

GDB Debugging Tutorial: A Comprehensive Guide

Introduction

The GNU Debugger (GDB) is a powerful debugging tool that helps developers analyze and fix problems in their programs. This tutorial covers essential aspects of using GDB, from basic setup to advanced debugging techniques.

Prerequisites

Before starting with GDB debugging, ensure you have:

- GDB installed on your system
- Basic understanding of C/C++ programming
- Access to a terminal/command line interface

Compilation with Debugging Symbols

To effectively use GDB, programs must be compiled with debugging symbols. These symbols provide GDB with crucial information about the program's structure.

```
# Basic compilation with debug symbols
gcc -g program.c -o program

# Compilation with shared libraries
gcc -g program.c -o program -lsome_library
```

The `-g` flag tells the compiler to include debugging information in the executable. Without these symbols, GDB's functionality would be limited.

Starting GDB

There are several ways to start GDB, depending on your needs:

Basic Start

```
gdb program
```

With Text User Interface (TUI)

```
gdb -tui program
```

The TUI mode provides a split-screen interface showing:

- Source code
- Command window
- Program output

With Command Line Arguments

```
# Method 1 (recommended)
gdb --args program arg1 arg2 arg3

# Method 2 (with TUI)
gdb -tui --args program arg1 arg2 arg3
```

Essential GDB Commands

Once inside GDB, you can use these fundamental commands:

Program Execution Controls

- `run` or `r` : Start program execution
- `start` : Begin execution and break at `main()`
- `continue` or `c` : Continue execution
- `next` or `n` : Step over (execute next line)
- `step` or `s` : Step into (enter function calls)
- `finish` : Run until current function returns

Breakpoint Management

- `break function_name` : Set breakpoint at function
- `break file:line` : Set breakpoint at specific line

- `info breakpoints` : List all breakpoints
- `delete breakpoint_number` : Remove specific breakpoint
- `disable breakpoint_number` : Temporarily disable breakpoint
- `enable breakpoint_number` : Re-enable disabled breakpoint

Inspection Commands

- `print variable` : Display variable value
- `backtrace` or `bt` : Show call stack
- `info locals` : Display local variables
- `info args` : Show function arguments
- `whatis variable` : Show variable type

Working with Shared Libraries

When debugging programs that use shared libraries (`.so` files on Linux or `.dll` files on Windows):

1. Compile with library linking:

```
gcc -g program.c -o program -lsome_library
```

2. GDB automatically loads shared libraries during debugging
3. Verify loaded libraries:
 - Use "Modules" tab in TUI mode
 - Use `info shared` command

TUI Mode Navigation

The Text User Interface provides several views:

Key Bindings

- `F5` : Run/Continue program
- `F6` : View variables
- `F7` : Step into function
- `F8` : Step over function

- `F11` : Show backtrace

Layout Controls

- `ctrl-x a` : Toggle TUI mode
- `ctrl-x 1` : Show single window
- `ctrl-x 2` : Show two windows
- `ctrl-L` : Refresh screen

Best Practices

1. Always Compile with Debug Symbols

- Use `-g` flag consistently
- Keep source files accessible

2. Set Strategic Breakpoints

- Place breakpoints at critical code sections
- Use conditional breakpoints for complex scenarios

3. Use TUI Mode for Better Visibility

- Helps track program flow
- Shows source code context

4. Inspect Variables Carefully

- Check variable values at key points
- Watch for unexpected changes

5. Handle Shared Libraries Properly

- Ensure proper linking during compilation
- Verify library loading during debugging

Advanced Features

Watch Points

```
watch variable      # Break when variable changes
rwatch variable     # Break when variable is read
awatch variable     # Break on read or write
```

Conditional Breakpoints

```
break file:line if condition
break function if argc > 1
```

Examining Memory

```
x/nfu address      # Examine memory at address
# n = number of units
# f = format (x/d/u/o/t/a/c/s)
# u = unit size (b/h/w/g)
```

Troubleshooting Common Issues

1. Missing Debug Symbols

- Symptom: Limited debugging information
- Solution: Recompile with `-g` flag

2. Shared Library Not Found

- Symptom: Library loading errors
- Solution: Check library path and linking

3. Source Files Not Found

- Symptom: Cannot display source code
- Solution: Ensure source files are in correct location

Conclusion

GDB is a versatile debugging tool that becomes more powerful as you learn its features. Regular practice with small programs will help build proficiency. Remember to:

- Always compile with debug symbols

- Use TUI mode for better visualisation
- Set strategic breakpoints
- Pay attention to shared library handling
- Keep source files accessible

This understanding will make debugging more efficient and help resolve program issues more effectively.

Advanced GDB Features: A Deep Dive

Conditional Breakpoints

Conditional breakpoints allow you to pause program execution only when specific conditions are met. This is particularly useful when debugging loops, recursive functions, or situations where you want to examine the program state under particular circumstances.

Basic Syntax

```
break location if condition
```

The `location` can be:

- A line number
- A function name
- A file:line combination

The `condition` can be any valid C expression that evaluates to true or false.

Examples with Explanations

Let's look at some practical examples:

Example 1: Breaking on Array Index

```
// Program code
for(int i = 0; i < 100; i++) {
    array[i] = process_data(i);
}

// GDB command
break 5 if i == 50
```

This breakpoint will only trigger when the loop variable `i` equals 50, allowing you to examine the program state at exactly the midpoint of the loop.

Example 2: Breaking on Value Condition

```
struct User {
    int id;
    char* name;
    float balance;
};

void process_user(struct User* user) {
    // Process user data
}

// GDB command
break process_user if user->balance < 0
```

This breakpoint triggers only when processing users with negative balances, helpful for debugging account-related issues.

Example 3: Multiple Conditions

```
void update_status(int status, char* message) {
    // Update system status
}

// GDB command
break update_status if status == 500 && message != 0
```

This breakpoint combines conditions to break only on error status (500) with non-null messages.

Dynamic Condition Updates

You can modify breakpoint conditions during debugging:

```
# List breakpoints
info breakpoints

# Change condition of breakpoint 1
condition 1 new_condition

# Remove condition (makes breakpoint unconditional)
condition 1
```

Examining Memory

Memory examination in GDB allows you to inspect raw memory contents, which is crucial for debugging pointer-related issues, buffer overflows, or understanding data structures at a low level.

Basic Syntax

```
x/nfu address
```

Where:

- **n** = number of units to display
- **f** = display format
- **u** = unit size

Format Specifiers (f)

- **x** = hexadecimal (default)
- **d** = decimal
- **u** = unsigned decimal
- **o** = octal
- **t** = binary
- **a** = address
- **c** = character
- **s** = string
- **i** = instruction

Unit Sizes (u)

- `b` = byte (1 byte)
- `h` = halfword (2 bytes)
- `w` = word (4 bytes)
- `g` = giant word (8 bytes)

Practical Examples

Example 1: Examining String Content

```
char* message = "Hello, World!";

// GDB commands
# Print as string
x/s message
# Output: "Hello, World!"

# Print first 5 characters as bytes
x/5cb message
# Output: 72 'H' 101 'e' 108 'l' 108 'l' 111 'o'
```

Example 2: Examining Integer Array

```
int numbers[] = {1, 2, 3, 4, 5};

// GDB commands
# Print 5 words in decimal
x/5dw numbers
# Output: 1 2 3 4 5

# Print 5 words in hex
x/5xw numbers
# Output: 0x00000001 0x00000002 0x00000003 0x00000004 0x00000005
```

Example 3: Examining Struct Memory Layout

```
struct Point {
    int x;
    int y;
} point = {10, 20};

// GDB commands
# Examine struct memory
x/2dw &point
# Output: 10 20

# Examine byte by byte
x/8cb &point
# Shows individual bytes in memory
```

Advanced Memory Examination Techniques

Following Pointers

```
int* ptr = &value;

// GDB commands
# Print pointer address
x/a &ptr

# Print value at pointer
x/d ptr
```

Examining Memory Regions

```
# Examine 16 bytes starting at address
x/16xb address

# Examine as instructions
x/5i $pc # Current instruction pointer
```

Best Practices for Memory Examination

1. **Address Verification** Always verify addresses before examination:

```
print &variable # Get address
x/nfu &variable # Examine memory
```

2. **Unit Size Selection** Choose appropriate unit size for data type:

- Use `b` for chars
- Use `w` for ints
- Use `g` for long longs/doubles

3. **Format Selection** Select format based on what you're examining:

- Use `x` for raw memory viewing
- Use `d` for numeric values
- Use `s` for strings
- Use `i` for instructions

Common Use Cases

1. Buffer Overflow Detection

```
# Examine memory before suspected overflow
x/32xb buffer

# Continue program
continue

# Re-examine to see changes
x/32xb buffer
```

2. String Manipulation Debugging

```
# View string content
x/s string_ptr

# View raw bytes
x/16xb string_ptr
```

3. Data Structure Investigation

```
# Examine struct layout
x/nfu &struct_var

# Follow pointer chain
x/a pointer
x/nfu *pointer
```

Tips for Effective Usage

1. **Combining Features** Use conditional breakpoints with memory examination:

```
break *address if *(int*)ptr == value
commands
x/nfu examination_address
continue
end
```

2. **Memory Watch Points** Set watchpoints on memory regions:

```
watch *((int*)address)
watch -location buffer # Watch entire buffer
```

3. **Automated Memory Inspection** Create GDB scripts for repeated memory analysis:

```
define inspect_buffer
x/32xb buffer
x/s buffer
end
```

Remember that memory examination requires careful handling of addresses and understanding of data layout. Always verify addresses before examination to avoid crashes or misleading results.

GDB Debugging Tutorial: A Comprehensive Guide

Document History and Attribution

This documentation originated from an interactive conversation with `qwen2.5-coder-3b-instruct-q4_0.gguf` (size: 1.86 GB), running locally through `llama.cpp` (<https://github.com/ggml-org/llama.cpp.git>). The original conversation explored GDB usage fundamentals and advanced debugging techniques. The content has been substantially enhanced, reorganised, and expanded by Claude 3.5 Sonnet (on 2025/February/23) to provide a more comprehensive and structured learning resource.

Key improvements include:

- Detailed explanations of complex concepts
- Additional practical examples and use cases
- Structured progression from basic to advanced topics
- Best practices and troubleshooting guidelines
- Integration of conditional breakpoints and memory examination techniques
- Enhanced code examples with explanatory comments

The initial exchange was obtained in JSON format from "llama.cpp":

```

{
  "conv": {
    "id": "conv-1740318651623",
    "lastModified": 1740320776805,
    "currNode": 1740320261296,
    "name": "Give me a GDB tutorial in simple terms as if I'm a noob. How to
use it, etc."
  },
  "messages": [
    {
      "id": 1740318651623,
      "convId": "conv-1740318651623",
      "type": "root",
      "timestamp": 1740318651623,
      "role": "system",
      "content": "",
      "parent": -1,
      "children": [
        1740318651641
      ]
    },
    {
      "id": 1740318651641,
      "timestamp": 1740318651641,
      "type": "text",
      "convId": "conv-1740318651623",
      "role": "user",
      "content": "Give me a GDB tutorial in simple terms as if I'm a noob. How
to use it, etc.",
      "parent": 1740318651623,
      "children": [
        1740318651701
      ]
    },
    {
      "id": 1740318651701,
      "convId": "conv-1740318651623",
      "type": "text",
      "timestamp": 1740318651701,
      "role": "assistant",
      "content": "Sure! GDB (GNU Debugger) is a powerful tool for debugging
your code. Here's a simple tutorial to help you get started:\n\n### Step 1:
Install GDB\nFirst, you need to install GDB on your system. It usually comes
pre-installed with many Linux distributions. If it's not, you can install it
using your package manager. For example, on Ubuntu, you can run:\n```\nbash\nsudo
apt-get install gdb\n```\n\n### Step 2: Open a Terminal\nOpen your terminal
(Ctrl+Alt+T on most Linux systems).\n\n### Step 3: Compile Your Code\nBefore

```

you can debug, you need to compile your code. For example, if you have a C program named `hello.c`, you can compile it with:

```
bash$ gcc hello.c -o hello
```

Step 4: Run GDB

Start GDB and attach it to your program by running:

```
bash$ gdb hello
```

Step 5: Use GDB Commands

Here are some basic GDB commands you can use:

- **Start the program**:** `run` or `start`
- **Print a variable**:** `print variable_name`
- **Continue execution**:** `continue` or `c`
- **Step into a function**:** `step` or `s`
- **Step over a function**:** `next` or `n`
- **Print current stack trace**:** `backtrace` or `bt`
- **Exit GDB**:** `quit` or `q`

Step 6: Example

Let's say you have the following simple C program:

```
c$ cat main.c
#include <stdio.h>
int main() {
    int num = 10;
    printf("Number is: %d\n", num);
    return 0;
}
```

Compile it:

```
bash$ gcc main.c -o main
```

Run GDB:

```
bash$ gdb main
```

Start the program:

```
bash$ run
```

Print the value of `num`:

```
bash$ print num
```

Continue execution:

```
bash$ continue
```

You should see the output:

```
Number is: 10
```

Step 7: Debugging

If you encounter an error, you can use GDB to step into the function and inspect variables. For example, if you have a function `add` that takes two arguments, you can step into it:

```
bash$ step
```

Print the values of the arguments:

```
bash$ print a
print b
```

Continue execution:

```
bash$ continue
```

Conclusion

GDB is a versatile tool for debugging your code. With basic knowledge of the commands, you can effectively debug even complex programs. Practice by working on small projects to get comfortable with GDB's features."

```

    "parent": 1740318651641,
    "children": [
        1740318769381
    ]
},
{
    "id": 1740318769381,
    "timestamp": 1740318769381,
    "type": "text",
    "convId": "conv-1740318651623",
    "role": "user",
    "content": "What about `gdb -tui`?",
    "parent": 1740318651701,
    "children": [
        1740318769492
    ]
},
{
    "id": 1740318769492,
    "convId": "conv-1740318651623",
    "type": "text",
    "timestamp": 1740318769492,
    "role": "assistant",
    "content": "`gdb -tui` (gdb with text user interface) is a graphical

```

version of GDB that provides a more user-friendly interface. While it's not as powerful as the text-based interface, it can be very helpful for beginners. Here's how to use it:

Step 1: Install GDB

Ensure GDB is installed on your system. On Ubuntu, you can install it using:

```
bash$ sudo apt-get install gdb
```

Step 2: Open a Terminal

Open your terminal.

Step 3: Compile Your Code

Compile your code as shown in the previous examples.

Step 4: Run GDB with Text User Interface

Start GDB with the `-tui` option:

```
bash$ gdb -tui hello
```

Step 5: Use GDB Commands

The GDB text user interface provides a graphical interface to the same commands you used earlier. Here are some key features:

- **Start the program**:** Click the "Run" button or press `F5`.
- **Print a variable**:** Use the "Variables" tab or press `F6`.
- **Continue execution**:** Use the "Run" button or press `F5`.
- **Step into a function**:** Use the "Step" button or press `F7`.
- **Step over a function**:** Use the "Step Over" button or press `F8`.
- **Print current stack trace**:** Use the "Backtrace" button or press `F11`.

Step 6: Example

Let's walk through the same example as before, but now using the text user interface:

Compile the program:

```
bash$ gcc main.c -o main
```

Run GDB:

```
bash$ gdb -tui main
```

Start the program:

- Click the "Run" button or press `F5`.

Print the value of `num`:

- Click the "Variables" tab or press `F6`.

Continue execution:

- Click the "Run" button or press `F5`.

You should see the output:

```
Number is: 10
```

Step 7: Debugging

If you encounter an error, you can use GDB to step into functions and inspect variables. For example, if you have a function `add` that takes two arguments, you can step into it:

- Click the "Step" button or press `F7`.

Print the values of the arguments:

- Click the "Variables" tab or press `F6`.

Continue execution:

- Click the "Run" button or press `F5`.

Conclusion

The text user interface in GDB can be very helpful for beginners. It provides a graphical representation of the commands you can use, making it easier to navigate and use. Practice by working on small projects to get comfortable with GDB's features."

```
,
  "parent": 1740318769381,
  "children": [
    1740318924730
  ]
},
{
  "id": 1740318924730,
  "timestamp": 1740318924730,
  "type": "text",
  "convId": "conv-1740318651623",
  "role": "user",
  "content": "What if a program takes arguments from the main() before starting?",
  "parent": 1740318769492,
  "children": [
    1740318924856
  ]
}
```



```

    },
    {
        "id": 1740318924856,
        "convId": "conv-1740318651623",
        "type": "text",
        "timestamp": 1740318924856,
        "role": "assistant",
        "content": "When working with GDB, especially when a program takes
command-line arguments from the `main()` function, you need to provide these
arguments to GDB when you start the program. This allows GDB to set breakpoints
and inspect the program state as it runs.\n\nHere's how you can start GDB with
command-line arguments:\n\n### Step 1: Compile Your Program\nCompile your
program as shown in the previous examples.\n\n### Step 2: Run GDB with Command-
Line Arguments\nStart GDB with the `-tui` option and provide the command-line
arguments. For example:\n```\nbash\ngdb -tui main arg1 arg2 arg3\n```\n\nReplace
`main` with the name of your program, and `arg1`, `arg2`, `arg3` with the
command-line arguments you want to pass to the program.\n\n### Step 5: Use GDB
Commands\nThe GDB text user interface provides a graphical interface to the
same commands you used earlier. Here are some key features:\n\n- **Start the
program**: Click the `Run` button or press `F5`.\n- **Print a variable**: Use
the `Variables` tab or press `F6`.\n- **Continue execution**: Use the `Run`
button or press `F5`.\n- **Step into a function**: Use the `Step` button or
press `F7`.\n- **Step over a function**: Use the `Step Over` button or press
`F8`.\n- **Print current stack trace**: Use the `Backtrace` button or press
`F11`.\n\n### Step 6: Example\nLet's walk through the same example as before,
but now using the text user interface and passing command-line
arguments:\n\nCompile the program:\n```\nbash\ngcc main.c -o main\n```\n\nRun
GDB:\n```\nbash\ngdb -tui main arg1 arg2 arg3\n```\n\nStart the program:\n- Click
the `Run` button or press `F5`.\n\nPrint the value of `num`:\n- Click the
`Variables` tab or press `F6`.\n\nContinue execution:\n- Click the `Run`
button or press `F5`.\n\nYou should see the output:\n```\nNumber is:
10\n```\n\n### Step 7: Debugging\nIf you encounter an error, you can use GDB to
step into functions and inspect variables. For example, if you have a function
`add` that takes two arguments, you can step into it:\n- Click the `Step`
button or press `F7`.\n\nPrint the values of the arguments:\n- Click the
`Variables` tab or press `F6`.\n\nContinue execution:\n- Click the `Run`
button or press `F5`.\n\n### Conclusion\nWhen working with GDB and a program
that takes command-line arguments, you need to provide these arguments when
starting GDB. This allows you to set breakpoints and inspect the program state
as it runs. Practice by working on small projects to get comfortable with GDB's
features.",
        "parent": 1740318924730,
        "children": [
            1740319191462
        ]
    },
    {
        "id": 1740319191462,

```

```

    "timestamp": 1740319191462,
    "type": "text",
    "convId": "conv-1740318651623",
    "role": "user",
    "content": "However, as far as I can recall if a program being debugged
through GDB takes arguments, the command line argument to start GDB properly
should be `gdb --args program_name arg1 arg2 arg3` or something similar.",
    "parent": 1740318924856,
    "children": [
        1740319191529
    ]
},
{
    "id": 1740319191529,
    "convId": "conv-1740318651623",
    "type": "text",
    "timestamp": 1740319191529,
    "role": "assistant",
    "content": "You're correct. When starting GDB for a program that takes
command-line arguments, you should use the `--args` option followed by the
program name and its arguments. This ensures that GDB knows to pass those
arguments to the program when it runs.\n\nHere's how you can start GDB with
command-line arguments using the `--args` option:\n\n### Step 1: Compile Your
Program\nCompile your program as shown in the previous examples.\n\n### Step 2:
Run GDB with Command-Line Arguments\nStart GDB with the `--args` option and
provide the program name and its arguments. For example:\n`bash\ngdb --args
main arg1 arg2 arg3`\n\nReplace `main` with the name of your program, and
`arg1`, `arg2`, `arg3` with the command-line arguments you want to pass to the
program.\n\n### Step 3: Use GDB Commands\nThe GDB text user interface provides
a graphical interface to the same commands you used earlier. Here are some key
features:\n\n- **Start the program**: Click the `Run` button or press
`F5`.\n- **Print a variable**: Use the `Variables` tab or press `F6`.\n-
**Continue execution**: Use the `Run` button or press `F5`.\n- **Step into a
function**: Use the `Step` button or press `F7`.\n- **Step over a function**:
Use the `Step Over` button or press `F8`.\n- **Print current stack trace**:
Use the `Backtrace` button or press `F11`.\n\n### Step 4: Example\nLet's walk
through the same example as before, but now using the text user interface and
passing command-line arguments:\n\nCompile the program:\n`bash\ngcc main.c -o
main`\n\nRun GDB:\n`bash\ngdb --args main arg1 arg2 arg3`\n\nStart
the program:\n- Click the `Run` button or press `F5`.\n\nPrint the value of
`num`:\n- Click the `Variables` tab or press `F6`.\n\nContinue execution:\n-
Click the `Run` button or press `F5`.\n\nYou should see the
output:\n`Number is: 10`\n\n### Step 5: Debugging\nIf you encounter an
error, you can use GDB to step into functions and inspect variables. For
example, if you have a function `add` that takes two arguments, you can step
into it:\n- Click the `Step` button or press `F7`.\n\nPrint the values of the
arguments:\n- Click the `Variables` tab or press `F6`.\n\nContinue
execution:\n- Click the `Run` button or press `F5`.\n\n### Conclusion\nUsing

```

the `--args` option with GDB ensures that the program receives the specified command-line arguments when it is executed. This is particularly useful when debugging programs that require specific arguments to run correctly. Practice by working on small projects to get comfortable with GDB's features.",

```
    "parent": 1740319191462,
    "children": [
        1740319406576
    ]
},
{
    "id": 1740319406576,
    "timestamp": 1740319406576,
    "type": "text",
    "convId": "conv-1740318651623",
    "role": "user",
    "content": "How can I start the TUI with the same arguments passed to the
program being debugged?",
    "parent": 1740319191529,
    "children": [
        1740319406758
    ]
},
{
    "id": 1740319406758,
    "convId": "conv-1740318651623",
    "type": "text",
    "timestamp": 1740319406758,
    "role": "assistant",
    "content": "To start the GDB text user interface (TUI) with the same
command-line arguments that the program being debugged receives, you can use
the --args option when starting GDB. This ensures that GDB passes the exact
command-line arguments to the program when it starts.\n\nHere's how you can do
it:\n\n### Step 1: Compile Your Program\nCompile your program as shown in the
previous examples.\n\n### Step 2: Run GDB with the TUI and Command-Line
Arguments\nStart GDB with the -tui option and the --args option, providing
the program name and its arguments. For example:\n\n`bash`ngdb -tui --args main
arg1 arg2 arg3\n\nReplace main with the name of your program, and
arg1, arg2, arg3 with the command-line arguments you want to pass to the
program.\n\n### Step 3: Use GDB Commands\nThe GDB text user interface provides
a graphical interface to the same commands you used earlier. Here are some key
features:\n\n- Start the program: Click the Run button or press
F5.\n- Print a variable: Use the Variables tab or press F6.\n-
Continue execution: Use the Run button or press F5.\n- Step into a
function: Use the Step button or press F7.\n- Step over a function:
Use the Step Over button or press F8.\n- Print current stack trace:
Use the Backtrace button or press F11.\n\n### Step 4: Example\nLet's walk
through the same example as before, but now using the text user interface and
passing command-line arguments:\n\nCompile the program:\n\n`bash`ngcc main.c -o
```

```

main\n```\n\nRun GDB:\n```\bash\ngdb -tui --args main arg1 arg2
arg3\n```\n\nStart the program:\n- Click the \"Run\" button or press
`F5`.\n\nPrint the value of `num`:\n- Click the \"Variables\" tab or press
`F6`.\n\nContinue execution:\n- Click the \"Run\" button or press `F5`.\n\nYou
should see the output:\n```\nNumber is: 10\n```\n\n### Step 5: Debugging\nIf
you encounter an error, you can use GDB to step into functions and inspect
variables. For example, if you have a function `add` that takes two arguments,
you can step into it:\n- Click the \"Step\" button or press `F7`.\n\nPrint the
values of the arguments:\n- Click the \"Variables\" tab or press
`F6`.\n\nContinue execution:\n- Click the \"Run\" button or press `F5`.\n\n###
Conclusion\nUsing the `--args` option when starting GDB with the TUI ensures
that the program receives the exact command-line arguments it needs when it
starts. This is particularly useful for debugging programs that require
specific arguments to run correctly. Practice by working on small projects to
get comfortable with GDB's features.",
    "parent": 1740319406576,
    "children": [
        1740319737476
    ]
},
{
    "id": 1740319737476,
    "timestamp": 1740319737476,
    "type": "text",
    "convId": "conv-1740318651623",
    "role": "user",
    "content": "I think debugging symbols must be added to the program while
compiling it, if the intention is to debug it later.",
    "parent": 1740319406758,
    "children": [
        1740319737693
    ]
},
{
    "id": 1740319737693,
    "convId": "conv-1740318651623",
    "type": "text",
    "timestamp": 1740319737693,
    "role": "assistant",
    "content": "Yes, adding debugging symbols to your program is crucial for
effective debugging. Debugging symbols, often referred to as debug information,
provide GDB with the necessary context to understand the program's structure,
variable values, and control flow.\n\nHere's how you can compile your program
with debugging symbols:\n\n### Step 1: Compile Your Program\nCompile your
program with the `-g` option, which enables debugging information. For example,
if you have a C program named `hello.c`, you can compile it with:\n```\bash\ngcc
-g hello.c -o hello\n```\n\nThe `-g` option tells the compiler to include
debugging information in the binary, which is essential for debugging with

```

GDB.\n\n### Step 2: Run GDB with the TUI\nNow that your program is compiled with debugging symbols, you can start GDB with the ``-tui`` option as described in the previous tutorials:\n\n```\nbash\ngdb -tui hello\n```\n\n### Step 3: Use GDB Commands\nYou can now use the GDB text user interface to debug your program. Here are some key features:\n\n- ****Start the program****: Click the `\\"Run\\"` button or press ``F5``.\n- ****Print a variable****: Use the `\\"Variables\\"` tab or press ``F6``.\n- ****Continue execution****: Use the `\\"Run\\"` button or press ``F5``.\n- ****Step into a function****: Use the `\\"Step\\"` button or press ``F7``.\n- ****Step over a function****: Use the `\\"Step Over\\"` button or press ``F8``.\n- ****Print current stack trace****: Use the `\\"Backtrace\\"` button or press ``F11``.\n\n### Step 4: Example\nLet's walk through the same example as before, but now using the text user interface and passing command-line arguments:\n\nCompile the program:\n\n```\nbash\ngcc -g main.c -o main\n```\n\nRun GDB:\n\n```\nbash\ngdb -tui main\n```\n\nStart the program:\n- Click the `\\"Run\\"` button or press ``F5``.\n\nPrint the value of ``num``:\n- Click the `\\"Variables\\"` tab or press ``F6``.\n\nContinue execution:\n- Click the `\\"Run\\"` button or press ``F5``.\n\nYou should see the output:\n\n```\nNumber is: 10\n```\n\n### Step 5: Debugging\nIf you encounter an error, you can use GDB to step into functions and inspect variables. For example, if you have a function ``add`` that takes two arguments, you can step into it:\n- Click the `\\"Step\\"` button or press ``F7``.\n\nPrint the values of the arguments:\n- Click the `\\"Variables\\"` tab or press ``F6``.\n\nContinue execution:\n- Click the `\\"Run\\"` button or press ``F5``.\n\n### Conclusion\nBy compiling your program with the ``-g`` option, you ensure that it includes debugging symbols, which are necessary for effective debugging with GDB. This allows you to inspect variable values, step through code, and understand the program's behavior at a granular level. Practice by working on small projects to get comfortable with GDB's features."

```

    "parent": 1740319737476,
    "children": [
      1740319988643
    ]
  },
  {
    "id": 1740319988643,
    "timestamp": 1740319988643,
    "type": "text",
    "convId": "conv-1740318651623",
    "role": "user",
    "content": "What if the program loads shared libraries (SO/DLL) to
function? What do I have to do then?",
    "parent": 1740319737693,
    "children": [
      1740319988839
    ]
  },
  {
    "id": 1740319988839,
    "convId": "conv-1740318651623",

```

```

    "type": "text",
    "timestamp": 1740319988839,
    "role": "assistant",
    "content": "When working with programs that load shared libraries (e.g.,
`.so` files on Linux or `.dll` files on Windows), you need to ensure that GDB
can access these libraries during debugging. Here's how you can do it:\n\n###
Step 1: Compile Your Program with Debugging Symbols\nCompile your program with
the `-g` option, which enables debugging information and links the program with
its dependencies. For example, if you have a C program named `main.c` that
links against a shared library `libexample.so`, you can compile it
with:\n```\nbash$ gcc -g main.c -o main -lexample\n```\n\nThe `-g` option tells
the compiler to include debugging information in the binary, and `-lexample`
links the program with the `libexample.so` library.\n\n### Step 2: Run GDB with
the TUI\nStart GDB with the `-tui` option as described in the previous
tutorials:\n\n```\nbash$ gdb -tui\n```\n\n### Step 3: Load the Shared
Libraries\nDuring the debugging session, GDB will automatically load the shared
libraries. You can verify this by checking the list of loaded modules:\n\n-
**Load shared libraries**: Use the `\"Modules\"` tab or press `F11`.\n\n-
**List loaded modules**: Use the `\"Loaded Modules\"` tab or press `F12`.\n\n### Step 4:
Use GDB Commands\nYou can now use the GDB text user interface to debug your
program. Here are some key features:\n\n-
**Start the program**: Click the `\"Run\"` button or press `F5`.\n\n-
**Print a variable**: Use the `\"Variables\"`
tab or press `F6`.\n\n-
**Continue execution**: Click the `\"Run\"` button or
press `F5`.\n\n-
**Step into a function**: Click the `\"Step\"` button or press
`F7`.\n\n-
**Step over a function**: Click the `\"Step Over\"` button or press
`F8`.\n\n-
**Print stack trace**: Click the `\"Backtrace\"` button or press
`F11`.\n\n### Step 5: Example\nLet's walk through an example where the program
loads a shared library `libexample.so`:\n\nCompile the program:\n```\nbash$ gcc -
g -l example -o main -lexample\n```\n\nRun GDB:\n```\nbash$ gdb -tui
main\n```\n\nVerify loaded modules:\n-
**Loaded Modules**: Use the `\"Loaded
Modules\"` tab or press `F12`.\n\n### Step 6: Debugging\nIf you encounter an
error, you can use GDB to step into the code and inspect the program's
behavior:\n\n-
**Step into a function**: Click the `\"Step\"` button or press
`F7`.\n\n-
**Inspect variable values**: Use the `\"Variables\"` tab or press
`F6`.\n\n-
**Print stack trace**: Click the `\"Backtrace\"` button or press
`F11`.\n\n### Step 7: Conclusion\nBy compiling your program with debugging
symbols and linking against shared libraries, you ensure that GDB can access
these libraries during debugging. This is particularly important when debugging
programs that depend on shared libraries. Practice by working on small projects
to get comfortable with GDB's features.",
    "parent": 1740319988643,
    "children": [
        1740320261041
    ]
},
{
    "id": 1740320261041,
    "timestamp": 1740320261041,

```

```

    "type": "text",
    "convId": "conv-1740318651623",
    "role": "user",
    "content": "Summarise this conversation in detail, step by step, and please write a tutorial in Markdown format as usual.",
    "parent": 1740319988839,
    "children": [
      1740320261296
    ]
  },
  {
    "id": 1740320261296,
    "convId": "conv-1740318651623",
    "type": "text",
    "timestamp": 1740320261296,
    "role": "assistant",
    "content": "### Summary of the Conversation\n\n1. **Compiling with Debugging Symbols**:\n    - **Command**: `gcc -g main.c -o main -lexample`\n    - **Explanation**: The `-g` flag enables debugging symbols, which are necessary for GDB to provide detailed information about the program's execution.\n\n2. **Running GDB with the TUI**:\n    - **Command**: `gdb -tui main`\n    - **Explanation**: The `-tui` option starts GDB in the text user interface (TUI), which provides a graphical interface similar to the GDB command-line interface.\n\n3. **Loading Shared Libraries**:\n    - **GDB Feature**: During the debugging session, GDB automatically loads the shared libraries specified in the link command.\n    - **Verification**: Use the \"Loaded Modules\" tab or press `F12` to list the loaded modules.\n\n4. **Using GDB Commands**:\n    - **Running Program**: Use the \"Run\" button or press `F5` to start the program execution.\n    - **Printing Variables**: Use the \"Variables\" tab or press `F6` to inspect variable values.\n    - **Continuing Execution**: Use the \"Run\" button or press `F5` to continue execution.\n    - **Stepping Into Functions**: Use the \"Step\" button or press `F7` to step into a function.\n    - **Stepping Over Functions**: Use the \"Step Over\" button or press `F8` to step over a function.\n    - **Printing Stack Trace**: Use the \"Backtrace\" button or press `F11` to print the current stack trace.\n\n5. **Example**:\n    - **Compilation**: `gcc -g -l example -o main -lexample`\n    - **Running GDB**: `gdb -tui main`\n    - **Verification**: Use the \"Loaded Modules\" tab or press `F12` to confirm the shared library is loaded.\n\n6. **Debugging**:\n    - **Step Into Function**: Use the \"Step\" button or press `F7` to step into the `add` function.\n    - **Inspect Variable Values**: Use the \"Variables\" tab or press `F6` to inspect the arguments of the `add` function.\n    - **Print Stack Trace**: Use the \"Backtrace\" button or press `F11` to print the current stack trace.\n\n7. **Conclusion**:\n    - **Efficiency**: Using debugging symbols and loading shared libraries ensures that GDB can provide detailed information during debugging, which is crucial for effective program analysis.\n\n### Tutorial in Markdown Format\n\n#### Step 1: Compiling with Debugging Symbols\n\nTo compile your program with debugging symbols, use the following command:\n\n```\nbash\ngcc -g main.c -o main -lexample\n```\n\n- **Explanation**:
```


The `-g` flag enables debugging symbols, which provide GDB with detailed information about the program's execution.

Step 2: Running GDB with the TUI

Start GDB in the TUI mode by running the following command:

```
bash ngdb -tui main
```

****Explanation**:** The `-tui` option starts GDB in the text user interface, providing a graphical interface similar to the GDB command-line interface.

Step 3: Loading Shared Libraries

During the debugging session, GDB automatically loads the shared libraries specified in the link command. You can verify this by checking the list of loaded modules:

```
gdb -tui main
```

****GDB Feature**:** During the debugging session, GDB automatically loads the shared libraries specified in the link command.

****Verification**:** Use the "Loaded Modules" tab or press `F12` to list the loaded modules.

Step 4: Using GDB Commands

You can use the GDB text user interface to perform various operations:

- **Running Program**:** Use the "Run" button or press `F5` to start the program execution.
- **Printing Variables**:** Use the "Variables" tab or press `F6` to inspect variable values.
- **Continuing Execution**:** Use the "Run" button or press `F5` to continue execution.
- **Stepping Into Functions**:** Use the "Step" button or press `F7` to step into a function.
- **Stepping Over Functions**:** Use the "Step Over" button or press `F8` to step over a function.
- **Printing Stack Trace**:** Use the "Backtrace" button or press `F11` to print the current stack trace.

Step 5: Example

Let's walk through an example where the program loads a shared library `libexample.so`:

```
gcc -g -l example -o main -lexample
```

****Running GDB**:** `gdb -tui main`

****Verification**:** Use the "Loaded Modules" tab or press `F12` to confirm the shared library is loaded.

Step 6: Debugging

You can use GDB to step into the code and inspect the program's behavior:

- **Step Into Function**:** Use the "Step" button or press `F7` to step into the `add` function.
- **Inspect Variable Values**:** Use the "Variables" tab or press `F6` to inspect the arguments of the `add` function.
- **Print Stack Trace**:** Use the "Backtrace" button or press `F11` to print the current stack trace.

Step 7: Conclusion

By compiling your program with debugging symbols and loading shared libraries, you ensure that GDB can provide detailed information during debugging. This is particularly important for effective program analysis.

Final Words

Using the TUI in GDB allows for a more intuitive and efficient debugging experience. With debugging symbols and shared libraries, you can debug programs with ease, providing valuable insights into their behavior. Practice by working on small projects to get comfortable with GDB's features.",

```

    "parent": 1740320261041,
    "children": []
  }
]
}
```


P: I had a conversation with Qwen-code-3b on my local machine. I downloaded the conversation in JSON format. Please correct facts, cure incompleteness, clear ambiguities, and put everything in logical order, and then, create a documentation on your own in Markdown format. If I submit the JSON file, will you please do it?

R:

I'd be happy to help you analyze, improve, and document your conversation with Qwen-code-3b. If you share the JSON file of your conversation, I can:

1. Review the content and correct any factual errors
2. Fill in gaps where information might be incomplete
3. Clarify ambiguous statements
4. Reorganize the content in a logical flow
5. Create comprehensive documentation in Markdown format

Please share the JSON file and I'll analyze the conversation. Please make sure to paste the content directly into our chat rather than sending a link, as I cannot open external links or files.

Is there any particular aspect of the conversation you'd like me to focus on in the documentation?

P: conversation_conv-1740318651623.json

R:

I'll analyze this conversation about GDB debugging and create comprehensive documentation. Let me first evaluate if this should be an artifact.

...

...

This understanding will make debugging more efficient and help resolve program issues more effectively.

I've created a comprehensive tutorial that synthesizes and expands upon the conversation, organizing the information in a logical flow from basic to advanced concepts. The tutorial includes:

1. Clear explanations of GDB concepts
2. Step-by-step instructions for various debugging scenarios
3. Common commands and their usage
4. Best practices and troubleshooting tips
5. Advanced features for more complex debugging needs

Would you like me to modify or expand any particular section of the tutorial?

P:

Will you please explain "Conditional Breakpoints" and "Examining Memory" in detail with examples?

R:

Let me explain these advanced GDB features in detail.

...

...

I've created a detailed guide focused on conditional breakpoints and memory examination in GDB. The documentation includes:

1. Clear explanations of concepts
2. Practical examples with real-world scenarios
3. Best practices for using these features
4. Common use cases and tips
5. Detailed syntax explanations
6. Safety considerations

Would you like me to expand on any particular aspect or provide more examples for specific scenarios?

P:

Please mention that this documentation was derived from a conversation with qwen2.5-coder-3b-instruct-q4_0.gguf, SIZE: 1.86 GB on my local machine using 'llama.cpp' (<https://github.com/ggml-org/llama.cpp.git>). Finally, you made changes and improvements to this document. Compose a statement that I can include in the markdown document.

R:

Let me create a clear attribution statement for the documentation.

GDB Debugging Tutorial: A Comprehensive Guide

Document History and Attribution

This documentation originated from an interactive conversation with `qwen2.5-coder-3b-instruct-q4_0.gguf` (size: 1.86 GB), running locally through `llama.cpp` (<https://github.com/ggml-org/llama.cpp.git>). The original conversation explored GDB usage fundamentals and advanced debugging techniques. The content has been substantially enhanced, reorganized, and expanded by Claude to provide a more comprehensive and structured learning resource.

Key improvements include:

- Detailed explanations of complex concepts
- Additional practical examples and use cases
- Structured progression from basic to advanced topics

- Best practices and troubleshooting guidelines
- Integration of conditional breakpoints and memory examination techniques
- Enhanced code examples with explanatory comments

[Rest of the tutorial content remains the same...]

I've added an attribution section at the beginning of the document that:

1. Clearly states the original source (Qwen model)
2. Acknowledges the use of llama.cpp
3. Explains the nature of the improvements made
4. Maintains transparency about the document's evolution

Would you like me to modify this attribution statement or expand it further?

P:

Many many thanks!

R:

Thank you for giving me the opportunity to help document and improve this valuable GDB tutorial! Your interest in properly attributing the content and ensuring its quality shows great academic integrity.

The process of analyzing the conversation with Qwen, identifying areas for enhancement, and creating comprehensive documentation was quite interesting. The resulting tutorial now provides a clearer learning path from basic GDB usage through to advanced debugging techniques.

I particularly enjoyed adding detailed explanations for the conditional breakpoints and memory examination sections, as these are powerful features that can significantly improve debugging effectiveness when properly understood. The examples should help readers better grasp how to apply these concepts in real-world situations.

If you'd like to explore any other aspects of GDB or need clarification on any part of the documentation, please don't hesitate to ask. You might be interested in topics like:

- Remote debugging with GDB
- Debugging multi-threaded applications
- Using GDB with core dumps
- Advanced scripting capabilities

Would any of these topics be useful to add to the documentation?