Cpp-Python GDB Debug Helper - Manuale Utente

Table of Contents

1. Introduzione

1.1 Cos'è Cpp-Python GDB Debug Helper?

Il Cpp-Python GDB Debug Helper è un'Interfaccia Utente Grafica (GUI) progettata per migliorare e semplificare il processo di debugging di applicazioni C/C++ utilizzando il GNU Debugger (GDB). Mira a fornire un'esperienza più intuitiva rispetto all'interfaccia a riga di comando di GDB, specialmente per attività come l'ispezione di strutture dati complesse e l'automazione di scenari di debugging ripetitivi.

1.2 A chi è rivolto?

Questo strumento è principalmente destinato agli sviluppatori C/C++ che usano GDB per il debugging e che trarrebbero beneficio da:

- Un'interfaccia visuale per le operazioni comuni di GDB.
- Una più facile ispezione di tipi di dati C++ complessi (strutture, classi, contenitori STL).
- Automazione di sequenze di debugging attraverso profili configurabili.
- Output strutturato dei dump delle variabili in formato JSON o CSV.

1.3 Caratteristiche Principali

- **Debugging Manuale Interattivo:** Avvia GDB, imposta breakpoint, esegui il tuo programma target e ispeziona le variabili.
- **Dumping Avanzato delle Variabili:** Utilizza uno script Python personalizzato per GDB per estrarre lo stato delle variabili C/C++, incluse strutture dati complesse come classi, struct, puntatori, array e `std::string`, in un formato JSON strutturato.
- **Profili di Debug Automatizzati:** Crea, gestisci ed esegui profili di debug. Ogni profilo può definire:
  - Eseguibile target e parametri del programma.
  - Molteplici "azioni" di debug, ognuna specificando un breakpoint, variabili da estrarre, formato di output finale (JSON/CSV), directory di output e pattern per i nomi dei file.
- **Analisi dei Simboli:** Analizza il tuo eseguibile compilato per estrarre informazioni su funzioni, variabili globali, tipi definiti dall'utente e file sorgente. Questi dati aiutano nella configurazione delle azioni di debug.
- **Ispezione Live dello Scope:** Durante la configurazione di un'azione, lo strumento può interrogare GDB in tempo reale per elencare le variabili (locali e argomenti) disponibili a un breakpoint specificato, consentendo una selezione precisa.
- **Ambiente Configurabile:** Imposta i percorsi per GDB, lo script dumper Python personalizzato e vari timeout per le operazioni di GDB.
- **Output Flessibile:** Salva i dati estratti in formati JSON o CSV con nomi di file personalizzabili utilizzando placeholder per una migliore organizzazione.
- **Logging nella GUI:** Visualizza i log dell'applicazione e l'output grezzo di GDB direttamente nell'interfaccia.

---

2. Requisiti di Sistema e Configurazione

2.1 Sistemi Operativi Supportati

- **Windows (Primario):** L'applicazione è sviluppata e testata principalmente su Windows. Utilizza il backend compatibile con Windows di `pexpect` (`PopenSpawn`) per un controllo robusto del processo.
- **Linux/macOS (Sperimentale):** L'applicazione dovrebbe essere compatibile con sistemi Unix-like poiché `pexpect` è multipiattaforma.

2.2 Python

- Python 3.7 o successivo è raccomandato.

2.3 Librerie Python Richieste

Sarà necessario installare le seguenti librerie Python. Puoi installarle usando pip: `pip install pexpect appdirs`

- **`pexpect`**: Per controllare GDB come processo figlio.
- **`appdirs`**: Utilizzato per determinare directory di configurazione e dati utente indipendenti dalla piattaforma (sebbene la configurazione principale sia ora salvata in modo relativo all'applicazione).
- **Tkinter**: Inclusa nelle installazioni standard di Python e utilizzata per la GUI.

2.4 Installazione di GDB

- È richiesta un'installazione funzionante di GNU Debugger (GDB).
- Assicurati che GDB sia aggiunto alla variabile d'ambiente `PATH` del tuo sistema o fornisci il percorso completo all'eseguibile di GDB nella configurazione dell'applicazione.
- Versioni di GDB 8.x e successive sono raccomandate per il miglior supporto allo scripting Python.

2.5 Compilazione della Tua Applicazione Target C/C++

- La tua applicazione C/C++ **deve essere compilata con i simboli di debug**.
- Per GCC/G++ o Clang, usa il flag `-g`: `g++ -g -o mioprogramma mioprogramma.cpp`.

- Evita alti livelli di ottimizzazione (es. `-O2`, `-O3`) se interferiscono con il debug. Considera l'uso di `-Og` (ottimizza per l'esperienza di debug).

---

3. Installazione ed Esecuzione

3.1 Esecuzione da Codice Sorgente

1. Assicurati che tutti i prerequisiti della Sezione 2 siano soddisfatti.
2. Scarica o clona il repository del codice sorgente.
3. Naviga nella directory radice del progetto (`cpp_python_debug`).
4. Esegui lo script principale come un modulo: `python -m cpp_python_debug`

3.2 Esecuzione della Versione Compilata (`--onedir`)

L'applicazione può essere impacchettata in una cartella di distribuzione usando PyInstaller.

1. Decomprimi o copia la cartella di distribuzione (es. `cpp_python_debug`) nella posizione desiderata. Questa cartella è auto-contenuta.
2. All'interno della cartella, trova ed esegui l'eseguibile principale (es. `cpp_python_debug.exe`).
3. Tutti i file generati dall'applicazione (configurazioni, log, dump) verranno creati all'interno di questa cartella, rendendola completamente portabile.

---

4. Struttura di File e Directory

L'applicazione crea e gestisce diversi file e directory. Comprendere questa struttura è fondamentale per trovare le tue configurazioni e i tuoi output.

- **Esecuzione da sorgente:** Tutti i percorsi sono relativi alla directory radice del progetto.
- **Esecuzione da versione compilata:** Tutti i percorsi sono relativi alla cartella che contiene l'eseguibile principale.
- **`config/`**
- **`gdb_debug_gui_settings.v2.json`**: Il file di configurazione principale. Memorizza tutte le tue impostazioni, inclusi percorsi, timeout e tutti i tuoi profili di debug. Il file è in formato JSON.
- **`logs/`**
- **`cpppythondebughelper_gui.log`**: Il file di log principale per l'applicazione GUI stessa. Utile per risolvere problemi della GUI.
- **`gdb_dumper_script_internal.log`**: Un file di log dedicato per lo script `gdb_dumper.py`. È estremamente utile per debuggare problemi che si verificano *all'interno* di GDB durante un dump di variabili.
- **`manual_gdb_dumps/`**: La directory in cui vengono memorizzati i file di dump temporanei (`.gdbdump.json`) dalla scheda "Manual Debug" prima che tu li salvi in una posizione finale.
- **`gdb_dumper_diagnostics/`**: (Opzionale) Se abiliti "Enable Diagnostic JSON Dump to File" nelle impostazioni, questa cartella conterrà una copia JSON grezza di ogni singolo dump di variabile, utile per il debug dello script dumper stesso.
- **`<Profile Output Directory>`**: La directory che specifichi nell'azione di un profilo è dove verranno salvati i file di dump finali (JSON o CSV) per l'esecuzione di quel profilo. L'applicazione creerà qui una sottocartella specifica per l'esecuzione (es. `MieiDump/MioProfilo_20231027_143000/`).

---

5. Guida Rapida

1. **Avvia l'Applicazione** come descritto nella Sezione 3.
2. **Configurazione Iniziale:** Al primo avvio, vai su **Options > Configure Application...**.
   - Nella scheda **Paths & Directories**, vai al tuo eseguibile GDB.
   - (Fortemente Raccomandato) Vai anche allo script `gdb_dumper.py` situato nella sottodirectory `core` del codice sorgente (o `cpp_python_debug/core` nella versione compilata).
   - Clicca **Save**.
3. **Esegui una Sessione di Debug Manuale:**
   - Vai alla scheda **Manual Debug**.
   - Seleziona il tuo eseguibile C/C++ compilato.
   - Inserisci un breakpoint (es. `main`).
   - Clicca **1. Start GDB**.
   - Clicca **2. Set Breakpoint**.
   - Clicca **3. Run Program**.
   - Quando il breakpoint viene raggiunto, inserisci un nome di variabile e clicca **4. Dump Variable**.
   - Osserva la scheda "Parsed JSON/Status Output". Mostrerà un messaggio di stato che conferma il dump e il percorso di un file temporaneo `.gdbdump.json`.
   - I pulsanti **Save as JSON** e **Save as CSV** diventeranno attivi. Usali per salvare i dati catturati in una posizione permanente.

---

6. Panoramica dell'Interfaccia Utente

- **Barra dei Menu**: Accedi a **Options** (Configurazione) e **Profiles** (Gestore Profili).
- **Stato della Configurazione Critica**: Mostra se GDB e lo script dumper sono configurati correttamente.
- **Pannello Modalità (Schede)**: Passa tra **Manual Debug** e **Automated Profile Execution**.
- **Area di Output e Log (Schede)**:
  - `GDB Raw Output`: Comunicazione testuale grezza con il processo GDB.
  - `Parsed JSON/Status Output`: Mostra il payload di stato ricevuto dallo script dumper o formatta JSON semplici.
  - `Application Log`: Messaggi di log relativi alla GUI.
- **Barra di Stato**: Brevi messaggi sullo stato corrente dell'applicazione.

---

7. Finestra di Configurazione (`Options > Configure Application...`)

Qui puoi impostare le impostazioni globali dell'applicazione.

- **Scheda Paths & Directories**: Imposta i percorsi assoluti a `gdb.exe` e `gdb_dumper.py`.
- **Scheda Timeouts**: Configura i timeout (in secondi) per varie operazioni di GDB per evitare che l'applicazione si blocchi.
- **Scheda Dumper Options**: Controlla il comportamento dello script `gdb_dumper.py` (es. profondità di ricorsione, numero massimo di elementi degli array). Puoi anche abilitare i dump diagnostici qui.

---

8. Modalità Debug Manuale in Dettaglio

Questa modalità fornisce un'interfaccia passo-passo per una singola sessione di debug. Il flusso di lavoro chiave da notare è:

- Quando clicchi su **4. Dump Variable**, lo script `gdb_dumper.py` salva lo stato della variabile direttamente in un file temporaneo (es. nella cartella `logs/manual_gdb_dumps/`).
- La GUI riceve solo un **messaggio di stato** che conferma questa azione e il percorso del file temporaneo.
- Devi quindi usare i pulsanti **Save as...** per copiare e convertire questi dati temporanei in una posizione permanente.

---

9. Gestore Profili ed Esecuzione Automatizzata

Questa è la funzionalità principale per l'automazione del debugging.

9.1 Gestore Profili (`Profiles > Manage Profiles...`)

Questa finestra è il centro per creare e gestire i tuoi scenari di debug automatizzati. Un profilo è composto da: 1. **Dettagli Profilo**: Nome, eseguibile target e parametri del programma. 2. **Dati di Analisi Simboli**: Puoi eseguire un'analisi sull'eseguibile target. Lo strumento usa GDB per trovare tutte le funzioni, variabili globali, ecc., e memorizza queste informazioni nel profilo. Questo ti aiuta a configurare le azioni in modo accurato. 3. **Azioni**: Un elenco di azioni di debug.

9.2 Editor di Azioni

Ogni azione definisce un'attività specifica da eseguire a un breakpoint. * **Posizione Breakpoint**: Dove GDB dovrebbe fermarsi. * **Variabili da Dumpare**: Quali variabili ispezionare a quel breakpoint. * **Formato Output**: Formato finale (JSON o CSV). * **Directory di Output**: La directory di base per i file di output. * **Pattern Nome File**: Un modello per nominare i file di output. * **Flusso di Esecuzione**: Se continuare dopo il dump e se eseguire il dump ad ogni hit o solo al primo.

9.3 Flusso di Esecuzione Automatizzata

1. Seleziona un profilo dal menu a tendina nella scheda "Automated Profile Execution".
2. Clicca **Run Profile**.
3. Il `ProfileExecutor` avvia GDB ed esegue il programma.
4. Quando un breakpoint viene raggiunto, l'azione corrispondente viene attivata.
5. Viene invocato lo script `gdb_dumper.py`, che estrae le variabili specificate in file intermedi `.gdbdump.json`.
6. L'applicazione principale elabora quindi questi file intermedi:
   - Se il formato desiderato è **JSON**, rinomina il file secondo il pattern.
   - Se il formato desiderato è **CSV**, legge il JSON, lo converte, salva il nuovo file `.csv` ed elimina il JSON intermedio.
7. Il "Produced Files Log" viene aggiornato in tempo reale con lo stato di ogni file creato.

---

10. Risoluzione Problemi / FAQ

D: GDB non trovato / Problemi con lo script Dumper / Nessun simbolo di debug. R: Assicurati che i percorsi configurati in **Options > Configure Application...** siano corretti. Controlla le schede `Application Log` e `GDB Raw Output` per messaggi di errore specifici da GDB o dallo script dumper.

D: L'applicazione si blocca o va in timeout. R: Il tuo programma target potrebbe richiedere molto tempo. Prova ad aumentare i timeout nella Finestra di Configurazione.

D: `gdb_dumper.py` sta fallendo. Come posso debuggarlo? A: 1. Controlla il file `logs/gdb_dumper_script_internal.log`. È il primo posto dove cercare errori che si verificano all'interno del dumper. 2. Per ancora più dettagli, abilita "**Enable Diagnostic JSON Dump to File**" nelle Opzioni Dumper. Questo salverà una copia JSON grezza di ogni dump nella directory `logs/gdb_dumper_diagnostics/`, permettendoti di vedere esattamente cosa sta producendo il dumper.

---

11. Casi d'Uso / Esempi

11.1 Dumpare un `std::vector`

- **Scenario:** Vuoi ispezionare il contenuto di un `std::vector<MyObject> myVector` ogni volta che viene modificato all'interno di una funzione `processVector`.
- **Setup Profilo:**
  - **Azione 1:** Breakpoint all'inizio di `processVector`.
  - **Azione 2:** Breakpoint alla fine di `processVector`.
  - Entrambe le azioni estraggono la variabile `myVector`.
- **Risultato:** Durante l'esecuzione del profilo, verranno creati file come `dump_vettori/MioProfilo_timestamp/processVector_myVector_timestamp.json` (o `.csv`), permettendoti di vedere lo stato del vettore prima e dopo l'elaborazione.

11.2 Tracciare una variabile globale

- **Scenario:** Devi monitorare come una variabile globale `globalCounter` cambia in punti chiave della tua applicazione.
- **Setup Profilo:** Crea più azioni, ognuna con un diverso breakpoint (es. `func_A`, `func_B`, `main.cpp:150`), ma tutte che estraggono la stessa variabile `globalCounter`.
- **Risultato:** Otterrai una serie di file con timestamp, uno per ogni volta che il contatore è stato estratto, permettendoti di tracciare il suo valore attraverso il flusso di esecuzione del programma.

11.3 Snapshot di dati complessi

- **Scenario:** La tua applicazione ha un grande oggetto di configurazione o di stato (`ApplicationState appState`) e vuoi fare uno snapshot completo di esso in un punto critico, come poco prima di un'attività di lunga durata.
- **Setup Profilo:** Un'azione a `longRunningTask.cpp:75` che estrae l'oggetto `appState`.
- **Risultato:** Verrà creato un file JSON dettagliato come `snapshot_stato/MioProfilo_timestamp/longRunningTask.cpp_75_appState_timestamp.json`, contenente una rappresentazione completa e annidata dello stato della tua applicazione.

---

12. Avanzato: Lo Script `gdb_dumper.py`

12.1 Ruolo e Interazione con GDB

Lo script `gdb_dumper.py` è il cuore del motore di estrazione dati. Viene eseguito all'interno del processo GDB e ha accesso all'API Python di GDB.

- **Logica di Serializzazione:**
  1. Usa `gdb.parse_and_eval()` per ottenere un oggetto `gdb.Value` che rappresenta una variabile C++.
  2. Attraversa ricorsivamente questo oggetto, rispettando le "Opzioni Dumper" (profondità massima, ecc.).
  3. Costruisce una rappresentazione Python (dizionario/lista) dei dati C++.
  4. Serializza questo oggetto Python in una stringa JSON.
  5. Salva la stringa JSON completa direttamente in un file intermedio specificato (es. `...nome.gdbdump.json`).
  6. Stampa un piccolo **payload di stato JSON** (che indica successo/fallimento e il percorso scritto) sull'output standard di GDB, racchiuso tra delimitatori speciali.
- **Elaborazione della GUI:** La GUI principale cattura questo payload di stato per capire l'esito del dump. In **Modalità Profilo**, elabora quindi il file intermedio per creare l'output finale specificato dall'utente (rinominando per JSON, convertendo per CSV).

12.2 File di Log del Dumper (`gdb_dumper_script_internal.log`)

Questo file di log, situato nella directory principale `logs`, è prezioso per il debug dello script dumper stesso. Registra passaggi interni, configurazioni ed errori che si verificano all'interno dell'ambiente GDB, che non sono visibili nel log principale dell'applicazione.

---

13. Appendice: Placeholder per Nomi File

I seguenti placeholder possono essere usati nel campo "Filename Pattern" (nell'Editor Azioni) per costruire il nome base dei tuoi file di

output. L'estensione finale del file (`.json` o `.csv`) è gestita automaticamente.

- `{profile_name}`: Il nome del profilo (sanificato per la sicurezza del filesystem).
- `{app_name}`: Il nome base dell'eseguibile target.
- `{breakpoint}`: La stringa della posizione del breakpoint (sanificata).
- `{variable}`: Il nome della variabile/espressione estratta (sanificato).
- `{timestamp}`: Un timestamp dettagliato (`ANNO MESE GIORNO_ORA MINUTO SECONDO_ms`).

Pattern Esempio: `dump_{app_name}_{breakpoint}_{variable}_{timestamp}` **Output Finale Esempio (se JSON):** `dump_mioprogramma_main_miaVar_20231027_143005_123.json` **File Dump Intermedio GDB Esempio:** `dump_mioprogramma_main_miaVar_20231027_143005_123.gdbdump.json`