GDB Debugger Reference

Compiling with Debugging Symbols

Pass the `-g` flag to your compiler:

```
z1234567@turing:~/csci241/Assign1$ g++ -Wall –std=c++11 -g -o assign1 assign1.cpp
```

Note: If you have a larger program with several files, each must be compiled with the `-g` flag, and it must also be set when you link.

If you have written a makefile, you can easily add the `-g` flag to the list of compiler variables:

```
#
# PROGRAM:    assign1
# PROGRAMMER: Ima Coder
# LOGON ID:   z1234567
# DATE DUE:   9/14/2027
#

# Compiler variables
CXX = g++
CXXFLAGS = -Wall -std=c++11 -g

# Rule to link object code files to create executable file
assign1: assign1.o
        $(CXX) $(CXXFLAGS) -o assign1 assign1.o
...
```

Setting Your Default Editor

It's very handy to be able to edit your source files from within the `gdb` debugger using the `edit` command. To enable this capability, you must specify a value for the shell environment variable `EDITOR`.

Change to your home directory and open the file `.bash_profile` in a text editor. Add the following line to the end of the file:

```
export EDITOR="/usr/bin/nano"
```

Note that you can specify a different pathname if you want a different editor such as Vim (`"/usr/bin/vim"`) or Emacs (`"/usr/bin/emacs"`).

Save the file and exit.

Then, either log out and log back in, or type the command

```
source .bash_profile
```

Starting the Debugger

Start the debugger with your executable program name as the first argument. For example, if the name of the executable file is `assign1`, then you need to type:

```
gdb assign1
```

Here's an example of what you'll typically see when the debugger starts:

```
z1234567@turing:~/csci241/Assign1$ gdb assign1
GNU gdb (Debian 7.7.1+dfsg-5) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from assign1...done.
(gdb)
```

You can now type debugger commands at the `gdb` prompt. Some of the most commonly used `gdb` commands are listed on the next two pages. This list is not exhaustive, and there are many other commands and options available.

Command	Description	Examples
Getting help		
`help`	Get a list of classes of debugger commands.	`help`
`help` *`class-name`*	Get a list of all commands in the specified command class.	`help breakpoints`
`help` *`command`*	Get documentation for a specific command within a class.	`help break`
Setting command-line arguments and redirection		
`set args` *`arg1 arg2`* …	Specify command-line arguments. You may also use this command to redirect input or output for the program.	`set args in.txt 20` `set args in.txt > out.txt`
`set args`	Cancel previous command-line arguments and redirection.	`set args`
Executing the program		
`run`	Run the program you are debugging. The program will run until it terminates or it hits a breakpoint.	`run`
`continue`	Continue executing program being debugged after it has hit a breakpoint. Execution will continue until termination or the next breakpoint. This command may be abbreviated as `c`.	`c`
`finish`	Continue executing until the current function returns.	`finish`
`next`	Execute next program statement, stepping over subroutine calls. This command may be abbreviated as `n`.	`n`
`next` *`n`*	Step over next *n* program statements.	`next 3` `n 4`
`step`	Execute next program statement, stepping into subroutine calls. This command may be abbreviated as `s`.	`s`
`step` *`n`*	Step into next *n* program statements.	`step 2`
`kill`	Kill execution of program being debugged.	`kill`
Breakpoints		
`break` *`n`*	Set a breakpoint on line *n* of the current source file. The command `tbreak` can be used instead to set a temporary breakpoint that will only be triggered once.	`break 51`
`break` *`filename`*`:`*`n`*	Set a breakpoint on line *n* of the specified source file.	`break other.cpp:32`
`break` *`function-name`*	Set a breakpoint at the beginning of the specified function.	`break buildArray`
`break` *`function-name`*`<`*`type`*`>`	Set a breakpoint at the beginning of the specified template function.	`break compare<int>`
`break` *`class-name`*`::`*`function-name`*	Set a breakpoint at the beginning of the specified C++ member function.	`break Date::print`
`break` *`class-name`*`<`*`type`*`>::`*`function-name`*	Set a breakpoint at the beginning of the specified C++ template member function.	`break Stack<int>::push`
`info breakpoints`	Show status of breakpoints. Can optionally be followed by a list of specific breakpoint numbers; defaults to all breakpoints	`info breakpoints` `info breakpoints 1 3`
`disable breakpoints` *`n1 n2`* …	Disable specified breakpoint numbers. Defaults to all. May be abbreviated as `disable`.	`disable 3`
`enable breakpoints` *`n1 n2`* …	Enable specified breakpoint numbers. Defaults to all. May be abbreviated as `enable`.	`enable 2 3`
`delete breakpoints` *`n1 n2`* …	Delete specified breakpoint numbers. Defaults to all. May be abbreviated as `delete`.	`delete 1`

Command	Description	Examples
Watchpoints		
`watch` *expression*	A watchpoint stops execution of your program whenever the value of the specified expression changes. A watchpoint is a specific type of breakpoint and can be enabled, disabled, or deleted using the same commands.	`watch playerName`
`watch -location` *expression*	Evaluates expression and watches the memory location to which it refers. `-location` may be abbreviated as `-l`.	`watch -l ptrName`
`info watchpoints`	Show status of watchpoints only. Can optionally be followed by a list of specific watchpoint numbers; defaults to all watchpoints	`info watchpoints` `info watchpoints 1 3`
Examining and modifying variables		
`whatis` *expression*	Print the data type of the specified expression.	`whatis num`
`print` *expression*	Print the current value of the specified expression or variable name. Can be abbreviated as `p`.	`print num` `p providerArray[3]`
`set var` *variable* `=` *expression*	Set the specified variable to the specified expression.	`set var x = 3`
`display expression`	Print value of *expression* each time debugger stops.	`display num`
`info display`	Lists expressions to display when program stops, with code numbers.	`info display`
`disable display` *n1 n2* …	Disable specified display expression code numbers. Defaults to all.	`disable 3`
`enable display` *n1 n2* …	Enable specified display expression code numbers. Defaults to all.	`enable 2 3`
`undisplay` *n1 n2* …	Cancel the specified display expression code numbers. Defaults to all.	`undisplay 1`
`info locals`	Prints values of local variables in the current stack frame.	`info locals`
Listing source code		
`list`	List ten more lines after or around previous listing.	`list`
`list` *n*	List ten lines around the specified line. Line number arguments may be preceded by a filename.	`list 51` `list Provider.cpp:20`
`list` *function*	List ten lines around the specified function.	`list buildArray`
`list` *class-name*`::`*function-name*	List ten lines around the specified C++ member function.	`list Provider::print`
`list +`	List the ten lines after the previous listing.	`list +`
`list -`	List the ten lines before the previous listing.	`list -`
`list` *x*`,`*y*	List the specified range of line numbers.	`list 10,35`
Program stack		
`backtrace`	Print backtrace of all program stack frames. May be abbreviated as `bt`.	`bt`
`backtrace full`	Print backtrace of all program stack frames, including local variables.	`bt full`
Other commands		
`make`	Run the `make` program using the rest of the line as arguments.	`make` `make clean`
`file` *executable-filename*	Use *executable-filename* as program to be debugged.	`file assign1`
`edit`	Edit a source or header file.	`edit`
`edit` *filename*`:`*n*	Edit at the specified line number in the specified file.	`edit Date.cpp:10`
`edit` *function*	Edit at the beginning of the specified function. May be optionally preceded by a filename.	`edit buildArray` `edit sorts.cpp:compare`
`edit` *class-name*`::`*function-name*	Edit at the beginning of the specified C++ member function.	`edit Provider::print`
`quit`	Exit the debugger. May be abbreviated `q`.	`quit`

How to Debug Using `gdb`

The `backtrace` command can give you an immediate sense of the sequence of method calls that resulted in your runtime error. That should help you to localize the last statement that executed before your program abnormally terminated. Unfortunately, the last statement executed by your program is not necessarily the one with a bug. Mistakes earlier in the program may not manifest immediately, particularly when it comes to a runtime error like a segmentation fault. It's also entirely possible to fix one runtime error only to reveal another one.

Usually, you will need to create at least one breakpoint in order to do anything useful. If you suspect that a particular function is causing your runtime error, place a breakpoint on that function. Alternatively, use the `list` command to list your source code and place a breakpoint on a specific line number. If you have no idea where the error is happening, start by putting a breakpoint on one of the first lines inside your `main()` function.

Use `run` to make the program run until it hits your first breakpoint. Remember that the lines displayed in `gdb` as the program is executing represent *the next statement to be executed*. Advance line-by-line through the program code using the `next` or `step` commands or continue running it until your next breakpoint by using the `continue` command. As you step through the program, you can examine the values of variables using the `print` and `display` commands or `info locals`.

To find the bug that is causing your runtime error, you need to know what the values of your variables **should be** at any given point in the program as well as what the values **actually are**. Using good test data can make this much easier!