT alla quale seguirà una risposta; “autodebug” che lancia il processo associato all'eseguibile ed invia le informazioni di debugging necessarie alla rivelazione dei bug a ChatGPT ottenendo poi il suo responso. Vi sono poi altri 3 comandi, due di questi relativi alla sola modalità “api\_key\_mode” mentre l'ultimo alla modalità “browser\_mode”. Si tratta di “export” e “import” nel primo caso, i quali consentono di importare e caricare nella storia della sessione una chat salvata nel percorso specificato e, inversamente, di esportare la chat presa dalla sessione corrente in un file YAML che viene creato al momento nel percorso specificato dall'utente. Nel secondo caso si tratta di “wait” che consente di regolare le tempistiche di attesa per la risposta di ChatGPT quando si utilizza l'interfaccia web per la comunicazione con il Chatbot. Appare evidente il perché di questa esclusiva, nel primo è superfluo nella modalità browser dare all'utente la possibilità di salvare la propria chat in locale, su un percorso da lui scelto, se si ha la possibilità di conservarla in remoto sul proprio account OpenAI e lo stesso per l'importazione. Quando al secondo, il comando “wait” non ha senso nella modalità con la chiave API poiché in questo caso ChatGPT invia il proprio responso non appena elaborato il contenuto, non c'è bisogno di aspettare un tempo stimato il quale garantisce l'invio della risposta completa come nella modalità browser nella quale un tempo troppo breve può troncare il contenuto del responso se ancora in elaborazione.
- **Launched Process State .** Altresì noto come Not Exited State oppure come Running or Stopped State. Questo stato si verifica partendo dal “Target Created State”, in particolare dopo che è stato dato, nel seguente ordine, prima il comando LLDB “breakpoint set” con un argomento valido e dopo il comando LLDB “run”. Comandi che solitamente si usano abbreviati e specificano il breakpoint nel main. Per semplicità si ha: “b –name main” e poi “r”. È importante specificare che tali operazioni pongono il processo associato all'obiettivo (target) del debugging in esecuzione. Durante l'esecuzione però possono esservi ragioni di stop (stop reasons) quali possono essere l'utilizzo

di “step”, “next” o l’incontro di un breakpoint impostato precedentemente, pertanto tale stato comprende anche queste situazioni. Tale stato infatti è caratterizzato dal fatto che il processo non ha concluso la sua esecuzione, non è uscito da essa. Il processo pertanto si troverà o in esecuzione (Running), se non viene fermato, oppure in fermo (Stopped), pertanto si trova nello stato di non uscita dall’esecuzione (Not Exited). Si esce dallo stato Launched Process State solo quando il processo termina la sua esecuzione. Entrando ora nella descrizione delle possibilità che offre tale stato, bisogna tenere presente che una volta nel “Launched Process State” si possono eseguire tutti i comandi compresi quelli associati al “Target Created State”. Sono tuttavia 5 i comandi che possono essere eseguiti solo in questo stato. Questi possono essere lanciati indipendentemente dalla modalità: “send” con il quale vengono recuperate e inviate le informazioni di debugging rispetto alla linea corrente scelta dall’utente; “code” che recupera il codice sorgente associato all’eseguibile caricato come target del debugging e lo stampa per intero; “save” che salva in locale o in remoto, a seconda della modalità di interazione con ChatGPT, lo storico dei messaggi scambiati con il ChatBot nella sessione; “load” che carica lo storico dei messaggi scambiati con il ChatBot dal file locale in “chatgpt-lldb-plugin/main/saves”, lo stesso generato durante l’utilizzo del comando “save” il quale è in formato YAML.

È bene specificare che i comandi utilizzabili nel “Target Created State” possono essere utilizzati anche nel “Launched Process State”, tuttavia non vale il viceversa.



**Figura 2.3.** Diagramma degli stati

## 2.4 Comandi

I comandi si trovano definiti nel file “commands.py” nella sottodirectory “main”. Sono ben 13 e forniscono tutti gli strumenti per effettuare sia un debugging manuale che automatico. Questi vengono importati con da LLDB alla sua apertura, LLDB infatti all'avvio legge il file “.lldbinit”, che generalmente si trova nella “home” come file nascosto, il quale contiene i comandi da eseguire all'avvio tra i quali durante l'installazione è stato aggiunto “command script import /home/user/chatgpt-lldb-plugin/main/commands.py”, questo importa “commands.py” che poi aggiunge singolarmente ogni comando con l'iterazione del comando LLDB “command script add -f commands.NOME\_COMANDO NOME\_COMANDO”. L'implementazione dei comandi verrà vista poi nel comando del codice, nella sottosezione commands.py. Proseguiamo con la descrizione dei comandi.

### 2.4.1 mode

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” nonché in qualsiasi modalità, inoltre non ha argomenti. Stampa a schermo la modalità di utilizzo. Questa, di default è l’“api\_key\_mode”, mentre l'altra disponibile è la “browser\_mode”.

### 2.4.2 switch

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” nonché in qualsiasi modalità, inoltre non ha argomenti. Passa dalla modalità corrente a quella alternativa. Come detto prima, la modalità di default è l’“api\_key\_mode”, quindi se lo si usa ad inizio sessione si passerà direttamente alla “browser\_mode”. Nel fare il viceversa, in caso di browser aperto, questo verrà chiuso.

### 2.4.3 code

Il comando può essere utilizzato solo nel “Launched Process State” e in qualsiasi modalità, inoltre non ha argomenti. Stampa a schermo l'intero codice sorgente associato all'eseguibile obiettivo del debugging. Le righe sono numerate.

### 2.4.4 send

Il comando può essere utilizzato solo nel “Launched Process State” e in qualsiasi modalità, inoltre non ha argomenti. Recupera il prompt contenente le informazioni sulla piattaforma di utilizzo, la versione di LLDB, il nome del programma e il codice numerato per righe e lo salva nella variabile “codeInfo”. Recupera poi il prompt contenente la linea corrente e le variabili visibili e le salva nella variabile “stepInfo”. Invia quindi i contenuti di entrambe le variabili a ChatGPT, questo solo la prima volta che viene utilizzato nella sessione corrente, le volte successive invierà solo il contenuto di “stepInfo” onde evitare uno spreco di risorse.

#### 2.4.5 ask

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” nonché in qualsiasi modalità, inoltre ha come argomento la richiesta che si desidera fare a ChatGPT. Una volta scritto sulla linea di comando di LLDB “ask LA\_TUA\_RICHIESTA” e premuto invio, ciò che fa il comando è inviare a ChatGPT la propria richiesta e stampare a schermo il responso del ChatBot. Tale risposta viene stampata non appena pronta in caso si utilizzi la modalità “api\_key\_mode”, mentre viene stampata dopo un tempo predefinito se si si utilizza la “browser\_mode”, tempo regolabile o in maniera statica e quindi fissa per tutte le sessioni a venire attraverso la modifica della variabile “timeForResponse” nel file di configurazione “browser\_config.py” oppure attraverso la modifica, sempre della stessa variabile ma direttamente nel corso della sessione con il comando “wait”, in tal caso la modifica è temporanea e dura fino a fine sessione.

#### 2.4.6 wait

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” e solo nella modalità browser, inoltre ha come argomento il numero di secondi che si desidera impostare come nuovo tempo di attesa per le risposte nella sessione corrente. Un tempo di attesa troppo breve farà sì che si ricevano risposte troncate da parte del ChatBot. Un tempo di attesa troppo lungo potrebbe rendere tedioso l'utilizzo del plugin. Dalle prove in fase di testing risulta soddisfacente l'attesa di 20 secondi, impostata come standard. Tale attesa è necessaria soltanto per la modalità browser poiché mentre nella “api\_key\_mode” non appena elaborato il responso da parte di ChatGPT questo viene inviato, quando si utilizza la “browser\_mode”, si ha a che fare con un'interfaccia web e bisogna quindi catturare quanto viene mostrato a schermo e poi stamparlo all'interno del debugger. Dato che nell'interfaccia web si stampa parola per parola e i tempi di risposta sono variabili, bisognerà aspettare il tempo necessario affinché venga riportata la risposta completa, altrimenti questa viene catturata parzialmente ovvero troncata.

#### 2.4.7 autodebug

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” nonché in qualsiasi modalità, inoltre non ha argomenti. Ciò che fa è recuperare le informazioni di debugging, inviarle a ChatGPT con il comando “send” e successivamente richiedere a ChatGPT se vi sono errori tenendo conto di quanto è successo runtime, descritto sempre come argomento di “ask”. Nella “browser\_mode” si aspetta un tempo limitato tra la “send” e “ask”. Questa attesa è necessaria poiché dopo aver inviato i dati di debugging con la “send”, ChatGPT elabora una risposta, seppure più breve rispetto il reponso per “ask”, e in questo tempo di elaborazione non è possibile fare ulteriori richieste, quindi bisogna aspettare ad immettere l'argomento di “ask” nel campo testuale e inviarlo poiché non ce lo consente. Generalmente 5 secondi sono sufficienti. Onde favorire la modifica di tale tempo di attesa senza complicare ulteriormente le cose, per convenzione si è posto come il “timeForResponse/4”. Modificando così il tempo di attesa per i responsi richiesti con “ask” il cui valore standard è 20 secondi, si viene a modificare anche il

tempo di attesa per i responsi della “send” che come detto sono inutili da un punto di vista informativo ma il cui tempo deve essere aspettato necessariamente affinché si possa procedere con le proprie richieste. Quindi, con “autodebug” verranno recuperate le medesime informazioni che si recuperano con la “send”, ovvero:

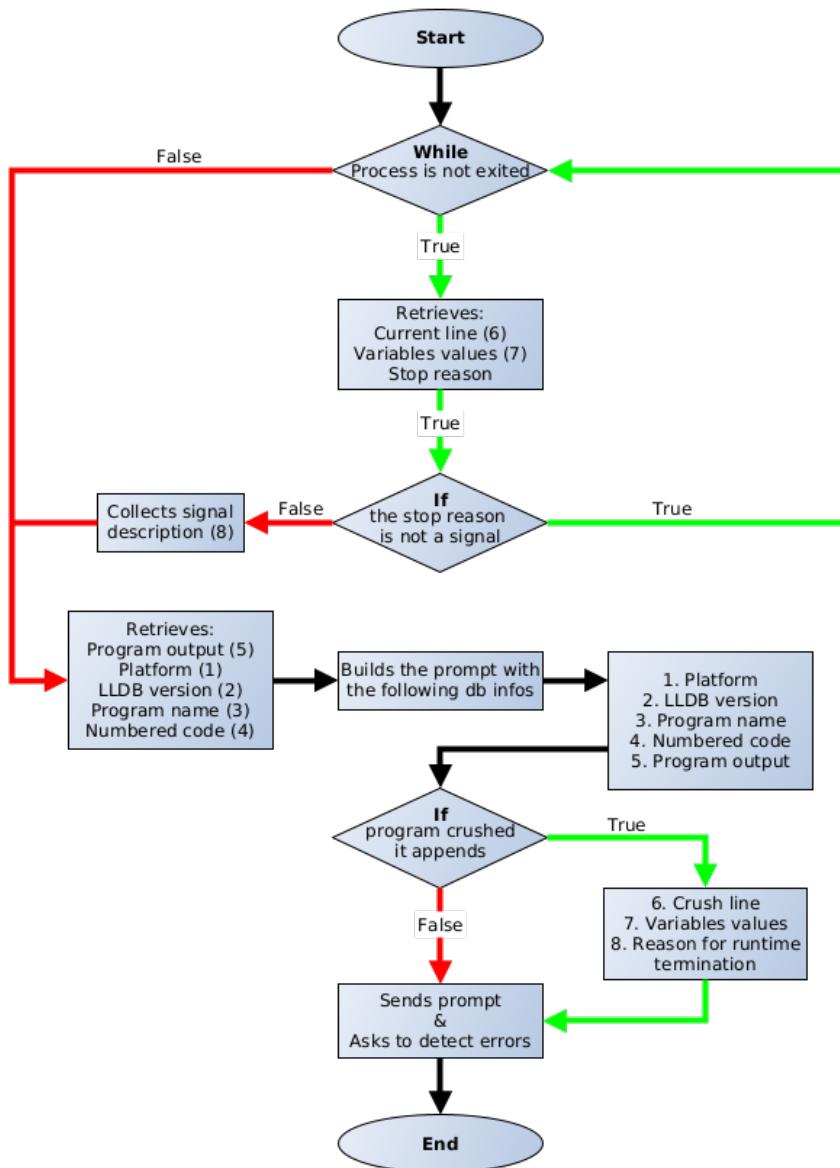
- Il prompt contenente le informazioni sulla piattaforma di utilizzo, la versione di LLDB, il nome del programma e il codice numerato per righe, che verrà inviato solo una volta in tutta la sessione;
- Il prompt contenente la linea corrente e le variabili visibili. Queste informazioni verranno inviate, tuttavia e al contrario della “send” standard, solo in caso di rilevazione di errori che interrompono l’esecuzione, in caso contrario risultano infatti inutili per non dire fuorvianti. Per recuperare il valore di queste variabili, bisogna infatti impostare un breakpoint e lanciare l’esecuzione del processo associato all’obiettivo (target) del debugging, poi le informazioni verranno catturate nel punto definito dal breakpoint, che in caso di esecuzione con terminazione regolare è la funzione “main” del programma per convenzione scelta nello sviluppo del plugin, ma questo breakpoint non è un punto in cui si sono rilevati errori e per di più se ci troviamo all’inizio della funzione “main” ci ritroveremo tutte variabili non inizializzate da leggere. Pertanto leggeremmo dei valori non inizializzati e in un punto che sicuramente non può essere fonte di errori. Pertanto è bene non inviare queste informazioni in caso di terminazione regolare. La ragione della convenzione verranno spiegate più dettagliatamente nell’implementazione della funzione “automated\_debugging” del file “stepping.py”, la quale viene chiamata indirettamente dal comando “autodebug”, qui daremo solo un’accenno.

Ma verranno recuperate anche ulteriori informazioni come vedremo poi quando verrà mostrata nel dettaglio l’implementazione del comando. Infatti, venendo il programma posto in esecuzione, viene catturato l’output e vengono inoltre salvate le seguenti informazioni:

- In caso di rilevazione di segnali che interrompono l’esecuzione, verrà recuperata la linea di rilevazione del segnale e specificata la ragione dello stop, ad esempio un “SIGABRT”, un “SIGSEGV: indirizzo non valido” o un “SIGFPE”.
- In caso di rilevazione di segnali che interrompono l’esecuzione, ma che compromettono l’indirizzo di memoria all’interno del program counter (PC) verrà specificata sia la ragione dello stop, ma senza indicare la linea corrente, poiché errata in quanto si è passati all’esecuzione di istruzioni il cui indirizzo è stato compromesso. Ciò potrebbe accadere in caso di rilevazione di un “SIGSEGV: indirizzo non valido” ad esempio.
- In caso di esecuzione regolare, senza alcuna interruzione, verrà preso come riferimento alla linea corrente il main come accennato poco fa, questo poiché onde utilizzare poi il comando “send” e il recupero delle informazioni elencate precedentemente è necessario che il programma sia in “Launched Process State” e questo non può accadere se l’esecuzione è terminata poiché ci si ritroverebbe nel “Target Created State”. Viene preso come riferimento il main poiché è

l'unica funzione comune a tutti i programmi, non ci si può nemmeno basare sul numero di righe in quanto queste variano da programma a programma e quindi è sembrato agli occhi dello sviluppatore la sola costante di riferimento comune a tutti i programmi in C e derivate.

Quindi mentre con la “send” recuperiamo e inviamo le note informazioni sull’obiettivo del debugging tranne la linea corrente e le variabili visibili in caso di terminazione regolare, con “ask” inviamo: l’output del programma, se presente il punto e la ragione dell’interruzione del programma e infine la richiesta di rilevazione dei bug nell’obiettivo del debugging. Il tutto appropriatamente formattato per essere letto da ChatGPT grazie al file “prompts.py”.



**Figura 2.4.** Diagramma di flusso semplificato del comando “autodebug”

#### 2.4.8 save

Il comando può essere utilizzato solo nel “Launched Process State” e in qualsiasi modalità, inoltre non ha argomenti. Quando viene utilizzato, viene recuperato il nome del file sorgente, essendo noto l’obiettivo del debugging, e viene poi creato il file “dbChat\_NOMEDELFILESORGENTE.yaml” nel quale viene salvata una lista di dizionari contenente lo storico dei messaggi scambiati con il ChatBot.

#### 2.4.9 load

Il comando può essere utilizzato solo nel “Launched Process State” e in qualsiasi modalità, inoltre non ha argomenti. Quando viene utilizzato, viene recuperato il nome del file sorgente, essendo noto l’obiettivo del debugging, e viene poi caricato il file “dbChat\_NOMEDELFILESORGENTE.yaml”, contenente la lista di dizionari con lo storico dei messaggi scambiati con il ChatBot. Questa lista di dizionari viene caricata nella variabile globale “history” la quale è designata per tenere traccia della conversazione con ChatGPT per la sessione corrente.

#### 2.4.10 export

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” e solo nell’“api\_key\_mode”, inoltre ha come argomento il percorso al file nel quale si desidera salvare la lista di dizionari contenente lo storico dei messaggi scambiati con ChatGPT. Essenzialmente agisce come “save”, ma consente di scegliere la destinazione del file con funzione di salvataggio e conseguentemente consente di avere più salvataggi per il debugging dello stesso target.

#### 2.4.11 import

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” e solo nell’“api\_key\_mode”, inoltre ha come argomento il percorso al file dal quale si desidera caricare la lista di dizionari contenente lo storico dei messaggi scambiati con ChatGPT. Essenzialmente agisce come “load”, ma consente di scegliere la destinazione del file con funzione di salvataggio che si desidera caricare.

#### 2.4.12 clear

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” nonché in qualsiasi modalità, inoltre non ha argomenti. Quando viene utilizzato, esegue due operazioni distinte ma con il medesimo fine, a seconda che si utilizzi l’“api\_key\_mode” oppure la “browser\_mode”. Nel primo caso svuota la lista di dizionari contenuta nella variabile history e quindi vengono persi tutti i messaggi scambiati con ChatGPT per la sessione in corso, mentre nel secondo viene aperta una nuova chat attraverso l’interfaccia web, perdendo il riferimento alla precedente e ottenendo, di fatto, il medesimo risultato dell’“api\_key\_mode”. Fondamentalmente consente all’utente di iniziare una nuova chat con il ChatBot.

#### 2.4.13 info

Il comando può essere utilizzato sia nel “Target Created State” che nel “Launched Process State” nonché in qualsiasi modalità, inoltre non ha argomenti. Una volta lanciato, stampa a schermo la lista dei comandi offerti dal plugin con una breve descrizione degli stessi e un esempio di utilizzo quando ritenuto necessario ai fini della comprensione.

## 2.5 Interazione con ChatGPT: Prompt

Per l’interazione con ChatGPT, i dati raccolti devono prima essere formattati correttamente[8, 33] in modo tale che possano essere compresi senza ambiguità, formattazione che avviene nel file “prompts.py” che si pone quale intermezzo tra i comandi in “commands.py” e le informazioni di debugging in “stepping.py”. In tal senso si è puntato all’utilizzo di proposizioni principali semplici evitando l’utilizzo di subordinate o periodi complessi. È stato inoltre favorita la rappresentazione schematica, ordinata e logica delle informazioni. Ad esempio le variabili visibili sono rappresentate in un elenco, il codice è numerato riga per riga e c’è una separazione tra le informazioni mutevoli e soggette a più invii come la linea corrente e le variabili visibili rispetto alle informazioni statiche come la piattaforma, la versione di LLDB, il nome dell’eseguibile e il codice del file sorgente associato. Entrando nel dettaglio i dati sono raggruppati fondamentamente in 3 sezioni:

- **codeInfo**, sono le informazioni che afferiscono principalmente alla descrizione di cosa si sta facendo e cosa si vuole da ChatGPT, fornendo all’IA le informazioni basilari per la correzione del codice sorgente associato al target del debugging. Vanno mandate una sola volta per sessione di debugging essendo statiche e variando solo al variare dell’obiettivo del debugging. Il prompt in “codeInfo” è l’insieme di altri tre prompts, salvati nelle seguenti variabili: “intro”, “code” e “conclusion”. In “intro” viene specificato il sistema utilizzato, la versione di LLDB che si sta utilizzando su tale sistema per effettuare il debugging e il nome del file sorgente associato all’obiettivo del debugging, creato a partire dall’eseguibile caricato in LLDB. Il codice sorgente si trova in “code” ed è numerato. Nell’“api\_key\_mode” si va a capo per ogni riga del codice così da semplificare la lettura, ma ciò non è possibile per la “browser\_mode” in quanto il simbolo “\n” che specifica l’accao nelle stringhe, viene letto da ChatGPT come l’equivalente di “Enter” e quindi come “invio”. In tal modo se lasciassimo i “\n” nella modalità browser, allora il prompt di “send” verrebbe spezzettato in tante parti quanti sono i “\n”. Grazie alla numerazione delle righe, nella sperimentazione è stato verificato che ChatGPT comprende anche senza gli accapo il codice inviato. Nella variabile “conclusion” si specifica, infine, che non ci si vuole una risposta da ChatGPT in quanto ulteriori info verranno mandate e devono essere processate. Nel caso di utilizzo del debugging manuale con “send” e “ask”, mancano ancora le “stepInfo” prima di poter ricevere una risposta soddisfacente le quali vanno mandate con “send” assieme a codeInfo la prima volta e poi da sole per le future iterazioni del comando nella sessione.

- **stepInfo**, sono le informazioni che afferiscono alla parte più tecnica del debugging. Possono e devono essere mandate più volte così che ChatGPT possa controllare come varino i valori assegnati alle variabili nel corso del debugging a seconda della linea corrente sulla quale ci si trova. Il prompt è diviso in 4 parti ciascuna delle quali è un prompt: “intro”, “current\_line”, “variables” e “conclusion”. Nella variabile “intro” si specifica che le informazioni che verranno visualizzate sono riferite a questo momento. La prima informazione rilevante viene presentata in “current\_line” dove viene specificata la linea corrente sulla quale si è fermi e quindi oltre i valori delle variabili sono soggetti a variazioni. In “variables” è presentata la seconda informazione rilevante, qui vengono esposti con un elenco numerato le variabili visibili specificandone rispettivamente: nome, valore, e indirizzo. Così come in “codeInfo”, nella variabile “conclusion” si chiede di non fornire risposte, poiché in questo caso vanno ancora specificate le richieste, le quali nel debugging manuale saranno fatte dall’utente mentre nel debugging automatico saranno provviste di ulteriori informazioni circa le interruzioni e le ragioni delle interruzioni prima di richiedere quali siano gli errori nel codice dell’obiettivo del debugging. È bene specificare che così come per “codeInfo”, anche in “stepInfo”, per le medesime ragioni, nella modalità browser non vengono inseriti i “\n” che a livello estetico garantiscono maggiore leggibilità all’autore del plugin e la cui assenza non causa alcun problema a ChatGPT per la comprensione dei prompt.
- **automatedDebuggingInfo**, sono le informazioni recuperabili solo attraverso il comando “autodebug” e afferiscono al risultato e all’andamento dell’esecuzione del processo associato all’obiettivo del debugging. Il prompt restituito dalla funzione “automatedDebuggingInfo” è composto da 3 parti, ciascuna delle quali è un prompt: “program\_output”, “debugging\_result” e “ask\_for\_errors”. Nella variabile “program\_output” viene salvato l’output del programma, il quale è racchiuso all’interno delle virgolette inglesi: ‘’. Ogni riga dell’output è separata dalla successiva da un semplice spazio, convenzione attuata in ragione del fatto che ChatGPT non ha problemi a comprendere l’output anche se tutto su una linea come osservato dall’autore nella fase di sperimentazione. L’importante è lasciare almeno uno spazio tra una riga e l’altra, pertanto si è optato per questa convenzione. In “debugging\_result” vi possono essere prompt differenti a seconda del fatto che si sono rilevate o meno interruzioni nell’esecuzione del programma. Riepilogando quanto meglio spiegato nella sottosezione “autodebug” della sezione “comandi”: se non sono rilevate il prompt specificherà che il programma oggetto del debugging ha terminato la sua esecuzione senza interruzioni; se sono state rilevate verrà specificata la linea di interruzione in quanto linea corrente e la ragione dello stop, un segnale; se sono state rilevate ma dovesse essere stato corrotto l’indirizzo di memoria per le istruzioni successive, allora la linea corrente è conseguentemente errata e viene indicata solo la ragione dello stop. Infine in “ask\_for\_prompts” verrà chiesto quanti e quali sono gli errori nel codice e di classificarli per tipologia. Dalle sperimentazioni è stato il prompt con i risultati migliori in quanto il responso di ChatGPT sarà meno condizionato dalla richiesta. Ad esempio, chiedendo di differenziare gli errori in logici, sintattici, semantici e di runtime

(bloccanti o meno), il ChatBot tenderà a raggruppare gli errori presenti secondo queste categorie e per compiacere la richiesta tenderà a commettere più errori rispetto al tipo di errore pur di farlo rientrare in una delle categorie. Questo per lo meno è quanto osservato dalle sperimentazioni sui prompt. Lasciare quindi indefinite le tipologie di errori che si vogliono scovare fa invece sì che più che sulla classificazione, ChatGPT si focalizzi sullo scovare gli errori, il che è lo scopo principale del suo utilizzo. Un altro prompt possibile potrebbe richiedere un'elenco degli errori differenziati per tipologia, ma sempre dalle sperimentazioni risulta che seppure vi sia un beneficio in termini di chiarezza del responso, in questo modo il ChatBot si focalizza troppo sull'elencazione dei possibili errori, tendendo quindi a dividere un errore in più punti pur di fare un elenco corposo. Per queste ragioni si è scelto il prompt attuale: “Quanti e quali sono gli errori del codice? Differenziali per tipologia.\n” che come nei casi precedenti nella modalità browser avrà uno spazio “ ” al posto di “\n”.



## Capitolo 3

# Commento del codice

### 3.1 Struttura delle directory

La directory principale è “chatgpt-lldb-plugin”. Essa contiene l’intero plugin. Al suo interno vi sono la sottodirectory “main” e i file di configurazione e a carattere informativo:

- “**\_\_init\_\_.py**”, indica che la directory “chatgpt-lldb-plugin” è un pacchetto Python, i cui moduli sono importabili;
- “**api\_key\_config.py**”, contiene l’OpenAI API Key scelta dall’utente per le interazioni con ChatGPT attraverso le omonime API.;
- “**browser\_config.py**”, contiene i percorsi di installazione di Chrome Brower e del corrispondente Chromedriver necessari al funzionamento dell’interazione via interfaccia web, oltre che la variabile per la regolazione del tempo di attesa per le risposte di ChatGPT;
- “**LICENSE**”, contiene la licenza MIT del plugin;
- “**README.md**”, contiene le dipendenza da installare autonomamente prima di procedere con l’installazione del pacchetto, contiene inoltre la procedura di installazione e una panoramica dell’utilizzo del plugin;
- “**requirements.txt**”, contiene i requisiti per il funzionamento del software che verranno installati con il processo di installazione del plugin nelle rispettive versioni per le quali è garantito il funzionamento del plugin;
- “**setup.py**”, garantisce l’installazione del pacchetto, quindi l’installazione dei requisiti e il caricamento del plugin in LLDB.

Nella sottodirectory “main” si hanno le sottosottodirectory “api\_connection”, “browser\_connection”, “saves” e i file che costituiscono il nucleo del plugin:

- “**\_\_init\_\_.py**”, indica che la sottodirectory “main” è un pacchetto Python, i cui moduli sono importabili;
- “**commands.py**”, contiene tutti i comandi che il pacchetto integra a quelli già presenti in LLDB;

- “**main.py**”, viene eseguita solo durante l’installazione del plugin e si occupa del caricamento del plugin in LLDB per conto di “setup.py”;
- “**prompts.py**”, contiene tutti i prompts con le informazioni di debugging appropriatamente formalizzate per essere inviate, comprese e rielaborate da ChatGPT;
- “**stepping.py**”, si occupa del recupero delle informazioni di debugging attraverso le API di LLDB;
- “**version.py**”, contiene la versione del plugin, la quale viene letta da “setup.py” al momento dell’installazione.

Nella sottosottodirectory “api\_connection” viene gestita la connessione con ChatGPT attraverso le API di OpenAI e contiene i file “\_\_init\_\_.py” e “api.messaging.py”. Nella sottosottodirectory “browser\_connection” viene sempre gestita la connessione con ChatGPT però attraverso il pacchetto Python “selenium” che consente il controllo dell’interfaccia web resa disponibile dal browser utilizzato. Contiene: “\_\_init\_\_.py”, “browser.messaging.py”, “cleaner.py” e “previous\_chrome\_pids.yaml”. Per concludere, la sottosottodirectory “saves” contiene i salvataggi delle conversazioni con ChatGPT creati attraverso il comando “save” di “commands.py”.

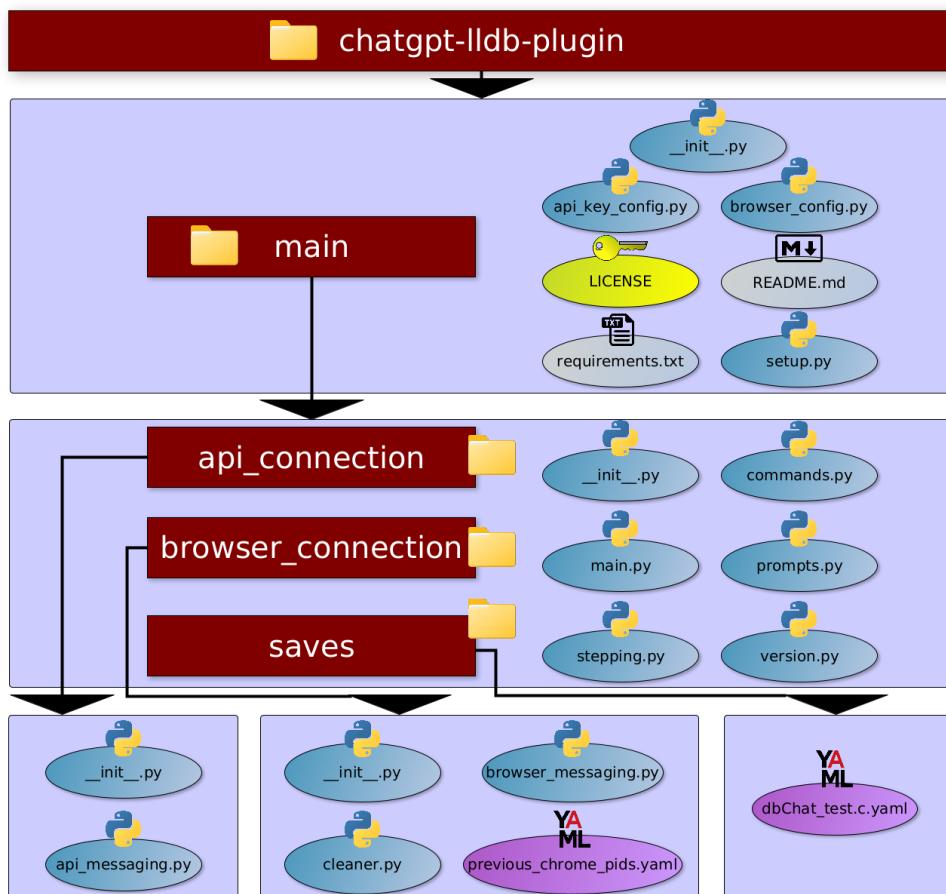


Figura 3.1. Schema della struttura delle directory

## 3.2 Directory 'chatgpt-lldb-plugin'

È stata pensata per contenere tutto il plugin, quindi il nucleo del programma in "main" e i file di configurazione per l'installazione e il tipo di modalità scelta, oltre che dei file a carattere informativo come il README e la licenza. Da qui in avanti si consiglia di consultare i commenti del codice con lo schema delle dipendenze dell'applicativo a lato.

### 3.2.1 Sottodirectory 'main'

#### 3.2.2 \_\_init\_\_.py

Contraddistingue la directory "chatgpt-lldb-plugin" come un pacchetto Python i cui moduli sono importabili. È stato lasciato vuoto non essendo necessaria alcuna inizializzazione aggiuntiva per il funzionamento del pacchetto.

**Listing 3.1.** chatgpt-lldb-plugin/\_\_init\_\_.py

#### 3.2.3 api\_key\_config.py

È stato ideato per consentire all'utente di impostare in modo semplice e statico l'OpenAI API Key per il funzionamento dell'interazione con ChatGPT attraverso le API omonime per Python. Il pacchetto "api.messaging" prenderà da qui la chiave per la gestione delle interazioni. Tuttavia questa chiave può anche non essere impostata, più avanti vedremo infatti che, in caso di mancata configurazione, è sempre possibile impostare l'OpenAI API Key come variabile d'ambiente per la sessione del terminale in corso. Avendo pensato questo file come per la configurazione e per essere modificato dall'utente è quindi minimale e si limita all'impostazione della variabile contenente l'OpenAI API Key piuttosto che alla gestione delle interazioni, onde evitare modifiche accidentali del codice da parte dell'utente che avrebbero potuto compromettere il funzionamento dell'applicativo. L'utente può recuperare la propria chiave API da sito di OpenAI all'indirizzo: platform.openai.com/account/api-keys.

```
1 OPENAI_API_KEY = "YOUR_OPENAI_API_KEY" #Exchange YOUR_OPENAI_API_KEY  
with your personal one.
```

**Listing 3.2.** chatgpt-lldb-plugin/api\_key\_config.py

#### 3.2.4 browser\_config.py

Così come il precedente file di configurazione, è pensato per essere modificato direttamente dall'utente, pertanto presenta solo le variabili necessarie per la gestione dell'interazione con ChatGPT tramite l'interfaccia web del browser che avviene invece nel pacchetto "browser\_connection" onde evitare modifiche accidentali dell'utente che possano compromettere la correttezza del plugin. Pertanto in questo file va specificato, nella variabile "chrome\_path", il percorso nel quale si è installato Chrome Browser, solitamente nei sistemi UNIX-like come GNU/Linux e MacOS è "/usr/bin/google-chrome-stable", mentre in Windows è "C:\Program Files\Google\Chrome\Application\chrome.exe". Nella variabile "chrome\_driver\_path"

va specificato invece il percorso al Chromedriver, il quale garantisce l'interazione tra il pacchetto Python “selenium” e il Chrome Browser. Il Chromedriver può essere scaricato dal web, facendo attenzione a trovare quello compatibile con la propria versione di Chrome Browser, e viene generalmente spostato in “/usr/bin/-chromedriver” nei sistemi UNIX-like e in “C:\Users\user\Desktop\chromedriver.exe” in Windows. La variabile globale “timeForResponse” specifica il tempo di attesa necessario all’elaborazione della risposta di ChatGPT alla richiesta specificata come argomento del comando “ask” di “commands.py”, oltre che al tempo di stampa sul browser della stessa. La cattura della risposta per mezzo del pacchetto Selenium deve essere effettuata dopo che la risposta di ChatGPT è stata scritta tutta, bisogna inoltre non aspettare troppo tempo onde rendere l’utente insofferente all’utilizzo dell’applicativo. Il tempo di 20 secondi è generalmente sufficiente per la ricezione completa delle risposte, lasciando qualche secondo di margine rispetto al tempo di stampa sul browser. L’utente da questo file può modificare tale tempo di attesa in modo statico in caso dovesse compiere richieste con risposta più breve o con risposta più lunga. È inoltre a disposizione il comando “wait” di “commands.py” che consente la modifica per la sessione in corso del tempo di attesa, da specificare come argomento del comando. È fondamentale dichiarare la variabile “timeForResponse” come globale, altrimenti non può essere modificata dal comando “wait” poiché questo dovrebbe modificare una variabile locale definita in un’altra funzione e questo non è possibile poiché “timeForResponse” sarebbe definita appunto al di fuori del blocco di “wait”. Ciò può essere fatto solo se la variabile in questione è globale.

```

1 #Specify the path for the chosen web browser and the relative
  webdriver executable.
2 chrome_path = r"/usr/bin/google-chrome-stable"
3 chrome_driver_path = r"/usr/bin/chromedriver"
4
5 #It is the time (seconds) you wait for ChatGPT responses: too big you
  wait to much; too short you may receive cut responses.
6 global timeForResponse
7 timeForResponse = 20

```

**Listing 3.3.** chatgpt-lldb-plugin/browser\_config.py

### 3.2.5 LICENSE

La licenza MIT consente l'utilizzo, la copia, la modifica, l'unione, la pubblicazione, la distribuzione, la sottolicenza e la vendita delle copie del software a condizione che tale licenza sia inclusa in tutte le copie del software. Specifica inoltre che il software è dato così com'è senza garanzie di alcun tipo. La scelta di utilizzare una licenza libera e permissiva come quella del MIT è motivata dalla volontà del creatore del progetto di lasciare libero l'utilizzo del plugin e lo sviluppo di tutte le sue potenzialità, anche da parte di altri programmati, anche se dovesse significare una proprietarizzazione di una copia del software. L'intenzione dell'autore è infatti pubblicare su GitHub il plugin ampliando l'utilizzo e il miglioramento del software a quanti più utenti e sviluppatori possibile. L'idea di una licenza proprietaria è stata scartata in principio sia perché limiterebbe la diffusione e il miglioramento del codice, sia perché ritengo pressoché inutile proteggere con una licenza proprietaria un codice scritto in Python, quindi interpretabile e non compilabile e pertanto visibile chiaramente e modificabile

da chiunque abbia una copia del software. D'altro canto non è stato sviluppato ai fini aziendali e quindi per porsi in una posizione di vantaggio nell'ambito della competitività del libero mercato, bensì al fine di presentare un lavoro di tesi che mostri le competenze dell'autore-laureando. La licenza MIT è quindi motivata anche dal disinteresse dell'autore per il controllo sull'utilizzo del copie del software in quanto ciò diverge dallo scopo del progetto: un buon lavoro di tesi. Resta comunque una soddisfazione personale se tutto il tempo e lavoro impiegato per questo progetto possa risultare veramente di aiuto a qualcuno semplificandogli il debugging e facendogli risparmiare tempo. Che questo possa accadere lo auguro dal profondo del mio cuore.

```
1 MIT License
2
3 Copyright (c) 2023 Carlo Capodilupo
4
5 Permission is hereby granted, free of charge, to any person obtaining
   a copy of this software and associated documentation files (the
   Software ), to deal in the Software without restriction,
   including without limitation the rights to use, copy, modify,
   merge, publish, distribute, sublicense, and/or sell copies of the
   Software, and to permit persons to whom the Software is furnished
   to do so, subject to the following conditions:
6
7 The above copyright notice and this permission notice shall be
   included in all copies or substantial portions of the Software.
8
9 THE SOFTWARE IS PROVIDED AS IS , WITHOUT WARRANTY OF ANY KIND,
   EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
   MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
   NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
   HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
   WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
   OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
   DEALINGS IN THE SOFTWARE.
```

**Listing 3.4.** chatgpt-lldb-plugin/LICENSE

### 3.2.6 README.md

Questo file è scritto usufruendo del linguaggio Markdown, un linguaggio di markup solitamente utilizzato per i file README in quanto consente di creare un testo formattato utilizzando un editor di testo semplice. Comprende tre sezioni:

- **Dependencies**, nella quale sono specificate le dipendenze, ovvero quei pacchetti necessari affinché il plugin possa essere installato e funzionare. Anzitutto viene specificato LLDB (versione == 11.0.1), il debugger nel quale integrare il plugin. In secondo luogo l'interprete Python3 (versione == 3.9.2) senza il quale non potremmo eseguire il codice Python nel quale è scritto il plugin. Infine il pacchetto Python 3 “setuptools” (versione 68.2.0) il quale è necessario per installare il plugin in modo semplice e automatico.
- **OS Independent Installation**, nella quale viene specificata l'installazione indipendentemente dal sistema operativo e comprendendo tra i vari passaggi: il download del plugin, il comando per l'installazione da terminale, le procedure

per configurare le due modalità di utilizzo disponibili che possono essere selezionate con il comando “switch”, come vedere i comandi che il plugin integra in lldb e la procedura di importazione manuale dei comandi a requisiti installati.

- **Overview**, nella quale viene offerta una panoramica su come possono essere utilizzati alcuni comandi offerti sia per il debugging manuale che per quello automatico.

```

1 <h1><p align="center">ChatGPT LLDB Plugin</p></h1>
2
3 ## Dependencies
4   - lldb == 11.0.1
5   - python3 == 3.9.2
6   - setuptools == 68.2.0
7
8 ## OS Independent Installation
9
10 1. Download chatgpt-lldb-plugin.zip .
11 2. Install the development package, use 'cd chatgpt-lldb-plugin',
12    then 'pip install -e .'
13 3. Open '/chatgpt-plugin/api_key_config.py' and replace 'YOUR_OPENAI_API_KEY' with your OpenAI API key or just enter 'export OPENAI_API_KEY=<YOUR_OPENAI_API_KEY>' from terminal before launching LLDB. Grab an API key from [https://platform.openai.com/account/api-keys](https://platform.openai.com/account/api-keys). If you don't want to use an API key, you can run the program as well with the 'browser_mode' option making use of your OpenAI account. In this case you first have to install Chrome and the corresponding Chromedriver and secondly update the '/chatgpt-plugin/browser_config.py' file. Once opened LLDB, remember you can switch among the 'api_key_mode' (default) and the 'browser_mode' using the 'switch' command.
14 4. Run 'lldb'. Use the 'info' command to see all the available commands.
15 5. If something during the installation went wrong you can manually import the plugin in your LLDB session with the LLDB command: 'command script import PATH/TO/chatgpt-lldb-plugin/main/commands.py '.
16
17 ## Overview
18 Basically you can send the source code you're debugging, the visible variables and the current line with the 'send' command. So before using it, make sure to have set at least a breakpoint and to have runned the program with the LLDB commands: 'b LINE_CHOSEN' and 'r '. You can now use 'send' and afterward the 'ask YOUR_REQUEST' command to ask anything you want to chatGPT. You can quit the LLDB session with 'q'. Example:
19
20 $ lldb EXECUTABLE_NAME
21 (lldb) b --name main
22 (lldb) r
23 (lldb) send
24 (lldb) ask Do you have any suggestions to improve my code?
25 (lldb) next
26 (lldb) send

```

```

26     (lldb) ask What's the line I'm stepping on?
27     (lldb) q
28
29 Use the command 'autodebug' to automatically track down bugs in your
  code. You just have to type it and wait for ChatGPT's response.

```

Listing 3.5. chatgpt-lldb-plugin/README.md

### 3.2.7 requirements.txt

In questo semplice file di testo sono indicati i requisiti per il corretto funzionamento del plugin nelle rispettive versioni per le quali è garantito il funzionamento del plugin. Si tratta di pacchetti Python che verranno installati grazie a "setup.py" il quale leggerà proprio da questo file i pacchetti da installare. Si tratta di: "cpython-lldb" (versione == 0.3.2) affinché si possa usufruire delle API di LLDB messe a disposizione per Python; "openai" (versone == 0.28.0) affinché si possa usufruire delle API di OpenAI sempre per Python; psutil (versione == 5.9.5) così che "cleaner.py" nel pacchetto "browser\_connection" possa gestire la chiusura del processo Chrome Browser ogni qualvolta si cambia modalità oppure in caso di chiusura di LLDB; PyYAML (versione == 6.0.1) per la gestione dei file YAML in cui vengono salvate le conversazioni tra utente e il ChatBot e nel quale viene conservato il PID del processo Chrome Browser sempre per la chiusura dello stesso attraverso la gestione in "cleaner.py"; selenium (versione == 4.9.0) per la gestione dell'interfaccia web nella quale, quando si è in modalità browser, vanno inserite le richieste per il ChatBot e prese le risposte dello stesso non appena elaborate, tutto ciò è molto altro avviene grazie al pacchetto selenium che una volta importato come modulo consente l'identificazione e la gestione di parti specifiche dell'interfaccia web attraverso il codice Python. È importante notare che versioni successive alla 4.9.0 per il pacchetto "selenium" compromettono il funzionamento del plugin e quindi della modalità browser.

```

1 cpython-lldb==0.3.2
2 openai==0.28.0
3 psutil==5.9.5
4 PyYAML==6.0.1
5 selenium==4.9.0

```

Listing 3.6. chatgpt-lldb-plugin/requirements.txt

### 3.2.8 setup.py

Qui viene gestita l'installazione del plugin. Anzitutto vengono importati i moduli necessari da "setuptools" e dalla libreria standard Python ("re", "os.path" e "subprocess"). Recupero il percorso completo della directory in cui si trova il file Python in questione ovvero "setup.py", grazie a tale percorso infatti possiamo attraverso un join recuperare il percorso al file "version.py" e salvarci la versione in una variabile. In un'altra variabile salviamo invece il contenuto del file "README.md" e in un'altra i requisiti letti dal file "requirements.txt". Il persorso non è necessario specificarlo trovandosi entrambi i file nella stessa directory di "setup.py". Se il modulo corrente è il modulo principale, ovvero è eseguito direttamente senza essere importato come

modulo da un altro programma, quindi quando si avvia l'installazione, allora al posto dell'implementazione predefinita del comando "develop", "setuptools" utilizzerà la classe "CustomDevelopCommand" la quale esegue il comando della CLI "chatgpt-lldb-plugin" con "subprocess.run" e quindi in sincrono rispetto l'esecuzione corrente, pertanto quella corrente verrà bloccata per il tempo necessario all'esecuzione del comando esterno e poi ripresa. L'esecuzione di questo comando, tuttavia, non avverrà in questo momento, ma più avanti. Dobbiamo infatti ancora definire cosa fa il comando "chatgpt-lldb-plugin" e avviare il comando "develop" che abbiamo appena personalizzato. Quanto fatto in questo passo, infatti, serve solo per creare il comando personalizzato che vogliamo eseguire nel corso dell'installazione. L'idea è questa: una volta installati i requisiti, chiamare la funzione main in "main.py" nella sottodirectory "main" la quale si occupa dell'importazione nel file ".lldbinit" del comando LLDB per l'importazione dei comandi aggiunti dal plugin. Questo è necessario poiché LLDB al suo avvio leggerà ed eseguirà tutti i comandi nel file ".lldbinit" e ci consente di importare ad ogni avvio i nostri nuovi comandi. L'importante è scrivere "command script import /home/user/chatgpt-lldb-plugin/main/commands.py" in questo file. Proseguendo, viene definita la funzione "setup", specifichiamo quindi il nome del programma, la versione ricavataci precedentemente, il nome dell'autore e la sua email, la descrizione del software, il contenuto del file README "README.md" salvatoci prima e il tipo di linguaggio dello stesso, ovvero Markdown. Viene inoltre specificato il sito web dell'autore, in questo caso la sua pagina su GitHub, i pacchetti del plugin, quindi "main", "api\_connection" e "browser\_connection", i requisiti per l'installazione del plugin che verranno installati nel processo di installazione del plugin, alcuni classificatori come l'ambiente di utilizzo del plugin ovvero la CLI, il linguaggio di programmazione del pacchetto ovvero Python 3, la licenza ovvero quella MIT, a chi è destinato il progetto ovvero l'ambito educativo e il sistema operativo, in tal caso avendo scritto il plugin in Python e non avendo utilizzato codice in modo dipendente rispetto al sistema operativo, il plugin è da considerarsi indipendente rispetto il sistema operativo. Si specifica anche la versione di Python 3 che deve essere maggiore o uguale alla 3.9.2. A questo punto possiamo dichiarare i "console scripts" ovvero comandi o programmi che possono essere eseguiti da una riga di comando dopo che il pacchetto è stato eseguito. Ne definiremo solo uno, il comando "chatgpt-lldb-plugin" che si occupa di chiamare la funzione "main" in "chatgpt-lldb-plugin/main/main.py", in questo modo, in qualsiasi momento possiamo utilizzare questo comando per importare in LLDB i nostri comandi. Questo può tornare utile in caso il file ".lldbinit" dovesse essere eliminato o sovrascritto con conseguente impossibilità di caricare i nostri comandi, il tutto senza reinstallare necessariamente il plugin. Adesso possiamo chiamare il nostro comando personalizzato, questo si occuperà di eseguire il comando CLI "chatgpt-lldb-plugin" il quale, come definito sopra, chiama la funzione main che carica i nuovi comandi in LLDB. Pertanto, una volta che si sarà avviato lo script "setup.py" a seguito dell'avvio dell'installazione con "cd chatgpt-lldb-plugin && pip install -e ." verranno prima installati i requisiti, definito il comando CLI "chatgpt-lldb-plugin" per l'importazione dei comandi del plugin e infine eseguito questo comando.

```
1 from setuptools import find_packages, setup
2 import re
3 from os import path
```

```

4 import subprocess
5
6 FILE_DIR = path.dirname(path.abspath(path.realpath(__file__)))
7 with open("README.md", "r", encoding="utf-8") as fh:
8     long_description = fh.read()
9
10 with open('requirements.txt') as f:
11     install_requirements = f.readlines()
12
13 with open(path.join(FILE_DIR, 'main', 'version.py')) as f:
14     version = re.match(r'^__version__ = "([\w\.]*)"$', f.read().strip())[1]
15
16 if __name__ == "__main__":
17     from setuptools.command.develop import develop
18
19     class CustomDevelopCommand(develop):
20         def run(self):
21             develop.run(self)
22             subprocess.run(["chatgpt-lldb-plugin"])
23
24 setup(
25     name="chatgpt-lldb-plugin",
26     version=version,
27     author="Carlo Capodilupo",
28     author_email="capodilupo@proton.me",
29     description="ChatGPT LLDB Plugin",
30     long_description=long_description,
31     long_description_content_type="text/markdown",
32     url="https://github.com/lupidux/chatgpt-lldb-plugin",
33     packages=find_packages(),
34     package_data={
35         'main': [
36             'api_connection/*',
37             'selenium_connection/*',
38         ],
39     },
40     install_requires=install_requirements,
41     classifiers=[
42         "Environment :: Console",
43         "Programming Language :: Python :: 3",
44         "License :: OSI Approved :: MIT License",
45         "Intended Audience :: Education",
46         "Operating System :: OS Independent",
47     ],
48     python_requires=">=3.9.2",
49     entry_points={
50         'console_scripts': [
51             'chatgpt-lldb-plugin = main:main',
52         ],
53     },
54     cmdclass={'develop': CustomDevelopCommand},
55 )

```

Listing 3.7. chatgpt-lldb-plugin/setup.py

### 3.3 Sottodirectory 'main'

- 3.3.1 Sottosottodirectory 'api\_connection'
- 3.3.2 Sottosottodirectory 'browser\_connection'
- 3.3.3 Sottosottodirectory 'saves'
- 3.3.4 \_\_init\_\_.py

Contraddistingue la sottodirectory "main" come un pacchetto Python i cui moduli sono importabili. È stato lasciato vuoto non essendo necessaria alcuna inizializzazione aggiuntiva per il funzionamento del pacchetto.

**Listing 3.8.** chatgpt-lldb-plugin/main/\_\_init\_\_.py

#### 3.3.5 commands.py

Importiamo i moduli necessari al funzionamento dello script. Dalla libreria standard abbiamo "os", "os.path", "time", "subprocess" e "multiprocessing", dal pacchetto "PyYAML" abbiamo "yaml". Abbiamo poi bisogno di importare i moduli interni al plugin che altro non sono che altri file. Per l'utilizzo della "api\_key\_modality" abbiamo bisogno del modulo "api.messaging" dal pacchetto "api\_connection". Per l'utilizzo della "browser\_connection" abbiamo invece bisogno del modulo "browser.messaging" dal pacchetto "browser\_connection". Abbiamo inoltre bisogno del modulo "browser\_config" dal quale recuperare il valore della variabile "timeForResponse". Il modulo "prompt" sarà poi fondamentale poiché da qui riprenderemo i prompt per l'utilizzo non solo dei comandi "send" e "autodebug", ma prenderemo anche il nome del file sorgente utile per i comandi "code", "save" e "load". Passiamo quindi in rassegna comando per comando, ognuno di questi riceve gli argomenti "debugger", "command", "result", "internal\_dict". Sebbene solitamente siano utilizzati solo i primi due, vanno inseriti tutti nella dichiarazione della funzione che definisce il comando. Il comando "mode" stampa a schermo la modalità d'uso per le interazioni con ChatGPT. Ve ne sono due: l'"api\_key\_mode" e la "browser\_mode". Prima di definire il comando, dichiariamo come globale la variabile "modality" e le assegnamo il valore "api\_key\_mode". In tal modo la modalità di default sarà quella che utilizza le API OpenAI, potremo poi cambiarla con il prossimo comando. È importante dichiarare "modality" globalmente, così che possa essere letta anche nelle altre funzioni che definiscono i comandi e a seconda del valore che si ha differenziare le operazioni da svolgere, diverse per ciascuna modalità sebbene il risultato ultimo sia equivalente. Il comando "switch" gestisce il passaggio da una modalità all'altra, pertanto può essere utilizzato in entrambe. All'interno del comando, definiamo anzitutto la funzione "cleaner\_caller" che ci consentirà di avviare in modo asincrono, e quindi parallelamente all'esecuzione del programma, lo script "cleaner.py". Per fare ciò utilizzeremo la funzione "Popen" della libreria "subprocess" e onde evitare di visualizzare eventuali messaggi di errore, specifichiamo il secondo argomento in modo tale che questo rediriga lo stream di output degli errori standard verso un "null device". Spiegheremo tra poco a cosa servirà eseguire in parallelo lo script "cleaner.py". Poniamo degli "if" condizionali. Se ci troviamo nella modalità "api\_key\_mode" (da

qui in poi anche detta “modalità API”) ciò che vogliamo è anzitutto chiudere il browser utilizzato nelle sessioni precedenti semmai fosse aperto e poi inizializzarne un altro. In caso di successo possiamo cambiare il valore di “modality” impostandolo su “browser\_mode” e porre in esecuzione parallela e concorrente “cleaner\_caller()”, script che monitora lo stato del processo “lldb” e in caso di chiusura dello stesso provvede in modo indipendente a chiudere anche il browser. Ci si potrebbe chiedere allora perché essere preventivi assicurandosi come prima istruzione dell’“if” di chiudere il browser della sessione precedente se alla fine della sessione il browser viene chiuso automaticamente. Ebbene, questa è una misura precauzionale, qualcosa potrebbe andare infatti storto semmai la priorità del processo dovesse essere abbassata a seguito di un utilizzo intenso della CPU da parte dell’utente, pertanto meglio agire in prevenzione. Semmai il processo della sessione precedente dovesse infatti risultare aperto, ciò comprometterebbe l’utilizzo della modalità browser. Se l’inizializzazione non ha avuto successo stampiamo un messaggio di errore, la modalità non è stata cambiata. Quanto al secondo “if” condizionale, quello che gestisce il passaggio in caso ci si trovi nella modalità browser per passare nella modalità API, in questo caso chiudiamo il browser aperto per l’utilizzo dell’interfaccia browser e assegnamo alla variabile “modality” il nome della modalità nella quale si vuole passare, ovvero “api\_key\_mode”. Il comando successivo è implementato dall’omonima funzione “code”. Si occupa di stampare a schermo l’intero codice associato all’eseguibile caricato come obiettivo di debugging. Tale codice sarà numerato riga per riga onde facilitare all’utente la lettura dello stesso. Recuperiamo il codice sorgente da “prompts.py” che lo recupera a sua volta da “stepping.py” e se il recupero è andato a buon fine viene stampato a schermo. In caso l’operazione non dovesse andare a buon fine, cosa che accade quando non è impostato un breakpoint e posto in esecuzione il programma, allora verrà stampato a schermo un messaggio di errore che specifica questa cosa. Si tratta di un’eccezione definita all’interno di “stepping.py” definita con un “try, except”, la vedremo poi al commento di “stepping.py”. Si passa adesso all’implementazione di uno dei comandi principali: “send”. Dichiariamo le variabili “codeAlreadySent” e “startedChat” come globali e assegniamole il booleano “False”, ciò significa che nella sessione corrente, quando la funzione che implementa il comando “send” non è ancora chiamata, noi consideriamo il codice come non inviato e la chat con ChatGPT via interfaccia web come non ancora aperta, tra poco entreremo nel dettaglio, l’idea è inviare il codice con le informazioni di sistema una sola volta e aprire una nuova chat una sola volta, altrimenti ad ogni “send” avremo una nuova chat e perderemmo quindi traccia di quanto avvenuto prima. Dichiariamo all’interno della funzione la variabile globale “modality” così da differenziare le operazioni da eseguire per ciascuna modalità. Dichiariamo poi la variabile nella quale vogliamo riportare le informazioni da inviare a ChatGPT e ricaviamo queste, già formattate, dal file “prompts.py”. Se qualcosa è andato storto e le informazioni ottenute non sono arrivate o sono arrivate solo parzialmente, possiamo proseguire, altrimenti verrà stampato il messaggio dell’eccezione apposita definita in “stepping.py” dalla quale “prompts.py” prende le info che fornisce noi qui. Dichiariamo internamente alla funzione la variabile globale “codeAlreadySent”, se il codice e le informazioni di sistema non sono state inviate, allora possiamo includere nelle informazioni da inviare anche queste e poi aggiungere le “stepInfo”, informazioni sulla linea corrente e le variabili visibili. Viceversa, non serve includere queste informazioni poiché sono

state già inviate, quindi inseriremo in “debuggingInfo” solamente le “stepInfo”. Ora, se siamo nella modalità API possiamo chiamare la funzione “send\_to\_chatgpt()” del pacchetto “api\_connection” e inviare l’argomento. Se siamo, invece, nella modalità browser dichiariamo internamente alla funzione “startedChat” e controlliamo se la chat non è stata ancora aperta, in questo caso possiamo iniziare una nuova chat e inviare le informazioni di debug tramite la funzione “send\_to\_chatgpt()” del pacchetto “browser\_connection”. Facciamo attenzione a specificare un secondo argomento, in questo modo, con “noReplayNoWait”, intendiamo dire alla funzione che non è nostra intenzione aspettare una risposta come avviene invece nel caso del comando “ask”. In questo modo ci evitiamo di aspettare il “timeForResponse” che di norma è di 20 secondi. Se, per entrambe le modalità, tutto è andato bene, allora possiamo segnare che il codice è stato inviato correttamente e non servirà poi rimandarlo. Anche la prossima funzione è importante poiché implementa il comando “ask” grazie al quale possiamo fare una richiesta a piacere a ChatGPT. Dichiariamo internamente alla funzione la variabile globale “modality” per differenziare le modalità. Recuperiamo l’input dell’utente e inseriamolo in “userRequest”. Dall’ottica dell’utente egli vedrà questo input che inserisce come fosse un argomento per “ask”. Se l’input utente è presente possiamo proseguire, altrimenti dobbiamo notificarlo dell’assenza della sua richiesta. Proseguendo, se siamo nella modalità API possiamo inviare la richiesta dell’utente e salvarla in “reply”. Nella modalità browser faremo lo stesso facendo però attenzione ad aprire una nuova chat solo se la chat della sessione ancora non è stata aperta, in tal caso la apriamo è impostiamo al booleano “True” la variabile “startedChat”. Dobbiamo fare inoltre attenzione a specificare un secondo argomento per la funzione “send\_to\_chatgpt.py” della modalità browser. Si tratta di “waitForReply” che fa sì che si aspetti il “timeForResponse” prima di catturare dall’interfaccia browser e inviare noi la risposta di ChatGPT. Se abbiamo la risposta, questa, che è salvata in “reply”, può essere stampata a schermo. Il prossimo comando serve a modificare, nel corso della sessione in corso e quindi solo temporaneamente, la variabile “timeForResponse” che definisce il tempo di attesa per le risposte alla richieste personalizzate dell’utente fatte con il comando “ask”. Sto parlando del comando “wait”. Se ci troviamo nella modalità API poniamo un messaggio per avvisare l’utente che in questa modalità non si può utilizzare tale comando, ma solo nella modalità browser. Se ci troviamo già nella modalità browser prendiamo da input il nuovo valore da utilizzare al posto di quello attuale. Da parte dell’utente questo valore risulta come fosse un argomento per il comando “wait”. Se l’argomento è presente allora lo assegniamo alla variabile globale “timeForResponse” dopo averlo convertito in intero. Altrimenti se l’argomento è mancante occorre ricordare all’utente di inserirlo. Si passa ora ad uno dei comandi cardine dell’applicativo, comando del quale al momento possiamo vedere solo il lato più superficiale. A seconda della modalità varia un singolo aspetto. Se siamo nella modalità API chiamiamo la funzione “automatedDebuggingInfo” di “prompts.py” la quale ritorna le informazioni sull’esecuzione del debugging automatico quindi se si è interrotta l’esecuzione con la ragione dello stop, l’output e la richiesta da inviare a ChatGPT per la correzione del codice. Possiamo ora usare il comando “send” che invierà il codice, le informazioni di sistema, la linea corrente del punto in cui si è fermata l’esecuzione se si è fermata nel corso del debugging automatico e le variabili visibili fino a quel punto. Adesso possiamo inviare le informazioni

recuperate dal debugging automatico attraverso il comando “ask”. È importante non utilizzare “send” prima dell’esecuzione del debugging automatico di “prompts.py” poiché abbiamo prima bisogno di far partire l’esecuzione per usare “send”, inoltre è bene usarla così da inviare le informazioni lungo una linea corrente rilevante come ad esempio quella nella quale l’esecuzione si è stoppata. Nella modalità browser, l’unica cosa che cambia è che tra la “send” e “ask” viene posto un “time.sleep()” pari ad un quarto del “timeForResponse” totale. Dobbiamo infatti aspettare un minimo prima di poter usare “ask”. Non ci è infatti possibile inviare nulla tramite l’interfaccia web se prima ChatGPT non ha elaborato la risposta per la “send”. Tale risposta è si irrilevante ma ChatGPT non può fare a meno di rispondere ad ogni nostro messaggio essendo un ChatBot così programmato. Pertanto questo breve tempo di attesa che di default è di 5 secondi risulta ad ora necessario. Si è deciso di far dipendere questo tempo da “timeForResponse” onde non elaborare un ulteriore comando per gestire questo tempo e al contempo non lasciare che un tempo fisso potesse compromettere il funzionamento di “autodebug” in caso di risposte a “send” più lunghe del tempo impostato. I prossimi 4 comandi si occupano del salvataggio e del caricamento delle chat con ChatGPT in modo da salvare o riutilizzare lo stato di una determinata sessione. Va detto tuttavia che vanno a perdere i breakpoint e lo stato del processo associato all’obiettivo di debugging. Prima del comando “save” anzitutto dichiariamo la variabile globale “alreadySaved” e assegnamole il booleano “False” in questo modo stiamo dicendo che per la sessione corrente non è stato ancora fatto alcun salvataggio, questa informazione ci sarà utile nella modalità browser, spiegheremo poi il perché. Recuperiamo il nome del file sorgente, ci è necessario per dare un nome univoco al file in cui salveremo il contenuto della chat e poi a recuperarlo. In caso di insuccesso verrà stampato il messaggio dell’eccezione definita in “stepping.py” da cui “prompts.py” recupera il nome del file sorgente. In caso di successo e quindi di nome del file sorgente presente possiamo procedere. Recuperiamo il percorso nel quale salvare la chat, si tratta della cartella “saves” che si trova nel pacchetto “main”. Utilizzando il nome del file sorgente definiamo il nome del file che conterrà la chat: “dbChat\_NOMEDELFILESORGENTE.yaml”. Unendo il percorso della directory “saves” al nome del file in cui salvare la chat otterremo il percorso al file in cui salvare la chat. Se la cartella “saves” non esiste allora la creiamo. Ora a seconda della modalità in cui ci troviamo agiremo diversamente. Se siamo nella modalità API recuperiamo e salviamo nella variabile locale “history” la storia dei messaggi scambiati con ChatGPT recuperabile dal pacchetto “api\_connection” che tiene traccia, solo per la sessione in corso, della conversazione con ChatGPT. Scriviamo quindi il contenuto di “history”, una lista di dizionari, nel file del quale abbiamo prima specificato il percorso. Se stiamo invece nella modalità browser dichiariamo internamente alla funzione la variabile globale “alreadySaved”. Se il salvataggio non è già avvenuto per la sessione corrente, controlliamo se esiste già un salvataggio all’interno del nostro account OpenAI, se esiste cancelliamolo per poterlo sostituire con il nuovo, altrimenti procediamo direttamente rinominando la chat corrente in modo univoco utilizzando il nome del file sorgente, in questo modo potremo poi ricondurci ad essa. Se l’operazione ha successo l’operazione restituisce il booleano “True” che viene assegnato in “alreadySaved”. Se la chat è già stata salvata per la sessione corrente, non possiamo fare nulla se non avvisare l’utente della cosa. C’è una ragione per la quale si utilizza la variabile “alreadySaved”, in caso

consentissimo infatti di fare più salvataggi, finiremmo per cancellare tutte le chat presenti nell'account OpenAI che utilizziamo per interagire con ChatGPT attraverso l'interfaccia browser. Infatti al prossimo utilizzo di "save" verrebbe rinominata la chat correntemente selezionata, poi al prossimo verrebbe cancellata in quanto il comando sovrascrive il salvataggio attuale e poi così di nuovo se riutilizzassimo altre due volte "save". Questo dipende non solo da come è stato implementato il comando ma soprattutto dal fatto che utilizzando l'interfaccia web i salvataggi si trovano tutti in colonna come in una pila di elementi all'eliminazione di uno il secondo va in testa e questo causa la selezione di questo. Al momento non si è potuto operare diversamente pertanto la funzione di salvataggio limiterà le sovrascrizioni ad una soltanto per sessione. Il prossimo comando è "load" con il quale carichiamo la chat salvata nell'applicativo all'interno della sessione corrente. Anzitutto recuperiamo il nome del file sorgente così da identificare univocamente la chat salvata. Se il nome è stato recuperato correttamente procediamo, altrimenti viene stampato il messaggio dell'eccezione definita in "stepping.py". Recuperiamo quindi il percorso alla cartella "saves", usiamo il nome del file sorgente per ricavare il nome del file utilizzato per salvare il contenuto della chat e uniamo questi due elementi per trovare il percorso completo al file cui siamo interessati. Se siamo nella modalità API, dovremo aprire il file di cui abbiamo recuperato il percorso in modalità di sola lettura e salvare nella variabile globale "history" il contenuto della lista di dizionari salvata. La variabile globale "history" contiene tutti i messaggi scambiati con ChatGPT salvati prima nel file "yaml" e viene inviata ogni volta a ChatGPT, costituisce infatti la memoria di ChatGPT per la sessione corrente. In caso di file non trovato si stampa un opportuno messaggio di errore e così uguale nel caso ci siano stati problemi per l'apertura del file. Se ci troviamo invece nella modalità browser dichiariamo internamente alla funzione la variabile globale "startedChat" e le restituiamo il contenuto di "select\_chat()" del pacchetto "browser\_connection". Utilizziamo come secondo argomento "load" solo per far sì che in questo caso solo venga stampato il messaggio di mancata presenza del salvataggio per il nome del file fornito. Tale messaggio è preferibile invece non stamparlo nel caso si stia utilizzando il comando "save" e pertanto abbiamo utilizzato questo argomento per specificare questa nostra volontà quando si utilizza il comando "save". Il prossimo comando è implementato grazie alla funzione expChat e prende il nome di "export" sebbene questo lo vedremo al momento dell'aggiunta dei comandi a LLDB, per ora infatti il comando si chiama "expChat" fino ad almeno alla ridenominazione. Il comando è utilizzabile solo nella modalità API, se ci si troverà nella modalità browser allora verrà stampato a schermo un messaggio per rendere la cosa nota all'utente. Il comando serve infatti ad esportare la chat all'interno di un file il cui percorso è specificato dall'utente, è una sorta di "salva con nome" e pertanto perde di significato nella modalità browser, anche in ragione del fatto che le chat sono salvate in automatico nel proprio account OpenAI quando si utilizza l'interfaccia browser. Nella modalità API, quindi, recuperiamo da input il percorso deciso dall'utente per il salvataggio del file, l'utente lo specificherà come argomento del comando. A questo percorso aggiungiamo l'estensione ".yaml". In caso le directory specificate non dovesse esistere vengono qui create. Si recupera poi la chat dalla variabile "history" del pacchetto "api\_connection" e scriviamola poi nel file apposito sovrascrivendolo se già presente. Il comando "import" così come il precedente possiede un'altra denominazione che verrà poi cambiata nel

momento di aggiunta dei comandi a LLDB. È stato definito solo per la modalità API e l'implementazione è speculare a quella di “export”. Recuperiamo da input il percorso del file dal quale si vuole importare la lista di dizionari con la conversazione con ChatGPT. Poi apriamo in sola lettura il file e salviamone il contenuto nella variabile globale “history” definita nel pacchetto “api\_connection”, la cosiddetta memoria di ChatGPT per la connessione API. In caso di mancata presenza del file oppure di errori nell'apertura stampiamo i rispettivi messaggi di errore. Con il prossimo comando di fatto è come se cancellassimo la memoria a ChatGPT per la sessione corrente. Nella modalità API ci basterà svuotare il contenuto della variabile globale “history” con la funzione “clear()” mentre nella modalità browser ci basta selezionare una nuova chat. Il comando “info” stampa l'elenco di tutti i comandi dell'applicativo provvisti di una breve descrizione e se necessario di un esempio di utilizzo. Infine definiamo la funzione “\_\_lldb\_init\_module” la quale viene chiamata automaticamente quando un modulo personalizzato per LLDB viene inizializzato, al suo interno possiamo scrivere i comandi LLDB che ci consentono di aggiungere i comandi da noi definiti in LLDB. Nel caso di “expChat” e “impChat” eseguiamo anche una ridenominazione in “export” e “import” che al momento della definizione delle rispettive funzioni non era possibile.

```
1 import prompts
2 import api_connection.api.messaging as api.messaging
3 import browser_connection.browser.messaging as browser.messaging
4 import browser_config
5 import os
6 from os import path
7 import time
8 import yaml
9 import subprocess
10 import multiprocessing
11
12 modality = "api_key_mode"
13 def mode(debugger, command, result, internal_dict):
14     global modality
15     print(modality)
16
17 def switch(debugger, command, result, internal_dict):
18     def cleaner_caller():
19         cleaner_path = path.join(path.dirname(path.abspath(path.
realpath(__file__))), "browser_connection", "cleaner.py")
20         subprocess.Popen(["python3", cleaner_path], stderr=subprocess.
DEVNULL)
21
22     global modality
23     if modality == "api_key_mode":
24         browser.messaging.quit()
25         initialization = browser.messaging.init()
26         if initialization == "done":
27             modality = "browser_mode"
28             cleaner_proc = multiprocessing.Process(target=
cleaner_caller)
29             cleaner_proc.start()
30         else:
31             print("You must log in into your ChatGPT account to use 'browser_mode'")
```

```

32     elif modality == "browser_mode":
33         browser_messaging.quit()
34         modality = "api_key_mode"
35
36 def code(debugger, command, result, internal_dict):
37     sourceCode = prompts.code(debugger)
38     if sourceCode:
39         print(sourceCode)
40
41 codeAlreadySent = False
42 startedChat = False
43 def send(debugger, command, result, internal_dict):
44     global modality
45     debuggingInfo = ""
46
47     codeInfo = prompts.codeInfo(debugger)
48     stepInfo = prompts.stepInfo(debugger)
49     if codeInfo and stepInfo is not None:
50         global codeAlreadySent
51         if codeAlreadySent is False:
52             debuggingInfo = codeInfo
53
54         debuggingInfo = debuggingInfo + stepInfo
55         if (modality == "api_key_mode"):
56             reply = api.messaging.send_to_chatgpt(debuggingInfo)
57         elif (modality == "browser_mode"):
58             global startedChat
59             if startedChat is False:
60                 startedChat = browser_messaging.new_chat()
61             reply = browser_messaging.send_to_chatgpt(debuggingInfo,
62 "noReplynoWait")
63
64         if reply:
65             codeAlreadySent = True
66
66 def ask(debugger, command, result, internal_dict):
67     global modality
68
69     userRequest = command.strip()
70     if userRequest:
71         if (modality == "api_key_mode"):
72             reply = api.messaging.send_to_chatgpt(userRequest)
73         elif (modality == "browser_mode"):
74             global startedChat
75             if startedChat is False:
76                 startedChat = browser_messaging.new_chat()
77             reply = browser_messaging.send_to_chatgpt(userRequest, "waitForReply")
78
79         if reply:
80             print(reply)
81     else:
82         print("You must place an argument: YOUR_REQUEST")
83
84 def wait(debugger, command, result, internal_dict):
85     if (modality == "api_key_mode"):
86         print("This command is supported only in 'browser_mode'")
```

```
87     elif (modality == "browser_mode"):
88         newWaitTime = command.strip()
89         if newWaitTime:
90             browser_config.timeForResponse = int(newWaitTime)
91         else:
92             print("You must place an argument: TIME_IN_SECONDS")
93
94 def autodebug(debugger, command, result, internal_dict):
95     if (modality == "api_key_mode"):
96         look_for_errors_request = prompts.automatedDebuggingInfo(
97             debugger)
98         debugger.HandleCommand('send')
99         debugger.HandleCommand(f'ask {look_for_errors_request}')
100    elif (modality == "browser_mode"):
101        look_for_errors_request = prompts.automatedDebuggingInfo(
102            debugger)
103        debugger.HandleCommand('send')
104        wait_to_write = browser_config.timeForResponse / 4
105        time.sleep(wait_to_write) #It is recommended to wait at least
106        5 seconds.
107        debugger.HandleCommand(f'ask {look_for_errors_request}')
108
109 alreadySaved = False
110 def save(debugger, command, result, internal_dict):
111     src_name = prompts.sourceFile_name(debugger)
112
113     if src_name is not None:
114         dirName = path.join(path.dirname(path.abspath(path.realpath(
115             __file__))), "saves")
116         fileName = "dbChat_"+src_name+".yaml"
117         filePath = path.join(dirName, fileName)
118
119         if not path.exists(dirName):
120             os.makedirs(dirName)
121
122         if (modality == "api_key_mode"):
123             history = api.messaging.retrieve_chat()
124             with open(filePath, "w") as file:
125                 yaml.dump(history, file)
126         elif (modality == "browser_mode"):
127             global alreadySaved
128             if alreadySaved is False:
129                 chatAlreadyExisting = browser.messaging.select_chat(
130                     src_name, "save")
131                 if chatAlreadyExisting:
132                     browser.messaging.delete_chat(src_name)
133                     alreadySaved = browser.messaging.rename_chat(src_name
134             )
135             else:
136                 print("Chat has already been saved")
137
138 def load(debugger, command, result, internal_dict):
139     src_name = prompts.sourceFile_name(debugger)
140
141     if src_name is not None:
142         dirName = path.join(path.dirname(path.abspath(path.realpath(
143             __file__))), "saves")
```

```

137     fileName = "dbChat_" + src_name + ".yaml"
138     filePath = path.join(dirName, fileName)
139
140     if (modality == "api_key_mode"):
141         try:
142             with open(filePath, 'r') as file:
143                 api.messaging.history = yaml.safe_load(file)
144
145         except FileNotFoundError:
146             print(f"There's no debugging state of {src_name}")
147         except yaml.YAMLError:
148             print(f"Failure in loading the debugging state of {src_name}")
149
150     elif (modality == "browser_mode"):
151         global startedChat
152         startedChat = browser.messaging.select_chat(src_name, "load")
153
154 def expChat(debugger, command, result, internal_dict):
155     if (modality == "api_key_mode"):
156         filePath = command.strip() + ".yaml"
157
158         dirName = path.dirname(filePath)
159         if not path.exists(dirName):
160             os.makedirs(dirName)
161
162         history = api.messaging.retrieve_chat()
163         with open(filePath, "w") as file:
164             yaml.dump(history, file)
165     elif (modality == "browser_mode"):
166         print("This command is supported only in 'api_key_mode'")
167
168 def impChat(debugger, command, result, internal_dict):
169     if (modality == "api_key_mode"):
170         filePath = command.strip()
171
172     try:
173         with open(filePath, 'r') as file:
174             api.messaging.history = yaml.safe_load(file)
175
176     except FileNotFoundError:
177         print(f"There's no debugging state here {filePath}")
178     except yaml.YAMLError:
179         print(f"Failure in loading the following debugging state {filePath}")
180
181     elif (modality == "browser_mode"):
182         print("This command is supported only in 'api_key_mode'")
183
184 def clear(debugger, command, result, internal_dict):
185     if (modality == "api_key_mode"):
186         api.messaging.history.clear()
187     elif (modality == "browser_mode"):
188         browser.messaging.new_chat()
189
190 def info(debugger, command, result, internal_dict):

```

```

191     print(f"ChatGPT LLDB Plugin commands: \n      {'mode'.ljust(10)} --\n      See the actual connection mode to ChatGPT, 'api_key_mode' is the\n      default one.\n      {'switch'.ljust(10)} -- Switch among '\n      api_key_mode'(default) and 'browser_mode' connection to ChatGPT.\n      {'code'.ljust(10)} -- See the source code of the debugging\n      target.\n      {'send'.ljust(10)} -- Sends informations about the\n      source code, the current line and the visible variables to ChatGPT\n      .\n      {'ask'.ljust(10)} -- Directly interact with ChatGPT\n      expressing your issues, ex.: 'ask YOUR_REQUEST'.\n      {'wait'.ljust(10)} -- Changes, for the current session, the time (seconds)\n      you wait for ChatGPT responses (supported only in 'browser_mode')\n      , ex.: 'wait TIME_IN_SECONDS'.\n      {'autodebug'.ljust(10)} --\n      Automatically track down code bugs with this compact command.\n      {'save'.ljust(10)} -- Save your current chat about the program\n      you're debugging (in 'browser_mode' you have just one saving per\n      session).\n      {'load'.ljust(10)} -- Load last saved chat about\n      the program you're debugging.\n      {'import'.ljust(10)} -- Import\n      a saved chat about the program you're debugging (supported only in\n      'api_key_mode'), ex.: 'import /home/user/PATH/T0/YOUR_FILE'.\n      {'export'.ljust(10)} -- Export your current chat about the\n      program you're debugging (supported only in 'api_key_mode'), ex.:\n      'export /home/user/PATH/T0/YOUR_FILE'.\n      {'clear'.ljust(10)} --\n      Clear your interactions with ChatGPT.\n      {'info'.ljust(10)} --\n      List the commands introduced by 'chatgpt-lldb-plugin' with a brief\n      description of each one.")

192
193 def __lldb_init_module(debugger, internal_dict):
194     debugger.HandleCommand('command script add -f commands.mode mode')
195
196     debugger.HandleCommand('command script add -f commands.switch switch')
197
198     debugger.HandleCommand('command script add -f commands.code code')
199
200     debugger.HandleCommand('command script add -f commands.send send')
201
202     debugger.HandleCommand('command script add -f commands.ask ask')
203     debugger.HandleCommand('command script add -f commands.wait wait')
204
205     debugger.HandleCommand('command script add -f commands.autodebug autodebug')
206     debugger.HandleCommand('command script add -f commands.save save')
207
208     debugger.HandleCommand('command script add -f commands.load load')
209
210     debugger.HandleCommand('command script add -f commands.expChat export')
211     debugger.HandleCommand('command script add -f commands.impChat import')
212
213     debugger.HandleCommand('command script add -f commands.clear clear')
214
215     debugger.HandleCommand('command script add -f commands.info info')

```

Listing 3.9. chatgpt-lldb-plugin/main/commands.py

### 3.3.6 main.py

Importiamo dalla libreria standard il modulo necessario al funzionamento dello script: “os.path”. In questo script definiamo una sola funzione: “main”. Questa verrà chiamata alla fine del processo di installazione del plugin, automatizzato grazie a “setuptools” e tutte le volte che da terminale si entrerà il comando “chatgpt-plugin-lldb”. Ciò che fa la funzione “main” di questo file python è provvedere all’importazione dei comandi del plugin e quindi all’integrazione del plugin in LLDB. Anzitutto recuperiamo il percorso completo al file “.lldbinit” che generalmente si trova nella home utente. Recuperiamo poi il percorso completo al file “commands.py” che contiene tutti i comandi implementati e dovrà quindi essere importato. Aggiungiamo ora il percorso completo di “commands.py” alla stringa che costituirà il comando LLDB per l’importazione in LLDB e salviamo la stringa completa in “import\_line”. Se il file esiste già, controlliamo se in esso è presente anche il comando LLDB che provvede all’importazione dei comandi all’interno di LLDB, se così è allora non dobbiamo fare nulla poiché l’importazione dei comandi è già predisposta, ma in caso contrario dobbiamo scrivere tale comando, specificando come opzione di apertura del file “.lldbinit” l’opzione “a” che sta per “append”, così da aggiungere il comando salvato nella “import\_line” e non sovrascrivere il file già presente. Se il file “.lldbinit” non esiste, allora dobbiamo crearlo e inserirvi il contenuto di “import\_line”, per fare questo aprimo il file con l’opzione “w”, che sta per “write”, così da creare il file. A questo punto, quando LLDB alla sua apertura andrà a leggere i comandi da eseguire all’avvio, i quali sono contenuti in “.lldbinit”, i nostri comandi verranno importati e l’utente non avrà bisogno di fare nulla in tutto ciò.

```

1 from os import path
2
3 def main():
4     lldbinit_path = path.expanduser("~/lldbinit")
5     plugin_commands_path = path.join(path.dirname(path.abspath(path.
6         realpath(__file__))), "commands.py")
7     print(plugin_commands_path)
8
9     import_line = f"command script import {plugin_commands_path}"
10
11    if path.exists(lldbinit_path):
12        with open(lldbinit_path, "r") as f:
13            contenuto = f.read()
14            already_present = import_line in contenuto
15            if not already_present:
16                with open(lldbinit_path, "a") as f:
17                    f.write(import_line + "\n")
18
19    elif not path.exists(lldbinit_path):
20        with open(lldbinit_path, "w") as file:
21            file.write(import_line + "\n")

```

Listing 3.10. chatgpt-lldb-plugin/main/main.py

### 3.3.7 prompts.py

Importiamo i moduli necessari al funzionamento dello script. Dalla libreria standard abbiamo “platform” e “io”, poi abbiamo i moduli interni al plugin che altro non sono

che altri file come quello che stiamo commentando. I moduli interni di cui abbiamo bisogno sono “stepping” e “commands”, il primo poiché è da “stepping.py” che ricaviamo le informazioni di cui abbiamo bisogno e che andiamo a formattare opportunamente per essere poi utilizzate in “commands.py”. Da quest’ultimo file ricaviamo invece la variabile globale “modality” per differenziare le operazioni da effettuare nella modalità API da quelle effettuate nella modalità browser. La funzione “code()” viene definita per essere poi utilizzata nell’implementazione del comando omonimo in “commands.py”. Recuperiamo dall’argomento “debugger”, il quale ci consente di interagire con debugger, il target ovvero l’obiettivo del debugging e lo passiamo alla funzione “get\_source\_code()” di “stepping.py”. Questa ritornerà il codice del file sorgente, associato all’eseguibile caricato nel debugger, numerato riga per riga. La funzione “code” lo ritorna a sua volta. In caso di fallimento dell’operazione verrà lanciata l’eccezione definita in “stepping.py” che stampa un messaggio informativo a schermo. La prossima funzione si occupa dell’implementazione della funzione “codeInfo()” che ritorna informazioni di sistema e codice numerato riga per riga con opportuna formattazione alla funzione “send()” che implementa l’omonimo comando in “commands.py”. Quindi, anzitutto recuperiamo il target dal debugger e grazie a questo il codice numerato riga per riga da “stepping.py”. Se il codice viene recuperato correttamente senza il lancio di alcuna eccezione, allora procediamo. Se ci troviamo nella modalità API definiamo 3 prompt che poi verranno concatenati. Il primo è “intro” e al suo interno inseriamo, in un testo opportuno, le informazioni sul sistema utilizzato recuperate grazie al modulo “platform”, la versione di LLDB e il nome del file sorgente, queste ultime recuperate grazie a “stepping.py”. Nel prompt “code” inseriamo il codice recuperato prima in un testo opportuno. Nel prompt “conclusion” chiediamo a ChatGPT di risparmiare “token” e di non risponderci ora. Sebbene non possiamo bloccare il “rispondere” del ChatBot possiamo posticipare la risposta attinente all’argomento fornito. A scopo informativo: i “token” sono la singola unità di testo presa come riferimento da OpenAI per il conteggio dell’input e output scambiato con ChatGPT. Nella modalità browser la sola differenza è che vengono rimossi i “\n” poiché questo comprometterebbe l’invio dei prompt. Utilizzando l’interfaccia web, infatti, ChatGPT interpreta “\n” come accapo e spezzetta l’input. Concateniamo e ritorniamo poi il prompt completo. La funzione “stepInfo()” si occupa di recuperare le informazioni sulla linea corrente e le variabili visibili e poi riorganizzarle in un prompt opportunamente formattato. Se la variabile globale “variables\_to\_be\_sent” definita in “stepping.py” è “True” e quindi è utile inviare le informazioni che ci accingiamo a recuperare e formattare, allora possiamo proseguire. Recuperiamo il target dal debugger e utilizziamolo per recuperare da “stepping.py” la linea corrente e le variabili visibili. Se il recupero avviene correttamente senza che venga lanciata alcuna eccezione allora proseguiamo altrimenti non restituiamo nulla alla funzione chiamante. Abbiamo sia una modalità API che browser, ma come abbiamo visto precedentemente la sola differenza sta nell’assenza, nella “browser\_mode”, dei “\n”. Pertanto si pone un’introduzione, si specifica la linea corrente e si elencano le variabili visibili riportandone: nome, valore, tipo e indirizzo. Poi si aggiunge una conclusione per chiedere di aspettare le mie richieste prima di rispondermi. Infine concateniamo tutti i prompt in “prompt” e ritorniamo la stringa, pronta per essere usata in “commands.py” nel comando “send”. La prossima funzione si occupa di formattare il prompt che sarà poi utilizzato in “commands.py”

come argomento per il comando “ask”. In questo prompt sarà quindi contenuto sia l’output del programma, l’informazione sul corretto avanzamento dell’esecuzione e nel caso se vi siano state interruzioni la ragione delle stesse, ma anche, se possibile specificarlo, dove queste sono avvenute, su quale linea. Ma sarà qui contenuta anche la richiesta che faremo a ChatGPT per identificare opportunamente i bug presenti nel codice. Pertanto eseguiamo la funzione “automated\_debugging” di “stepping.py” che ritorna l’output del programma e svolge altresì delle operazioni fondamentali che andremo poi a vedere nel dettaglio. Salviamo l’output in un prompt opportunamente formattato. Se ci sono stati errori che hanno interrotto l’esecuzione ma non è stato compromesso l’indirizzo della prossima istruzione da eseguire contenuto nel program counter (PC), allora specifichiamo nel prompt la linea di interruzione e la ragione dello stop. Se ci sono stati errori che hanno interrotto l’esecuzione e l’indirizzo di memoria della prossima istruzione di memoria contenuto nel program counter (PC) è stato compromesso, allora non possiamo indicare la linea corrente ma solo la ragione dello stop. La linea corrente in questo caso è infatti compromessa e sicuramente non corretta. Se, invece, non ci sono state interruzioni specifichiamo in un prompt che non ve ne sono state. Specifichiamo infine le nostre richieste in un prompt a parte nel quale operiamo le medesime differenziazioni operate prima circa la modalità di utilizzo. Nella modalità browser rimuoviamo quindi il “\n”. Concateniamo i prompt e ritorniamoli pronti per essere utilizzati nel comando “autodebug” dello script “commands.py”. Rimandiamo alla sezione “Interazione con ChatGPT: Prompt” la spiegazione di come e perché sono stati così organizzati i prompt. Nell’ultima funzione, “sourceFile\_name”, dopo aver recuperato il target lo utilizziamo come argomento della funzione “get\_source\_file\_name()” di “stepping.py” per recuperare il nome del file sorgente che sarà utilizzato poi in “commands.py” in diverse occasioni. Si nota che in realtà sia per la funzione “sourceFile\_name” che per “code()” si sarebbe potuto evitare di metterle in “prompts.py”, del resto avremmo potuto importare direttamente “stepping” in “commands.py”, tuttavia si è preferito organizzare queste funzioni in questo modo onde favorire un’organizzazione più ordinata e modulare dell’applicativo e oserei dire anche a più livelli. Così come “commands.py” si occupa dei comandi per il debugger, questa attinge da “prompts.py” che formatta opportunamente i dati da utilizzare per l’implementazione dei comandi e “prompts.py” attinge a sua volta da “stepping.py” che recupera i dati direttamente dal debugger grazie alle API Python fornite da LLDB.

```
1 import platform
2 import stepping
3 import io
4 import commands
5
6 def code(debugger):
7     target = debugger.GetSelectedTarget()
8     code = stepping.get_source_code(target)
9
10    return code
11
12 def codeInfo(debugger):
13     target = debugger.GetSelectedTarget()
14     code = stepping.get_source_code(target)
15
```

```

16     if code is not None:
17         if commands.modality == "api_key_mode":
18             intro = f"Utilizzo il sistema {platform.platform()} su
cui ho installato {stepping.get_lldb_version()}. Con LLDB sto
facendo il debugging dell'eseguibile di '{stepping.
get_sourceFile_path(target)}'. "
19             code = f"Il codice sorgente : \n{code}\n"
20             conclusion = "Non devi rispondermi. Riceverai ulteriori
info nel prossimo messaggio.\n"
21         elif commands.modality == "browser_mode":
22             intro = f"Utilizzo il sistema {platform.platform()} su
cui ho installato {stepping.get_lldb_version()}. Con LLDB sto
facendo il debugging dell'eseguibile di '{stepping.
get_sourceFile_path(target)}'. "
23             code = code.replace('\n', '')
24             code = f"Il codice sorgente : {code} "
25             conclusion = "Non devi rispondermi. Riceverai ulteriori
info nel prossimo messaggio. "
26             prompt = intro + code + conclusion
27
28     return prompt
29 else:
30     return None
31
32 def stepInfo(debugger):
33     if stepping.variables_to_be_sent is True:
34         target = debugger.GetSelectedTarget()
35         cline = stepping.get_current_line(target)
36         vars = stepping.get_local_variables(target)
37
38         if cline and vars is not None:
39             if commands.modality == "api_key_mode":
40                 intro = f"In questo momento: \n"
41                 current_line = f"    1. La linea corrente    la numero
: {cline} \n"
42                 output_string = io.StringIO()
43                 output_string.write("    2. Le variabili nel blocco
sono: \n")
44                 for i, var in enumerate(vars):
45                     output_string.write(f"          - La variabile {i
+1} ha nome '{var[0]}', valore '{var[1]}', tipo '{var[2]}' e
indirizzo '{var[3]}' .\n")
46                 variables = output_string.getvalue()
47                 output_string.close()
48                 conclusion = "Non devi rispondermi. Ti fornir le
mie richieste al prossimo messaggio.\n"
49             elif commands.modality == "browser_mode":
50                 intro = f"In questo momento: "
51                 current_line = f"    1. La linea corrente    la numero
: {cline} . "
52                 output_string = io.StringIO()
53                 output_string.write("    2. Le variabili nel blocco
sono: ")
54                 for i, var in enumerate(vars):
55                     output_string.write(f"          - La variabile {i
+1} ha nome '{var[0]}', valore '{var[1]}', tipo '{var[2]}' e
indirizzo '{var[3]}' . ")

```

```

56         variables = output_string.getvalue()
57         output_string.close()
58         conclusion = "Non devi rispondermi. Ti fornir le
59         mie richieste al prossimo messaggio. "
60         prompt = intro + current_line + variables + conclusion
61
62     return prompt
63 else:
64     return None
65
66 else:
67     prompt = ""
68 return prompt
69
70 def automatedDebuggingInfo(debugger):
71     program_output = stepping.automated_debugging(debugger)
72     program_output = f"L'output del programma il seguente: '{
73     program_output}'. "
74
75     if stepping.blocking_errors is True:
76         if stepping.memory_read_failed is False:
77             debugging_result = f"Il programma durante il debugging si
78             interrotto alla linea {stepping.line_of_crash}. La ragione
79             dello stop : {stepping.sig_detected}. "
80         elif stepping.memory_read_failed is True:
81             debugging_result = f"Il programma durante il debugging si
82             interrotto. La ragione dello stop : {stepping.sig_detected}.
83             "
84     elif stepping.blocking_errors is False:
85         debugging_result = "Il programma durante il debugging ha
86         concluso la sua esecuzione senza interruzioni. "
87
88     if commands.modality == "api_key_mode":
89         ask_for_errors = "Quanti e quali sono gli errori del codice?
90         Differenziali per tipologia.\n"
91     elif commands.modality == "browser_mode":
92         ask_for_errors = "Quanti e quali sono gli errori del codice?
93         Differenziali per tipologia. "
94     prompt = program_output + debugging_result + ask_for_errors
95
96 return prompt
97
98 def sourceFile_name(debugger):
99     target = debugger.GetSelectedTarget()
100    name = stepping.get_sourceFile_name(target)
101
102    if name is not None:
103        return name
104    else:
105        return None

```

**Listing 3.11.** chatgpt-lldb-plugin/main/prompts.py

### 3.3.8 stepping.py

Importiamo i moduli necessari al funzionamento dello script. Dalla libreria standard abbiamo “subprocess” e “os”, dal pacchetto “cpython-lldb” abbiamo “lldb”. Non

abbiamo bisogno di importare i moduli interni al plugin poiché “stepping.py” interagisce direttamente con il debugger. Questo avviene attraverso l’utilizzo delle API per Python fornite dalla casa madre di LLDB. La prima funzione è “get\_lldb\_version()” e viene utilizzata per ricavarci la versione di LLDB che stiamo utilizzando per il debugging. Utilizziamo “subprocess.run” per eseguire in sincrono il comando da terminale “lldb -v”, aspettiamo che finisca per riacquisire il controllo e salviamo l’output sprovvisto degli accapo in “lldb\_version” che ritorniamo pronto per essere utilizzato dalla funzione “codeInfo()” di “prompts.py”. La prossima funzione è “get\_frame()”. Questa ci ritorna il frame corrente dopo averlo recuperato dall’argomento target che ci viene fornito. Quindi recuperiamo dal target il processo, da questo il thread e da quest’ultimo il frame corrente. La prossima funzione è “get\_current\_line()”. Usiamo un “try, except” per lanciare un’eccezione in caso di fallimento dell’operazione ritornando “None”. Dal frame corrente possiamo recuperare, partendo dall’oggetto “SBFrame” e poi da “SBLLineEntry”, la linea corrente e ritornarla. È in formato “unsigned integer 32-bit”. La prossima funzione riceve come argomento il target e si occupa di recuperare il nome del file sorgente associato all’obiettivo del debugging creato nel debugger. Si tratta della funzione: “get\_sourceFile\_name()”. Questa prova a recuperare il frame corrente e da questo il nome del file sorgente utilizzando gli oggetti “SBFrame”, “SBLLineEntry” e “SBFileSpec” e poi lo ritorna. Se qualcosa va storto viene lanciata un’eccezione che stampa la ragione dell’eccezione. Questa occorre quando non si è nello stato “Launched Process State” ovvero quando il processo non è stato lanciato oppure è già terminato la cosa si risolve impostando un breakpoint e lanciando l’esecuzione del processo. La prossima funzione è “get\_sourceFile\_path()”. Questa riceve il target e dopo aver recuperato il frame corrente, utilizza l’oggetto “SBFrame”, poi “SBLLineEntry” e infine “SBFileSpec” per recuperare la directory del file sorgente. Mettendo assieme la directory recuperata con il nome del file recuperato attraverso la funzione “get\_sourceFile\_path()” si ottiene il percorso completo al file sorgente che possiamo ora ritornare. In caso qualcosa dovesse andare storto viene lanciata l’opportuna eccezione. La prossima funzione è “source\_code\_in\_array()” e si occupa di leggere il file sorgente copiandolo riga per riga nell’array “lines” che poi ritorna. Questo partendo dal percorso al file sorgente fornитigli come argomento. In caso di percorso fornito non corretto o di file sorgente mancante, viene lanciata un’eccezione che stampa a schermo l’errore. La prossima funzione è “add\_numbers\_to\_lines()”. Scorre l’array e aggiunge ad ogni riga un numero che va da 1 in poi. Sono aggiunti degli spazi per facilitare la leggibilità del codice più avanti. Viene ritornato l’array, dove ogni suo elemento è una riga numerata del codice. La prossima funzione è “get\_source\_code()” e, ricevuto il target, recupera il percorso al file sorgente con la funzione “get\_sourceFile\_path()”. Se il path è stato recuperato correttamente viene formito come argomento a “source\_code\_in\_array()” che ritorna un array contenente il codice riga per riga. Questo array viene passato a “add\_numbers\_to\_lines()” che si occupa di aggiungere i numeri di riga. Viene poi ritornata una stringa che concatena tutte le linee che formano l’array con la funzione “join()” preoccupandosi di aggiungere anche gli accapo (\n). La prossima funzione recupera le variabili visibili, si tratta di “get\_variables()”. Dichiariamo l’array che conterrà le variabili e assegnamole ad esso dopo averle recuperare con la funzione “get\_locals()” dell’oggetto “SBFrame”. In questo modo recuperiamo le variabili del blocco in cui ci troviamo ma in alternativa, se volessimo recuperarle tutte, potremmo

utilizzare “get\_all\_variables()”. Scorriamo poi le variabili in “vars” e per ognuna di queste recuperiamo: “nome”, “valore”, “tipo” e “indirizzo”. Ora avremo una matrice che ha per righe le variabili e per colonne il nome, valore, tipo e indirizzo di ciascuna variabile. Ritorniamo la matrice, pronta per essere spaccata in “prompts.py”. Ultima e più complessa funzione è “automated\_debugging()” la quale regola l’automatizzazione del debugging e fornisce dei dati significativi a “prompts.py”, e quindi informazioni per “commands.py” ed esegue inoltre delle operazioni fondamentali per l’utilizzo del comando “autodebug”. Si può dire che è il nucleo del comando stesso. Dichiariamo globalmente “sig\_detected” come una stringa vuota, qui inseriremo il nome del segnale che riporteremo poi nel prompt come ragione dello stop in caso di interruzioni. Dichiariamo globalmente anche “variables\_to\_be\_sent” e “blocking\_errors” e assegniamo loro rispettivamente i booleani “True” e “False”. Questo ad indicare che al momento le variabili possono essere inviate e che non sono stati rilevati errori bloccanti. Va specificato che le variabili al momento devono poter essere inviate, altrimenti attraverso il debugging manuale, con il comando “send” non si invierebbe mai il prompt fornito da “stepInfo()”, funzione definita in “prompts.py”. Quindi, qualora nel debugging automatico dovesse rivelarsi inutile inviare le variabili visibili poiché magari la linea corrente è fissa sul “main”, a quel punto non le invieremmo, ma solo a quel punto. Entriamo nella funzione “automated\_debugging”. Dichiariamo internamente alla funzione le variabili globali “line\_of\_crash” e “memory\_read\_failed” oltre alle globali dichiarate prima. Utilizzeremo tutte queste variabili sia qui che per differenziare i prompt in “prompts.py”. Recuperiamo il target, ovvero l’obiettivo del debugging, dal debugger. Eliminiamo i breakpoint esistenti se ve ne sono onde non avere problemi nell’esecuzione che seguirà. Impostiamo un breapoint all’inizio della funzione “main” del programma che stiamo debuggando. Vediamo se c’è già un processo associato al target, ovvero se è già in esecuzione, in caso lo terminiamo. Ora ci sono le condizioni per lanciare il processo, ma prima di fare questo è bene recuperare l’output e catturarlo. È qualcosa che non potevamo fare prima poiché avevamo bisogno del target e che precauzionalmente non si è voluto fare dopo per evitare che la doppia esecuzione potesse avere ripercussioni sull’automatizzazione del debugging. Recuperiamo quindi il percorso completo al file eseguibile, non il sorgente come fatto prima in un’altra funzione. Eseguiamo l’eseguibile di cui abbiamo recuperato il percorso in modo asincrono e quindi contemporaneamente all’esecuzione dell’applicativo. Per fare ciò utilizziamo la funzione “subprocess.Popen” e quindi specifichiamo gli argomenti “stdout” e “stderr” per catturare l’output standard e l’output degli errori standard. Il processo viene avviato e se ne aspetta la terminazione, questo lo si fa con la funzione “communicate()”. Ritroveremo gli output desiderati nelle variabili “output” e “error” che concateniamo. Sostituiamo poi gli accapo (\n) con degli spazi così da evitarci dopo la differenziazione per il prompt con l’output a seconda della modalità di utilizzo. L’output è ora catturato e si trova nella variabile “output\_string”, pronto per essere ritornato alla funzione chiamante “automatedDebuggingInfo” di “prompt.py”. Procediamo e lanciamo quindi l’esecuzione attraverso la creazione di un processo, stampiamo anche una riga per mostrare che ora inizia l’output dello stesso che verrà a generarsi. Fintanto che il processo non termina, quindi fintanto che siamo nel “Launched Process State”, ripeteremo delle operazioni ciclicamente. Queste sono: recuperare il file sorgente, la linea corrente che chiamiamo “line\_of\_crash” e in caso di crash sarà

la linea prima dello stesso, il thread ovvero il filone di esecuzione e grazie a questo passiamo all’istruzione successiva. Attualmente non c’è corruzione dell’indirizzo di memoria dell’istruzione successiva da eseguire, custodita nel program counter (PC), quindi impostiamo “memory\_read\_failed” al booleano “False”. Le cose potrebbero cambiare da qui a poco. Recuperiamo la ragione dello stop, che solitamente è o lo step over, oppure la presenza di un breakpoint. Se la ragione dello stop è un segnale, allora recuperiamo la descrizione della ragione dello stop e salviamola in “signal\_info”. Se abbiamo a che fare con un “SIGABRT” il segnale rilevato sarà questo così come per “SIGFPE”. Nel caso del “SIGSEGV” le cose variano un poco. Dobbiamo controllare se il “segmentation fault” non abbia causato una corruzione dell’indirizzo di memoria alla prossima istruzione, contenuta nel PC. Questa verifica si attua recuperando il frame corrente, quindi il contesto di esecuzione attuale che ci consente di esaminare lo stato del programma e delle variabili in questo punto specifico dell’esecuzione, dal frame recuperiamo l’indirizzo nel PC e proviamo a leggerlo. Se la lettura fallisce allora l’indirizzo non è valido, allora è stato corrotto (“memory\_read\_failed = True”) e le variabili visibili e preferibile non mandarle poiché non sappiamo in quale linea è avvenuto il problema che ha generato il “SIGSEGV”, potrebbe benissimo essere capitato qualche riga fa sebbene la rilevazione sia giunta dopo. Per qualsiasi altro segnale che viene rilevato porremo la descrizione direttamente in “sig\_detected”. Ora che abbiamo rilevato la presenza di un segnale che ha stoppato l’esecuzione del programma dobbiamo operare degli accorgimenti. Se “sig\_detected” non è una stringa vuota quindi il segnale è stato rilevato, eliminiamo i breakpoint precedenti, quindi quello all’inizio della funzione “main”. Se l’indirizzo nel PC non è stato corrotto possiamo considerare affidabile la linea antecedente all’interruzione irregolare dell’esecuzione e allora impostare un breakpoint su quella linea specificando il file sorgente dell’obiettivo del debugging. In caso contrario impostiamo un breakpoint all’inizio della funzione “main”. È necessario impostare un breakpoint per poi lanciare l’esecuzione e mantenersi nel “Launched Process State”, senza di esso, infatti, si finirebbe per terminare l’esecuzione uscendo da questo stato, fuori dal quale non si può usare il comando “send”. Allora terminiamo il processo attuale la cui esecuzione al momento è in stop. Possiamo qui stampare una riga che mostra la fine dell’esposizione dell’output. Con il gruppo di istruzioni successive lanciamo nuovamente il processo prestando attenzione a non stampare a schermo l’output, onde non creare confusione all’utente. Ora possiamo finalmente riportare in “blocking\_errors” la presenza di errori che hanno interrotto l’esecuzione impostando “blocking\_errors” a “True”, usciamo allora dal ciclo e, non potendo entrare nel prossimo “if” dato che l’esecuzione è terminata con uno stato di uscita diverso da zero essendoci stati degli errori che hanno causato l’interruzione dell’esecuzione, ritorniamo l’output caricato prima in “output\_string” alla funzione chiamante in “prompts.py”, questa ritornerà poi il prompt in “commands.py” e trovandoci nel “Launched Process State” potremo allora usare il comando “send” all’interno di “autodebug” per comunicare con ChatGPT. Questo avviene se dovessimo rilevare un segnale, ma cosa succede nel frattempo se non dovessimo rilevarlo? E se questo non venisse rilevato mai? Fintanto che stiamo eseguendo il programma oppure questa è stata stoppata a seguito dell’utilizzo di “step over”, il ciclo si ripete. Il ciclo terminerà quando non si è più né in esecuzione né in stop e quindi quando si è usciti dall’esecuzione. Se non si sono rilevati segnali lanciati a seguito di un’uscita con

errore dall'esecuzione del programma possiamo entrare nell'ultimo "if". L'esecuzione è terminata senza errori quindi lo stato in cui ci troviamo è il "Target Created State". Il breakpoint all'inizio della funzione "main" è ancora presente, allora non dobbiamo far altro che lanciare l'esecuzione per entrare nel "Launched Process State" così da garantirci l'utilizzo di "send". Sempre per evitare confusione all'utente eviteremo di stampare l'output che verrà fuori da questa esecuzione. Infine anche qui ritorniamo "output\_string" alla funzione chiamante in "prompts.py". Per qualsiasi curiosità circa gli oggetti utilizzati del modulo "lldb" si rimanda alla documentazione ufficiale all'indirizzo: [lldb.llvm.org/python\\_reference/](http://lldb.llvm.org/python_reference/).

```

1 import subprocess
2 import lldb
3 import os
4
5 def get_lldb_version():
6     lldb_version = subprocess.run("lldb -v", shell=True,
7         capture_output=True, text=True, check=True)
8     lldb_version = lldb_version.stdout.replace("\n", "")
9     return lldb_version
10
11 def get_frame(target):
12     process = target.GetProcess()
13     if not process.IsValid():
14         return
15
16     thread = process.GetThreadAtIndex(0)
17     if not thread.IsValid():
18         return
19
20     frame = thread.GetSelectedFrame()
21     if not frame.IsValid():
22         return
23
24     return frame
25
26 def get_current_line(target):
27     try:
28         line = get_frame(target).GetLineEntry().GetLine()
29     except:
30         return None
31
32 def get_sourceFile_name(target):
33     try:
34         frame = get_frame(target)
35         file_name = frame.GetLineEntry().GetFileSpec().GetFilename()
36     except:
37         print("You have to set a breakpoint and run the target first")
38
39 def get_sourceFile_path(target):
40     try:
41         frame = get_frame(target)
42         dirName = frame.GetLineEntry().GetFileSpec().GetDirectory()
43         fileName = get_sourceFile_name(target)
44

```

```
45     sourceFilePath = dirName+"/"+fileName
46     return sourceFilePath
47 except:
48     sourceFilePath = None
49     print("You have to set a breakpoint and run the target first"
50 )
51 def source_code_in_array(filePath):
52     try:
53         lines = []
54         with open(filePath, "r") as file:
55             for line in file:
56                 lines.append(line.rstrip('\n'))
57
58         return lines
59
60     except FileNotFoundError:
61         print(f"Error: File '{filePath}' not found")
62         return []
63
64 def add_numbers_to_lines(lines):
65     for i, line in enumerate(lines):
66         lines[i] = f"    {(i+1):>3} {line}"
67
68     return lines
69
70 def get_source_code(target):
71     path=get_sourceFile_path(target)
72     if path is not None:
73         lines = source_code_in_array(path)
74         lines = add_numbers_to_lines(lines)
75
76         return '\n'.join(lines)
77     else:
78         return None
79
80 def get_local_variables(target):
81     try:
82         vars = []
83         vars = get_frame(target).get_locals() #Alternatively, use:
84         get_all_variables().
85
86         variables = []
87         for var in vars:
88             variable = [var.GetName(), var.GetValue(), var.
89             GetTypeName(), var.GetLoadAddress()]
90             variables.append(variable)
91
92
93         return variables
94     except:
95         return None
96
97 sig_detected = ""
98 variables_to_be_sent = True
99 blocking_errors = False
100 def automated_debugging(debugger):
101     global line_of_crash, sig_detected, variables_to_be_sent,
102         blocking_errors, memory_read_failed
```

```

98     target = debugger.GetSelectedTarget()
99     target.DeleteAllBreakpoints()
100    target.BreakpointCreateByName("main")
101    process = target.GetProcess()
102    if process.IsValid():
103        process.Kill()
104
105    executable_dir = target.GetExecutable().GetDirectory()
106    executable_filename = target.GetExecutable().GetFilename()
107    executable_path = os.path.join(executable_dir,
108                                   executable_filename)
109    proc = subprocess.Popen(executable_path, shell=True, stdout=
110                           subprocess.PIPE, stderr=subprocess.PIPE, text=True) #In case of
111                           blocking error it is not guaranteed the full output.
112    output, error = proc.communicate()
113    output_string = output + error
114    output_string = output_string.replace('\n', ' ')
115
116    process = target.LaunchSimple(None, None, None)
117    print("----- Execution Started -----")
118
119    while process.GetState() != lldb.eStateExited:
120        debugging_file = get_sourceFile_path(target)
121        line_of_crash = get_current_line(target)
122        thread = process.GetThreadAtIndex(0)
123        thread.StepOver()
124
125        memory_read_failed = False
126        stop_reason = thread.GetStopReason()
127        if stop_reason == lldb.eStopReasonSignal:
128            signal_info = thread.GetStopDescription(256)
129            if "SIGABRT" in signal_info:
130                sig_detected = "SIGABRT"
131            elif "SIGSEGV" in signal_info:
132                sig_detected = "SIGSEGV indirizzo non valido"
133
134            frame = target.GetProcess().GetSelectedThread().
135            GetSelectedFrame()
136            address = frame.GetPCAddress()
137            memory_data = target.ReadMemory(address, 4, lldb.
138            SBError())
139            if memory_data is None:
140                memory_read_failed = True
141                variables_to_be_sent = False
142            elif "SIGFPE" in signal_info:
143                sig_detected = "SIGFPE"
144            else:
145                sig_detected = signal_info
146
147            if sig_detected != "":
148                target.DeleteAllBreakpoints()
149                if memory_read_failed is False:
150                    target.BreakpointCreateByLocation(debugging_file,
151                                         line_of_crash)
152                else:
153                    target.BreakpointCreateByName("main")

```

```

148         if process.IsValid() and process.GetState() == lldb.eStateStopped:
149             process.Kill()
150             print("----- Execution Terminated -----")
151
152             stdout_fd = os.dup(1)
153             os.close(1)
154             os.open(os.devnull, os.O_WRONLY)
155             process = target.LaunchSimple(None, None, None)
156             os.dup2(stdout_fd, 1)
157             os.close(stdout_fd)
158
159             blocking_errors = True
160             break
161
162     if process.GetState() == lldb.eStateExited and process.GetExitStatus() == 0:
163         print("----- Execution Terminated -----")
164         stdout_fd = os.dup(1)
165         os.close(1)
166         os.open(os.devnull, os.O_WRONLY)
167         process = target.LaunchSimple(None, None, None)
168         os.dup2(stdout_fd, 1)
169         os.close(stdout_fd)
170
171     variables_to_be_sent = False
172     blocking_errors = False
173
174     return output_string

```

Listing 3.12. chatgpt-lldb-plugin/main/stepping.py

### 3.3.9 version.py

In questo script avviene una semplice assegnazione. Viene assegnata alla variabile speciale “`__version__`” la stringa che contiene la versione del plugin, così come si usa convenzionalmente nel software Python. Il contenuto di tale variabile verrà poi letto da “`setup.py`” al momento dell’installazione del plugin onde consentire a “`setuptools`” di proseguire con l’installazione. Indicare la versione è un’operazione utile se si ha intenzione di rilasciare versioni successive. In questo modo, quando si aggiornerà il plugin, “`setuptools`” confronterà il numero della versione del pacchetto installato con quello nuovo e potrà quindi rimuovere i file precedenti in favore dei nuovi.

```
1 __version__ = "0.1.0"
```

Listing 3.13. chatgpt-lldb-plugin/main/version.py

## 3.4 Sottosottodirectory 'api\_connection'

### 3.4.1 \_\_init\_\_.py

Contraddistingue la sottosottodirectory "api\_connection" come un pacchetto Python i cui moduli sono importabili. È stato lasciato vuoto non essendo necessaria alcuna inizializzazione aggiuntiva per il funzionamento del pacchetto.

**Listing 3.14.** chatgpt-lldb-plugin/main/api\_connection/\_\_init\_\_.py

### 3.4.2 api.messaging.py

Importiamo i moduli necessari al funzionamento dello script. Dalla libreria standard abbiamo "sys", "os" e "os.path", dal pacchetto "openai" abbiamo "openai". Dobbiamo importare anche il modulo interno al plugin "api\_key\_config", dal quale recuperare la OpenAI API Key per il corretto funzionamento della modalità API. Per importare questo modulo, tuttavia, dobbiamo ottenere il percorso assoluto della directory "chatgpt-lldb-plugin", directory che sta due livelli sopra il file corrente, e aggiungerlo all'inizio della lista dei percorsi in "sys.path" che è la lista dei percorsi di ricerca che Python utilizza per trovare i moduli importati. In questo modo possiamo importare moduli dalla directory "chatgpt-lldb-plugin" che altrimenti non sarebbe possibile importare trovandoci due livelli sotto di essa. Importato "api\_key\_config" possiamo procedere. Definiamo due funzioni che ci consentono di accedere all'OpenAI API Key seppure in modo differente. La prima funzione, "get\_from\_script()", ritorna la variabile globale statica "OPENAI\_API\_KEY", presa dallo script "api\_key\_config" che è designato per essere modificato dall'utente che voglia impostare una l'OpenAI API Key in maniera statica per tutte le sessioni a venire di LLDB. La seconda funzione, "get\_from\_env()" recupera, dall'oggetto contenente le variabili d'ambiente del sistema operativo, la variabile d'ambiente "OPENAI\_API\_KEY" impostata dall'utente entrando da terminale il comando "export OPENAI\_API\_KEY=<YOUR\_OPENAI\_API\_KEY>" e la ritorna. Va precisato che in tal caso la variabile d'ambiente dovrà essere reimpostata ogni volta che si apre un nuovo terminale dentro il quale eseguire LLDB. In caso la variabile d'ambiente "OPENAI\_API\_KEY" non dovesse essere stata impostata allora viene alzatato un "EnvironmentError", di cui viene fornita la descrizione. Adesso definiamo delle operazioni globali che verranno eseguite quando si importa questo script come modulo in "commands.py" che viene a sua volta importato ad ogni apertura di LLDB grazie al comando LLDB specificato in ".lldbinit" al momento dell'installazione, file dal quale LLDB prende i comandi da eseguire al suo avvio. Tali operazioni verranno quindi effettuate ad ogni avvio di LLDB. Salviamo in "api\_key" la chiave API recuperata dal modulo "api\_key\_config". Se l'utente non ha posto la sua chiave all'interno della variabile globale statica "OPENAI\_API\_KEY" del file che definisce il modulo omonimo, allora proviamo a recuperare la variabile d'ambiente. Se non è presente lanciamo l'eccezione prima definita con "raise" e stampiamone la descrizione. Se, invece, la chiave API nel file preposto è stata inserita dall'utente, allora possiamo salvare in "openai.api\_key" tale chiave che verrà poi utilizzata per interagire con ChatGPT. In tal modo la variabile d'ambiente viene presa in

considerazione solo se non è stata già impostata la chiave API, mentre in caso fosse già stata impostata allora la salviamo subito dove opportuno. Passiamo ora alla definizione di “send\_to\_chatgpt()”, la funzione che si occupa dell’interazione con il ChatBot, in questo caso tramite l’utilizzo della chiave API dell’utente. Dichiariamo la variabile “history” che conterrà la lista di dizionari nella quale verrà organizzata la chat con ChatGPT. Ciascun dizionario rappresenta un elemento della conversazione e ha due chiavi: “role” e “content” che specificano rispettivamente il mittente del messaggio e il contenuto del messaggio. L’insieme di tutti i messaggi e quindi di tutti i dizionari è la conversazione. Assegname inoltre un primo messaggio che definisce il comportamento di ChatGPT. Entriamo ora all’interno della funzione. Questa riceve come argomento il contenuto del messaggio da inviare al ChatBot. Dichiariamo quindi, internamente alla funzione, la variabile globale “history” e aggiungiamo un dizionario alla lista dei dizionari già creata. Questo dizionario avrà “user” come “role” e il contenuto ricevuto come argomento come “content”. Usiamo un “try, except” e proviamo ad inviare come messaggio la lista di dizionari rappresentante la nostra conversazione precisando il modello che si intende utilizzare. Per il rapporto qualità-prezzo attuale “gpt-3.5-turbo” risulta un’ottima scelta. Modificando questa stringa si può selezionare un modello differente come “gpt4”, ad esempio, che a fronte di un costo maggiore è specializzato nella risoluzione di sistemi complessi con accuratezza[22]. Tutti i modelli sono presentati all’indirizzo: platform.openai.com/docs/models/overview. È essenziale inviare tutta la lista dei messaggi poiché, per come è stato pensato l’utilizzo delle API da OpenAI, ChatGPT non ha stato, quindi non è in grado di memorizzare le conversazioni. Sarà quindi nostra premura occuparci di salvare opportunamente tutti i messaggi organizzati come dizionari in una lista da salvare in “history” e inviarla ogni volta. Potremmo anche specificare altre variabili come “max\_tokens” che specifica il limite massimo di token utilizzabili da generare nel completamento della chat, ma non vogliamo porre questo limite. Non definiremo nemmeno la variabile “temperature” lasciandola quindi al valore di default “1”. La temperatura regola il grado di randomicità della risposta. Tanto più è piccola tanto più ChatGPT ci fornirà delle risposte di cui è sicura e poco varie, tanto più sarà alta tanto più darà sfogo alla sua creatività, osando nelle risposte, tale valore può essere impostato tra 0 e 2. Recuperiamo poi il messaggio il contenuto del messaggio di risposta, se è arrivato, e salviamolo nel dizionario da appendere alla lista. Ritorniamo poi il contenuto alla funzione chiamante in “commands.py” affinché possa stamparlo a schermo o meno a seconda che questa sia “ask()” o “send()”. In caso dovesse esserci un errore di autenticazione oppure un errore dovuto a problemi d’invocazione dell’API OpenAI o nell’elaborazione della richiesta allora verranno lanciate le apposite eccezioni. Infine definiamo la funzione “retrieve\_chat()” che ritorna la conversazione con ChatGPT custodita in “history” alle funzioni definite in “commands.py” che implementano i comandi per salvare la chat.

```
1 import openai
2 import sys
3 import os
4 from os import path
5
6 program_path = path.abspath(path.join(path.dirname(__file__), '..', '..'))
```

```

7 sys.path.insert(0, program_path)
8 import api_key_config
9
10 def get_from_script():
11     api_key = api_key_config.OPENAI_API_KEY
12     return api_key
13
14 def get_from_env():
15     api_key = os.environ.get('OPENAI_API_KEY')
16     if api_key is None:
17         raise EnvironmentError("The environment variable
18 OPENAI_API_KEY is not set")
19     return api_key
20
21 api_key = get_from_script()
22 if (api_key == "YOUR_OPENAI_API_KEY"):
23     try:
24         api_key = get_from_env()
25     except EnvironmentError as e:
26         print(e)
27 openai.api_key = api_key
28
29 history = [{"role": "system", "content" : "Sei un gentile e utile
30 assistente per il debugging e la correzione del codice."}]
31 def send_to_chatgpt(content):
32     global history
33     history.append({"role": "user", "content": content})
34
35     try:
36         completion = openai.ChatCompletion.create(
37             model="gpt-3.5-turbo",
38             messages=history
39         )
40
41         reply = completion.choices[0].message.content
42         if completion and len(completion.choices) > 0:
43             history.append({"role": "assistant", "content": reply})
44             return reply
45
46     except openai.error.AuthenticationError as e:
47         print("Authentication error:", e)
48     except openai.error.OpenAIErr as e:
49         print("OpenAI error:", e)
50
51 def retrieve_chat():
52     return history

```

**Listing 3.15.** chatgpt-lldb-plugin/main/api\_connection/api\_messaging.py

## 3.5 Sottosottodirectory 'browser\_connection'

### 3.5.1 \_\_init\_\_.py

Contraddistingue la sottosottodirectory "browser\_connection" come un pacchetto Python i cui moduli sono importabili. È stato lasciato vuoto non essendo necessaria

alcuna inizializzazione aggiuntiva per il funzionamento del pacchetto.

**Listing 3.16.** chatgpt-lldb-plugin/main/browser\_connection/\_\_init\_\_.py

### 3.5.2 browser.messaging.py

Importiamo i moduli necessari al funzionamento dello script. Dalla libreria standard abbiamo "time", "socket", "threading", "subprocess", "sys" e "os.path", dal pacchetto "yaml" abbiamo "yaml". Dal pacchetto selenium abbiamo "webdriver", poi da questo i moduli "By", "Keys", "WebDriverWait" ed "expected\_conditions". Dal pacchetto "selenium" importiamo inoltre le eccezioni di cui abbiamo bisogno, sono cinque. Dobbiamo importare ancora il modulo interno al plugin "browser\_config", dal quale recuperare il percorso all'eseguibile "Chrome Browser" e all'eseguibile "Chromedriver" e ultimo ma non meno importante la variabile "timeForResponse" che definisce il tempo da attendere prima di catturare l'elaborato di ChatGPT. Per importare questo modulo, tuttavia, dobbiamo ottenere il percorso assoluto della directory "chatgpt-lldb-plugin", directory che sta due livelli sopra il file corrente, e aggiungerlo all'inizio della lista dei percorsi in "sys.path" che è la lista dei percorsi di ricerca che Python utilizza per trovare i moduli importati. In questo modo possiamo importare moduli dalla directory "chatgpt-lldb-plugin" che altrimenti non sarebbe possibile importare trovandoci due livelli sotto di essa. Importato "browser\_config" possiamo procedere. Definiamo due variabili globali e inseriamovi i percorsi degli eseguibili recuperati da "browser\_config.py". Definiamo la funzione "init()" che serve per inizializzare il browser e verrà chiamata al momento del passaggio dalla "api\_key\_mode" alla "browser\_mode" in occasione dell'utilizzo del comando "switch" definito in "commands.py". All'interno della funzione dichiariamo la variabile globale "driver", salviarmo in "url" indirizzo da aprire col browser per interagire con ChatGPT via interfaccia web, cerchiamo una porta libera per la nostra connessione e lanciamo l'esecuzione di Chrome Browser fornendo URL e porta libera. Verifichiamo poi che l'utente abbia effettuato il login e superato eventuali CAPTCHA (test per determinare se l'utente è umano oppure un bot), se così è avvenuto impostiamo il driver per interfacciarci con il browser web e assegniamolo a "driver". Ritorniamo la stringa "done" che segnalare il successo dell'inizializzazione. La prossima funzione è progettata per trovare e restituire una porta di rete disponibile sul sistema. Creiamo una socket di rete utilizzando la libreria "socket" e assicuriamoci che venga chiusa correttamente quando il suo utilizzo è terminato grazie al costrutto "with". Associamo la socket appena creata ad un'indirizzo IP vuoto e ad una porta 0, in tal modo sarà il sistema operativo ad assegnare automaticamente una porta disponibile. Impostiamo un'opzione socket per consentire il riutilizzo dell'indirizzo IP e della porta. Infine restituiamo il numero di porta associato alla socket dopo che è stato assegnato automaticamente dal sistema operativo. In questo avremo una porta disponibile per utilizzare ChatGPT via browser. Con la prossima funzione lanciamo il Chrome Browser nella modalità di debugging remoto, questa consente di sfruttare "selenium" come strumento di automazione. Riceviamo come argomenti l'URL da aprire e la porta su cui tenere la connessione. Definiamo la funzione per aprire Chrome Browser. Al suo interno creiamo un elenco "chrome\_cmd" contenente i comandi da eseguire per avviare Google Chrome. Questi includono il percorso

al programma Chrome “chrome\_path”, l’impostazione della porta di debugging remoto, la specifica di un percorso per il profilo utente e l’URL del sito web da aprire. Utilizzando il modulo “subprocess”, avviamo un processo separato per eseguire il comando “chrome\_cmd”, che avvierà a sua volta Google Chrome con le opzioni specificate. Il parametro “stderr=subprocess.DEVNULL” serve a instradare l’output di errore in un “null device”, impedendo così che gli errori vengano stampati a schermo. Ora acquisiamo il PID del processo appena avviato e salviamolo all’interno di “previous\_chrome\_pids.yaml” dopo aver recuperato il percorso al file. In questo modo avremo poi la possibilità di chiudere il browser web facendo riferimento al PID, in modo tale da non lasciarlo aperto quando si cambia modalità, quando non si prosegue con il login oppure quando si esce da LLDB, onde non lasciare questo compito all’utente. Definita la funzione “open\_chrome()” creiamo un thread che la esegua in parallelo grazie al modulo “threading” ed eseguiamolo in parallelo. La prossima funzione, “wait\_for\_human\_verification()” risulta ad ora necessaria poiché non è stato risultato possibile automatizzare il login. Alle volte infatti bisogna superare dei CAPTCHA. Va anche detto che la schermata di login spesso cambia. Per questo lasciamo l’onere di effettuare il login e la verifica umana all’utente. Definiamo questa funzione. Stampiamo un avviso che fornisce le informazioni su quanto sta accadendo. Apriamo un ciclo “while” infinito. Ad ogni iterazione diremo all’utente di specificare se si è fatto il login, se non lo si è ancora fatto, oppure se si intende rinunciare al cambio di modalità e chiudere il browser. Se l’utente ha effettuato il login la funzione ritorna “y” ad indicare il successo dell’operazione e si esce dal ciclo. Se all’utente serve altro tempo e utilizza “n” aspettiamo 5 secondi. Se l’utente decide con “q” di rinunciare al cambio di modalità chiudiamo il browser con la funzione “quit()” che definiremo dopo e ritorniamo l’input. Infine se l’utente dà come input altro fuorché quanto richiesto gli si ricorda le opzioni valide. La prossima funzione si occupa di creare e configurare il driver “WebDriver” di Chrome Browser che può interagire con un’istanza di Chrome in modalità debugging remoto, necessario a “selenium” per interfacciarsi con Chrome Browser. Definiamo quindi “setup\_driver()” che riceve come argomento il numero di porta libero trovato prima. Al suo interno creiamo un’istanza di “webdriver.ChromeOptions()”, che consente di configurare le opzioni per il driver di Google Chrome. Aggiungiamo l’opzione sperimentale “debuggerAddress”, praticamente collega il driver “WebDriver” a un’istanza di Chrome in esecuzione con il debugging remoto abilitato. Ora possiamo creare un’istanza del driver “WebDriver” di Chrome Browser utilizzando il percorso dell’eseguibile del Chromedriver salvato in “chrome\_driver\_path” e le opzioni di configurazione specificate in “chrome\_options”. Questo driver sarà configurato per interagire con Chrome in modalità di debugging remoto. Infine, l’istanza del driver “WebDriver” di Chrome configurato viene restituita dalla funzione. Ora che possiamo affidarci al driver configurato per interagire attraverso il pacchetto “selenium” con Chrome Browser, definiamo le funzioni per la gestione di ChatGPT via interfaccia web. Definiamo “new\_chat()”, funzione con la quale possiamo selezionare una nuova chat. Questa ci tornerà utile nei comandi “send”, “ask” e “clear” definiti in “commands.py”. Dichiarato internamente alla funzione la variabile globale driver, utilizziamola per trovare l’elemento che ci appare a schermo col nome di “New chat” e clicchiamoci sopra. Se le operazioni hanno avuto successo ritorniamo il booleano “True”. La prossima funzione che andiamo a definire è “send\_to\_chatgpt()”.

Omonima della modalità API, ottiene lo stesso risultato sebbene sia implementata diversamente e sia più lenta. Dichiariamo internamente alla funzione la variabile globale "driver" e utilizziamola per selezionare l'area di testo dove inserire l'input per ChatGPT, inseriamo qui il contenuto del messaggio da inviare, ricevuto come argomento della funzione e premiamo il tasto "Enter" per inviare il messaggio. Ora si differenziano le operazioni da effettuare in base all'opzione specificata come secondo argomento della funzione: "wait". In caso dovesse essergli stata assegnata la stringa "noReplyNoWait" significa che a chiamare la nostra funzione è stato il comando "send", pertanto non ci interessa aspettare una risposta da ChatGPT e aspettiamo soltanto un secondo così da catturare quel minimo di elaborato del ChatBot per testimoniare attraverso l'assegnazione di un testo parziale alla variabile "reply", che l'interazione con ChatGPT è avvenuta correttamente. In caso alla variabile "wait" sia stata invece assegnata la stringa "waitForReply", significa che il comando che ha chiamato la funzione "send\_to\_chatgpt()" è "ask" e quindi è attesa una risposta completa da parte di ChatGPT. Dobbiamo aspettare l'elaborazione della risposta, catturarla e poi inviarla. Il tempo di attesa sarà definito dalla variabile globale "timeForResponse" definita nel file "browser\_config.py". Avvisiamo l'utente dell'attesa e aspettiamo il tempo detto. Una volta fatto questo possiamo recuperare l'ultimo messaggio della chat dall'interfaccia web, altro non sarà se non la risposta di ChatGPT al nostro messaggio. Ritorniamo la risposta per testimoniare l'avvenuta interazione nel caso la funzione sia stata chiamata dal comando "send" oppure per stampare la risposta di ChatGPT ad una nostra richiesta se la funzione chiamante è stata quella che implementa il comando "ask". La prossima funzione, "select\_chat()", riceve come argomenti il nome del target, ad esempio "test.c", e lo salva in "name" opportunamente formattato. Riceve anche l'argomento "option" che utilizzeremo poi per differenziare il caso in cui la funzione sia chiamata da "save" oppure da "load". Cerchiamo poi l'elemento con questo nome e lo selezioniamo, se tutto è andato regolarmente restituiamo il booleano "True", altrimenti, ma solo nel caso di opzione "load", lanciamo un'eccezione che avvisa l'utente della mancata presenza della chat richiesta. Risulta infatti superfluo avvisare l'utente della mancata presenza della chat in caso la funzione fosse stata chiamata dal comando "save", ciò che faremmo è salvarla. Sapere che prima non c'erano altri salvataggi interessa poco all'utente. La prossima funzione è "delete\_chat()", viene utilizzata nel comando "save" per cancellare il salvataggio della chat precedente, se presente. La funzione cerca il pulsante del cestino affianco alla chat selezionata, lo clicca, poi cerca il pulsante rosso "Delete" per confermare l'operazione e clicca anche su questo. In caso di problemi viene lanciata un'opportuna eccezione notificando l'utente del fallimento dell'operazione in un messaggio. La funzione successiva è "rename\_chat()" e costituisce uno dei tasselli fondamentali per l'implementazione del comando "save" di "commands.py". All'interno di questa definiamo la funzione "std\_exception()" che si occupa di notificare l'utente del fatto che il sito di OpenAI che garantisce l'interazione con ChatGPT via web è stato modificato e quindi devono essere aggiornate le procedure di automazione operate attraverso il pacchetto "selenium". Utilizziamo un "try, except". Formattiamo il nome del target ricevuto come argomento, ad esempio "test.c", e salviamolo nella variabile "name". Questo sarà il nome che vogliamo dare alla chat a cui contiamo di riferirci nuovamente in futuro. Cerchiamo e selezioniamo prima la chat corrente, alla cima della lista. Poi

cerchiamo e clicchiamo il bottone a forma di matita della chat selezionata, posto alla sinistra del bottone a forma di cestino. Ora il nome della chat corrente diventerà modificabile, selezioniamo quindi la casella di input e inseriamovi il nuovo nome salvato prima in “name”. Cerchiamo e clicchiamo il pulsante di conferma, a forma di spunta, che si trova al posto della matita presente prima. Ritorniamo il booleano “True” a segnalare la corretta esecuzione di tutte le istruzioni. In caso venisse lanciata una delle 4 eccezioni gestite stampiamo il rispettivo messaggio di errore. L’ultima funzione ci consente infine di chiudere il browser web aperto. Viene utilizzato in questo script quando si desiste dal fare login e “human verification” dando come input “q”, in “commands.py” quando si usa il comando ‘switch’ per passare dalla modalità browser alla modalità API e si utilizza anche nello script “cleaner.py” quando si vuole monitorare lo stato del processo “lldb” per chiudere alla chiusura di questo anche il browser web. La funzione “quit()” è così implementata. Recuperiamo il percorso al file YAML “previous\_chrome\_pids.yaml”, da qui prenderemo il PID della sessione di Chrome Brower avviata. Proviamo ad aprire, in sola lettura, il file di cui abbiamo recuperato il percorso completo e salviamoci i PID nella variabile “chrome\_pids”. Se il contenuto non è nullo e si tratta di una lista allora procediamo. Per ciascun PID della lista in “chrome\_pids” terminiamo il processo identificato da tale PID instradando l’output di errore in un “null device” per non avere eventuali messaggi a schermo. Se la lista invece non è valida significa che non avendo alcun PID non posso chiudere il browser web e quindi l’utente deve procedere manualmente, lo avvisiamo di ciò. In caso si dovesse verificare un’eccezione durante l’apertura del file YAML allora stampiamo un messaggio contenente i dettagli dell’eccezione. È bene precisare che il browser chrome, per come è stato organizzato l’intero applicativo, viene chiuso tante volte quante viene aperto, pertanto, in realtà, sarà uno e uno soltanto il PID salvato in “previous\_chrome\_pids.yaml” e quindi la lista recuperata da questo file contiene un solo elemento, pertanto l’utilizzo della stessa risulta un po’ esagerato. È stata lasciata questa struttura poiché si potrebbe sempre pensare di cambiare l’utilizzo di “quit()” nel comando “switch”. Questo fatto cambierebbe le carte in gioco e allora la lista si rivelerebbe una buona struttura dati capace di garantire robustezza al sistema, se non altro in questo script.

```
1 from selenium import webdriver
2 from selenium.webdriver.remote.webdriver import By
3 from selenium.webdriver.common.keys import Keys
4 from selenium.webdriver.support.ui import WebDriverWait
5 from selenium.webdriver.support import expected_conditions as EC
6 from selenium.common.exceptions import NoSuchElementException,
7     InvalidElementStateException, ElementNotInteractableException,
8     StaleElementReferenceException, TimeoutException
9
10 import time
11 import socket
12 import threading
13 import subprocess
14 import sys
15 from os import path
16 import yaml
17
18 program_path = path.abspath(path.join(path.dirname(__file__), '...', '...'))
19 sys.path.insert(0, program_path)
```

```
17 import browser_config
18
19 chrome_path = browser_config.chrome_path
20 chrome_driver_path = browser_config.chrome_driver_path
21
22 def init():
23     global driver
24     url = r"https://chat.openai.com"
25     free_port = find_available_port()
26     launch_chrome_with_remote_debugging(free_port, url)
27     response = wait_for_human_verification()
28     if response == "y":
29         driver = setup_webdriver(free_port)
30         return "done"
31
32 def find_available_port():
33     with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
34         s.bind(('', 0))
35         s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
36     return s.getsockname()[1]
37
38 def launch_chrome_with_remote_debugging(port, url):
39     def open_chrome():
40         chrome_cmd = [chrome_path, f"--remote-debugging-port={port}",
41                       '--user-data-dir=remote-profile', url]
42         chrome_process = subprocess.Popen(chrome_cmd, stderr=
43                                         subprocess.DEVNULL)
44         chrome_pid = chrome_process.pid
45         if chrome_pid:
46             chrome_pids_path = path.join(path.dirname(path.abspath(
47                 path.realpath(__file__))), "previous_chrome_pids.yaml")
48             with open(chrome_pids_path, "w") as file:
49                 yaml.dump([chrome_pid], file)
50
51     chrome_thread = threading.Thread(target=open_chrome)
52     chrome_thread.start()
53
54 def wait_for_human_verification():
55     print("You need to manually complete the log-in or the human
56     verification if required.")
57     while True:
58         user_input = input("Enter 'y' if you have completed the log-
59         in or the human verification, 'n' to check again, or 'q' to abort
60         the login procedure: ").lower()
61
62         if user_input == 'y':
63             return user_input
64         elif user_input == 'n':
65             print("\nWaiting for you to complete the human
66             verification")
67             time.sleep(5)
68         elif user_input == 'q':
69             quit()
70             return user_input
71         else:
72             print("\nInvalid input. Please enter 'y', 'n', or 'q'.")
```

```

67 def setup_webdriver(port):
68     chrome_options = webdriver.ChromeOptions()
69     chrome_options.add_experimental_option("debuggerAddress", f"127.0.0.1:{port}")
70     driver = webdriver.Chrome(executable_path=chrome_driver_path,
71                               options=chrome_options)
72
73     return driver
74
75 def new_chat():
76     global driver
77     new_chat = driver.find_element(By.LINK_TEXT, "New chat")
78     new_chat.click()
79
80     return True
81
82 def send_to_chatgpt(content, wait):
83     global driver
84     input = driver.find_element(By.TAG_NAME, "textarea")
85     input.send_keys(content)
86     input.send_keys(Keys.RETURN)
87
88     if wait == "noReplynoWait":
89         time.sleep(1)
90     elif wait == "waitForReply":
91         tfr = browser_config.timeForResponse
92         print(f"Waiting {tfr} seconds for ChatGPT reply (wait time is
93               editable in 'browser_config')")
94         time.sleep(tfr)
95
96     reply = driver.find_elements(by=By.CSS_SELECTOR, value="div.text-
97                                   base")[-1].text
98     return reply
99
100 def select_chat(target_name, option):
101     try:
102         name = "dbChat_" + target_name
103         chat_button = driver.find_element(By.LINK_TEXT, name)
104         chat_button.click()
105
106         return True
107     except NoSuchElementException:
108         if option == "load":
109             print(f"There's no debugging state of {target_name}")
110
111 def delete_chat(target_name):
112     try:
113         bin_button = WebDriverWait(driver, 10).until(EC.
114             presence_of_element_located((By.XPATH, "(//button[@class='p-1
115               hover:text-white'])[2]")))
116         bin_button.click()
117         delete_button = WebDriverWait(driver, 10).until(EC.
118             presence_of_element_located((By.XPATH, "//button[@class='btn
119               relative btn-danger']//div[contains(text(), 'Delete')]")))
120         delete_button.click()
121
122     except StaleElementReferenceException:

```

```

116         print("Operation failed. Try again")
117
118     def rename_chat(target_name):
119         def std_exception():
120             print("The https://chat.openai.com/ html code has been
121             changed. Patch is required.")
122
123         try:
124             name = "dbChat_"+target_name
125
126             chat_button = driver.find_element(By.CSS_SELECTOR, "li:first-
127             child")
128             chat_button.click()
129             modify_button = WebDriverWait(driver, 10).until(EC.
130             presence_of_element_located((By.XPATH, "(//button[@class='p-1
131             hover:text-white'])[1]")))
132             modify_button.click()
133
134             text_input = driver.find_element(By.XPATH, "//input[@type='
135             text']")
136             driver.execute_script("arguments[0].value = arguments[1];",
137             text_input, name)
138
139             check_button = WebDriverWait(driver, 10).until(EC.
140             presence_of_element_located((By.XPATH, "(//button[@class='p-1
141             hover:text-white'])[1]")))
142             check_button.click()
143
144             return True
145
146     except InvalidElementStateException:
147         std_exception()
148     except ElementNotInteractableException:
149         std_exception()
150     except StaleElementReferenceException:
151         print("Operation failed. Try again")
152     except TimeoutException:
153         print("An error has occurred on ChatGPT, the chat has been
154         lost")
155
156     def quit():
157         chrome_pids_path = path.join(path.dirname(path.abspath(path.
158             realpath(__file__))), "previous_chrome_pids.yaml")
159         try:
160             with open(chrome_pids_path, 'r') as file:
161                 chrome_pids = yaml.safe_load(file)
162                 if chrome_pids is not None and isinstance(chrome_pids,
163                     list):
164                     for pid in chrome_pids:
165                         subprocess.run(["kill", "-9", str(pid)], stderr=
166                         subprocess.DEVNULL)
167                     else:
168                         print("Can't close previous chrome session, proceed
169                         manually.")
170                 except Exception as e:
171                     print(f"Error in YAML file reading:", e)

```

Listing 3.17. chatgpt-lldb-plugin/main/browser\_connection/browser\_messaging.py

### 3.5.3 cleaner.py

Importiamo i moduli necessari al funzionamento dello script. Dalla libreria standard abbiamo “time” e “atexit”, dal pacchetto “psutil” abbiamo “psutil”. Importiamo poi il modulo interno al plugin “browser\_messaging”. Questo script si occupa del monitoraggio del processo “lldb” e ha come scopo quello di chiudere il browser web utilizzato nella modalità browser alla chiusura di LLDB. Definiamo la funzione “cleanup()”, questa chiama la funzione “quit()” dello script “browser\_messaging.py” e termina il processo di Chrome Browser aperto durante la sessione della modalità browser. Definiamo poi la funzione “lldb\_monitor()” dedita al monitoraggio del processo “lldb” che si sta utilizzando per il debugging. Usiamo un ciclo infinito per cercare il processo di LLDB in esecuzione. Con “psutil.process\_iter()” scansioniamo tutti i processi in esecuzione. Se viene trovato un processo chiamato “lldb”, salviamo il suo PID in “lldb\_process” e usciamo dal ciclo. Se il processo non viene trovato si esce dal ciclo. Se il processo è in stato “zombie”, ovvero è terminato ma il processo padre non ha eseguito la chiamata di sistema “wait”, allora devono essere liberate le risorse da questo utilizzate, chiamiamo “cleanup()” e usciamo dal ciclo. Definita questa funzione, registriamo la funzione “cleanup()” per essere eseguita alla chiusura del programma utilizzando “atexit.register()” e chiamiamo “lldb\_monitor()”. In questo modo, in caso il monitoraggio dovesse essere interrotto, eccezione che gestiamo, verrà comunque eseguita la funzione “cleanup()”.

```

1 import psutil
2 import time
3 import atexit
4 import browser_messaging
5
6 def cleanup():
7     browser_messaging.quit()
8
9 def lldb_monitor():
10    try:
11        lldb_process = None
12        while True:
13            for proc in psutil.process_iter(['pid', 'name']):
14                if proc.info['name'] == 'lldb':
15                    lldb_process = psutil.Process(proc.info['pid'])
16                    break
17
18            if lldb_process is None:
19                print("LLDB process not found... exiting")
20                break
21
22            if lldb_process.status() == psutil.STATUS_ZOMBIE:
23                cleanup()
24                break
25
26            time.sleep(1)
27
28    except KeyboardInterrupt:
29        print("LLDB tracking interrupted")
30
31 atexit.register(cleanup)
```

```
32 lldb_monitor()
```

**Listing 3.18.** chatgpt-lldb-plugin/main/browser\_connection/cleaner.py

### 3.5.4 previous\_chrome\_pids.yaml

In questo file YAML viene salvata una lista dei PID dei processi di Chrome Browser che vengono aperti a seguito dell'utilizzo della funzione “`launch_chrome_with_remote_debugging()`” dello script “`browser_messaging.py`”. Per ogni volta che la funzione viene chiamata viene aperto un processo e si aggiunge un PID alla lista. Per come abbiamo implementato l'applicativo, per ogni volta che la funzione viene chiamata è assicurata la chiusura del processo associato al PID contenuto nella lista, pertanto tale lista si mantiene di lunghezza unitaria.

```
1 - 28388
```

**Listing 3.19.** chatgpt-lldb-plugin/main/browser\_connection/previous\_chrome\_pids.yaml

## 3.6 Sottosottodirectory 'saves'

### 3.6.1 dbChat\_test.c.yaml

In questo file YAML abbiamo salvato, a titolo esemplificativo, la lista di dizionari che rappresenta la conversazione con il ChatBot per il debugging di un programma con nome “`test.c`”. Questo file viene creato a seguito dell'utilizzo del comando “`save`” definito in “`commands.py`”. Il contenuto può essere poi importato nella sessione di debugging con il comando “`load`”. È stato scelto il linguaggio YAML poiché è un linguaggio di serializzazione dei dati leggibile dall'uomo, quindi consente di verificare il contenuto scambiato tra l'utente e ChatGPT facilmente in fase di sviluppo ed eventualmente consente di correggere quanto di errato nei prompt. Inoltre rappresenta i dati in modo chiaro e in modo più semplice rispetto il formato per lo scambio di dati JSON.

```
1 - content: Sei un gentile e utile assistente per il debugging e la
      correzione del
      codice.
2   role: system
3 - content: "Utilizzo il sistema Linux-5.10.0-25-amd64-x86_64-with-
4   glibc2.31 su cui\
5     \ ho installato lldb version 11.0.1 . Con LLDB sto facendo il
6     debugging dell'eseguibile\
7     \ di '/home/carlo/test.c'. Il codice sorgente \xE8: \n      1  #
8       include <stdio.h>\n\n      2  \n      3  int main() {\n      4  FILE *pfile;\n      5  int \
9    ch;\n      6  \n      7  puts(\"Opening file...\\\" );\n      8  pfile =\
10   fopen(\"FILE.TXT\", \"r\\\" );\n      9  \n      10  puts(\"\
11  Displaying file\
12  \\ ...\\\" );\n      11  while ((ch = fgetc(pfile)) != EOF) {\n      12  printf(\"\
13  <%c>\\\", ch);\n      13  }\\n      14  \n      15  puts(\"\
14  Closing file ...\\\" \\\n      15
```

```

12      );\n      16      fclose(pfile);\n      17      \n      18      return\n0;\n      19  }\n\n  Non devi rispondermi. Riceverai ulteriori info nel prossimo\nmessaggio.\nIn questo\\
  \\ momento:\n    1. La linea corrente \xE8 la numero: 11 .\n    2.\nLe variabili\\
  \\ nel blocco sono: \n          - La variabile 1 ha nome 'pfile',\nvalore '0x0000000000000000',\\
  \\ tipo 'FILE *' e indirizzo '140728260435768' .\n          - La\nvariabile 2 ha nome\\
  \\ 'ch', valore '32765', tipo 'int' e indirizzo '140728260435764'\n.\nNon devi rispondermi.\n
  \\ Ti fornir\xF2 le mie richieste al prossimo messaggio.\n"
role: user
20 - content: "Grazie per le informazioni fornite. Prima di continuare\ncon le richieste,\n  \\ vorrei annotare che l'indirizzo fornito per le variabili sembra\nnon essere corretto.\n  \\ Gli indirizzi dovrebbero essere numeri esadecimali e possono\nvariare ad ogni\\
  \\ esecuzione del programma. Potresti controllare se la stessa\ninformazione \xE8\\
  \\ visualizzata in un formato diverso? Potrebbe essere utile per\nil debug successivo.\n\\
  \\ Al fine di fornirti il supporto di debug, potresti fornirmi\ndelle informazioni\\
  \\ sul problema specifico che stai riscontrando?"\nrole: assistant
28 - content: "L'output del programma \xE8 il seguente: 'Segmentation\nfault '. Il programma\\
  \\ durante il debugging si \xE8 interrotto alla linea 11. La\nragione dello stop\\
  \\ \xE8: SIGSEGV indirizzo non valido. Quanti e quali sono gli\nerrori del codice?\n  \\ Differenziali per tipologia."
32 role: user
33 - content: "L'errore principale che sta causando il segmentation\nfault \xE8 la mancanza\\
  \\ di gestione di un'apertura fallita del file nel blocco di\ncodice. In caso di\\
  \\ errore nell'apertura del file, la variabile pfile rimarr\xE0\neguale a NULL e\\
  \\ l'accesso a pfile nella successiva condizione di ciclo while\ncausere\xE0 un SIGSEGV\\
  \\ (segmentation fault) perch\xE9 si sta cercando di\ndereferenziare un puntatore\\
  \\ NULL.\nPer risolvere questo problema, \xE8 buona pratica\ncontrollare il valore\\
  \\ restituito dalla funzione fopen() per verificare se l'apertura\ndel file \xE8\\
  \\ stata eseguita correttamente. In caso contrario, \xE8\nnecessario gestire l'errore\\
  \\ in modo appropriato.\nEcco un esempio di come potrebbe essere\nmodificato il\\
  \\ codice per gestire l'apertura del file in modo sicuro:\n\n```\n#include <stdio.h>\n\\

```

```
43  \nint main() {\n    FILE *pfile;\n    int ch;\n\n    puts(\"Opening\n    file...\" );\n    pfile\\n\n    \\ = fopen(\"FILE.TXT\", \"r\");\\n\\n    if (pfile == NULL) {\\n\n        fprintf(stderr,\\n\n        \\ \"Error opening file\\n\");\\n        return 1; // o un valore\n        diverso da zero per\\n\n        \\ indicare un errore\\n    }\\n\\n    puts(\"Displaying file ...\" );\\n\n    while ((ch =\\n\n        \\ fgetc(pfile)) != EOF) {\\n        printf(\"%c\", ch);\\n    }\\n\\n    puts(\"Closing\\n\n    file ...\" );\\n    fclose(pfile);\\n\\n    return 0;\\n}\\n\\n\\n\\nFacendo questa modifica,\\n\nil programma gestirà correttamente l'eventuale apertura\nfallita del file\\n\n    e non si verificherà il segmentation fault.\n\nrole: assistant
```

**Listing 3.20.** chatgpt-lldb-plugin/main/saves/dbChat\_test.c.yaml



## Capitolo 4

# Risultati sperimentali

### 4.1 Insieme dei dati

Per l'analisi delle prestazioni dell'applicativo si è resa necessaria la redazione di esperimenti mirati, atti a valutare la capacità di ChatGPT di rilevare gli errori grazie alle informazioni recuperate e fornite dal plugin oltre che i vantaggi di utilizzo del plugin stesso. Per ogni classe di errori presa in esame sono state realizzate una o più istanze dello stesso, garantendo quindi, per ogni tipologia di errore, uno o più di un esempio. Ciascun esempio altro non è che un programma in C. Gli errori di logica, sintassi e semantica sono stati raggruppati in quanto affini come macro-categoria rispetto tutti gli altri errori trattati poi, i quali sono invece classificabili come errori di runtime. Si hanno un totale di 15 tipologie di errori prese in esame e di 27 diversi programmi realizzati ad hoc per le due sperimentazioni a seguire. L'insieme dei dati è limitato, ma vi sono delle ragioni perché non è stato potuto fare altrimenti:

- **Dataset documentato mancante:** non è stato trovato un insieme di programmi buggati abbastanza grande e al contempo documentato, quindi con la tipologia dell'errore specificato assieme ad una descrizione dello stesso. Pertanto il candidato ha provveduto personalmente a creare uno ad uno i programmi necessari alle sperimentazioni corredandoli degli errori che ChatGPT avrebbe poi dovuto scovare.
- **Impossibilità di automatizzare i controlli:** non è stato possibile trovare un modo di automatizzare il controllo dei responsi del ChatBot, infatti bisognerebbe realizzare un programma che per ciascun esempio fa controllare ad un'IA se l'IA utilizzata nella sperimentazione ha scovato correttamente i bug, per fare ciò si ha quindi bisogno di un controllore di un grado di intelligenza maggiore e tale compito al momento può solo che essere adempiuto da un'essere umano.

Di seguito sono presentate le diverse tipologie d'errore prese in esame provviste delle rispettive descrizioni.

- **Errori di logica, sintassi e semantica:**
  - **Errori di logica:** Questi errori sono causati da un ragionamento improprio nell'implementazione del programma. Significa che il codice può

funzionare esattamente come previsto, ma il risultato finale non è ciò che ci si aspettava per via di un'interpretazione errata o perché è stata applicata in modo errato la logica del problema. Gli errori di logica spesso richiedono una revisione approfondita dell'algoritmo o dell'approccio alla soluzione.

- **Errori di sintassi:** Questi errori sono simili agli errori grammaticali in un testo. Si verificano quando il codice non rispetta le regole del linguaggio di programmazione. Ad esempio, quando si ha una parentesi mancante, un punto e virgola dimenticato o una parola chiave del linguaggio scritta in modo errato. Questi errori di sintassi sono generalmente facili da individuare perché il compilatore o l'interprete del linguaggio di programmazione segnalerà un errore.
- **Errori semanticici:** sono errori più subdoli. In questo caso il codice può essere corretto dal punto di vista sintattico, ma produce risultati inaspettati o non desiderati a causa di una comprensione errata del significato delle istruzioni. Ad esempio, potrebbero essere sommate due variabili di tipo diverso senza convertirle adeguatamente, causando risultati errati. Gli errori semanticici possono essere più difficili da individuare rispetto agli errori di sintassi e richiedono una comprensione più profonda del linguaggio di programmazione e del problema che stai risolvendo.
- **Buffer overflow:** Si verifica quando un programma scrive dati oltre i limiti di un buffer (una zona di memoria temporanea), causando la sovrascrittura di dati importanti o vulnerabilità di sicurezza.
- **Divisione per zero:** Questo errore si verifica quando si tenta di dividere un numero per zero, cosa che matematicamente non è definita. Può causare crash del programma o risultati imprevisti.
- **Accesso a memoria non allocata o non valida:** Si verifica quando un programma cerca di accedere a una zona di memoria a cui non ha il permesso o che non è stata allocata. Può causare crash o comportamenti imprevedibili.
- **Puntatori nulli:** Si verificano quando si cerca di dereferenziare un puntatore che punta a nulla o non è stato inizializzato correttamente. Può causare crash del programma.
- **Overflow aritmetico:** Si verifica quando il risultato di un'operazione aritmetica supera il limite massimo rappresentabile in una variabile, causando risultati non validi.
- **Underflow aritmetico:** Si verifica quando il risultato di un'operazione aritmetica è più piccolo del limite minimo rappresentabile in una variabile, causando risultati non validi.
- **Ricorsione infinita:** Si verifica quando una funzione chiama se stessa in modo ricorsivo senza una condizione di uscita adeguata, causando un loop infinito.

- **Accesso a variabili non inizializzate:** Si verifica quando si utilizzano variabili prima che vengano inizializzate con un valore valido. Può causare risultati imprevedibili.
- **Accesso fuori dai limiti dell'array:** Si verifica quando si cerca di accedere a un elemento di un array al di fuori dei suoi limiti dichiarati. Può causare comportamenti imprevedibili o errori di segmentazione.
- **File non trovato o non accessibile:** Si verifica quando un programma cerca di accedere a un file che non esiste o al quale non ha i permessi di accesso.
- **Comportamento non definito:** Si riferisce a situazioni in cui il comportamento di un programma non è specificato o prevedibile dalle specifiche del linguaggio di programmazione. Può variare da un ambiente di esecuzione all'altro.
- **Loop infiniti:** Si verificano quando un ciclo (come un ciclo while o for) non ha una condizione di uscita adeguata, causando l'esecuzione continua del ciclo senza fine.

## 4.2 Descrizione degli esperimenti

Ogni sperimentazione verrà affrontata utilizzando il comando “autodebug”, in grado di semplificare e raggruppare le funzioni sviluppate nel plugin automatizzando quindi il processo di debugging di ciascun eseguibile (ottenuto con “gcc -o test test.c -lm -g”) caricato come target in LLDB . Seguirà la stampa a schermo del responso di ChatGPT contenente gli errori scovati dal ChatBot. I responsi sono quindi analizzati uno per uno, poi i risultati sono astratti e categorizzati in una tabella. Solo nel caso della sottosezione “Loop infiniti” della I sperimentazione l’uso del comando “autodebug” verrà meno e saranno, invece, usati “send” e “ask” in modo equivalente senza ricadere nell’esecuzione infinita dei loop. In questo caso, infatti, “autodebug” metterebbe in esecuzione i loop infiniti e solo manualmente ciò è evitabile.

Sono presentate due sperimentazioni differenti. Per ciascun esempio trattato si è considerato soltanto il primo tentativo. Nella I sperimentazione si vuole vedere se ChatGPT sia in grado e in che misura di rilevare gli errori nei programmi grazie alle informazioni fornite e ricavate attraverso l’applicativo realizzato. Nella II sperimentazione si vuole vedere quali sono i vantaggi di utilizzare l’applicativo rispetto al solo invio del codice a ChatGPT via web. Si osserva quanto siano rilevanti le informazioni di debugging recuperate attraverso l’applicativo affinché ChatGPT scovi gli errori.

## 4.3 I Sperimentazione

In questa sezione, per ogni tipologia di errore verranno presentati uno o più esempi di codice C contenenti lo stesso. Verrà anche presentata la schermata con il responso di ChatGPT e una breve descrizione delle istanze dell’errore per ciascuna tipologia d’errore specificando se ChatGPT con la sua risposta ha rilevato o meno il problema e se sì in quale misura.

### 4.3.1 Errori di logica, sintassi e semantica

#### test\_1.c

Qui vi sono due errori. Il primo, di tipo sintattico poiché si attua un'assegnazione all'interno di un “if” condizionale, cosa sempre verificata che modifica inoltre il valore di “foo”. Il secondo errore è di tipo logico, infatti sembra si voglia assegnare un nuovo valore a “foo” utilizzando tuttavia un’operatore di confronto. Nel responso di ChatGPT vengono trovati correttamente gli errori. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     int foo = 3;
5
6     if (foo = 5)
7         foo == 7;
8
9     printf("%d\n", foo);
10
11    return 0;
12 }
```

**Listing 4.1.** i\_testing/programs/test\_1.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_1
(lldb) target create "test_1"
Current executable set to '/home/carlo/I_sperimentazione/test_1' (x86_64).
(lldb) autodebug
----- Execution Started -----
5
----- Execution Terminated -----
Dai risultati che hai ottenuto durante l'esecuzione del programma, possiamo individuare alcuni errori nel tuo codice. Ecco un'analisi differenziata per tipologia:
1. Errore di assegnazione nel condizionale:
   - Nella riga 6, hai utilizzato l'operatore di assegnazione `=` anziché l'operatore di confronto `==`. Questo significa che stai assegnando il valore 5 alla variabile `foo` invece di confrontare se `foo` è uguale a 5. Questo errore può causare un comportamento imprevisto nel tuo programma.

2. Errore nell'uso dell'operatore di confronto:
   - Nella riga 7, hai nuovamente utilizzato l'operatore di confronto `==` invece dell'operatore di assegnazione `=`. Questo significa che non stai assegnando il valore 7 alla variabile `foo`, ma stai solo confrontando se `foo` è uguale a 7. Questa istruzione non ha effetto sul valore di `foo`.

3. Mancanza delle parentesi graffe nell'istruzione if:
   - Anche se la riga 7 ha un solo comando all'interno dell'istruzione if, è buona pratica utilizzare le parentesi graffe per delimitare il corpo dell'istruzione. Sebbene non sia un errore sintattico, l'omissione delle parentesi può causare confusione o errori futuri se dovessi aggiungere altre istruzioni all'interno del blocco dell'if.

Spero che questa analisi ti sia utile per comprendere gli errori presenti nel tuo codice. Se hai domande o bisogno di ulteriori chiarimenti, sarò qui per aiutarti.
(lldb) 
```

**Figura 4.1.** Debugging di test\_1.c

**test\_2.c**

Qui si vuole stampare un valore randomico dell'intervallo [i,j]. L'errore è di tipo logico e sta nel non aver considerato la precedenza dell'operatore "%" rispetto alla sottrazione. La formula corretta è "val = i + (rand() % (j - i + 1))", facendo quindi uso delle parentesi per garantire l'ordine corretto di esecuzione delle operazioni. Nel responso di ChatGPT l'errore non viene trovato.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 int main() {
6     int i = 6;
7     int j = 7;
8     int val;
9
10    srand(time(NULL));
11
12    val = i + (rand() % j - i + 1);
13
14    printf("Here's a random value in [%d,%d]: %d\n", i, j, val);
15
16    return 0;
17 }
```

**Listing 4.2.** i\_testing/programs/test\_2.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_2
(lldb) target create "test_2"
Current executable set to '/home/carlo/I_sperimentazione/test_2' (x86_64).
(lldb) autodebug
----- Execution Started -----
Here's a random value in [6,7]: 5
----- Execution Terminated -----
Dopo aver analizzato il tuo codice, non sembra ci siano errori sintattici evidenti. Tuttavia, ci sono alcune problematiche potenziali che potrebbero causare risultati inaspettati. Ecco i punti da considerare:

1. La chiamata alla funzione `time(NULL)` all'interno di `srand` potrebbe generare lo stesso valore di tempo iniziale ad ogni esecuzione del programma. Questo significa che `rand` genererà sempre lo stesso sequenza di numeri casuali. Sarà bene spostare la chiamata `srand` all'esterno del ciclo principale per garantire un inizializzazione diversa ad ogni esecuzione.

2. La formula `val = i + (rand() % j - i + 1)` potrebbe generare un valore fuori dall'intervallo [i, j] se `i` è maggiore di `j`. In questo caso, la formula diventerà `val = i + (rand() % 0) + 1`, che risulterà in un errore. Per evitare ciò, puoi verificare se `i` è maggiore di `j` e in tal caso scambiare i loro valori.

3. L'output del programma stampa il valore `val` generato casualmente. Nel tuo caso, l'output è "Here's a random value in [6,7]: 5", che indica che `val` assume il valore `5`. Tuttavia, questo valore non sembra essere valido all'interno del l'intervallo [i, j]. Potrebbe essere il risultato della formula non corretta o di altri problemi.

Ti suggerisco di controllare attentamente i punti sopra elencati e cercare di risolvere le eventuali problematiche in modo da ottenere un comportamento corretto del programma.
(lldb) 
```

**Figura 4.2.** Debugging di test\_2.c

**test\_3.c**

Non vi sono errori nel codice di questo programma. Va solo tenuto presente che non avendo assegnato alla variabile globale j alcun valore, allora questa verrà inizializzata a zero secondo lo standard C. Nel responso di ChatGPT l'errore non viene trovato.

```

1 #include <stdio.h>
2
3 int i = 5;
4 int j;
5
6 void do_nothing() {
7 }
8
9 int foo(int j) {
10    for (i=0; i<j; i++)
11        do_nothing();
12    return j;
13 }
14
15 void ineedj() {
16    printf("j is %d\n", j);
17 }
18
19 int main() {
20    int j;
21
22    j = foo(i);
23    ineedj();
24
25    return 0;
26 }
```

**Listing 4.3.** i\_testing/programs/test\_3.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_3
(lldb) target create "test_3"
Current executable set to '/home/carlo/I_sperimentazione/test_3' (x86_64).
(lldb) autodebug
----- Execution Started -----
j is 0
----- Execution Terminated -----
Dopo aver analizzato il codice sorgente, ho identificato due errori nel tuo programma:

1. Nella funzione `foo()`, la variabile `i` viene utilizzata per il loop `for`, ma essa è definita come una variabile globale all'inizio del programma. Questo significa che modificando il valore di `i` all'interno della funzione, si sta effettivamente modificando la variabile globale. Questo potrebbe non essere l'intento desiderato. Per risolvere questo errore, puoi dichiarare una variabile locale all'interno della funzione `foo()` per l'iterazione del loop.

2. Nella funzione `main()`, dichiari una variabile `j` locale, ma non la inizializzi con un valore. Di conseguenza, quando viene passata alla funzione `foo()`, il suo valore sarà imprevedibile, poiché non è stato inizializzato. Per correggere questo errore, puoi inizializzare la variabile `j` alla riga 20 con un valore appropriato.

I due errori sopra menzionati sono differenziali in base alla tipologia di problematica che presentano. Il primo errore è legato all'uso non opportuno di una variabile globale, mentre il secondo errore riguarda la mancata inizializzazione di una variabile locale.
(lldb) 
```

**Figura 4.3.** Debugging di test\_3.c

**test\_4.c**

L'errore sta nel confronto operato nell'“if” condizionale. Invece che i contenuti delle stringhe, vengono confrontati i puntatori delle stringhe. Si tratta di un errore logico. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono proposte le correzioni.

```
1 #include <stdio.h>
2
3 int main() {
4     char* string1 = "Hello";
5     char* string2 = "World";
6
7     if (string1 == string2)
8         printf("%s and %s are equals.\n", string1, string2);
9     else
10        printf("%s and %s are different.\n", string1, string2);
11
12    return 0;
13 }
```

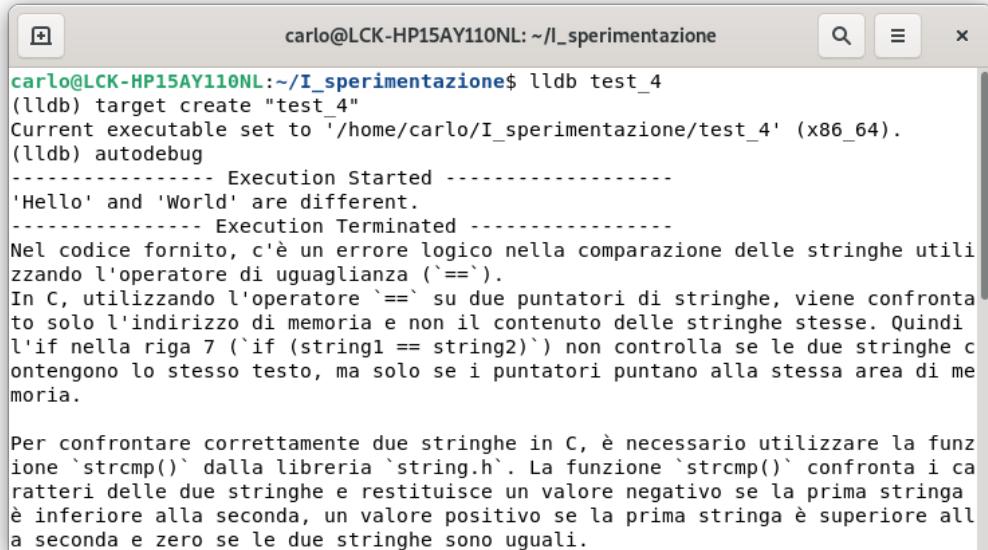
**Listing 4.4.** i\_testing/programs/test\_4.c

**test\_5.c**

Nel programma non vengono inseriti i “break” per ciascun caso dello “switch”. Ciò porta ad eseguire sia il caso scelto che quelli successivi. Si tratta di un errore logico. Nel responso di ChatGPT l'errore viene identificato correttamente. Viene fatta un'osservazione sulle parentesi graffe che risulta erronea, poi proposte le correzioni.

```
1 #include <stdio.h>
2
3 int main() {
4     int choice = 2;
5     switch (choice) {
6         case 1:
7             printf("Case 1\n");
8         case 2:
9             printf("Case 2\n");
10        default:
11            printf("Default case\n");
12    }
13
14    return 0;
15 }
```

**Listing 4.5.** i\_testing/programs/test\_5.c



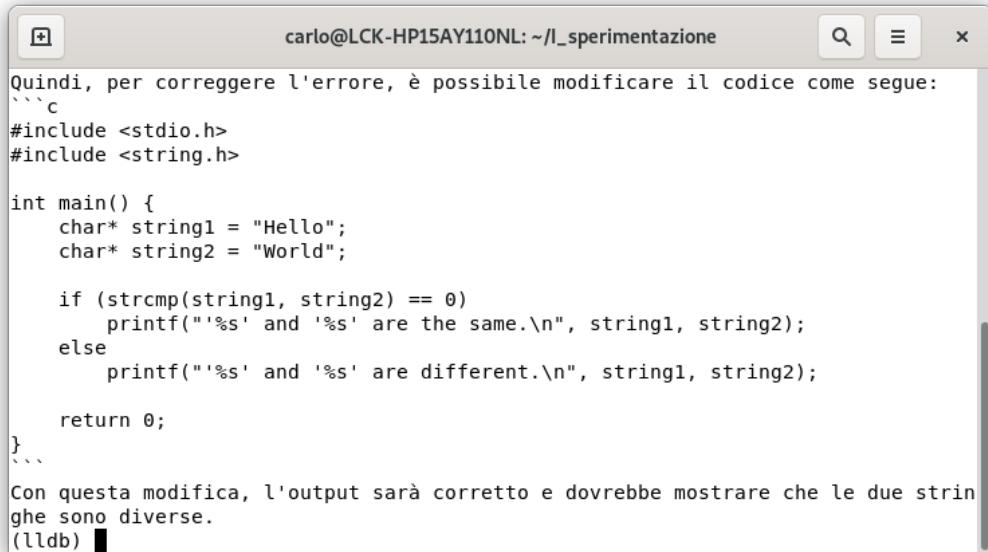
```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_4
(lldb) target create "test_4"
Current executable set to '/home/carlo/I_sperimentazione/test_4' (x86_64).
(lldb) autodebug
----- Execution Started -----
'Hello' and 'World' are different.
----- Execution Terminated -----
Nel codice fornito, c'è un errore logico nella comparazione delle stringhe utilizzando l'operatore di uguaglianza ('==').
In C, utilizzando l'operatore '==' su due puntatori di stringhe, viene confrontato solo l'indirizzo di memoria e non il contenuto delle stringhe stesse. Quindi l'if nella riga 7 ('if (string1 == string2)') non controlla se le due stringhe contengono lo stesso testo, ma solo se i puntatori puntano alla stessa area di memoria.

Per confrontare correttamente due stringhe in C, è necessario utilizzare la funzione `strcmp()` dalla libreria `string.h`. La funzione `strcmp()` confronta i caratteri delle due stringhe e restituisce un valore negativo se la prima stringa è inferiore alla seconda, un valore positivo se la prima stringa è superiore all'a seconda e zero se le due stringhe sono uguali.

```

(a) Debugging di test\_4.c



```

Quindi, per correggere l'errore, è possibile modificare il codice come segue:
```c
#include <stdio.h>
#include <string.h>

int main() {
    char* string1 = "Hello";
    char* string2 = "World";

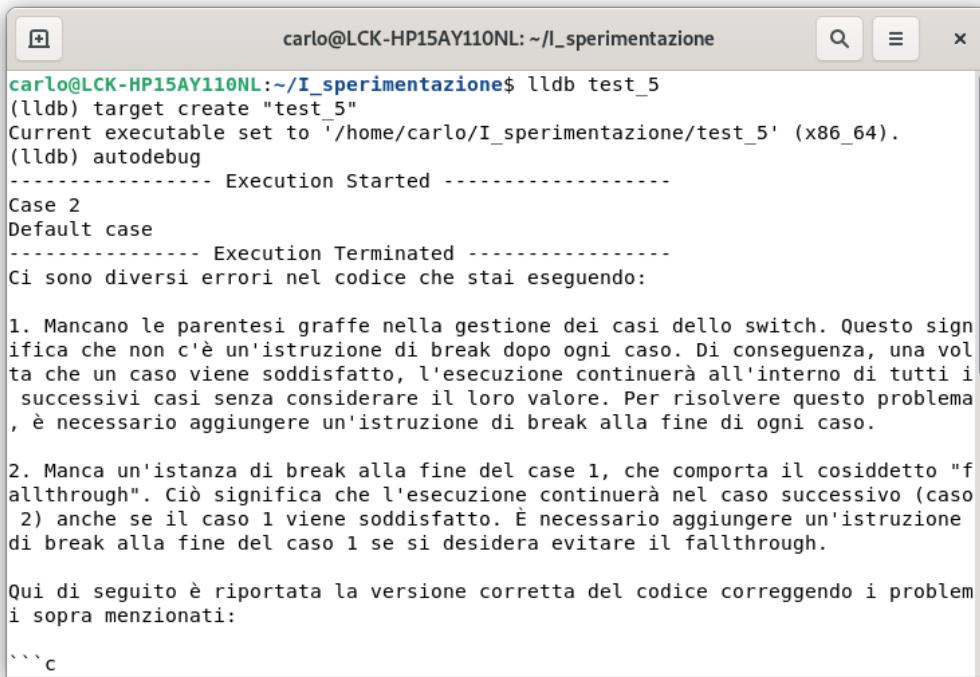
    if (strcmp(string1, string2) == 0)
        printf("%s and %s are the same.\n", string1, string2);
    else
        printf("%s and %s are different.\n", string1, string2);

    return 0;
}
```
Con questa modifica, l'output sarà corretto e dovrebbe mostrare che le due stringhe sono diverse.
(lldb) █

```

(b) Debugging di test\_4.c (continuo)

**Figura 4.4.** Debugging di test\_4.c



```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_5
(lldb) target create "test_5"
Current executable set to '/home/carlo/I_sperimentazione/test_5' (x86_64).
(lldb) autodebug
----- Execution Started -----
Case 2
Default case
----- Execution Terminated -----
Ci sono diversi errori nel codice che stai eseguendo:

1. Mancano le parentesi graffe nella gestione dei casi dello switch. Questo significa che non c'è un'istruzione di break dopo ogni caso. Di conseguenza, una volta che un caso viene soddisfatto, l'esecuzione continuerà all'interno di tutti i successivi casi senza considerare il loro valore. Per risolvere questo problema, è necessario aggiungere un'istruzione di break alla fine di ogni caso.

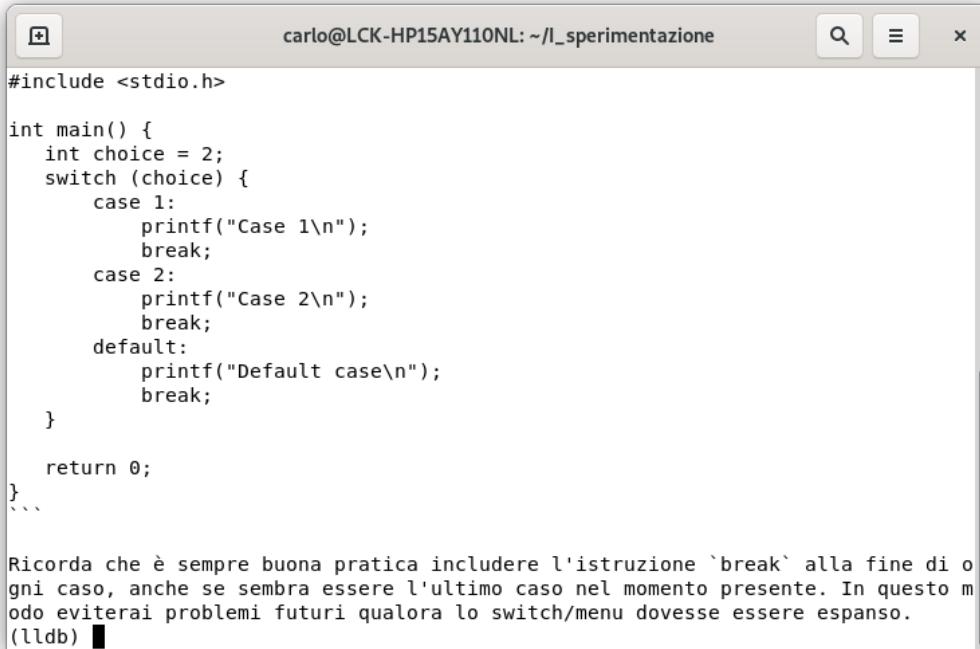
2. Manca un'istanza di break alla fine del case 1, che comporta il cosiddetto "fallthrough". Ciò significa che l'esecuzione continuerà nel caso successivo (caso 2) anche se il caso 1 viene soddisfatto. È necessario aggiungere un'istruzione di break alla fine del caso 1 se si desidera evitare il fallthrough.

Qui di seguito è riportata la versione corretta del codice correggendo i problemi sopra menzionati:

```c

```

(a) Debugging di test\_5.c



```

#include <stdio.h>

int main() {
    int choice = 2;
    switch (choice) {
        case 1:
            printf("Case 1\n");
            break;
        case 2:
            printf("Case 2\n");
            break;
        default:
            printf("Default case\n");
            break;
    }

    return 0;
}

Ricorda che è sempre buona pratica includere l'istruzione `break` alla fine di ogni caso, anche se sembra essere l'ultimo caso nel momento presente. In questo modo eviterai problemi futuri qualora lo switch/menu dovesse essere espanso.
(lldb) █

```

(b) Debugging di test\_5.c (continuo)

**Figura 4.5.** Debugging di test\_5.c

### 4.3.2 Buffer overflow

#### test\_6.c

L'errore nel programma si verifica quando una stringa con un certo numero di caratteri viene copiata in un array di caratteri di dimensione inferiore. Il buffer overflow causa la sovrascrittura dati delle altre variabili nello stack. Nel risponso di ChatGPT l'errore viene identificato correttamente sebbene vi sia un'osservazione sulle variabili inizializzate erronea. Vengono proposte le correzioni.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char name[10];
6     int var1 = 1, var2 = 2;
7
8     printf("var1 = %d; var2 = %d\n", var1, var2);
9
10    puts("Where do I live? See..."); 
11    strcpy(name, "...Roccasecca dei Volsci");
12
13    printf("<%s> is length %d\n", name, strlen(name));
14    printf("var1 = %d; var2 = %d\n", var1, var2);
15
16    return 0;
17 }
```

**Listing 4.6.** i\_testing/programs/test\_6.c

#### test\_7.c

L'errore consiste nel dichiarare un array di caratteri di dimensione inferiore ai caratteri immessi nello stesso, infatti non viene lasciato spazio per il terminatore nullo. Nel risponso di ChatGPT l'errore viene identificato correttamente. Vi è un'osservazione errata sull'inclusione degli header file. Vengono proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     char string[5] = "Hello";
5
6     return 0;
7 }
```

**Listing 4.7.** i\_testing/programs/test\_7.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_6
(lldb) target create "test_6"
Current executable set to '/home/carlo/I_sperimentazione/test_6' (x86_64).
(lldb) autodebug
----- Execution Started -----
var1 = 1; var2 = 2
Where do I live? See...
<...Roccasecca dei Volsci> is length 24
var1 = 543778148; var2 = 543253347
----- Execution Terminated -----
Sulla base dell'output fornito, sono presenti alcune problematiche nel codice:

1. Overflow del buffer: La riga 11 `strcpy(name, "...Roccasecca dei Volsci");` provoca un overflow del buffer `name[10]` in quanto la stringa da copiare ha una lunghezza di 24 caratteri. Questo comportamento è imprevedibile e può causare errori nel programma.

2. Valori non inizializzati: La riga 14 `printf("var1 = %d; var2 = %d\n", var1, var2);` mostra dei valori sorprendenti per le variabili `var1` e `var2`, che non sono stati inizializzati esplicitamente. Questo potrebbe essere un comportamento indesiderato.

3. Dimensione del buffer: Il buffer `name` è dichiarato con una dimensione di 10 caratteri, ma la stringa da copiare ha una lunghezza maggiore. È necessario aumentare la dimensione del buffer `name` per evitare l'overflow.

Per risolvere questi errori, è necessario:

1. Aumentare la dimensione del buffer `name` in modo che sia sufficiente per contenere la stringa da copiare.
2. Inizializzare le variabili `var1` e `var2` con valori desiderati prima di utilizzarle.

```

(a) Debugging di test\_6.c

```

3. Utilizzare una funzione di copia della stringa che impedisca l'overflow del buffer, come ad esempio `strncpy` con la dimensione adeguata del buffer.

Ecco un possibile codice corretto:

```c
#include <stdio.h>
#include <string.h>

int main() {
    char name[25]; // Aumentata la dimensione del buffer

    int var1 = 1, var2 = 2;

    printf("var1 = %d; var2 = %d\n", var1, var2);

    puts("Where do I live? See...");

    strncpy(name, "...Roccasecca dei Volsci", sizeof(name) - 1); // Utilizzata strncpy per evitare l'overflow
    name[sizeof(name) - 1] = '\0'; // Aggiunto il terminatore di stringa

    printf("<%s> is length %zu\n", name, strlen(name)); // Aggiunto il correttore formato %zu per mostrare la dimensione di tipo size_t
    printf("var1 = %d; var2 = %d\n", var1, var2);

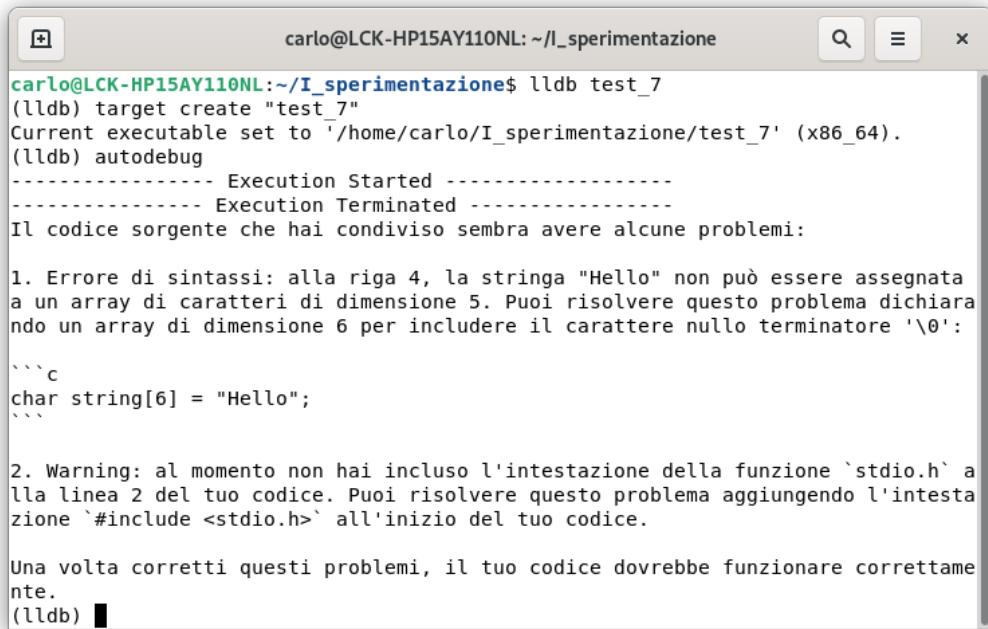
    return 0;
}
```

Spero che questa correzione ti sia d'aiuto.
(lldb)

```

(b) Debugging di test\_6.c (continuo)

**Figura 4.6.** Debugging di test\_6.c



```
carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_7
(lldb) target create "test_7"
Current executable set to '/home/carlo/I_sperimentazione/test_7' (x86_64).
(lldb) autodebug
----- Execution Started -----
----- Execution Terminated -----
Il codice sorgente che hai condiviso sembra avere alcune problemi:

1. Errore di sintassi: alla riga 4, la stringa "Hello" non può essere assegnata a un array di caratteri di dimensione 5. Puoi risolvere questo problema dichiarando un array di dimensione 6 per includere il carattere nullo terminatore '\0':

```c
char string[6] = "Hello";
```

2. Warning: al momento non hai incluso l'intestazione della funzione `stdio.h` alla linea 2 del tuo codice. Puoi risolvere questo problema aggiungendo l'intestazione `#include <stdio.h>` all'inizio del tuo codice.

Una volta corretti questi problemi, il tuo codice dovrebbe funzionare correttamente.
(lldb) █
```

**Figura 4.7.** Debugging di test\_7.c

### 4.3.3 Divisione per zero

#### test\_8.c

L'errore sta nella divisione per zero. Il denominatore, infatti, seppure non è evidente, è inizializzato a 0. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono suggeriti degli accorgimenti.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int a = 1;
6     int b = 0;
7     int c = log(pow(sqrt(a), b) - sin(b));
8
9     int result = a / c;
10    printf("Result: %d\n", result);
11
12    return 0;
13 }
```

Listing 4.8. i\_testing/programs/test\_8.c

carlo@LCK-HP15AY110NL:~/I\_sperimentazione\$ lldb test\_8  
(lldb) target create "test\_8"  
Current executable set to '/home/carlo/I\_sperimentazione/test\_8' (x86\_64).  
(lldb) autodebug  
----- Execution Started -----  
----- Execution Terminated -----  
L'errore "Floating point exception" indica un errore in virgola mobile durante l'esecuzione del programma. Questo può essere dovuto a diverse cause nel codice fornito.

1. Errore di divisione per zero: Il calcolo di 'c' nella linea 7 coinvolge la sottrazione di sin(b) da pow(sqrt(a), b). Tuttavia, se il risultato di questa espressione è zero, si avrà un errore di divisione per zero nella linea 9. Questo può accadere se 'a' o 'b' sono valori non validi che producono un risultato nullo nella formula. Puoi aggiungere un controllo per evitare la divisione per zero e gestire i casi in cui 'a' o 'b' potrebbero assumere valori che portano a errori.

2. Errore di valori non validi per funzioni matematiche: Alcune funzioni matematiche come log, pow, sqrt e sin potrebbero generare errori quando ricevono valori non validi come argomenti. Ad esempio, se provi a calcolare log di un valore ne

(a) Debugging di test\_8.c

carlo@LCK-HP15AY110NL:~/I\_sperimentazione\$ gativo o a calcolare la radice quadrata di un valore negativo, potrebbe verificarsi un errore. Assicurati che gli argomenti passati a queste funzioni siano validi e che soddisfino le regole matematiche.

3. Potenziale overflow o underflow: Non sono evidenti potenziali problemi di overflow o underflow nella porzione di codice fornita. Tuttavia, è sempre buona pratica controllare che i valori assegnati alle variabili rimangano all'interno dei limiti consentiti dal tipo di dato. Ad esempio, controlla che 'a' e 'b' non abbiano valori troppo grandi o troppo piccoli che possano causare overflow o underflow nei calcoli.

Ti suggerirei di controllare attentamente i valori delle variabili 'a' e 'b' prima di eseguire i calcoli e di implementare un controllo per evitare la divisione per zero. Inoltre, assicurati che i valori passati alle funzioni matematiche siano validi per evitare errori in virgola mobile.

Se hai ulteriori dubbi o necessiti di ulteriori spiegazioni, sarò lieto di assisterti.  
(lldb) █

(b) Debugging di test\_8.c (continuo)

Figura 4.8. Debugging di test\_8.c

#### 4.3.4 Accesso a memoria non allocata o non valida

##### test\_9.c

L'errore principale è l'accesso ad una zona non allocata dell'array e quindi a memoria non allocata, infatti si sta tentando di accedere all'undicesima posizione sebbene l'array sia stato allocato dinamicamente per averne tutt'al più dieci. Un'altro errore è la differenziazione del puntatore di un array dopo averlo deallocated e quindi di un puntatore nullo. Nel risponso di ChatGPT gli errori vengono identificati correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int *array = (int *)malloc(10 * sizeof(int));
6     array[11] = 42;
7     printf("%d", array[11]);
8
9     free(array);
10    *array = 42;
11
12    return 0;
13 }
```

**Listing 4.9.** i\_testing/programs/test\_9.c

##### test\_10.c

L'errore qui è nella doppia deallocazione di memoria. Rientra nell'accesso a memoria non allocata in quanto per deallocate la memoria la seconda volta si tenta di accedere ad un'area di memoria già deallocated. Nel risponso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int *array;
6
7     puts("Malloc an array ...");
8     array = malloc(sizeof(int) * 5);
9
10    if (array) {
11        puts("Malloc succeeded");
12        puts("Free the array... ");
13        free(array);
14    }
15
16    puts("Free the array... ");
17    free(array);
18
19 }
```

**Listing 4.10.** i\_testing/programs/test\_10.c

```
carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_9
(lldb) target create "test_9"
Current executable set to '/home/carlo/I_sperimentazione/test_9' (x86_64).
(lldb) autodebug
----- Execution Started -----
malloc(): corrupted top size
Process 83355 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Gli errori nel tuo codice possono essere suddivisi in due categorie: errori di accesso alla memoria e errori di utilizzo di memoria deallocated.

Errori di accesso alla memoria:

1. Alla riga 6, stai accedendo ad un'area di memoria oltre i limiti dell'array allocato. L'indice massimo consentito per l'array di dimensione 10 è 9. Questo provoca un errore di accesso alla memoria.

Errori di utilizzo di memoria deallocated:

1. Alla riga 9, stai cercando di assegnare un valore all'indirizzo di memoria puntato da `array` dopo averlo liberato con `free(array)`. Questo provoca un error
```

(a) Debugging di test\_9.c

```
e di utilizzo di memoria deallocated.

Questi errori combinati causano un comportamento indefinito del programma, che potrebbe causare il fallimento dell'allocazione di memoria (come nel tuo caso) o comportamenti imprevisti.

Per risolvere questi errori, puoi seguire le correzioni che ho suggerito nel mio precedente messaggio:

1. Cambia la dimensione dell'array alla riga 5 da `10` a `12` per includere l'elemento con indice 11.

2. Rimuovi l'assegnazione all'elemento con indice 11 alla riga 6.

3. Rimuovi l'assegnazione al puntatore `array` alla riga 10, in quanto non è più valido dopo la chiamata a `free(array)`.

Queste correzioni dovrebbero risolvere gli errori nel tuo codice e consentirti di eseguirlo correttamente senza errori.
(llldb) █
```

(b) Debugging di test\_9.c (continuo)

Figura 4.9. Debugging di test\_9.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_10
(lldb) target create "test_10"
Current executable set to '/home/carlo/I_sperimentazione/test_10' (x86_64).
(lldb) autodebug
----- Execution Started -----
Malloc an array ...
Malloc succeeded
Free the array...
Free the array...
free(): double free detected in tcache 2
Process 84588 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Gli errori nel codice sono:

1. Double Free Error: Si verifica quando un'area di memoria viene liberata due volte. Nel tuo caso, hai chiamato la funzione `free(array)` due volte: una volta all'interno dell'if statement (linea 13) e una volta alla linea 17. La seconda chiamata a `free(array)` causa un errore di "double free".

Puoi risolvere questo errore rimuovendo la seconda chiamata a `free(array)` alla linea 17.

Differenziali per tipologia:
- 1 Errore di Double Free.
(lldb) 

```

Figura 4.10. Debugging di test\_10.c

### test\_11.c

L'errore si manifesta nel momento in cui viene passato il puntatore “myval” alla funzione “do\_something\_with(myval)” e poi si tenta di accedervi. Infatti “myval” non è stato inizializzato correttamente dalla funzione “initialize()” poiché questa ritorna un puntatore locale che cessa di esistere non appena la funzione ritorna, pertanto “myval” punterà ad un'area di memoria non valida. Quindi ciò che succede dopo con “do\_something\_with(myval)” costituisce accesso ad un'area di memoria non valida. Nel risponso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2
3 char *initialize() {
4     char string[80];
5     char* ptr = string;
6
7     return ptr;
8 }
9
10 void do_something_with(char *myval){
11 }
12
13 int main() {
14     char *myval = initialize();
15     do_something_with(myval);
16     printf("myval: %s\n", myval);
17
18     return 0;
19 }

```

Listing 4.11. i\_testing/programs/test\_11.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_11
(lldb) target create "test_11"
Current executable set to '/home/carlo/I_sperimentazione/test_11' (x86_64).
(lldb) autodebug
----- Execution Started -----
myval:
----- Execution Terminated -----
Alcuni errori nel codice sono:

1. Variabile locale restituita: Nella funzione `initialize()`, stai restituendo un puntatore a una variabile locale `string` che viene allocata nello stack. Questo è un comportamento indefinito e potrebbe causare problemi. È meglio allocare `string` dinamicamente utilizzando `malloc` o dichiarare la variabile come statica.

2. Utilizzo errato del puntatore `ptr`: Nella funzione `initialize()`, stai restituendo un puntatore `ptr` che viene inizializzato con l'indirizzo della variabile locale `string`. Tuttavia, dopo la chiamata di `initialize()`, `ptr` diventa un puntatore non valido poiché `string` viene distrutta una volta che la funzione termina. È necessario allocare la memoria in modo dinamico in modo che sia valida anche al di fuori della funzione.

3. Funzione `do_something_with()`: Questa funzione è vuota e non sta facendo alcuna operazione con il parametro `myval`. Se non hai intenzione di utilizzare il parametro, puoi dichiararlo come `(void)` per evitare avvertimenti del compilatore

```

(a) Debugging di test\_11.c

```

re.

4. Formato di stampa non coerente: Nella printf alla riga 16, stai cercando di stampare una stringa utilizzando `%s` come formato di stampa. Tuttavia, dato che `myval` è un puntatore, potresti ottenere un comportamento indefinito o una stampa di valori non validi. Dovresti utilizzare `%p` per stampare l'indirizzo di `myval`.

Suggerimenti per correggere questi errori:

1. Modifica la riga 4 in `char* string = malloc(sizeof(char) * 80);` per allocare `string` dinamicamente. Ricordati di liberare la memoria quando hai finito di utilizzarla.

2. Modifica la riga 7 in `return string;` per restituire il puntatore corretto.

3. Puoi rimuovere completamente la funzione `do_something_with()` se non è necessaria per il tuo programma.

4. Modifica la riga 16 in `printf("myval: %p\n", (void*)myval);` per stampare l'indirizzo di `myval`.

Assicurati di includere l'header `stdlib.h` per utilizzare la funzione `malloc` e `free` correttamente.
(lldb)

```

(b) Debugging di test\_11.c (continuo)

Figura 4.11. Debugging di test\_11.c

### 4.3.5 Puntatori nulli

#### test\_12.c

L'errore in questo programma è quello di deferenziare (ovvero accedere al valore memorizzato nella posizione di memoria a cui il puntatore fa riferimento) un puntatore precedentemente inizializzato a “NULL” e quindi che non punta a nessuna memoria valida. Si tratta di errore per l'utilizzo di puntatori nulli. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     int *ptr = NULL;
5     *ptr = 42;
6
7     return 0;
8 }
```

**Listing 4.12.** i\_testing/programs/test\_12.c

### 4.3.6 Overflow aritmetico

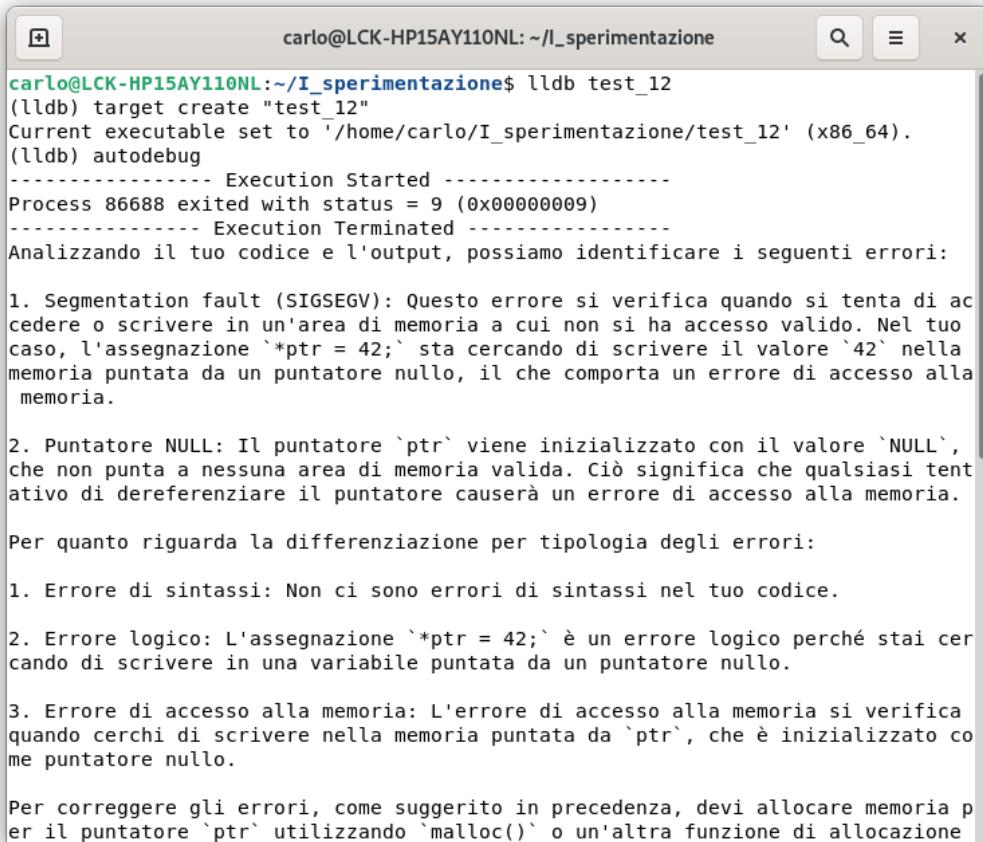
#### test\_13.c

Qui è presente un errore di overflow. Viene infatti effettuata una somma nella quale uno dei due termini è inizializzato al valore massimo per il tipo “int”. Precisamente si tratta di un errore di overflow aritmetico. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     int num = 2147483647;
5     int res = num + 1;
6
7     printf("Summing 1 to %d results in: %d\n", num, res);
8
9     return 0;
10 }
```

**Listing 4.13.** i\_testing/programs/test\_13.c



```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_12
(lldb) target create "test_12"
Current executable set to '/home/carlo/I_sperimentazione/test_12' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 86688 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Analizzando il tuo codice e l'output, possiamo identificare i seguenti errori:

1. Segmentation fault (SIGSEGV): Questo errore si verifica quando si tenta di accedere o scrivere in un'area di memoria a cui non si ha accesso valido. Nel tuo caso, l'assegnazione `*ptr = 42;` sta cercando di scrivere il valore `42` nella memoria puntata da un puntatore nullo, il che comporta un errore di accesso alla memoria.

2. Puntatore NULL: Il puntatore `ptr` viene inizializzato con il valore `NULL`, che non punta a nessuna area di memoria valida. Ciò significa che qualsiasi tentativo di dereferenziare il puntatore causerà un errore di accesso alla memoria.

Per quanto riguarda la differenziazione per tipologia degli errori:

1. Errore di sintassi: Non ci sono errori di sintassi nel tuo codice.

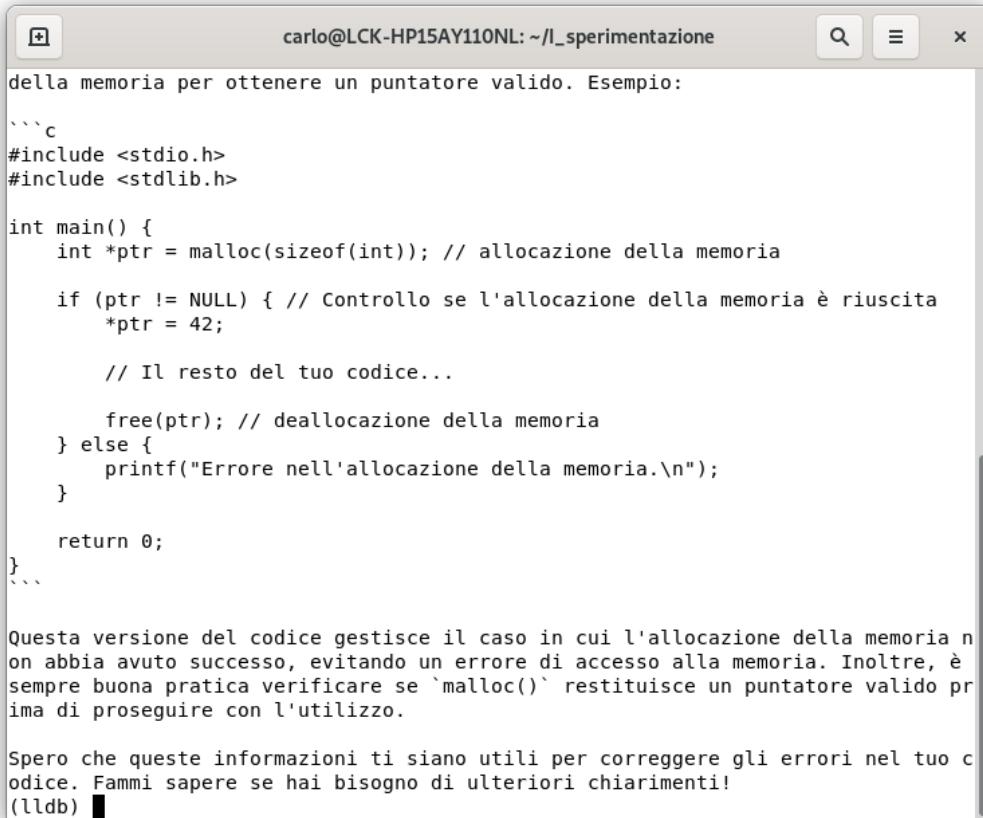
2. Errore logico: L'assegnazione `*ptr = 42;` è un errore logico perché stai cercando di scrivere in una variabile puntata da un puntatore nullo.

3. Errore di accesso alla memoria: L'errore di accesso alla memoria si verifica quando cerchi di scrivere nella memoria puntata da `ptr`, che è inizializzato come puntatore nullo.

Per correggere gli errori, come suggerito in precedenza, devi allocare memoria per il puntatore `ptr` utilizzando `malloc()` o un'altra funzione di allocazione

```

(a) Debugging di test\_12.c



```

carlo@LCK-HP15AY110NL:~/I_sperimentazione
della memoria per ottenere un puntatore valido. Esempio:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *ptr = malloc(sizeof(int)); // allocazione della memoria

    if (ptr != NULL) { // Controllo se l'allocazione della memoria è riuscita
        *ptr = 42;

        // Il resto del tuo codice...

        free(ptr); // deallocazione della memoria
    } else {
        printf("Errore nell'allocazione della memoria.\n");
    }

    return 0;
}
```

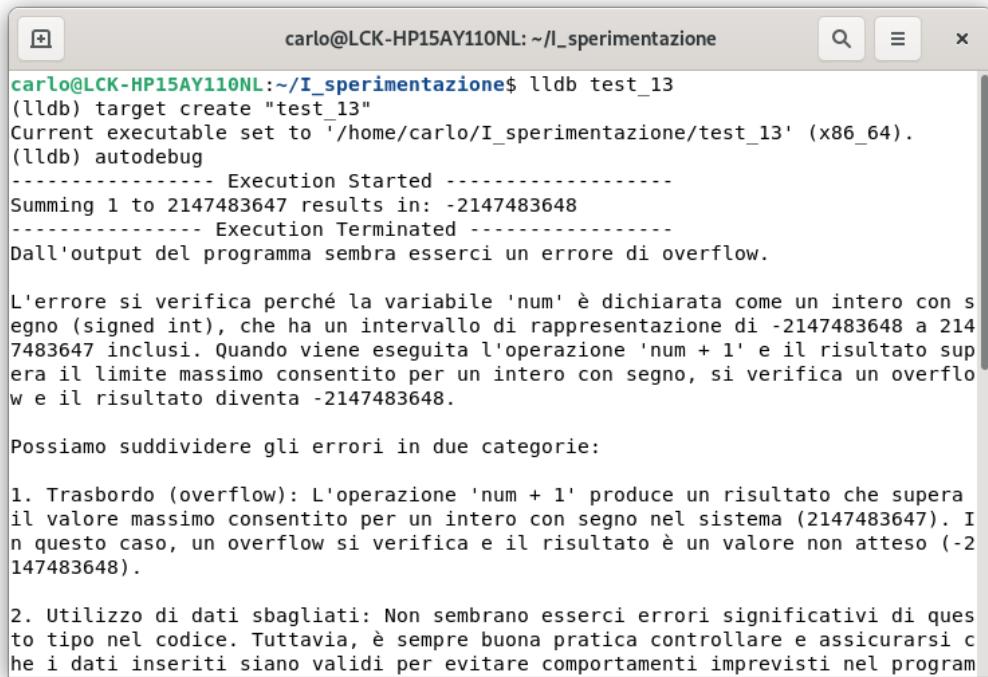
```

Questa versione del codice gestisce il caso in cui l'allocazione della memoria non abbia avuto successo, evitando un errore di accesso alla memoria. Inoltre, è sempre buona pratica verificare se `malloc()` restituisce un puntatore valido prima di proseguire con l'utilizzo.

Spero che queste informazioni ti siano utili per correggere gli errori nel tuo codice. Fatti sapere se hai bisogno di ulteriori chiarimenti!

(b) Debugging di test\_12.c (continuo)

**Figura 4.12.** Debugging di test\_12.c



```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_13
(lldb) target create "test_13"
Current executable set to '/home/carlo/I_sperimentazione/test_13' (x86_64).
(lldb) autodebug
----- Execution Started -----
Summing 1 to 2147483647 results in: -2147483648
----- Execution Terminated -----
Dall'output del programma sembra esserci un errore di overflow.

L'errore si verifica perché la variabile 'num' è dichiarata come un intero con segno (signed int), che ha un intervallo di rappresentazione di -2147483648 a 2147483647 inclusi. Quando viene eseguita l'operazione 'num + 1' e il risultato supera il limite massimo consentito per un intero con segno, si verifica un overflow e il risultato diventa -2147483648.

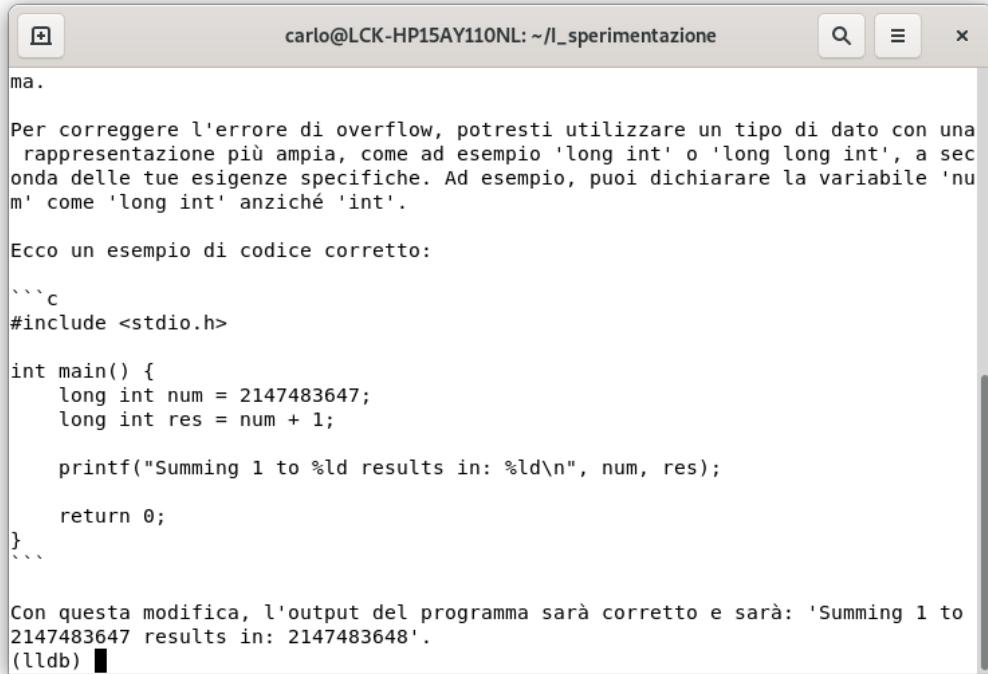
Possiamo suddividere gli errori in due categorie:

1. Trasbordo (overflow): L'operazione 'num + 1' produce un risultato che supera il valore massimo consentito per un intero con segno nel sistema (2147483647). In questo caso, un overflow si verifica e il risultato è un valore non atteso (-2147483648).

2. Utilizzo di dati sbagliati: Non sembrano esserci errori significativi di questo tipo nel codice. Tuttavia, è sempre buona pratica controllare e assicurarsi che i dati inseriti siano validi per evitare comportamenti imprevisti nel program

```

(a) Debugging di test\_13.c



```

ma.

Per correggere l'errore di overflow, potresti utilizzare un tipo di dato con una rappresentazione più ampia, come ad esempio 'long int' o 'long long int', a seconda delle tue esigenze specifiche. Ad esempio, puoi dichiarare la variabile 'num' come 'long int' anziché 'int'.

Ecco un esempio di codice corretto:

```c
#include <stdio.h>

int main() {
    long int num = 2147483647;
    long int res = num + 1;

    printf("Summing 1 to %ld results in: %ld\n", num, res);

    return 0;
}
```

Con questa modifica, l'output del programma sarà corretto e sarà: 'Summing 1 to 2147483647 results in: 2147483648'.
(lldb) █

```

(b) Debugging di test\_13.c (continuo)

Figura 4.13. Debugging di test\_13.c

### 4.3.7 Underflow aritmetico

#### test\_14.c

Qui è presente un errore di underflow. Viene infatti diviso per due un numero già molto piccolo che viene quindi approssimato a zero sebbene zero non sia. Precisamente si tratta di un errore di underflow aritmetico. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     float num = 1.0e-45;
5     float res = num / 2;
6
7     printf("Dividing %e by 2 results in: %e\n", num, res);
8
9     return 0;
10 }
```

**Listing 4.14.** i\_testing/programs/test\_14.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_14
(lldb) target create "test_14"
Current executable set to '/home/carlo/I_sperimentazione/test_14' (x86_64).
(lldb) autodebug
----- Execution Started -----
Dividing 1.401298e-45 by 2 results in: 0.000000e+00
----- Execution Terminated -----
Dai risultati che hai fornito, sembra che il codice non abbia errori di compilazione o di esecuzione deterministicici. Tuttavia, il risultato ottenuto potrebbe essere inaspettato a causa dell'uso di tipi di dato floating point.

Alcuni punti da notare:
1. La variabile `num` viene inizializzata con il valore `1.0e-45`. Questa è una piccola quantità e potrebbe essere rappresentata come un valore denormalizzato o zero a causa delle limitazioni del calcolo in virgola mobile.
2. La variabile `res` viene assegnata al risultato di `num / 2`. Dal momento che `num` è un valore molto piccolo, il risultato potrebbe essere zero se la precisione non è sufficiente.
3. La printf formatta e stampa i valori di `num` e `res` utilizzando il formato `%e`, che rappresenta i numeri in notazione scientifica. In questo caso, entrambi i valori vengono stampati come zero in notazione scientifica (`0.000000e+00`).

Quindi, non ci sono errori specifici del codice, ma il risultato ottenuto potrebbe essere diverso da quello atteso a causa della rappresentazione in virgola mobile e della precisione limitata.

Se desideri ottenere un risultato più preciso, potresti considerare l'utilizzo di tipi di dato con maggiore precisione, come `double` o `long double`, al posto di `float`. Inoltre, potresti voler prendere in considerazione l'utilizzo di funzioni o librerie specifiche per il calcolo in virgola mobile per ottenere risultati più accurati.
(lldb) 
```

**Figura 4.14.** Debugging di test\_14.c

### 4.3.8 Ricorsione infinita

#### test\_15.c

In questo caso l'errore sta nel non aver gestito il caso nel quale per il calcolo del fattoriale sia fornito un numero negativo. Se viene fornito un numero negativo alla funzione “factorial(n)”, infatti, si genera una ricorsione infinita dato che non viene mai raggiunto il caso base. Si ha quindi a che fare con un errore di ricorsione infinita. Nel responso di ChatGPT l'errore viene identificato e correttamente sebbene sia riportato che il caso base non è gestito correttamente per “n==0” il che è falso. Vengono proposte le correzioni.

```

1 #include <stdio.h>
2
3 int factorial(int n) {
4     if (n == 0) {
5         return 1;
6     } else {
7         return n * factorial(n - 1);
8     }
9 }
10
11 int main() {
12     int n = -5;
13     int result = factorial(n);
14     printf("Factorial of %d is %d\n", n, result);
15     return 0;
16 }
```

**Listing 4.15.** i\_testing/programs/test\_15.c

### 4.3.9 Accesso a variabili non inizializzate

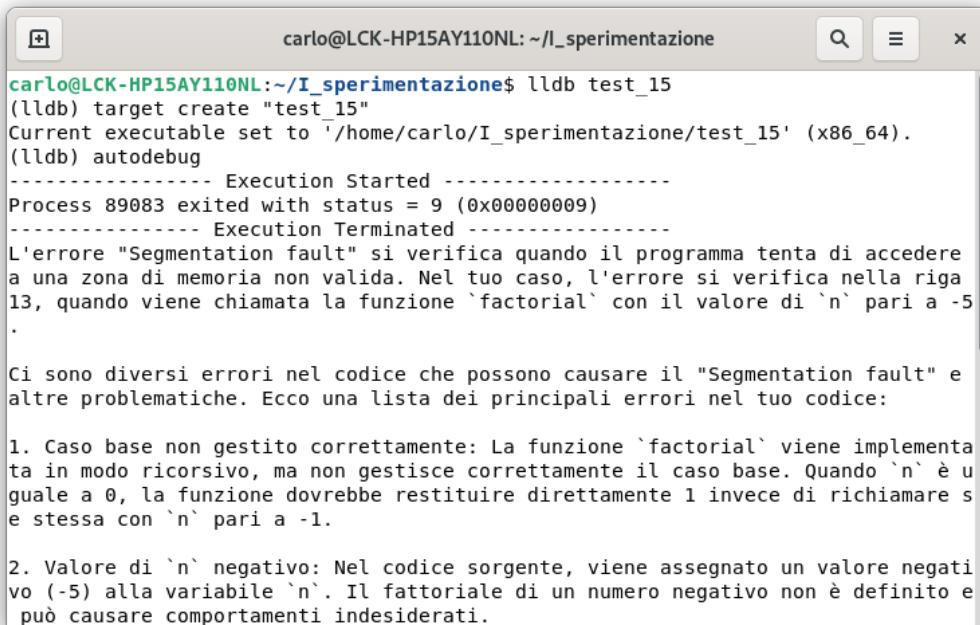
#### test\_16.c

In questo caso si ha una doppia dichiarazione della variabile “i”. Una all'inizio del main e una all'interno del ciclo for. Sono due variabili indipendenti con scope (ambito) limitato alle rispettive parti di codice in cui sono dichiarate. Pertanto quando nell'ultima stampa si stamperà la “i”, questa avrà un valore casuale, poiché non è stata inizializzata al contrario della “i” locale al ciclo “for”. Si tratta quindi di un accesso ad una variabile non inizializzata. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     int mming = 0;
5     int i;
6
7     for (int i = 1; i <= 5; i++) {
8         mming += i;
9     }
10    printf("Sum is %d\n", mming);
11    printf("i after the loop is %d\n", i);
12    return 0;
13 }
```

**Listing 4.16.** i\_testing/programs/test\_16.c



```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_15
(lldb) target create "test_15"
Current executable set to '/home/carlo/I_sperimentazione/test_15' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 89083 exited with status = 9 (0x00000009)
----- Execution Terminated -----
L'errore "Segmentation fault" si verifica quando il programma tenta di accedere
a una zona di memoria non valida. Nel tuo caso, l'errore si verifica nella riga
13, quando viene chiamata la funzione `factorial` con il valore di `n` pari a -5
.

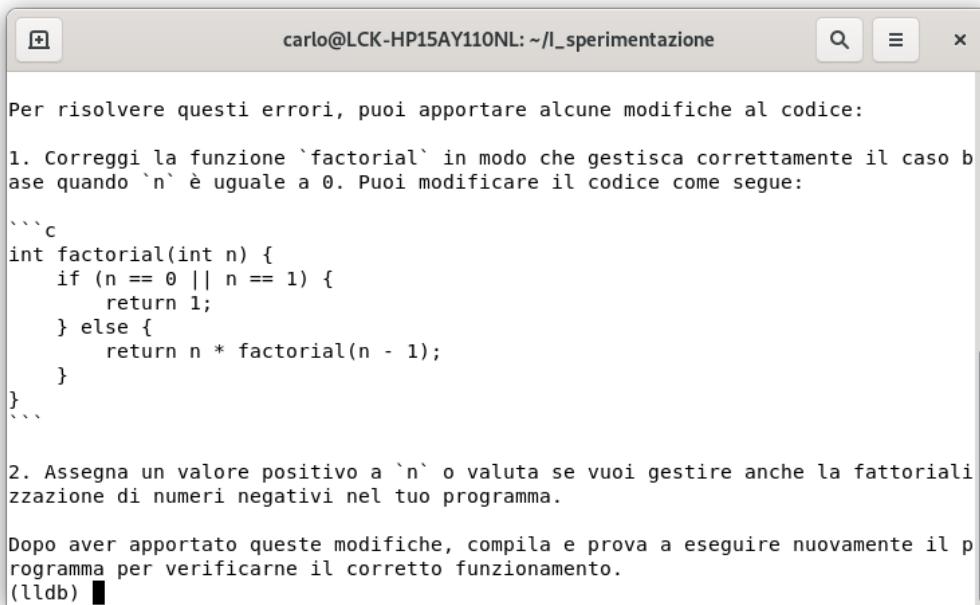
Ci sono diversi errori nel codice che possono causare il "Segmentation fault" e
altre problematiche. Ecco una lista dei principali errori nel tuo codice:

1. Caso base non gestito correttamente: La funzione `factorial` viene implementata
in modo ricorsivo, ma non gestisce correttamente il caso base. Quando `n` è uguale a 0,
la funzione dovrebbe restituire direttamente 1 invece di richiamare se stessa con `n` pari a -1.

2. Valore di `n` negativo: Nel codice sorgente, viene assegnato un valore negativo (-5)
alla variabile `n`. Il fattoriale di un numero negativo non è definito e
può causare comportamenti indesiderati.

```

(a) Debugging di test\_15.c



```

Per risolvere questi errori, puoi apportare alcune modifiche al codice:

1. Correggi la funzione `factorial` in modo che gestisca correttamente il caso base quando `n` è uguale a 0. Puoi modificare il codice come segue:

```c
int factorial(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
```

2. Assegna un valore positivo a `n` o valuta se vuoi gestire anche la fattorializzazione di numeri negativi nel tuo programma.

Dopo aver apportato queste modifiche, compila e prova a eseguire nuovamente il programma per verificarne il corretto funzionamento.
(lldb) █

```

(b) Debugging di test\_15.c (continuo)

**Figura 4.15.** Debugging di test\_15.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_16
(lldb) target create "test_16"
Current executable set to '/home/carlo/I_sperimentazione/test_16' (x86_64).
(lldb) autodebug
----- Execution Started -----
Sum is 15
'i' after the loop is 32767
----- Execution Terminated -----
Dall'output che hai fornito, possiamo identificare due problemi nel codice:

1. All'inizio del tuo programma, hai dichiarato una variabile `i` alla riga 5, ma successivamente hai dichiarato un'altra variabile `i` all'interno del ciclo `for` alla riga 7. Questo causa un'ombra della variabile `i` dichiarata all'esterno del ciclo. Per risolvere questo problema, puoi rimuovere la dichiarazione di `int i;` alla riga 5.

2. Dopo il ciclo `for`, alla riga 11, stai tentando di stampare il valore di `i` , che è dichiarato solo all'interno del ciclo. Questo provoca un errore perché la variabile `i` è fuori dal suo ambito quando cerchi di utilizzarla. Puoi risolvere questo problema dichiarando la variabile `i` fuori dal ciclo `for` alla riga 6, anziché all'interno del ciclo.

Dopo aver apportato queste modifiche, il tuo codice corretto dovrebbe essere:

```

(a) Debugging di test\_16.c

```

```c
#include <stdio.h>

int main() {
    int mming = 0;
    int i;

    for (i = 1; i <= 5; i++) {
        mming += i;
    }
    printf("Sum is %d\n", mming);
    printf("i' after the loop is %d\n", i);

    return 0;
}
```

Con queste modifiche, il programma dovrebbe funzionare correttamente e l'output dovrebbe essere: `Sum is 15 i' after the loop is 6`. Spero che questa risposta sia stata utile! Fammi sapere se hai altre domande.
(lldb) █

```

(b) Debugging di test\_16.c (continuo)

**Figura 4.16.** Debugging di test\_16.c

**test\_17.c**

In questo caso sono stati dichiarati ma non inizializzati gli elementi in gioco. Questo vale sia per le variabili, sia per l'array allocato staticamente che per quello allocato dinamicamente. Pertanto si avranno dei valori casuali. Si tratta di un errore di accesso a variabili non inizializzate. Nel responso di ChatGPT l'errore viene identificato correttamente sebbene venga considerata mancante la deallocazione della memoria il che è falso. Vengono proposte le correzioni.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     int i, j, k;
6     int numbers[5];
7     int *array;
8
9     puts("Printing variables:");
10    printf(" i = %d\n", i);
11    printf(" j = %d\n", j);
12    printf(" k = %d\n", k);
13
14    puts("Printing an array:");
15    for (i = 0; i < 5; i++) {
16        printf(" numbers[%d] = %d\n", i, numbers[i]);
17    }
18
19    array = malloc(sizeof(int) * 5);
20    if (array) {
21        puts("Printing malloc'ed array:");
22        for (i = 0; i < 5; i++) {
23            printf(" array[%d] = %d\n", i, array[i]);
24        }
25        free(array);
26    }
27
28    return 0;
29 }
```

**Listing 4.17.** i\_testing/programs/test\_17.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_17
(lldb) target create "test_17"
Current executable set to '/home/carlo/I_sperimentazione/test_17' (x86_64).
(lldb) autodebug
----- Execution Started -----
Printing variables:
  i = 0
  j = 0
  k = 32765
Printing an array:
  numbers[0] = 0
  numbers[1] = 0
  numbers[2] = 0
  numbers[3] = 0
  numbers[4] = -1377004944
Printing malloc'ed array:
  array[0] = 0
  array[1] = 0
  array[2] = 0
  array[3] = 0
  array[4] = 0
----- Execution Terminated -----
Dal tuo output, sembra che ci siano alcune variabili non inizializzate e alcuni errori nella gestione dell'allocazione e della deallocazione della memoria dinamica. Ecco una lista degli errori rilevati:

1. Variabili non inizializzate: le variabili intere `i`, `j` e `k` vengono utilizzate prima di essere inizializzate. Questo comporta letture di valori non definiti, che vengono visualizzati come numeri casuali. Per correggere questo, è nece

```

(a) Debugging di test\_17.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ 
ssario assegnare dei valori iniziali a queste variabili.

2. Accesso ad array non inizializzato: l'array `numbers` viene utilizzato senza essere inizializzato, quindi i suoi elementi contengono valori non definiti. Per evitare questo errore, è necessario assegnare dei valori iniziali agli elementi dell'array.

3. Errore nella gestione della memoria dinamica: l'array `array` viene allocato dinamicamente utilizzando la funzione `malloc`, ma non viene inizializzato. Questo significa che i suoi elementi contengono valori non definiti. Inoltre, l'array non viene mai de-allocato utilizzando la funzione `free`. Per correggere questi errori, è necessario allocare e inizializzare correttamente l'array e poi deallocare la memoria utilizzando la funzione `free` quando non è più necessario.

Per risolvere questi errori, puoi apportare le seguenti modifiche al codice:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int i, j, k;
    int numbers[5] = {0}; // Inizializza l'array numbers con zeri

    int *array;

    i = j = k = 0; // Inizializza le variabili i, j, k con zero
    puts("Printing variables:");
}
```

```

(b) Debugging di test\_17.c (continuo)

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ 
printf("  i = %d\n", i);
printf("  j = %d\n", j);
printf("  k = %d\n", k);

puts("Printing an array:");
for (i = 0; i < 5; i++) {
    printf("  numbers[%d] = %d\n", i, numbers[i]);
}

array = malloc(sizeof(int) * 5);
if (array) {
    for (i = 0; i < 5; i++) {
        array[i] = 0; // Inizializza gli elementi dell'array con zeri
    }
}

puts("Printing malloc'ed array:");
for (i = 0; i < 5; i++) {
    printf("  array[%d] = %d\n", i, array[i]);
}

free(array);
}

return 0;
}..
```
Questa versione del codice corregge gli errori menzionati sopra e dovrebbe produrre l'output corretto.
(lldb) 

```

(c) Debugging di test\_17.c (continuo)

Figura 4.17. Debugging di test\_17.c

**test\_18.c**

Qui non si è inizializzata la variabile “currmin” con un valore dell’array, pertanto, avendo in sé un valore casuale ciò crea poi problemi nel momento in cui in “currmin” si ha un valore più piccolo dei valori presenti nell’array. Si tratta di un accesso ad una variabile non inizializzata. Nel responso di ChatGPT l’errore viene identificato e correttamente. Vengono anche proposte le correzioni.

```

1 #include <stdio.h>
2
3 int minval(int*, int);
4
5 int main() {
6     int A[] = {1, 1, 1, 1, 1};
7     int n = sizeof(A) / sizeof(A[0]);
8
9     int minimum = minval(A, n);
10
11    return 0;
12 }
13
14 int minval(int *A, int n) {
15     int currmin;
16
17     for (int i=0; i<n; i++)
18         if (A[i] < currmin)
19             currmin = A[i];
20     return currmin;
21 }
```

**Listing 4.18.** i\_testing/programs/test\_18.c

**test\_19.c**

Qui non viene inizializzato il puntatore. Si verifica quindi un accesso ad una variabile non inizializzata. Nel responso di ChatGPT l’errore viene identificato correttamente sebbene contenga delle considerazioni errate sull’allocazione dinamica e successiva deallocazione. Vengono proposte le correzioni.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5
6     char* ptr;
7     strcpy(ptr, "hello world!");
8     printf("%s", ptr);
9
10    return 0;
11 }
```

**Listing 4.19.** i\_testing/programs/test\_19.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_18
(lldb) target create "test_18"
Current executable set to '/home/carlo/I_sperimentazione/test_18' (x86_64).
(lldb) autodebug
----- Execution Started -----
----- Execution Terminated -----
Dal codice fornito, ho individuato due errori:

1. Errore di inizializzazione della variabile `currmin` nella funzione `minval()`. La variabile `currmin` viene utilizzata prima di essere inizializzata. Per risolvere questo problema, è necessario inizializzare `currmin` con un valore iniziale valido, ad esempio con `currmin = A[0]` all'inizio della funzione.

2. Errore nell'operatore di confronto `<` nel ciclo `for` della funzione `minval()`. Attualmente, il confronto viene fatto tra `A[i]` e `currmin`, ma l'errore consiste nel fatto che `currmin` non è mai inizializzato e quindi potrebbe contenere un valore indeterminato. Invece, il confronto dovrebbe essere inizializzato con il primo elemento dell'array `A`, modificando la riga 15 come segue: `int currmin = A[0];`.

Questi errori possono causare risultati inattesi nel programma, in particolare nel calcolo del valore minimo dell'array `A`.
(lldb)

```

Figura 4.18. Debugging di test\_18.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_19
(lldb) target create "test_19"
Current executable set to '/home/carlo/I_sperimentazione/test_19' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 93178 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Ci sono diversi errori nel tuo codice:

1. Errore del puntatore non inizializzato: Il puntatore `ptr` viene dichiarato, ma non viene assegnato nessun indirizzo di memoria valido prima di essere utilizzato con `strcpy`. Questo porta a un errore di segmentazione perché stai cercando di copiare una stringa in un'area di memoria non valida.

2. Errore di accesso alla memoria non allocata: Poiché `ptr` non è stato inizializzato correttamente, stai cercando di copiare la stringa "hello world!" in un indirizzo di memoria casuale o non allocato, causando un errore di segmentazione.

3. Mancanza della liberazione della memoria: Hai utilizzato `malloc` per allocare memoria per `ptr`, ma non hai mai deallocato la memoria utilizzando `free`. Questo può causare una perdita di memoria nel tuo programma.

Per risolvere questi errori, puoi modificare il tuo codice come segue:
```c
#include <stdio.h>
#include <string.h>
```

```

(a) Debugging di test\_19.c

```

#include <stdlib.h>

int main() {
    char* ptr = malloc(strlen("hello world!") + 1); // Alloca memoria
    if (ptr == NULL) {
        fprintf(stderr, "Errore di allocazione della memoria.\n");
        return 1;
    }
    strcpy(ptr, "hello world!"); // Copia la stringa
    printf("%s", ptr);

    free(ptr); // Dealloca la memoria

    return 0;
}

In questo modo, il codice alloca la memoria per `ptr` utilizzando `malloc`, controlla se l'allocazione ha avuto successo, copia la stringa "hello world!" in `ptr`, stampa `ptr`, dealloca infine la memoria utilizzata.

Ricorda di includere `stdlib.h` per l'utilizzo delle funzioni `malloc` e `free`.

Queste modifiche dovrebbero risolvere gli errori nel tuo codice.+Fammi sapere se hai bisogno di ulteriori chiarimenti o assistenza.
(lldb)

```

(b) Debugging di test\_19.c (continuo)

Figura 4.19. Debugging di test\_19.c

### 4.3.10 Accesso fuori dai limiti dell'array

#### test\_20.c

Si dichiara e alloca staticamente un array di cinque elementi, ma si tenta poi di accedere allo stesso fuori dai limiti entro i quali è stato definito. Si tratta di un accesso fuori dai limiti dell'array. Nel responso di ChatGPT l'errore viene identificato correttamente sebbene sia fatta una considerazione errata sulle librerie. Vengono proposte le correzioni.

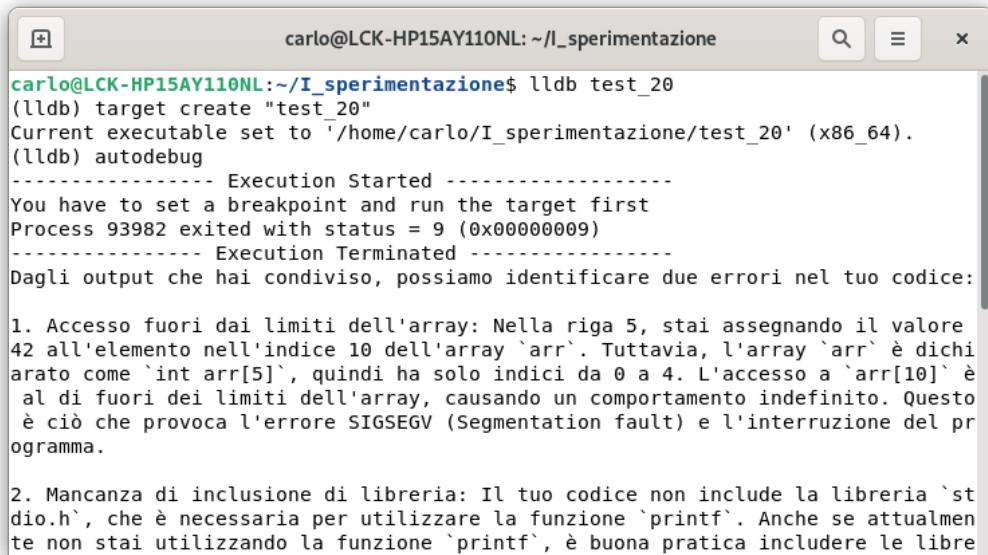
```
1 #include <stdio.h>
2 int main() {
3     int arr[5];
4     arr[10] = 42;
5     return 0;
6 }
```

**Listing 4.20.** i\_testing/programs/test\_20.c  
test\_21.c

In questo caso si itera lo stesso errore per ben due volte. Sebbene sia corretta l'assegnazione dei valori negli array, si sbaglia poi la gestione degli indici all'interno dei cicli “for” che si occupano della stampa degli elementi. Si parte quindi da un “off-by-one error” che causa poi un accesso fuori dai limiti dell'array. Nel responso di ChatGPT l'errore viene identificato correttamente, sebbene vengano identificati altri due errori che però non sussistono.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int i;
5     int numbers[5];
6     int *array;
7
8     puts("This array has five elements (0 to 4)");
9     for (i = 0; i < 5; i++) {
10         numbers[i] = i;
11     }
12     for (i = 0; i <= 5; i++) {
13         printf(" numbers[%d] = %d\n", i, numbers[i]);
14     }
15
16     puts("malloc an array ...");
17     array = malloc(sizeof(int) * 5);
18     if (array) {
19         puts("This malloc'ed array also has five elements (0 to 4)");
20         for (i = 0; i < 5; i++) {
21             array[i] = i;
22         }
23         for (i = 0; i < 10; i++) {
24             printf(" array[%d] = %d\n", i, array[i]);
25         }
26         free(array);
27     }
28     return 0;
29 }
```

**Listing 4.21.** i\_testing/programs/test\_21.c



```

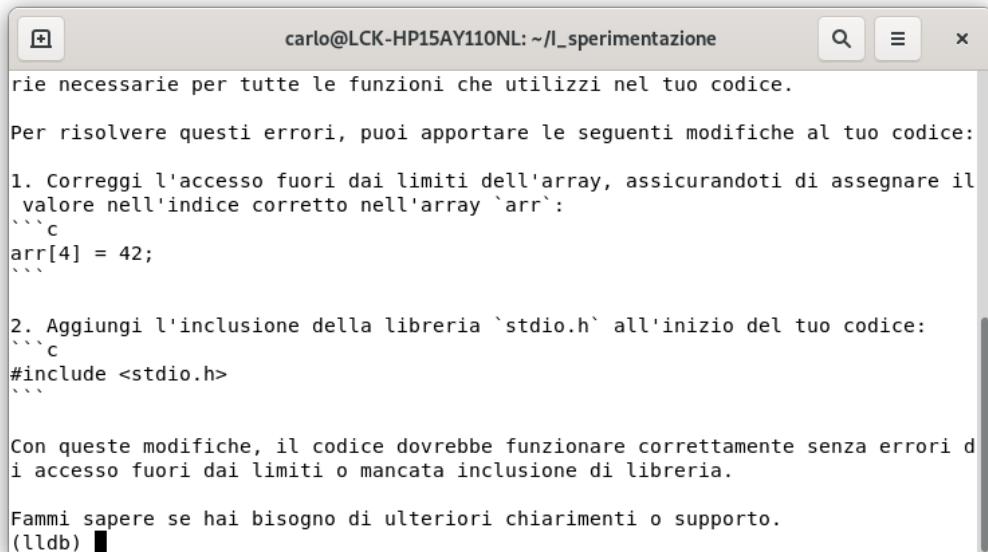
carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_20
(lldb) target create "test_20"
Current executable set to '/home/carlo/I_sperimentazione/test_20' (x86_64).
(lldb) autodebug
----- Execution Started -----
You have to set a breakpoint and run the target first
Process 93982 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Dagli output che hai condiviso, possiamo identificare due errori nel tuo codice:

1. Accesso fuori dai limiti dell'array: Nella riga 5, stai assegnando il valore
42 all'elemento nell'indice 10 dell'array `arr`. Tuttavia, l'array `arr` è dichiarato come `int arr[5]`, quindi ha solo indici da 0 a 4. L'accesso a `arr[10]` è
al di fuori dei limiti dell'array, causando un comportamento indefinito. Questo
è ciò che provoca l'errore SIGSEGV (Segmentation fault) e l'interruzione del pr
ogramma.

2. Mancanza di inclusione di libreria: Il tuo codice non include la libreria `st
dio.h`, che è necessaria per utilizzare la funzione `printf`. Anche se attualmen
te non stai utilizzando la funzione `printf`, è buona pratica includere le libre

```

(a) Debugging di test\_20.c



```

rie necessarie per tutte le funzioni che utilizzi nel tuo codice.

Per risolvere questi errori, puoi apportare le seguenti modifiche al tuo codice:

1. Correggi l'accesso fuori dai limiti dell'array, assicurandoti di assegnare il
valore nell'indice corretto nell'array `arr`:
```c
arr[4] = 42;
```

2. Aggiungi l'inclusione della libreria `stdio.h` all'inizio del tuo codice:
```c
#include <stdio.h>
```

Con queste modifiche, il codice dovrebbe funzionare correttamente senza errori d
i accesso fuori dai limiti o mancata inclusione di libreria.

Fammi sapere se hai bisogno di ulteriori chiarimenti o supporto.
(lldb) █

```

(b) Debugging di test\_20.c (continuo)

**Figura 4.20.** Debugging di test\_20.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_21
(lldb) target create "test_21"
Current executable set to '/home/carlo/I_sperimentazione/test_21' (x86_64).
(lldb) autodebug
----- Execution Started -----
This array has five elements (0 to 4)
numbers[0] = 0
numbers[1] = 1
numbers[2] = 2
numbers[3] = 3
numbers[4] = 4
numbers[5] = 21956
malloc an array ...
This malloc'ed array also has five elements (0 to 4)
array[0] = 0
array[1] = 1
array[2] = 2
array[3] = 3
array[4] = 4
array[5] = 0
array[6] = 133441
array[7] = 0
array[8] = 0
array[9] = 0
----- Execution Terminated -----
Ci sono diversi errori nel codice:

```

(a) Debugging di test\_21.c

```

1. Out-of-bounds access: Nella riga 14, il ciclo for dovrebbe iterare solo fino a `i < 5`, ma invece itera fino a `i <= 5`. Questo causa un accesso fuori dai limiti dell'array `numbers`, perché l'indice 5 non esiste. Questo errore provoca un comportamento non definito.

2. Out-of-bounds access dopo l'allocazione dinamica: Nella riga 27, il ciclo for dovrebbe iterare solo fino a `i < 5`, ma invece itera fino a `i < 10`. Questo causa un accesso fuori dai limiti dell'array `array`, perché l'array è stato allocato solo con dimensione 5. Questo anche provoca un comportamento non definito.

3. Mancato controllo della malloc: Nella riga 20, viene assegnato il risultato della malloc alla variabile `array`, ma non viene eseguito alcun controllo per verificare se la malloc ha avuto successo o meno. Se la malloc fallisce, restituirà NULL, e l'accesso a `array` nella riga 23 provocherà un errore.

4. Mancata deallocazione della memoria: Dopo l'utilizzo dell'array `array`, dovrà essere deallocated la memoria utilizzando `free(array)`. Questo non viene fatto nel codice.

5. Variabili non inizializzate: Le variabili `numbers` e `array` vengono dichiarate ma non vengono inizializzate. È una buona pratica inizializzare le variabili per evitare problemi derivanti da valori non definiti.

Quindi, gli errori nel codice sono principalmente di due tipi: accesso fuori dai limiti dell'array e gestione errata della memoria dinamica.

```

(b) Debugging di test\_21.c (continuo)

Figura 4.21. Debugging di test\_21.c

**test\_22.c**

Qui si dichiara un array di caratteri e lo si alloca staticamente affinché possa contenere 10 elementi. Gliene si assegnano 5 e poi si cerca di concatenare a questo array di caratteri una stringa di 6 caratteri. Ciò genera un errore per accesso fuori dai limiti dell'array poiché non si è tenuto in considerazione che la stringa "World" ha ben 6 caratteri, l'ultimo dei quali è il terminatore nullo "\0". Il terminatore nullo inoltre andrebbe aggiunto anche all'array di caratteri in quanto poi viene trattato come una stringa. Nel risponso di ChatGPT l'errore viene identificato correttamente sebbene vi sia una considerazione errata sull'inclusione dell'header file "string.h".

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main() {
5     char charArray[10] = {'H', 'e', 'l', 'l', 'o'};
6     printf("Character Array: %s\n", charArray);
7     char *string = "World";
8     printf("String: %s\n", string);
9
10    strcat(charArray, string);
11    printf("Modified Character Array: %s\n", charArray);
12
13    return 0;
14 }
```

**Listing 4.22.** i\_testing/programs/test\_22.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_22
(lldb) target create "test_22"
Current executable set to '/home/carlo/I_sperimentazione/test_22' (x86_64).
(lldb) autodebug
----- Execution Started -----
Character Array: Hello
String: World
Modified Character Array: HelloWorld
----- Execution Terminated -----
Dal codice fornito, posso identificare alcuni errori:

1. Errore nella dichiarazione e inizializzazione dell'array `charArray`: Nella riga 5, hai dichiarato un array di caratteri `charArray` di dimensione 10, ma hai inizializzato solo i primi 5 elementi con caratteri. Gli elementi rimanenti del l'array non sono inizializzati e quindi avranno valori indefiniti. Per evitare comportamenti imprevedibili, dovresti inizializzare completamente l'array o definire la dimensione correttamente per contenere tutti i caratteri necessari.

2. Utilizzo errato della funzione `strcat`: Nella riga 10, stai utilizzando `strcat` per concatenare `charArray` con `string`. Tuttavia, l'array `charArray` ha dimensione 10, quindi non è sufficientemente grande per contenere i caratteri extra di `string`. Questo può causare un buffer overflow e comportamenti indefiniti. Dovresti assicurarti che l'array di destinazione abbia abbastanza spazio per contenere la stringa concatenata.

3. Mancata inclusione dell'header per `strcat`: Non hai incluso l'header `string.h` (riga 2) prima di utilizzare la funzione `strcat`. Anche se il tuo codice potrebbe funzionare senza l'inclusione diretta dell'header in alcuni ambienti, è buona pratica includerlo esplicitamente per garantire la corretta dichiarazione dei prototipi di funzione.

Fammi sapere se hai bisogno di ulteriori chiarimenti o assistenza con la correzione dei problemi.
(lldb) 
```

**Figura 4.22.** Debugging di test\_22.c

### 4.3.11 File non trovato o non accessibile

#### test\_23.c

L'errore è causato dalla mancanza di gestione della situazione nella quale il file, che si apre in sola lettura, non è stato aperto correttamente. In questo caso specifico, infatti, non viene trovato alcun file e viene restituito un puntatore nullo il cui utilizzo porta poi ad un errore più generale di deferenziazione (accesso o utilizzo del valore cui il puntatore punta) di un puntatore nullo. Il bug è caratterizzabile come errore di file non trovato o non accessibile. Nel responso di ChatGPT l'errore viene identificato correttamente.

```

1 #include <stdio.h>
2
3 int main() {
4     FILE *pfile;
5     int ch;
6
7     puts("Opening file...");
8     pfile = fopen("FILE.TXT", "r");
9
10    puts("Displaying file ...");
11    while ((ch = fgetc(pfile)) != EOF) {
12        printf("<%c>", ch);
13    }
14
15    puts("Closing file ...");
16    fclose(pfile);
17
18    return 0;
19 }
```

**Listing 4.23.** i\_testing/programs/test\_23.c

### 4.3.12 Comportamento non definito

#### test\_24.c

Qui l'errore sta nel leggere e modificare la stessa variabile nella stessa espressione senza una sequenza ben definita. Il risultato dipenderà dal compilatore. Osserviamo infatti come questo vari a seconda del fatto che si valuti prima il secondo o il primo membro. Si tratta di un errore di comportamento indefinito. Nel responso di ChatGPT l'errore viene identificato correttamente sebbene venga fatta una considerazione errata circa un accesso fuori dai limiti dell'array che non sussiste. Vengono proposte le correzioni.

```

1 #include <stdio.h>
2
3 int main() {
4     int a[10];
5     int n = 3;
6
7     a[n] = n++;
8     printf("a[%d] = %d, n = %d\n", a[n], n);
9     return 0;
10 }
```

**Listing 4.24.** i\_testing/programs/test\_24.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_23
(lldb) target create "test_23"
Current executable set to '/home/carlo/I_sperimentazione/test_23' (x86_64).
(lldb) autodebug
----- Execution Started -----
Opening file...
Displaying file ...
Process 96500 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Basandoci sull'output che hai fornito, l'errore principale nel tuo codice è un errore di segmentazione (segmentation fault). Questo errore si verifica quando il programma tenta di accedere a una porzione di memoria per la quale non ha i permessi o che non esiste.

Guardando il codice, possiamo identificare alcune possibili cause per l'errore di segmentazione:

1. Il file "FILE.TXT" potrebbe non esistere nella directory corrente. Assicurati che il file esista e che tu abbia i permessi necessari per leggerlo.

2. Potrebbe esserci un problema nell'apertura del file. Assicurati che l'apertura del file con la funzione "fopen" sia andata a buon fine controllando che "pf" non sia nullo dopo la chiamata. In caso contrario, verifica se hai specificat

```

(a) Debugging di test\_23.c

```

o il percorso corretto del file.

3. Potresti avere problemi nel controllo dell'EOF nel ciclo while. Assicurati che la condizione `(ch = fgetc(pfile)) != EOF` sia corretta per il tipo di carattere restituito da "fgetc". Potrebbe essere necessario utilizzare il tipo "int" anziché "char" per il valore restituito da "fgetc" al fine di confrontarlo con EOF correttamente.

4. Potrebbe esserci un problema nella chiusura del file. Assicurati di chiudere correttamente il file con la funzione "fclose" dopo aver finito di leggerlo.

Analizzando meglio i punti 1 e 2, sembra che il problema principale sia che il file "FILE.TXT" non sia stato trovato nella directory corrente, portando all'errore di segmentazione.

Ti suggerisco di controllare attentamente il percorso e il nome del file per assicurarti che siano corretti. Inoltre, verifica anche i permessi di accesso al file.

Se il problema persiste, ti consiglio di eseguire il programma in modalità di debug e utilizzare il comando `bt` in LLDB per visualizzare la traccia dello stack e ottenere ulteriori informazioni sugli errori nell'applicazione.
(lldb) █

```

(b) Debugging di test\_23.c (continuo)

Figura 4.23. Debugging di test\_23.c

```
carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_24
(lldb) target create "test_24"
Current executable set to '/home/carlo/I_sperimentazione/test_24' (x86_64).
(lldb) autodebug
----- Execution Started -----
a[n] = 3, n = 4
----- Execution Terminated -----
Dal codice fornito, posso individuare due tipi di errori:

1. Errore di sequenza delle operazioni: Nella riga 7 del codice, viene assegnato il valore di `n++` all'elemento `a[n]`. Tuttavia, l'ordine delle operazioni non è definito in questa espressione, quindi il comportamento è indefinito. È una buona pratica evitare di modificare una variabile in un'espressione in cui viene utilizzata contemporaneamente.

2. Accesso fuori dai limiti dell'array: Nella riga 7, si tenta di assegnare il valore incrementato di `n` all'elemento `a[n]`. Tuttavia, l'array `a` è un array di dimensione 10, quindi l'accesso a `a[n]` quando `n` vale 3 comporta un accesso fuori dai limiti dell'array.

Questi errori possono essere corretti nel modo seguente:

1. Modifica la riga 7 da `a[n] = n++;` a `a[n] = n; n++;` per evitare il comportamento indefinito e garantire che `n` venga incrementata dopo l'assegnazione a `a[n]`.

2. Modifica la riga 5 da `int n = 3;` a `int n = 2;` per garantire che `n` sia nell'intervallo consentito per l'array `a`.

Queste correzioni dovrebbero aiutare a risolvere gli errori nel codice.
(lldb) █
```

Figura 4.24. Debugging di test\_24.c

**test\_25.c**

L'espressione che si occupa di assegnare un valore a "result" è causa di un comportamento indefinito poiché il codice contiene più di una modifica della stessa variabile nella stessa espressione. A seconda di come si valuta l'espressione il risultato varia, non è quindi definito e dipenderà dal compilatore. Si ha a che fare con un errore di comportamento indefinito. Nel risponso di ChatGPT l'errore viene identificato correttamente.

```

1 #include <stdio.h>
2
3 int main() {
4     int x = 5;
5     int y = 10;
6     int result;
7
8     result = x + y++ + x++;
9     printf("Il risultato : %d\n", result);
10
11    return 0;
12 }
```

**Listing 4.25.** i\_testing/programs/test\_25.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_25
(lldb) target create "test_25"
Current executable set to '/home/carlo/I_sperimentazione/test_25' (x86_64).
(lldb) autodebug
----- Execution Started -----
Il risultato è: 20
----- Execution Terminated -----
Dal codice fornito, dal messaggio di output e dal fatto che il programma ha term
inato senza interruzioni, sembra che il codice sia corretto e non ci siano error
i nel programma stesso.

Tuttavia, vorrei sottolineare che l'utilizzo degli operatori di post-incremento
nelle espressioni come in questo caso può portare a risultati imprevedibili e a
comportamenti non definiti.

Nel codice fornito, hai utilizzato 2 operatori di post-incremento ('y++' e 'x++')
all'interno dell'espressione `x + y++ + x++`. Il comportamento di espressioni
con più incrementi o decrementi sulla stessa variabile non è definito dallo stan
dard ANSI C e può variare tra le diverse implementazioni del compilatore.

Quindi, anche se il programma sembra funzionare correttamente nell'esecuzione co
rente, è considerata una pratica rischiosa e non raccomandata da utilizzare. È
meglio evitare l'utilizzo di più operatori di incremento o decremento sulla stes
sa variabile all'interno della stessa espressione.

Per ottenere risultati consistenti e prevedibili, è consigliabile utilizzare sol
o un operatore di incremento o decremento per volta o separare le espressioni in
operazioni distinte per evitare comportamenti ambigui.
(lldb) █
```

**Figura 4.25.** Debugging di test\_25.c

### 4.3.13 Loop infiniti

#### test\_26.c

Questo programma è istanza di una classe di errori che non è possibile debuggare in modo automatico, poiché si andrebbe avanti in un'esecuzione continua. Si può procedere solo manualmente con gli altri comandi definiti nel plugin. Osserviamo che non è stata definita una condizione di terminazione per il loop messo in piedi dall'operatore “while”. Questo genera come errore un loop infinito. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```
1 #include <stdio.h>
2
3 int main() {
4     int i;
5
6     while (i < 5) {
7         printf("Inside while loop\n");
8     }
9
10    return 0;
11 }
```

**Listing 4.26.** i\_testing/programs/test\_26.c

#### test\_27.c

Così come il precedente, anche questo programma è istanza di una classe di errori che non è possibile debuggare in modo automatico, poiché si andrebbe avanti in un'esecuzione continua. Si può procedere solo manualmente con gli altri comandi definiti nel plugin. Il problema sta nel porre come condizione di esistenza del ciclo while una tautologia senza fornire una condizione di uscita all'interno del corpo del ciclo. Nel responso di ChatGPT l'errore viene identificato correttamente. Vengono anche proposte le correzioni.

```
1 #include <stdio.h>
2
3 int main() {
4     while (1) {
5         printf("This will run forever\n");
6     }
7
8     return 0;
9 }
```

**Listing 4.27.** i\_testing/programs/test\_27.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_26
(lldb) target create "test_26"
Current executable set to '/home/carlo/I_sperimentazione/test_26' (x86_64).
(lldb) b --name main
Breakpoint 1: where = test_26`main + 8 at test_26.c:6:11, address = 0x0000000000000013d
(lldb) r
Process 99132 launched: '/home/carlo/I_sperimentazione/test_26' (x86_64)
Process 99132 stopped
* thread #1, name = 'test_26', stop reason = breakpoint 1.1
  frame #0: 0x00005555555513d test_26`main at test_26.c:6:11
  3   int main() {
  4     int i;
  5
-> 6       while (i < 5) {
  7         printf("Inside while loop\n");
  8     }
  9
(lldb) send
(lldb) ask Elencami gli errori nel codice
Sì, ci sono un paio di errori nel tuo codice:

1. La variabile `i` nell'istruzione `int i;` alla linea 4 non viene inizializzata con un valore. È una buona pratica assegnare un valore iniziale alle variabili per evitare comportamenti imprevisti. Potresti inizializzare `i` con `int i = 0;`.

2. Nella riga 6, stai utilizzando la variabile `i` nel controllo del ciclo while senza averla inizializzata in precedenza. Questo può portare a un comportamento

```

(a) Debugging di test\_26.c

```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ 
indefinito. Puoi correggere questo problema inizializzando `i` con un valore, come menzionato sopra.

3. Nella riga 6 il ciclo while non ha un'istruzione che incrementa il valore di `i`, quindi il ciclo continuerà all'infinito. Per evitare questo, puoi incrementare `i` all'interno del ciclo while con `i++` o un'altra istruzione che aumenta il suo valore di volta in volta.

Ecco una versione corretta del tuo codice:

```c
#include <stdio.h>

int main() {
    int i = 0;

    while (i < 5) {
        printf("Inside while loop\n");
        i++;
    }

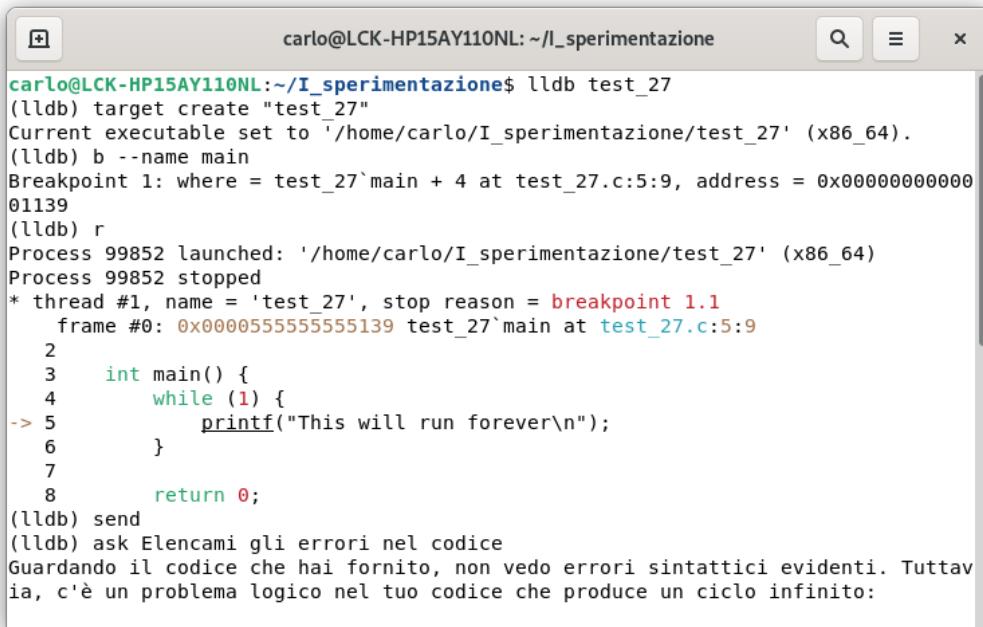
    return 0;
}
```

Queste correzioni risolvono gli errori nel tuo codice e consentono al programma di eseguirsi correttamente.
(lldb) 

```

(b) Debugging di test\_26.c (continuo)

Figura 4.26. Debugging di test\_26.c

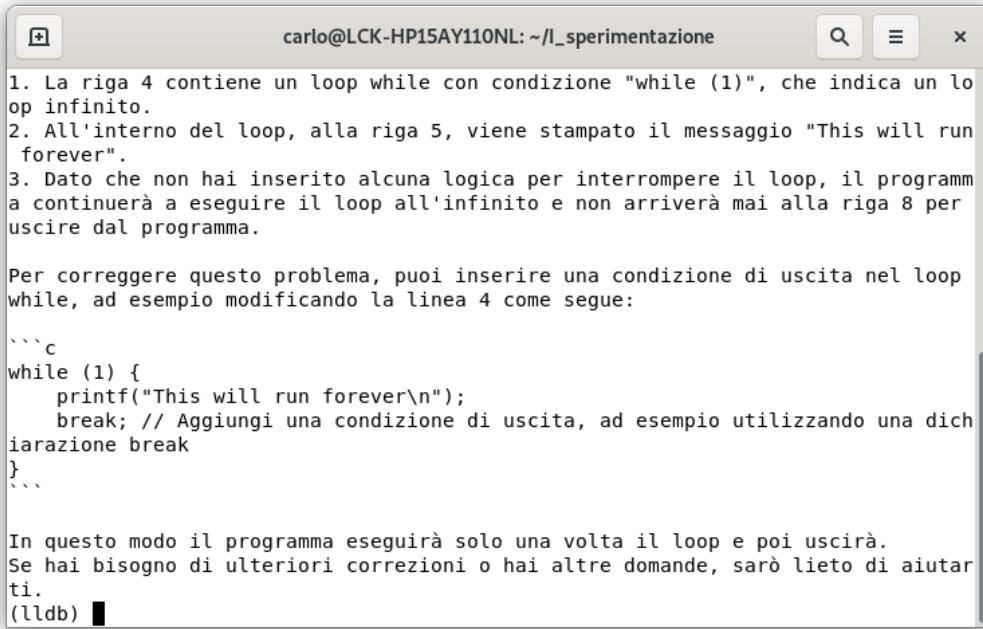


```

carlo@LCK-HP15AY110NL:~/I_sperimentazione$ lldb test_27
(lldb) target create "test_27"
Current executable set to '/home/carlo/I_sperimentazione/test_27' (x86_64).
(lldb) b --name main
Breakpoint 1: where = test_27`main + 4 at test_27.c:5:9, address = 0x0000000000000139
(lldb) r
Process 99852 launched: '/home/carlo/I_sperimentazione/test_27' (x86_64)
Process 99852 stopped
* thread #1, name = 'test_27', stop reason = breakpoint 1.1
  frame #0: 0x000055555555139 test_27`main at test_27.c:5:9
      2
      3     int main() {
      4         while (1) {
->  5             printf("This will run forever\n");
      6         }
      7
      8     return 0;
(lldb) send
(lldb) ask Elencami gli errori nel codice
Guardando il codice che hai fornito, non vedo errori sintattici evidenti. Tuttavia, c'è un problema logico nel tuo codice che produce un ciclo infinito:

```

(a) Debugging di test\_27.c



1. La riga 4 contiene un loop while con condizione "while (1)", che indica un loop infinito.  
 2. All'interno del loop, alla riga 5, viene stampato il messaggio "This will run forever".  
 3. Dato che non hai inserito alcuna logica per interrompere il loop, il programma continuerà a eseguire il loop all'infinito e non arriverà mai alla riga 8 per uscire dal programma.

Per correggere questo problema, puoi inserire una condizione di uscita nel loop while, ad esempio modificando la linea 4 come segue:

```

```c
while (1) {
    printf("This will run forever\n");
    break; // Aggiungi una condizione di uscita, ad esempio utilizzando una dichiarazione break
}
```

```

In questo modo il programma eseguirà solo una volta il loop e poi uscirà.  
 Se hai bisogno di ulteriori correzioni o hai altre domande, sarò lieto di aiutarti.

(b) Debugging di test\_27.c (continuo)

Figura 4.27. Debugging di test\_27.c

## 4.4 II Sperimentazione

Andiamo ora ad effettuare dei test chiave per la sperimentazione dell'applicativo. I test che seguiranno si propongono di porre in evidenza le differenze qualitative del responso di ChatGPT in relazione alle informazioni inviate. Cosa cambia se inviamo soltanto il codice a ChatGPT? È in grado di darci una risposta soddisfacente? Cosa succede, invece, se oltre al codice inviamo anche le altre informazioni di debugging recuperate con LLDB? La risposta di ChatGPT sarà più mirata e precisa? Il punto è osservare se ChatGPT è in grado, grazie a delle informazioni più dettagliate, di dare risposte più soddisfacenti e quantificare i vantaggi di utilizzo dell'applicativo. Vedere insomma come l'intelligenza artificiale si comporta in relazione a un prompt più rifinito e completo e di quanto queste risposte si discostano da quelle date ricevendo come prompt il solo codice. Dai dati emersi si proseguirà poi nell'analisi dei risultati con una riflessione ingegneristica sui corollari che derivano dalla presente sperimentazione, ovvero quando e in quali casi l'utilizzo dell'applicativo può rivelarsi più vantaggioso che in altri e quindi quando si possono ottenere i risultati migliori dall'applicativo. A scopo riepilogativo, specifichiamo nuovamente le informazioni inviate dall'applicativo:

- **Dati di sistema**, questi comprendono il nome del sistema operativo, la versione dello stesso, il processore utilizzato e altre info supplementari sul sistema. Esempio: “Linux-5.10.0-23-amd64-x86\_64-with-glibc2.31”.
- **Versione di LLDB**, la versione di LLDB debugger installata sul proprio sistema e che si sta utilizzando per il debugging del programma. Esempio: “lldb version 11.0.1”.
- **Nome del file sorgente**, potrebbe essere utile conoscere il nome del file sorgente, questo perché magari si tratta di un file con una funzione specifica, pensiamo ad esempio ad una componente di un sistema operativo. Questo, inoltre, delinea in maniera netta il linguaggio di programmazione del file sorgente.
- **Codice del programma**, il codice sorgente numerato riga per riga del programma che si sta debuggando. Questa è un'informazione che nel corso di questa sperimentazione manderemo in solitaria a ChatGPT prima e poi assieme alle altre informazioni di debugging.
- **Linea corrente**, questa sarà la prima riga del main in caso il programma non abbia subito interruzioni, mentre sarà la linea precedente al crash del programma in caso di un'interruzione dell'esecuzione in runtime. Segue che mentre nel primo caso, quello con esecuzione senza interruzioni, non è utile, nel secondo risulta invece essenziale.
- **Variabili visibili**, così come per la linea corrente, questa informazione acquisisce una reale utilità in caso di interruzione del programma dato che fornisce i valori delle variabili in gioco alla linea corrente e quindi un attimo prima che il programma subisca una terminazione improvvisa.

- **Output del programma**, ciò che viene stampato a schermo durante l'esecuzione del programma. Comprende la stampa anche di eventuali errori.
- **Ragione dell'interruzione in runtime**, se viene rilevato un segnale durante l'esecuzione del programma che ne interrompe l'esecuzione, allora viene specificato il nome del segnale in questione quale responsabile dell'accaduto, oltre che la linea di rilevazione dello stesso.

Per ogni test verrà quindi mostrato prima il file sorgente, corredata da eventuali file di supporto, e successivamente le schermate del responso di ChatGPT, prima in relazione al solo codice inviato, poi a tutte le altre informazioni di debugging.

È bene precisare che le sperimentazioni che seguono riguardano esempi di programmi la cui esecuzione viene interrotta con la comparsa degli errori. Per come è stato infatti progettato l'applicativo, come già detto, il comando “autodebug” pone lo stato “Launched Process State” nel quale poi vengono utilizzati, internamente al comando, i comandi “send” e “ask”. In caso di esecuzione con interruzioni, il breakpoint viene posto lungo la linea corrente, antecedente di uno “step” rispetto la terminazione improvvisa. In caso, invece, di esecuzione senza interruzioni, il breakpoint viene posto sulla funzione “main()” e quindi le informazioni sulla linea corrente e le variabili visibili perdono di significato. Pertanto, onde sperimentare e osservare la differenza qualitativa, si è deciso di utilizzare esempi la cui esecuzione viene interrotta così che tutte le informazioni di debugging possano essere utili alla rilevazione degli errori da parte di ChatGPT. Il nostro scopo, infatti, è vedere come ChatGPT riesce a sfruttare tali informazioni e quanto bene riesce a farlo valutando il massimo vantaggio che ne si può trarre, è bene perciò che tali informazioni siano rilevanti e non che vengano ridotte al solo codice sorgente, cosa che renderebbe la sperimentazione priva di senso.

Inoltre, onde mettere alla prova ChatGPT nella rilevazione dei bug ed osservare meglio i vantaggi di utilizzo dell'applicativo, si è deciso di mascherare gli errori, dove con “mascherare” si intende nascondere l'errore in due accezioni differenti:

- **mascheramento di calcolo** : ciò avviene quando l'errore è si riscontrabile dal codice statico ma solo dopo aver effettuato dei calcoli, quindi per essere evidenziato all'IA non basta vedere il codice, ma dovrà effettuare i calcoli autonomamente se non possiede le informazioni di debugging recuperate dal plugin. Esempio: (ii\_test\_1.c) .
- **mascheramento di input** : ciò avviene quando l'errore non è riscontrabile dal codice in quanto questo si manifesta quando ci si serve di funzioni o dati importati da file esterni, per essere evidenziato quindi l'IA dovrà essere in grado di valutare tutte le possibili cause di errore manifestabili a seconda di ciò che si potrebbe ricevere in input se non ha a disposizione le informazioni di debugging recuperate dal plugin. Esempio: (ii\_test\_4.c) .

Sostanzialmente il mascheramento mette alla prova ChatGPT sul fronte della potenza di calcolo la quale deve essere adeguata in caso non si posseggano le informazioni di debugging tratte dall'esecuzione del programma.

#### 4.4.1 Test 1

In questo esempio, già trattato nella precedente sperimentazione (test\_8.c), si ha una divisione per zero alla riga 9. L'inizializzazione a “0” del divisore viene mascherata attraverso l'utilizzo di più operazioni matematiche quindi per mascheramento di calcolo, pertanto dal codice non risulta evidente che “ $c = 0$ ” a meno che non vengano eseguiti dei calcoli. Differenziamo i responsi di ChatGPT:

- **Responso per l'invio del solo codice:** ChatGPT riporta incongruenze circa il tipo fornito e restituito dagli operatori matematici utilizzati oltre che due considerazioni errate: una sull'ordinamento delle operazioni e una sugli errori di sintassi. Non riesce a trovare l'errore di divisione per zero.
- **Responso per l'invio delle informazioni di debugging:** ChatGPT ci ammonisce della “floating point exception” (definita dal segnale rilevato “SIGFPE”). La divisione per zero viene rilevata subito.

Nel caso preso ad esempio ChatGPT necessita delle informazioni di debugging per rilevare l'errore principale del codice.

```

1 #include <stdio.h>
2 #include <math.h>
3
4 int main() {
5     int a = 1;
6     int b = 0;
7     int c = log(pow(sqrt(a),b) - sin(b));
8
9     int result = a / c;
10    printf("Result: %d\n", result);
11
12    return 0;
13 }
```

**Listing 4.28.** ii\_testing/programs/ii\_test\_1.c

```

carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_1
(lldb) target create "ii_test_1"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_1' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 108432 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Ci sono tre tipi di errori nel codice:

1. Errori di sintassi:
   - Manca l'inclusione delle librerie `stdio.h` e `math.h`.
   - Manca un punto e virgola alla fine della riga 7.

2. Errori semantici:
   - L'uso di `sqrt(a)` come argomento di `pow` è errato perché `sqrt` restituisce un valore in virgola mobile mentre `pow` accetta due argomenti in virgola mobile. Puoi correggere questo errore usando `sqrt((double)a)` per ottenere il risultato in virgola mobile.
   - La funzione `log` restituisce un valore in virgola mobile, ma il suo risultato viene assegnato a una variabile di tipo `int`. È necessario eseguire un cast esplicito per convertire il valore in virgola mobile in un intero, ad esempio `int c = (int)log(...)`.
   - La divisione viene effettuata tra due variabili intere `a` e `c`, ma l'assegnazione avviene in una variabile di tipo `int`. Per assicurare il corretto calcolo della divisione, è necessario eseguire un cast esplicito del risultato di `a` diviso `c` a `int`, ad esempio `int result = (int)a / c`.

3. Errori di logica:
   - La formula utilizzata per calcolare `c` è potenzialmente problematica. La radice quadrata di `a` dovrebbe essere calcolata prima di elevare il risultato al la potenza `b`. Pertanto, è necessario aggiungere parentesi per garantire l'ordine corretto delle operazioni. Puoi correggere questo errore utilizzando `pow(sqrt((double)a), b)`.

Ecco il codice corretto con gli errori corretti:
```c
#include <stdio.h>
#include <math.h>

int main() {
    int a = 1;
    int b = 0;
    double c = (int)log(pow(sqrt((double)a), b) - sin(b));

    int result = (int)a / c;
    printf("Result: %d\n", result);

    return 0;
}
```

```

(a) Debugging di ii\_test\_1.c con invio del solo codice a ChatGPT

```

3. Errori di logica:
   - La formula utilizzata per calcolare `c` è potenzialmente problematica. La radice quadrata di `a` dovrebbe essere calcolata prima di elevare il risultato al la potenza `b`. Pertanto, è necessario aggiungere parentesi per garantire l'ordine corretto delle operazioni. Puoi correggere questo errore utilizzando `pow(sqrt((double)a), b)`.

Ecco il codice corretto con gli errori corretti:
```c
#include <stdio.h>
#include <math.h>

int main() {
    int a = 1;
    int b = 0;
    double c = (int)log(pow(sqrt((double)a), b) - sin(b));

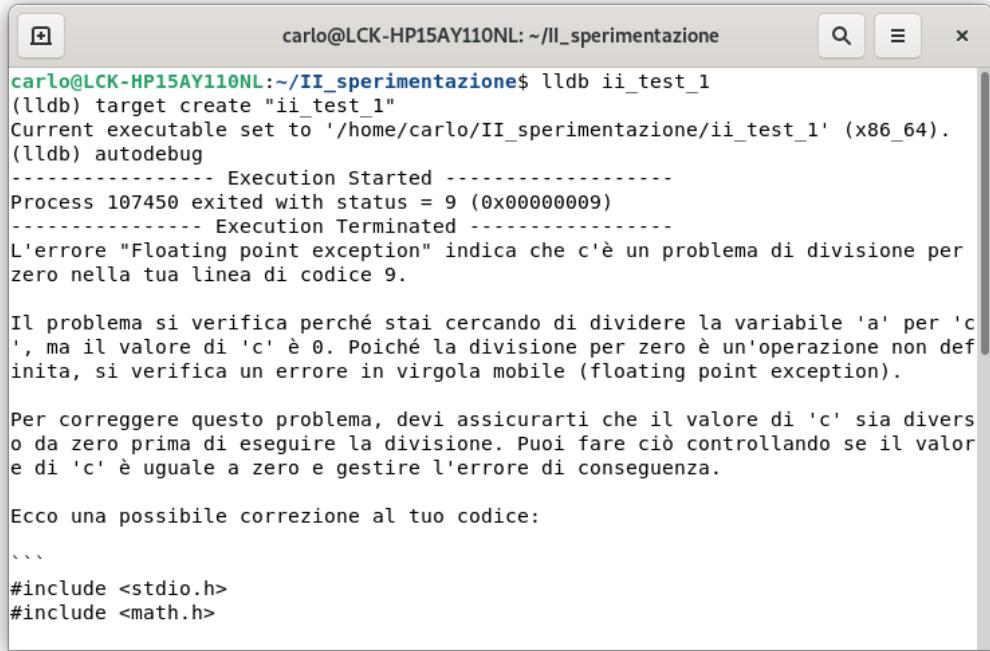
    int result = (int)a / c;
    printf("Result: %d\n", result);

    return 0;
}
```

```

(b) Debugging di ii\_test\_1.c con invio del solo codice a ChatGPT (continuo)

**Figura 4.28.** Debugging di ii\_test\_1.c con invio del solo codice a ChatGPT



```

carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_1
(lldb) target create "ii_test_1"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_1' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 107450 exited with status = 9 (0x00000009)
----- Execution Terminated -----
L'errore "Floating point exception" indica che c'è un problema di divisione per zero nella tua linea di codice 9.

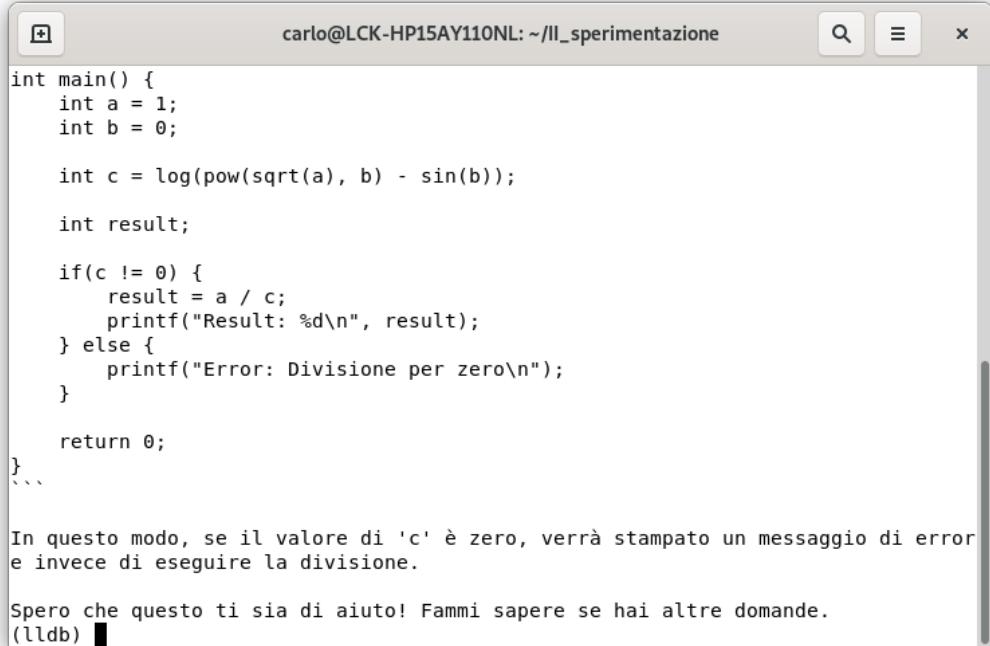
Il problema si verifica perché stai cercando di dividere la variabile 'a' per 'c', ma il valore di 'c' è 0. Poiché la divisione per zero è un'operazione non definita, si verifica un errore in virgola mobile (floating point exception).

Per correggere questo problema, devi assicurarti che il valore di 'c' sia diverso da zero prima di eseguire la divisione. Puoi fare ciò controllando se il valore di 'c' è uguale a zero e gestire l'errore di conseguenza.

Ecco una possibile correzione al tuo codice:

```
#include <stdio.h>
#include <math.h>
```
```

(a) Debugging di ii\_test\_1.c con invio delle informazioni di debugging a ChatGPT



```

int main() {
    int a = 1;
    int b = 0;

    int c = log(pow(sqrt(a), b) - sin(b));

    int result;

    if(c != 0) {
        result = a / c;
        printf("Result: %d\n", result);
    } else {
        printf("Error: Divisione per zero\n");
    }

    return 0;
}..
```

In questo modo, se il valore di 'c' è zero, verrà stampato un messaggio di errore invece di eseguire la divisione.

Spero che questo ti sia di aiuto! Fammi sapere se hai altre domande.

(b) Debugging di ii\_test\_1.c con invio delle informazioni di debugging a ChatGPT (continuo)

**Figura 4.29.** Debugging di ii\_test\_1.c con invio delle informazioni di debugging a ChatGPT

#### 4.4.2 Test 2

In questo esempio, già trattato nella precedente sperimentazione (test\_23.c), si ha un errore dovuto alla mancata presenza del file che si vuole aprire, errore che è stato mascherato per mascheramento di input. Pertanto la chiamata alla funzione “fopen()” restituirà un puntatore nullo il cui utilizzo causerà la terminazione improvvisa del programma. Differenziamo i responsi di ChatGPT:

- **Responso per l'invio del solo codice:** ChatGPT rileva due possibili errori circa l'apertura e la chiusura dei file individuando correttamente le criticità del codice in questione, sebbene venga fatta una considerazione errata sull'inclusione dell'intestazione.
- **Responso per l'invio delle informazioni di debugging:** ChatGPT ci dice subito che durante l'esecuzione del programma si è verificato un errore di segmentazione (definito dal segnale rilevato “SIGSEGV”) indicando la linea dello stesso ed evidenziando come questo sembra essere stato causato dalla mancata apertura del file.

In questo caso ChatGPT è in grado di rilevare i possibili errori del codice anche senza le informazioni di debugging, sebbene sia in grado di circoscrivere meglio l'errore se queste sono fornite.

```
1 #include <stdio.h>
2
3 int main() {
4     FILE *pfile;
5     int ch;
6
7     puts("Opening file...");
8     pfile = fopen("FILE.TXT", "r");
9
10    puts("Displaying file ...");
11    while ((ch = fgetc(pfile)) != EOF) {
12        printf("<%c>", ch);
13    }
14
15    puts("Closing file ...");
16    fclose(pfile);
17
18    return 0;
19 }
```

**Listing 4.29.** ii\_testing/programs/ii\_test\_2.c

```

carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_2
(lldb) target create "ii_test_2"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_2' (x86_64).
(lldb) autodebug
----- Execution Started -----
Opening file...
Displaying file ...
Process 108590 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Nel codice fornito, sono presenti i seguenti errori:

1. Errore di inclusione dell'intestazione: Il codice utilizza la funzione `puts()` , ma l'intestazione `<stdio.h>` non è inclusa. Per correggere l'errore, è sufficiente includere l'intestazione `<stdio.h>` all'inizio del file:

```c
#include <stdio.h>
```

2. Potenziale errore di apertura del file: Il codice apre un file chiamato "FILE.TXT" utilizzando `fopen()` , ma non viene eseguito un controllo per verificare se l'apertura del file è stata effettuata correttamente. Se il file non esiste o non può essere aperto, il programma potrebbe andare in errore. Per correggere l'errore, è consigliabile controllare se il puntatore restituito da `fopen()` è nullo. Ad esempio:

```

(a) Debugging di ii\_test\_2.c con invio del solo codice a ChatGPT

```

```c
pfile = fopen("FILE.TXT", "r");
if (pfile == NULL) {
    printf("Impossibile aprire il file.");
    return 1;
}
```

3. Errore di chiusura del file: Il codice chiude il file utilizzando la funzione `fclose()` . Tuttavia, non viene eseguito alcun controllo sull'avvenuta chiusura del file. Potrebbe essere utile eseguire un controllo sull'operazione di chiusura tramite la funzione `fclose()` . Ad esempio:

```c
if (fclose(pfile) != 0) {
    printf("Impossibile chiudere il file.");
    return 1;
}
```

Questi sono gli errori individuati nel codice fornito, differenziati per tipologia. È buona prassi controllare sempre gli errori di apertura e chiusura dei file , così come l'inclusione delle intestazioni corrette nel codice.
(lldb) █

```

(b) Debugging di ii\_test\_2.c con invio del solo codice a ChatGPT (continuo)

**Figura 4.30.** Debugging di ii\_test\_2.c con invio del solo codice a ChatGPT

```
carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_2
(lldb) target create "ii_test_2"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_2' (x86_64).
(lldb) autodebug
----- Execution Started -----
Opening file...
Displaying file ...
Process 107614 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Dai dati che hai fornito, sembra che il programma abbia causato un errore di segmentazione all'indirizzo non valido sulla linea 11. Questo significa che il programma sta cercando di accedere a una regione di memoria a cui non ha il permesso di accedere.

Dall'analisi del codice sorgente, sembra che il problema possa essere causato da un errore di apertura del file. Se il file "FILE.TXT" non esiste nella directory corrente o se non hai i diritti di accesso, la chiamata a fopen fallirà e il puntatore al file (pfile) rimarrà nullo. Successivamente, nella riga 11, il programma cerca di leggere un carattere dal file usando la funzione fgetc su pfile, che è nullo, causando quindi l'errore di segmentazione.

Per correggere l'errore, assicurati che il file "FILE.TXT" esista nella directory corrente e che tu abbia i diritti di accesso appropriati. Puoi anche controllare il valore di ritorno della chiamata a fopen per gestire eventuali errori di apertura del file in modo appropriato.

Inoltre, verifica che la variabile ch sia correttamente inizializzata prima dell'uso. Potresti assegnarle un valore di inizializzazione predefinito al momento della dichiarazione, come ad esempio: int ch = 0;.

Rivedendo il codice, sembrano essere questi gli errori presenti. Spero che queste informazioni ti siano utili per risolvere il problema!
(lldb) █
```

Figura 4.31. Debugging di ii\_test\_2.c con invio delle informazioni di debugging a ChatGPT

#### 4.4.3 Test 3

In questo esempio, già trattato nella precedente sperimentazione (test\_9.c), si ha un accesso a memoria non allocata. L'indice dell'array al quale si accede viene preso dal file “input.txt”, di fatto mascherando l'errore per mascheramento di input poiché l'indice erroneo non è presente nel codice ed è letto da file, ma anche per mascheramento di calcolo poiché l'indice viene coinvolto in alcuni calcoli matematici che è necessario eseguire per trarne il valore. L'array viene allocato per contenere 10 elementi di tipo “int”, da quello con indice 0 a 9. L'indice fornito dal file tuttavia è 10. Questo causa l'errore in questione non essendo stata allocata abbastanza memoria per contenere anche l'undicesimo elemento e viene quindi lanciato il segnale “SIGABRT”. Differenziamo i responsi di ChatGPT:

- **Responso per l'invio del solo codice:** ChatGPT identifica correttamente l'errore, ammondendo il programmatore sulla necessità di controllare l'indice dell'array ricevuto in input. Evidenzia inoltre altre criticità, una delle quali insussistente.
- **Responso per l'invio delle informazioni di debugging:** ChatGPT identifica correttamente l'errore ed evidenzia inoltre altre criticità, ma la motivazione dietro l'errore risulta fuorviante, la ragione dell'accesso fuori dai limiti dell'array non è l'operazione che si effettua sull'indice, ma la possibilità che questo, ricevuto in input, non sia valido rispetto le dimensioni dell'array.

In questo caso le informazioni di debugging non sembra siano state utilizzate dal ChatBot data la somiglianza dei due responsi. In ogni caso l'errore viene trovato sebbene con delle imprecisioni.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 int main() {
6     int *array = (int *)malloc(10 * sizeof(int));
7     int index;
8
9     FILE *file = fopen("input.txt", "r");
10    fscanf(file, "%d", &index);
11
12    index = log10(index)*(21/2);
13    printf("index: %d\n", index);
14    array[index] = 42;
15    printf("array[%d]: %d\n", index, array[index]);
16
17    fclose(file);
18    free(array);
19
20    return 0;
21 }
```

**Listing 4.30.** ii\_testing/programs/ii\_test\_3.c

1 10

**Listing 4.31.** ii\_testing/programs/input.txt

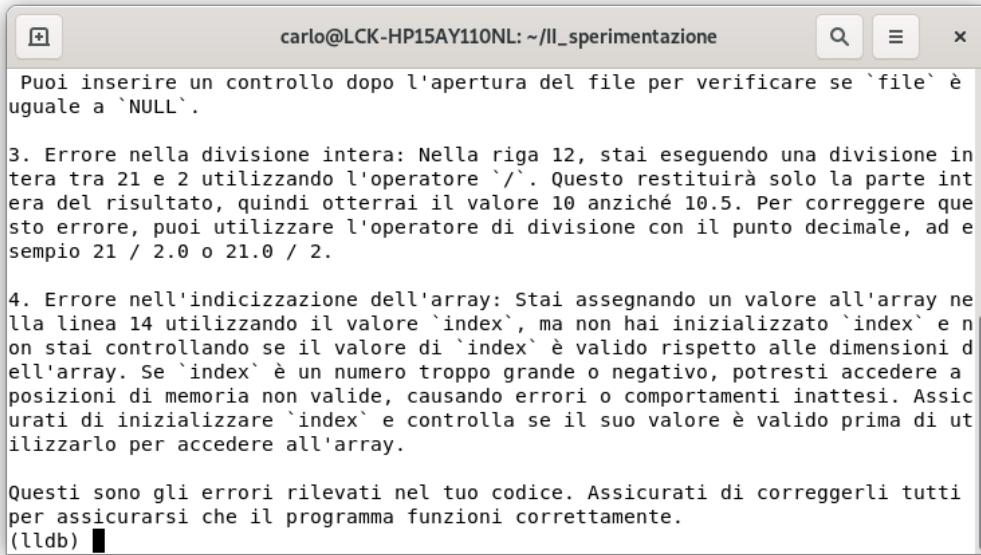


```
carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_3
(lldb) target create "ii_test_3"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_3' (x86_64).
(lldb) autodebug
----- Execution Started -----
index: 10
array[index]: 42
munmap_chunk(): invalid pointer
Process 108802 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Il tuo codice presenta i seguenti errori:

1. Errore di inclusione della libreria: Le librerie `stdio.h`, `stdlib.h` e `mat.h` non sono state incluse nel tuo codice. Questo può causare errori durante la compilazione. Per correggere questo errore, devi includere queste librerie all'inizio del tuo codice.

2. Errore nell'apertura del file di input: Non stai verificando se il file di input viene aperto correttamente prima di iniziare a leggere i dati. Se il file non può essere aperto correttamente, il programma dovrà gestire questa situazione.
```

(a) Debugging di ii\_test\_3.c con invio del solo codice a ChatGPT



```
Puoi inserire un controllo dopo l'apertura del file per verificare se `file` è uguale a `NULL`.

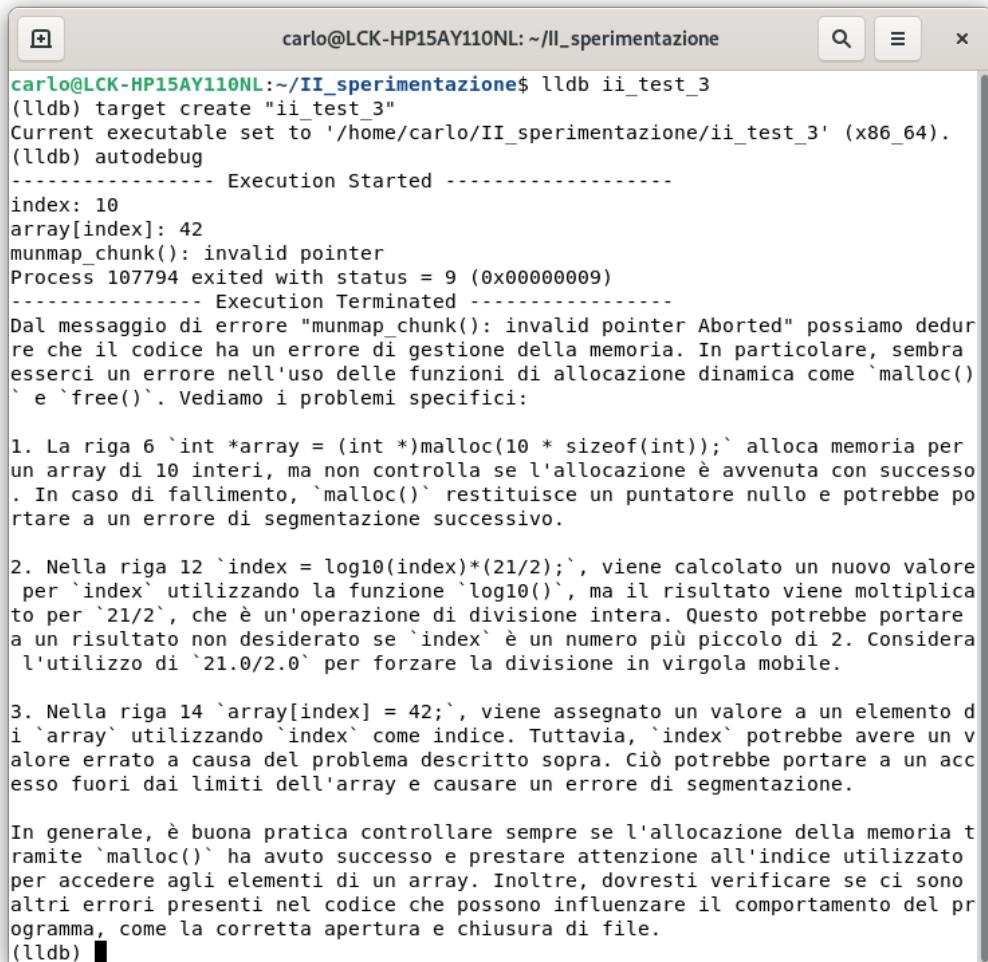
3. Errore nella divisione intera: Nella riga 12, stai eseguendo una divisione intera tra 21 e 2 utilizzando l'operatore `/`. Questo restituirà solo la parte intera del risultato, quindi otterrai il valore 10 anziché 10.5. Per correggere questo errore, puoi utilizzare l'operatore di divisione con il punto decimale, ad esempio 21 / 2.0 o 21.0 / 2.

4. Errore nell'indicizzazione dell'array: Stai assegnando un valore all'array nella linea 14 utilizzando il valore `index`, ma non hai inizializzato `index` e non stai controllando se il valore di `index` è valido rispetto alle dimensioni dell'array. Se `index` è un numero troppo grande o negativo, potresti accedere a posizioni di memoria non valide, causando errori o comportamenti inattesi. Assicurati di inizializzare `index` e controlla se il suo valore è valido prima di utilizzarlo per accedere all'array.

Questi sono gli errori rilevati nel tuo codice. Assicurati di correggerli tutti per assicurarsi che il programma funzioni correttamente.
(lldb) █
```

(b) Debugging di ii\_test\_3.c con invio del solo codice a ChatGPT (continuo)

**Figura 4.32.** Debugging di ii\_test\_3.c con invio del solo codice a ChatGPT



carlo@LCK-HP15AY110NL:~/II\_sperimentazione\$ lldb ii\_test\_3  
(lldb) target create "ii\_test\_3"  
Current executable set to '/home/carlo/II\_sperimentazione/ii\_test\_3' (x86\_64).  
(lldb) autodebug  
----- Execution Started -----  
index: 10  
array[index]: 42  
munmap\_chunk(): invalid pointer  
Process 107794 exited with status = 9 (0x00000009)  
----- Execution Terminated -----  
Dal messaggio di errore "munmap\_chunk(): invalid pointer Aborted" possiamo dedurre che il codice ha un errore di gestione della memoria. In particolare, sembra esserci un errore nell'uso delle funzioni di allocazione dinamica come `malloc()` e `free()`. Vediamo i problemi specifici:

1. La riga 6 `int \*array = (int \*)malloc(10 \* sizeof(int));` alloca memoria per un array di 10 interi, ma non controlla se l'allocazione è avvenuta con successo. In caso di fallimento, `malloc()` restituisce un puntatore nullo e potrebbe portare a un errore di segmentazione successivo.
2. Nella riga 12 `index = log10(index)\*(21/2);`, viene calcolato un nuovo valore per `index` utilizzando la funzione `log10()`, ma il risultato viene moltiplicato per `21/2`, che è un'operazione di divisione intera. Questo potrebbe portare a un risultato non desiderato se `index` è un numero più piccolo di 2. Considera l'utilizzo di `21.0/2.0` per forzare la divisione in virgola mobile.
3. Nella riga 14 `array[index] = 42;`, viene assegnato un valore a un elemento di `array` utilizzando `index` come indice. Tuttavia, `index` potrebbe avere un valore errato a causa del problema descritto sopra. Ciò potrebbe portare a un accesso fuori dai limiti dell'array e causare un errore di segmentazione.

In generale, è buona pratica controllare sempre se l'allocazione della memoria tramite `malloc()` ha avuto successo e prestare attenzione all'indice utilizzato per accedere agli elementi di un array. Inoltre, dovresti verificare se ci sono altri errori presenti nel codice che possono influenzare il comportamento del programma, come la corretta apertura e chiusura di file.  
(lldb) █

**Figura 4.33.** Debugging di ii\_test\_3.c con invio delle informazioni di debugging a ChatGPT

#### 4.4.4 Test 4

In questo esempio, già trattato nella sperimentazione precedente (test\_15.c), si ha una ricorsione infinita a causa del calcolo del fattoriale di un numero negativo. Ciò avviene in quanto non ci si può ricondurre al caso base. Oltre al file “ii\_test\_4.c”, andiamo ad utilizzare, includendolo con la direttiva “include “getN.h””, il file di intestazione “getN.h” nel quale riportiamo il prototipo della funzione “getN()” definita nel file sorgente “getN.c”. Sarà la funzione “getN()” a darci il valore, qui negativo, del quale calcolare il fattoriale, funzione che purtroppo è definita in un file sorgente esterno, pertanto dal solo codice l’errore risulta mascherato per mascheramento di input. Differenziamo i responsi di ChatGPT:

- **Responso per l’invio del solo codice:** ChatGPT non rileva l’errore principale e fa considerazioni irrilevanti specificando essa stessa, tra l’altro, che non è in grado di identificare errori relativi al file “get.h” in quanto il suo contenuto non è stato fornito.
- **Responso per l’invio delle informazioni di debugging:** ChatGPT sa che l’esecuzione del programma è stata interrotta da un “segmentation fault” (segnalato dal segnale “SIGSEGV”). Da qui riconduce la causa dell’errore al valore “-5”, assegnato alla variabile n, il quale essendo negativo genera una ricorsione infinita.

In questo caso le informazioni di debugging sono utili poiché consentono una corretta identificazione dell’errore, informazioni che ChatGPT stessa richiede al fine di verificare la correttezza del programma.

```

1 #include <stdio.h>
2 #include "getN.h"
3
4 int factorial(int n) {
5     if (n == 0) {
6         return 1;
7     } else {
8         return n * factorial(n - 1);
9     }
10}
11
12 int main() {
13     int n = getN();
14     int result = factorial(n);
15     printf("Factorial of %d is %d\n", n, result);
16
17     return 0;
18}
```

**Listing 4.32.** ii\_testing/programs/ii\_test\_4.c

```

1 ifndef GETN_H
2 define GETN_H
3
4 int getN();
```

```
6 #endif
```

**Listing 4.33.** ii\_testing/programs/getN.h

```
1 int getN() {
2     return -5;
3 }
```

**Listing 4.34.** ii\_testing/programs/getN.c

```
carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_4
(lldb) target create "ii_test_4"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_4' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 108957 exited with status = 9 (0x00000009)
----- Execution Terminated -----
Sono presenti alcuni errori nel codice sorgente.
Ecco una lista degli errori identificati e la loro tipologia:

1. Errore di inclusione:
   - Linea 2: L'header file "getN.h" viene incluso con le virgolette (" ") anziché con i segni di maggiore o minore (< >). Per includere un header file standard, è necessario utilizzare i segni di maggiore o minore.

2. Errore di definizione e implementazione:
   - Linee 1-2: La sintassi utilizzata per includere gli header files è scorretta.
   a. L'header file "stdio.h" dovrebbe essere incluso tra i segni di maggiore o minore (< >), mentre l'header file "getN.h" dovrebbe essere incluso tra doppi apici (" ").
```

(a) Debugging di ii\_test\_4.c con invio del solo codice a ChatGPT

```
- Linee 4-10: La definizione e l'implementazione della funzione "factorial" sembrano corrette dal punto di vista del codice, ma potrebbero essere soggette a errori di logica. Assicurati di testare attentamente la funzione per verificare che la logica sia corretta.

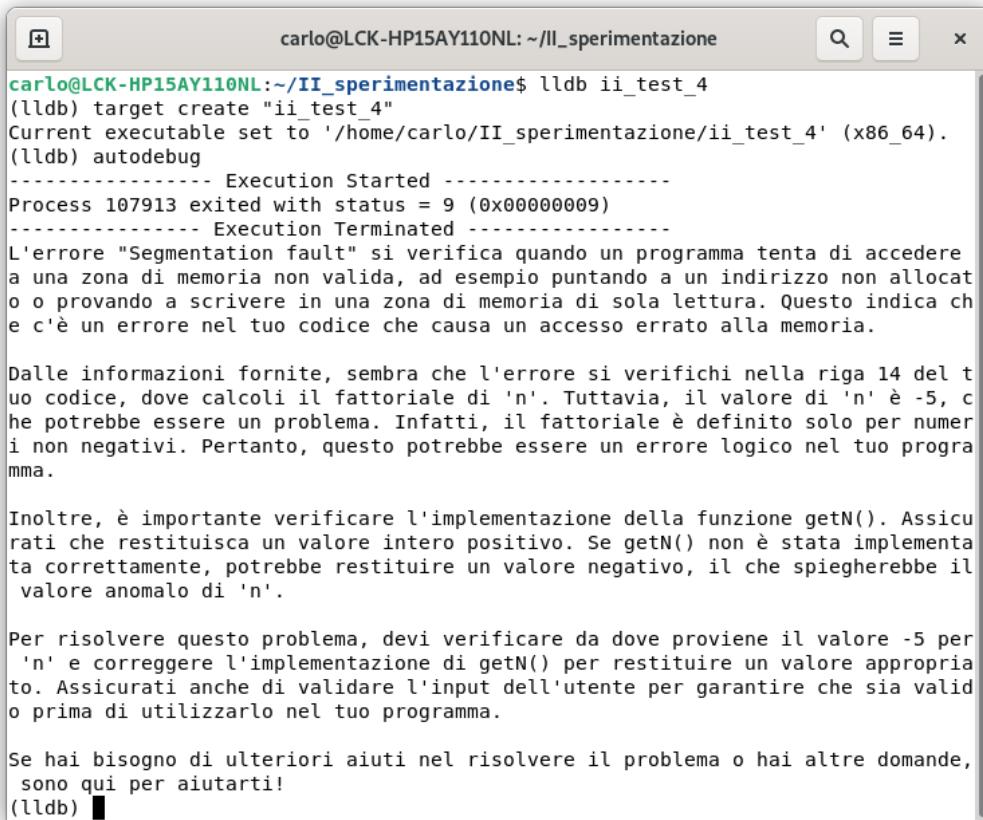
3. Errore di chiamata di funzione:
   - Linea 13: La funzione "getN()" viene chiamata senza l'argomento richiesto. Potresti voler passare un argomento specifico in base alla definizione della funzione "getN" nel file "getN.h".

4. Errore di formato di stampa:
   - Linea 15: Il formato di stampa utilizzato nella funzione "printf" sembra corretto, ma assicurati che il tipo di dato dell'argomento "n" sia compatibile con "%d" nel caso specifico.

5. Nessuna dichiarazione del file "getN.h" presente:
   - Non sono in grado di identificare gli eventuali errori o problemi relativi al file "getN.h" in quanto il suo contenuto non è stato fornito.
(lldb) █
```

(b) Debugging di ii\_test\_4.c con invio del solo codice a ChatGPT (continuo)

**Figura 4.34.** Debugging di ii\_test\_4.c con invio del solo codice a ChatGPT



```
carlo@LCK-HP15AY110NL:~/II_sperimentazione$ lldb ii_test_4
(lldb) target create "ii_test_4"
Current executable set to '/home/carlo/II_sperimentazione/ii_test_4' (x86_64).
(lldb) autodebug
----- Execution Started -----
Process 107913 exited with status = 9 (0x00000009)
----- Execution Terminated -----
L'errore "Segmentation fault" si verifica quando un programma tenta di accedere a una zona di memoria non valida, ad esempio puntando a un indirizzo non allocato o provando a scrivere in una zona di memoria di sola lettura. Questo indica che c'è un errore nel tuo codice che causa un accesso errato alla memoria.

Dalle informazioni fornite, sembra che l'errore si verifichi nella riga 14 del tuo codice, dove calcoli il fattoriale di 'n'. Tuttavia, il valore di 'n' è -5, che potrebbe essere un problema. Infatti, il fattoriale è definito solo per numeri non negativi. Pertanto, questo potrebbe essere un errore logico nel tuo programma.

Inoltre, è importante verificare l'implementazione della funzione getN(). Assicurati che restituisca un valore intero positivo. Se getN() non è stata implementata correttamente, potrebbe restituire un valore negativo, il che spiegherebbe il valore anomalo di 'n'.

Per risolvere questo problema, devi verificare da dove proviene il valore -5 per 'n' e correggere l'implementazione di getN() per restituire un valore appropriato. Assicurati anche di validare l'input dell'utente per garantire che sia valido prima di utilizzarlo nel tuo programma.

Se hai bisogno di ulteriori aiuti nel risolvere il problema o hai altre domande, sono qui per aiutarti!
(lldb) █
```

**Figura 4.35.** Debugging di ii\_test\_4.c con invio delle informazioni di debugging a ChatGPT

## 4.5 Analisi dei risultati

Sono state effettuate ben due sperimentazioni. Vediamo per ciascuna di esse quali considerazioni se ne possono trarre:

- **I Sperimentazione.** La prima sperimentazione prende in esame 27 programmi che vanno a istanziare ben 15 tipologie di errori differenti. Il responso di ChatGPT si è qui basato sulle informazioni di debugging recuperate e inviate attraverso l'applicativo sviluppato. Lo scopo della sperimentazione è rilevare le capacità di ChatGPT di scovare i bug nel codice sorgente di ciascun programma preso in esame grazie alle informazioni ricavate attraverso l'applicativo realizzato. Per ogni esempio è stato considerato soltanto il primo responso del ChatBot applicando quindi una politica di sperimentazione di tipo ‘first try’. Si osserva (Tabella 4.1) come: nel 55,6% dei casi il problema sia stato rilevato, identificando quindi l'errore senza ambiguità e senza riportare l'esistenza di errori non sussistenti; nel 37% dei casi il problema è rilevato in modo impreciso, ovvero è stato sì identificato l'errore, ma sono state considerate come erronee anche altre parti del codice sorgente che risultavano corrette oppure l'errore evinto dal responso non è stato identificato con chiarezza; mentre nel restante 7,4% dei casi il problema non è stato trovato. La sperimentazione rileva quindi una buona capacità dell'IA di OpenAI di rilevare i problemi del codice grazie alle informazioni di debugging fornite, ChatGPT si presenta come uno strumento valido per il debugging del codice.
- **II Sperimentazione.** La seconda sperimentazione prende in esame 4 programmi che vanno ad istanziare 4 tipologie di errore già analizzate nella sperimentazione precedente. Questi errori vengono mascherati per mascheramento di input e per mascheramento di calcolo così da osservare meglio quando le informazioni di debugging possano pesare ai fini della rilevazione degli errori partendo dalla prospettiva che nella vita reale non sempre gli errori sono evidenti dal solo codice, ma spesso si manifestano quando si riceve un input inaspettato o vi è un'alterazione di alcuni valori. Come specificato nell'introduzione alla II sperimentazione sono stati scelti programmi in cui l'errore causa una terminazione improvvisa in ragione del fatto che in questo caso il comando ‘autodebug’ bloccherà l'esecuzione alla linea antecedente la terminazione improvvisa collezionando tutte informazioni di debugging raccoglibili in modo automatico così da effettuare un confronto chiaro di quando si utilizzano tutte le informazioni di debugging rese disponibili dall'applicativo e quando si utilizza il solo codice. La differenza sostanziale con la I sperimentazione, infatti, sta nel fatto che qui si va ad osservare il responso di ChatGPT in base a due tipi di prompt inviati ad essa in modo alternativo. Prima viene inviato soltanto il codice sorgente del programma oggetto del debugging, poi vengono inviate anche tutte le altre informazioni di debugging recuperate attraverso l'applicativo implementato. Lo scopo della seconda sperimentazione è valutare i vantaggi dell'utilizzo del plugin partendo dall'osservazione di come cambia il responso di ChatGPT con l'invio di ulteriori informazioni. Si rileva, infatti, quanto l'efficacia dell'IA sviluppata da OpenAI risulta maggiore con l'invio

| Tipologia di errore                   | Programma | Problema non trovato | Problema rilevato in modo impreciso | Problema rilevato |
|---------------------------------------|-----------|----------------------|-------------------------------------|-------------------|
| Logica, sintassi, semantica           | test_1.c  |                      |                                     | ✓                 |
| Logica, sintassi, semantica           | test_2.c  | ✓                    |                                     |                   |
| Logica, sintassi, semantica           | test_3.c  | ✓                    |                                     |                   |
| Logica, sintassi, semantica           | test_4.c  |                      |                                     | ✓                 |
| Logica, sintassi, semantica           | test_5.c  |                      | ✓                                   |                   |
| Buffer overflow                       | test_6.c  |                      | ✓                                   |                   |
| Buffer overflow                       | test_7.c  |                      | ✓                                   |                   |
| Divisione per zero                    | test_8.c  |                      |                                     | ✓                 |
| Accesso a memoria non allocata        | test_9.c  |                      |                                     | ✓                 |
| Accesso a memoria non allocata        | test_10.c |                      |                                     | ✓                 |
| Accesso a memoria non allocata        | test_11.c |                      |                                     | ✓                 |
| Puntatori nulli                       | test_12.c |                      |                                     | ✓                 |
| Overflow aritmetico                   | test_13.c |                      |                                     | ✓                 |
| Underflow aritmetico                  | test_14.c |                      |                                     | ✓                 |
| Ricorsione infinita                   | test_15.c |                      | ✓                                   |                   |
| Accesso a variabili non inizializzate | test_16.c |                      |                                     | ✓                 |
| Accesso a variabili non inizializzate | test_17.c |                      | ✓                                   |                   |
| Accesso a variabili non inizializzate | test_18.c |                      |                                     | ✓                 |
| Accesso a variabili non inizializzate | test_19.c |                      | ✓                                   |                   |
| Accesso fuori dai limiti dell'array   | test_20.c |                      | ✓                                   |                   |
| Accesso fuori dai limiti dell'array   | test_21.c |                      | ✓                                   |                   |
| Accesso fuori dai limiti dell'array   | test_22.c |                      | ✓                                   |                   |
| File non trovato o non accessibile    | test_23.c |                      |                                     | ✓                 |
| Comportamento non definito            | test_24.c |                      | ✓                                   |                   |
| Comportamento non definito            | test_25.c |                      |                                     | ✓                 |
| Loop infiniti                         | test_26.c |                      |                                     | ✓                 |
| Loop infiniti                         | test_27.c |                      |                                     | ✓                 |
|                                       | Risultato | 2/27                 | 10/27                               | 15/27             |

**Tabella 4.1.** Resoconto dei risultati ottenuti nel corso della I sperimentazione

delle informazioni di debugging recuperate dall'applicativo. In questi 4 esempi si nota (Tabella 4.2) che qualcosa vi sia:

- **Invio del solo codice:** ChatGPT non riesce mai a centrare il problema senza commettere almeno un errore o riportando delle imprecisioni; centra il problema riportando alcune imprecisioni o errori insussistenti nel 50% dei casi; non riesce a trovare il problema nel restante 50% dei casi.
- **Invio delle informazioni di debugging:** ChatGPT centra il problema correttamente senza ambiguità nel 75% dei casi; mentre centra il problema riportando alcune imprecisioni o errori insussistenti nel restante 25% dei casi; in nessun caso non riesce a trovare il problema.

A ben vedere i vantaggi di utilizzare l'applicativo risultano innegabili: si osserva un incremento del 50% della possibilità di rilevare il problema e un incremento del 75% di rilevarlo senza imprecisioni. Pertanto l'utilizzo dell'applicativo si rivela vantaggioso poiché le informazioni di debugging recuperate dal plugin consentono di centrare meglio il problema, a maggior ragione se gli errori non risultano evidenti. Vi è anche un corollario interessante che può discendere dalle buone prestazioni nonostante il mascheramento di input e di calcolo. Il fatto che l'applicativo risulti vantaggioso quando si dipende da valori che si ricevono in input o subiscono modificazioni per mezzo di operazioni sembra suggerire che l'applicativo risulta tanto più utile quanto si ha a che fare con programmi complessi, quindi facenti parte di un sistema con più file interdipendenti contenenti codice sorgente e in cui la visibilità degli errori è sottesa all'esecuzione del codice. L'applicativo potrebbe quindi rivelarsi un ottimo strumento nelle mani di sviluppatori software che implementano programmi complessi, con delle dipendenze, come quelli che possono essere scritti quando si realizza un sistema operativo oppure un videogioco. In ogni caso ciò emerge di certo dalla seconda sperimentazione, considerando le ragioni della limitatezza dell'(insieme dei dati) (dataset), è che l'utilizzo del plugin per il miglioramento del debugging risulta vantaggioso e consente di centrare i problemi del codice fino al 50% più efficientemente rispetto alla sola analisi del codice operata da un'IA come ChatGPT.

| Invio del solo codice a ChatGPT                 |                      |                                     |                   |
|---|----------------------|-------------------------------------|-------------------|
| Programma                                       | Problema non trovato | Problema rilevato in modo impreciso | Problema rilevato |
| ii_test_1.c                                     | ✓                    |                                     |                   |
| ii_test_2.c                                     |                      | ✓                                   |                   |
| ii_test_3.c                                     |                      | ✓                                   |                   |
| ii_test_4.c                                     | ✓                    |                                     |                   |
| Risultato                                       | 2/4                  | 2/4                                 | 0/4               |
| Invio delle informazioni di debugging a ChatGPT |                      |                                     |                   |
| Programma                                       | Problema non trovato | Problema rilevato in modo impreciso | Problema rilevato |
| ii_test_1.c                                     |                      |                                     | ✓                 |
| ii_test_2.c                                     |                      |                                     | ✓                 |
| ii_test_3.c                                     |                      | ✓                                   |                   |
| ii_test_4.c                                     |                      |                                     | ✓                 |
| Risultato                                       | 0/4                  | 1/4                                 | 3/4               |

**Tabella 4.2.** Resoconto dei risultati ottenuti nel corso della II sperimentazione

L'analisi dei risultati delle sperimentazioni ci porta quindi a dire che l'integrazione di ChatGPT nel debugger si rivela uno strumento utile per scovare gli errori nel codice per una moltitudine di tipologie di errore. Tale strumento, inoltre, risulta vantaggioso

poiché le informazioni di debugging recuperate dall'applicativo consentono di trovare errori più facilmente. Il compromesso (trade-off) migliore risulta quello di utilizzare il plugin per l'analisi di programmi che presentano terminazioni improvvise e nei quali l'errore è mascherato (sia per la ricezione di input o la visibilità sottesa all'esecuzione di calcoli) poiché qui le informazioni di debugging risultano complete e rilevanti alla rilevazione del problema.



# Capitolo 5

## Conclusioni

La tesi presentata dal candidato è stata proposta con l'obiettivo di migliorare il debugging[20]. Per tale scopo è stato realizzato un applicativo in grado di integrare le funzionalità dell'intelligenza artificiale all'interno di un debugger al fine di rendere più efficiente il debugging, facendo da apripista alla possibilità di effettuare un debugging automatico.

Sono state fornite quindi le motivazioni[11] che hanno spinto il candidato ad affrontare questo tema, da qui i requisiti da rispettare per l'assolvimento del compito e un progetto che descrivesse l'applicativo nei suoi aspetti chiave. Da qui l'implementazione del plugin a cui hanno fatto seguito le sperimentazioni per testare le capacità dell'IA integrata e definire i vantaggi di utilizzo dell'applicativo. Dalle sperimentazioni è emerso che ChatGPT si rivela uno strumento utile per l'identificazione dei problemi del codice, l'IA trova i bug nel 92,6% dei casi e li rileva senza ambiguità o evidenziando errori insussistenti nel 55,6% dei casi. Si tratta di un risultato soddisfacente sul quale si può essere ottimisti in futuro visti i continui miglioramenti che OpenAI apporta al ChatBot[22]. Il secondo aspetto emerso dalle sperimentazioni è che le informazioni di debugging recuperate attraverso l'applicativo sono in grado di facilitare la scoperta degli errori da parte di ChatGPT, migliorando la sua efficienza fino al 50% in più rispetto a quanto non si avrebbe se fosse inviato soltanto il codice. Inoltre, il debugging automatico garantito dal comando “autodebug” del plugin consente di ottenere i risultati migliori in quei programmi i cui errori causano terminazioni improvvise e sono mascherati da calcoli complessi o valori presi in altri file sorgente. L'integrazione di ChatGPT in LLDB[4] quindi si rivela utile e l'applicativo che la implementa si presenta come uno strumento vantaggioso per il miglioramento del debugging dei programmi scritti in linguaggio C[17] e derivati.

Per concludere la tesi portata avanti dal candidato ha soddisfatto i requisiti che si era proposta di soddisfare: progettando, implementando e sperimentando l'efficacia di un applicativo in grado di migliorare il debugging.



# Bibliografia

- [1] LLDB Python API. [https://lldb.llvm.org/python\\_reference/](https://lldb.llvm.org/python_reference/). Accessed: 2023-10-10.
- [2] OpenAI Python API. <https://platform.openai.com/docs/guides/gpt>. Accessed: 2023-10-10.
- [3] Selenium Documentation. <https://www.selenium.dev/selenium/docs/api/py/api.html>. Accessed: 2023-10-10.
- [4] The LLDB Debugger. <https://lldb.llvm.org/>. Accessed: 2023-10-10.
- [5] Fiorella Artuso, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Debugging debug information with neural networks. *IEEE Access*, 10:54136–54148, 2022. <https://ieeexplore.ieee.org/document/9779237>.
- [6] Cristian Assaiante, Daniele Cono D’Elia, Giuseppe Antonio Di Luna, and Leonardo Querzoni. Where did my variable go? poking holes in incomplete debug information. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 935–947, 2023. <https://arxiv.org/abs/2211.09568>.
- [7] Aram Bahrini, Mohammadsadra Khamoshifar, Hossein Abbasimehr, Robert J Riggs, Maryam Esmaeili, Rastin Mastali Majdabadkohne, and Morteza Pasehvar. Chatgpt: Applications, opportunities, and threats. In *2023 Systems and Information Engineering Design Symposium (SIEDS)*, pages 274–279. IEEE, 2023. <https://ieeexplore.ieee.org/document/10137850>.
- [8] Jialun Cao, Meiziniu Li, Ming Wen, and Shing-chi Cheung. A study on prompt design, advantages and limitations of chatgpt for deep learning program repair. *arXiv preprint arXiv:2304.08191*, 2023. <https://arxiv.org/abs/2304.08191>.
- [9] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. Who’s debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1034–1045, 2021. <https://dl.acm.org/doi/abs/10.1145/3445814.3446695>.
- [10] Clark Evans, Oren Ben-Kiki, and I döt Net. Yaml ain’t markup language (yaml™) version 1.2.2, 2021. <https://yaml.org/spec/1.2.2/>.

- [11] Robert L. Glass. Persistent software errors, 1981. <https://ieeexplore.ieee.org/document/1702822>.
- [12] Don Gotterbarn. Not all codes are created equal: The software engineering code of ethics, a success story. *Journal of Business Ethics*, 22:81–89, 1999. <https://link.springer.com/article/10.1023/A:1006172527640>.
- [13] Md Asraful Haque and Shuai Li. The potential use of chatgpt for debugging and bug fixing, 2023. <https://publications.eai.eu/index.php/airo/article/view/3276>.
- [14] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: how misclassification impacts bug prediction, 2013. <https://ieeexplore.ieee.org/document/6606585>.
- [15] Mladan Jovanovic and Mark Campbell. Generative artificial intelligence: Trends and prospects. *Computer*, 55(10):107–112, 2022. <https://ieeexplore.ieee.org/document/9903869>.
- [16] Brian W Kernighan and PJ Plauger. The elements of programming style, 1978, 1976. <https://archive.org/details/elementsofprogra00kernrich>.
- [17] Brian W Kernighan and Dennis M Ritchie. The c programming language. 2002. [www.ir.juit.ac.in:8080/jspui/bitstream/123456789/5360/1/Brian%20W%20Kernighan%2C%20Dennis%20M%20Ritchie%20-%20The%20Ans%20C%20Programming%20Language%20.pdf](http://www.ir.juit.ac.in:8080/jspui/bitstream/123456789/5360/1/Brian%20W%20Kernighan%2C%20Dennis%20M%20Ritchie%20-%20The%20Ans%20C%20Programming%20Language%20.pdf).
- [18] Herb Krasner. The cost of poor software quality in the us: A 2022 report. *Consortium for Information Software Quality (CISQ)*, pages 1–61, 2022. [www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/](http://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2022-report/).
- [19] Andreas Kuehn and Milton Mueller. Analyzing bug bounty programs: An institutional perspective on the economics of software vulnerabilities. In *2014 TPRC Conference Paper*, 2014. [https://papers.ssrn.com/sol3/papers.cfm?abstract\\_id=2418812](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2418812).
- [20] Renee McCauley, Sue Fitzgerald, Gary Lewandowski, Laurie Murphy, Beth Simon, Lynda Thomas, and Carol Zander. Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2):67–92, 2008. [www.tandfonline.com/doi/abs/10.1080/089934008021145812](http://www.tandfonline.com/doi/abs/10.1080/089934008021145812).
- [21] S Nyamathulla, P Ratnababu, Nazma Sultana Shaik, et al. A review on selenium web driver with python. *Annals of the Romanian Society for Cell Biology*, pages 16760–16768, 2021. [www.annalsofrscb.ro/index.php/journal/article/view/7087](http://www.annalsofrscb.ro/index.php/journal/article/view/7087).
- [22] R OpenAI. GPT-4 technical report, 2023. <https://arxiv.org/abs/2303.08774>.
- [23] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray,

- John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022. <https://arxiv.org/abs/2203.02155>.
- [24] Veselin Raychev. Learning to find bugs and code quality problems-what worked and what not? In *2021 International Conference on Code Quality (ICQ)*, pages 1–5. IEEE, 2021. <https://ieeexplore.ieee.org/document/9392977>.
- [25] Giampaolo Rodola. psutil — pypi. <https://pypi.org/project/psutil/>.
- [26] John R Searle. Minds, brains, and programs, 1980. [www.cambridge.org/core/journals/behavioral-and-brain-sciences/article/abs/minds-brains-and-programs/DC644B47A4299C637C89772FACC2706A](http://www.cambridge.org/core/journals/behavioral-and-brain-sciences/article/abs/minds-brains-and-programs/DC644B47A4299C637C89772FACC2706A).
- [27] Kiran Sridhar and Ming Ng. Hacking for good: Leveraging hackerone data to develop an economic model of bug bounties. *Journal of Cybersecurity*, 7(1):tyab007, 2021. <https://academic.oup.com/cybersecurity/article/7/1/tyab007/6168453>.
- [28] Mark Summerfield. *Programming in Python 3: a complete introduction to the Python language*. Addison-Wesley Professional, 2010. <https://cs.smu.ca/~porter/csc/227/ProgrammingInPython3.pdf>.
- [29] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. Toward understanding compiler bugs in gcc and llvm. In *Proceedings of the 25th international symposium on software testing and analysis*, pages 294–305, 2016. <https://dl.acm.org/doi/abs/10.1145/2931037.2931074>.
- [30] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 283–294, 2011. <https://dl.acm.org/doi/abs/10.1145/1993498.1993532>.
- [31] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 26–36, 2011. <https://dl.acm.org/doi/abs/10.1145/2025113.2025121>.
- [32] Jeff Younker. Setuptools: Harnessing your code. *Foundations of Agile Python Development*, pages 81–102, 2008. [https://link.springer.com/chapter/10.1007/978-1-4302-0635-4\\_4](https://link.springer.com/chapter/10.1007/978-1-4302-0635-4_4).
- [33] Chenyuan Zhang, Hao Liu, Jiutian Zeng, Kejing Yang, Yuhong Li, and Hui Li. Prompt-enhanced software vulnerability detection using chatgpt. *arXiv preprint arXiv:2308.12697*, 2023. <https://arxiv.org/abs/2308.12697>.