Cpp-Python GDB Debug Helper - User Manual

Table of Contents

- 1. Introduction
- 2. System Requirements & Setup
- 3. Installation and Execution
- 4. File and Directory Structure
- 5. Quick Start Guide
- 6. User Interface Overview
- 7. Configuration Window (Options > Configure Application...)
- 8. Manual Debug Mode in Detail
- 9. Profile Manager & Automated Execution
- 10. Troubleshooting / FAQ
- 11. Use Cases / Examples
- 12. Advanced: The gdb_dumper.py Script
- 13. Appendix: Filename Placeholders

1. Introduction

1.1 What is Cpp-Python GDB Debug Helper?

The Cpp-Python GDB Debug Helper is a Graphical User Interface (GUI) designed to enhance and simplify the process of debugging C/C++ applications using the GNU Debugger (GDB). It aims to provide a more user-friendly experience compared to the GDB command-line interface, especially for tasks like inspecting complex data structures and automating repetitive debugging scenarios.

1.2 Who is it for?

This tool is primarily aimed at C/C++ developers who use GDB for debugging and would benefit from:

- A visual interface for common GDB operations.
- Easier inspection of complex C++ data types (structs, classes, STL containers).
- Automation of debugging sequences through configurable profiles.
- Structured output of variable dumps in JSON or CSV formats.

1.3 Key Features

- **Interactive Manual Debugging:** Start GDB, set breakpoints, run your target program, and inspect variables.
- **Advanced Variable Dumping:** Utilizes a custom GDB Python script to dump the state of C/C++ variables, including complex data structures like classes, structs, pointers, arrays, and `std::string`, into a structured JSON format.
- **Automated Debug Profiles:** Create, manage, and execute debug profiles. Each profile can define:
  - Target executable and program parameters.
  - Multiple debug "actions", each specifying a breakpoint, variables to dump, final output format (JSON/CSV), output directory, and filename patterns.
- **Symbol Analysis:** Analyze your compiled executable to extract information about functions, global variables, user-defined types, and source files. This data aids in configuring debug actions.
- **Live Scope Inspection:** When configuring an action, the tool can query GDB live to list variables (locals and arguments) available at a specified breakpoint, allowing for precise selection.
- **Configurable Environment:** Set paths for GDB, the custom Python dumper script, and various timeouts for GDB operations.
- **Flexible Output:** Save dumped data in JSON or CSV formats with customizable filenames using placeholders for better organization.
- **GUI Logging:** View application logs and raw GDB output directly within the interface.

---

2. System Requirements & Setup

2.1 Supported Operating Systems

- **Windows (Primary):** The application is primarily developed and tested on Windows. It uses the `pexpect` library's Windows-compatible backend (`PopenSpawn`) for robust process control.
- **Linux/macOS (Experimental):** The application should be compatible with Unix-like systems as `pexpect` is cross-platform.

2.2 Python

- Python 3.7 or newer is recommended.

2.3 Required Python Libraries

You will need to install the following Python libraries. You can install them using pip: `pip install pexpect appdirs`

- **`pexpect`**: For controlling GDB as a child process.
- **`appdirs`**: Used for determining platform-independent user configuration and data directories (though the primary configuration is now stored relative to the application).
- **Tkinter**: This is included with standard Python installations and is used for the GUI.

2.4 GDB Installation

- A working installation of the GNU Debugger (GDB) is required.
- Ensure that GDB is either added to your system's `PATH` environment variable or you provide the full path to the GDB executable in the application's configuration.
- GDB versions 8.x and newer are recommended for the best Python scripting support.

2.5 Compiling Your C/C++ Target Application

- Your C/C++ application **must be compiled with debugging symbols**.
- For GCC/G++ or Clang, use the `-g` flag: `g++ -g -o myprogram myprogram.cpp`.
- Avoid high levels of optimization (e.g., `-O2`, `-O3`) if they interfere with debugging. Consider using `-Og` (optimize for the debug experience).

---

3. Installation and Execution

3.1 Running from Source Code

1. Ensure all prerequisites from Section 2 are met.
2. Download or clone the source code repository.
3. Navigate to the root directory of the project (`cpp_python_debug`).
4. Run the main script as a module: `python -m cpp_python_debug`

3.2 Running the Compiled (`--onedir`) Version

The application can be packaged into a distribution folder using PyInstaller.

1. Unzip or copy the distribution folder (e.g., `cpp_python_debug`) to your desired location. This folder is self-contained.
2. Inside the folder, find and run the main executable (e.g., `cpp_python_debug.exe`).
3. All files generated by the application (configurations, logs, dumps) will be created inside this folder, making it fully portable.

---

4. File and Directory Structure

The application creates and manages several files and directories. Understanding this structure is key to finding your configurations and output.

- **When running from source:** All paths are relative to the project's root directory.
- **When running the compiled version:** All paths are relative to the folder containing the main executable.
- **`config/`**
- **`gdb_debug_gui_settings.v2.json`**: The main configuration file. It stores all your settings, including paths, timeouts, and all your debug profiles. This file is in JSON format.
- **`logs/`**
- **`cpppythondebughelper_gui.log`**: The main log file for the GUI application itself. Useful for troubleshooting GUI issues.
- **`gdb_dumper_script_internal.log`**: A dedicated log file for the `gdb_dumper.py` script. This is extremely useful for debugging issues that occur *inside* GDB during a variable dump.
- **`manual_gdb_dumps/`**: The directory where temporary dump files (`.gdbdump.json`) from the "Manual Debug" tab are stored before you save them to a final location.
- **`gdb_dumper_diagnostics/`**: (Optional) If you enable "Enable Diagnostic JSON Dump to File" in the settings, this folder will contain a raw JSON copy of every single variable dump, which is useful for debugging the dumper script itself.
- **`<Profile Output Directory>`**: The directory you specify in a profile's action is where the final dump files (JSON or CSV) for that profile run will be saved. The application will create a run-specific subfolder here (e.g., `MyDumps/MyProfile_20231027_143000/`).

---

5. Quick Start Guide

1. **Launch the Application** as described in Section 3.
2. **Initial Configuration:** On first launch, go to **Options > Configure Application...**.
    - In the **Paths & Directories** tab, browse to your GDB executable.
    - (Strongly Recommended) Also browse to the `gdb_dumper.py` script located in the `core` subdirectory of the source code (or `cpp_python_debug/core` in the compiled version).
    - Click **Save**.
3. **Your First Manual Debug Session:**
    - Go to the **Manual Debug** tab.
    - Select your compiled C/C++ executable.
    - Enter a breakpoint (e.g., `main`).
    - Click **1. Start GDB**.
    - Click **2. Set Breakpoint**.
    - Click **3. Run Program**.
    - When the breakpoint is hit, enter a variable name and click **4. Dump Variable**.
    - Observe the "Parsed JSON/Status Output" tab. It will show a status message confirming the dump and the path to a temporary `.gdbdump.json` file.
    - The **Save as JSON** and **Save as CSV** buttons will become active. Use them to save the captured data to a permanent location.

---

6. User Interface Overview

6.1 Menu Bar

- **Options:** "Configure Application...", "Exit".
- **Profiles:** "Manage Profiles...".

6.2 Critical Configuration Status Area

Displays status of GDB executable and Dumper script. Includes a "Configure..." button.

6.3 Mode Panel (Tabs)

- **Manual Debug Tab:** For interactive, step-by-step debugging.
- **Automated Profile Execution Tab:** For running pre-configured debug sequences.

6.4 Output and Log Area (Tabs)

- **GDB Raw Output Tab:** Raw text communication with the GDB process.
- **Parsed JSON/Status Output Tab:** Displays the status payload received from the GDB dumper script or pretty-prints simple JSON.
- **Application Log Tab:** Log messages from the GUI application itself.

6.5 Status Bar

Brief messages about the application's current state or last operation.

---

7. Configuration Window (`Options > Configure Application...`)

Organized into tabs:

7.1 Paths & Directories Tab

- **GDB Executable Path:** Full path to GDB. *Crucial.*
- **GDB Python Dumper Script Path:** Full path to `gdb_dumper.py`. *Strongly recommended for full functionality.*

7.2 Timeouts Tab (seconds)

Configure timeouts for GDB operations: GDB Start, GDB Command, Program Run/Continue, Dump Variable, Kill Program, GDB Quit.

7.3 Dumper Options Tab

Control the behavior of `gdb_dumper.py`: Max Array Elements, Max Recursion Depth, Max String Length, and options for diagnostic logging.

---

8. Manual Debug Mode in Detail

This mode provides a step-by-step interface for a single debug session.

8.1 Workflow 1. **Set Target & Parameters:** Specify the executable and any command-line arguments. 2. **Set Breakpoint & Variable:** Define where to stop and what to inspect. 3. **Control Session:** Use the numbered buttons (`1. Start GDB`, `2. Set Breakpoint`, `3. Run Program`, `4. Dump Variable`, `Stop GDB`) to control the flow. 4. **Dump Data:** The "Dump Variable" action saves the variable's state to a temporary `.gdbdump.json` file. 5. **Save Data:** After a successful dump, the "Save as..." buttons become active, allowing you to save the captured data permanently as JSON or CSV.

8.2 Interpreting Output * **GDB Raw Output:** Shows all communication with GDB, including the status message from the dumper script. * **Parsed JSON/Status Output:** Displays the status payload from the dumper, confirming the action and providing the path to the temporary file.

---

9. Profile Manager & Automated Execution

This is the core feature for automating debugging.

9.1 Profile Manager (`Profiles > Manage Profiles...`)

This window is the hub for creating and managing your automated debug scenarios. A profile consists of: 1. **Profile Details**: Name, target executable, and program parameters. 2. **Symbol Analysis Data**: You can run an analysis on the target executable. The tool uses GDB to find all functions, global variables, etc., and stores this information in the profile. This helps you accurately set up actions. 3. **Actions**: A list of debug actions.

9.2 Action Editor

Each action defines a specific task to be performed at a breakpoint. * **Breakpoint Location**: Where GDB should stop. * **Variables to**

Dump: Which variables to inspect. * **Output Format**: Final format (JSON or CSV). * **Output Directory**: The base directory for the output files. * **Filename Pattern**: A template for naming the output files. * **Execution Flow**: Whether to continue after the dump and whether to dump on every hit or just the first.

9.3 Automated Execution Flow

1. Select a profile from the dropdown on the "Automated Profile Execution" tab.
2. Click **Run Profile**.
3. The `ProfileExecutor` starts GDB and runs the program.
4. When a breakpoint is hit, the corresponding action is triggered.
5. The `gdb_dumper.py` script is invoked, which dumps the specified variable(s) to intermediate `.gdbdump.json` files.
6. The main application then processes these intermediate files:
    - If the desired format is **JSON**, it renames the file according to the pattern.
    - If the desired format is **CSV**, it reads the JSON, converts it, saves the new `.csv` file, and deletes the intermediate JSON.
7. The "Produced Files Log" is updated in real-time with the status of each file created.

---

10. Troubleshooting / FAQ

Q: GDB not found / Dumper script issues / No debugging symbols. A: Ensure your configured paths in **Options > Configure Application...** are correct. Check the `Application Log` and `GDB Raw Output` tabs for specific error messages from GDB or the dumper script.

Q: The application hangs or times out. A: Your target program might be taking a long time. Try increasing the timeouts in the Configuration Window.

Q: How can I get more debug information from `gdb_dumper.py`? A: 1. Check the `logs/gdb_dumper_script_internal.log` file. This is the first place to look for errors happening inside the dumper. 2. For even more detail, enable "**Enable Diagnostic JSON Dump to File**" in the Dumper Options. This saves a raw JSON copy of every dump to the `logs/gdb_dumper_diagnostics/` directory, allowing you to see exactly what the dumper is producing.

---

11. Use Cases / Examples

11.1 Dumping a `std::vector`

- **Scenario:** You want to inspect the contents of a `std::vector<MyObject> myVector` every time it's modified inside a `processVector` function.
- **Profile Setup:**
    - **Action 1:** Breakpoint at the start of `processVector`.
    - **Action 2:** Breakpoint at the end of `processVector`.
    - Both actions dump the `myVector` variable.
- **Result:** When the profile runs, files like `vector_dumps/MyProfile_timestamp/processVector_myVector_timestamp.json` (or `.csv`) will be created, allowing you to see the state of the vector before and after processing.

11.2 Tracing a Global Variable

- **Scenario:** You need to track how a global variable `globalCounter` changes at different key points in your application.
- **Profile Setup:** Create multiple actions, each with a different breakpoint (e.g., `func_A`, `func_B`, `main.cpp:150`), but all dumping the same variable `globalCounter`.
- **Result:** You will get a series of timestamped files, one for each time the counter was dumped, allowing you to trace its value through the program's execution flow.

11.3 Snapshots of Complex Data

- **Scenario:** Your application has a large configuration or state object (`ApplicationState appState`) and you want to take a complete snapshot of it at a critical point, like just before a long-running task.
- **Profile Setup:** An action at `longRunningTask.cpp:75` that dumps the `appState` object.
- **Result:** A detailed JSON file like `app_state_snapshots/MyProfile_timestamp/longRunningTask.cpp_75_appState_timestamp.json` will be created, containing a full, nested representation of your application's state.

---

12. Advanced: The `gdb_dumper.py` Script

12.1 Role and Interaction with GDB

The `gdb_dumper.py` script is the core of the data extraction engine. It runs within the GDB process and has access to GDB's Python API.

- **Serialization Logic:**

    1. Uses `gdb.parse_and_eval()` to get a `gdb.Value` object representing a C++ variable.
    2. Recursively traverses this object, respecting the "Dumper Options" (max depth, etc.).
    3. Constructs a Python dictionary/list representation of the C++ data.
    4. Serializes this Python object into a JSON string.
    5. Saves the full JSON string directly to a specified intermediate file (e.g., `...name.gdbdump.json`).
    6. Prints a small **status JSON payload** (indicating success/failure and the path written) to GDB's standard output, bracketed by special delimiters.

- **GUI Processing:** The main GUI captures this status payload to understand the outcome of the dump. In **Profile Mode**, it then processes the intermediate file to create the final user-specified output (renaming for JSON, converting for CSV).

12.2 Dumper Log File (`gdb_dumper_script_internal.log`)

This log file, located in the main `logs` directory, is invaluable for debugging the dumper script itself. It records internal steps, configurations, and errors that occur within the GDB environment, which are not visible in the main application log.

---

13. Appendix: Filename Placeholders

The following placeholders can be used in the "Filename Pattern" field (in the Action Editor) to construct the base name of your output files. The final file extension (`.json` or `.csv`) is managed automatically.

- `{profile_name}`: The name of the profile (sanitized for filesystem safety).
- `{app_name}`: Base name of the target executable.
- `{breakpoint}`: The breakpoint location string (sanitized).
- `{variable}`: The variable/expression name being dumped (sanitized).
- `{timestamp}`: A detailed timestamp (`YYYYMMDD_HHMMSS_ms`).

Example Pattern: `dump_{app_name}_{breakpoint}_{variable}_{timestamp}` **Example Final Output (if JSON):** `dump_myprogram_main_myVar_20231027_143005_123.json` **Example Intermediate GDB Dump File:** `dump_myprogram_main_myVar_20231027_143005_123.gdbdump.json`