Setup

Compilation

```
g++ -c -g -Og linkedlist.cpp
g++ -o mylinkedlist linkedlist.o
```

```
-O0: no optimization
-Og: optimization compatible with gdb
```

Invocation

```
gdb <program-name>
gdb (then use file <executable to load an executable)
```

Using a GUI

Emacs

- Start emacs: emacs
- Start gdb: <ESC>x gdb
- emacs prompts for the executable name

gdb built-in gui

- start tui: <ctrl-x>a
- split window: <ctrl-x>2 (source + assembly)
- merge windows: <ctrl-x>1 (back to source only)
- up and down arrow: move up and down in the source window
- move up in command window: <ctrl-p>
- move down in command window : <ctrl-n>

Other GUIs

- gdbgui
  - `https://github.com/cs01/gdbgui`
- Eclipse CDT:
  - `https://www.eclipse.org/cdt`
- Eclipse's Standalone Debugger:
  - `https://wiki.eclipse.org/CDT/StandaloneDebugger`
- KDevelop:
  - `https://www.kdevelop.org`
- NetBeans:
  - `https://www.netbeans.org`
- with the GDB server plugin:
  - `https://plugins.netbeans.org/plugin/37426/gdbserver`

Commands

Starting the debugging session

run

- Purpose: starts the debugging session.
- Syntax: *run <program args>* where *program args* are the command line arguments to be passed to the program

start

- Purpose: sets a temporary breakpoint at the beginning of main and starts the debugging session
- Syntax: *start <program args>*

Loading executable - refreshing symbols

file

- Purpose: loads an executable
- Syntax: *file <executable name>*

Directory seach path

directory

- directory <directory-name>

Ending the debugging session

kill

- Purpose: ends the debugging session
- Syntax: *kill*

Exit gdb

quit

Breakpoints

break <options> shortcut: b - sets a breakepoint

- break linenum
- break function
- break filename:linenum
- break filename:function
- break +offset
- break -offset

tbreak <line-number>

- set a breakpoint enabled only for one stop

rbreak <regex> sets a breakpoint according to a regular expression

- Example:
  - rbreak getnod*

conditional break

- break <option> if <condition>
- break 118 if k==2

disable disables an existing breakpoint

- disable [list]

enable

- enable [list]
- enable once list
- enable count *count* list
- enable delete [list]
  - enabled for deletion - like with tbreak

ignore [list] *count*

- ignores the breakpoint *count* times

delete [list]

- Purpose: used in conjuction with info break to remove a break point
- Syntax: *delete <number>* where number is the breakpoint you want to remove
- Note: without any parameters deletes all breakpoints.

watch <variable|address> [if <condition>]

- Purpose: monitors the changes to a variable. Stops when the memory location that holds the variable is modified
- Syntax: watch

info break

- Purpose: displays all the break and watch points and their corresponding number
- Syntax: *info break*

continue

- Purpose: continues with the execution
- Syntax: *continue* - short form: *c*

Displaying variables

print

- Purpose: prints the contents of a variable
- syntax: *print <variable name>*
- Short form: *p <variable name>*
- Example: print var

printf

- C-like fromatted printf
- printf FORMAT, <variable-list>

dprintf dynamic printf

- Syntax: dprintf location, FORMAT, <variable-list>
- location is similar to a breakpoint location.

display <expression>

- Adds <expression> to the list of expressons to display each time the program stops
- Variations
  - display /fmt <expression>
  - delete display *dnums*
  - disable display *dnums*
  - enable display *dnums*

display

- displays the current values of the expressions on the list

info display

- Prints the list of expressions previously set up to display atumotically.

info

- info args
- info registers
- info locals
- info variables [-t] [<regular-expression>]
- info registers

Examining memory - Artificial arrays

arrays

- Syntax:

```
print *<pointer>@count
print (<type>[count])* pointer
```

x/FMT *ADDRESS*

- FMT: count format size (no spaces)
- *ADDRESS* is an expression for the memory address to examine
- Examples
  - x/5i - 5 machine-level instructions
  - x/8fg - print 8 double precision numbers

- Format
  - x hexadecimal
  - d signed decimal
  - u unsigned decimal
  - c character
  - i disassembled instruction
  - s c-strings
  - f floating point
  - a address
  - t binary
- Size
  - b byte
  - h 2-byte blocks (halfword)
  - w 4-byte blocks (word)
  - g 8-byte blocks (giant word)

Navigation

next [/count/]

- Purpose: advances the execution one single (or *count* times) source instruction, if the instruction is a function call it does not step into the function
- Syntax: *next n*

step [/count/]

- Purpose: advance to the next instruction if the instruction is not a function call (behaves like next here) or step into a function if the instruction is a function call.
- Syntax: *step s*

nexti

- executes one machine instruction, does not step into the function

stepi

- executes one machine instruction.

until *location*

- Continue running until a souce line past the current line is reached. Used for loops.
- Continue running until location is reached

finish

- continue running until just after the function in the active stack frame returns.

return <value>

- immediately returns from a function. You can specify a return value

jump *location*

- skips instructions
- set *tbreak* first if you don't want execution to continue
- Example
  - set tbreak +2
  - jump +1 - skips one line, breaks

Examining the stack

where

- Purpose: display the stack frame, that is, the nested list of functions called at the point where execution stopped
- Syntax: *where*

frame

- Purpose: move to a given frame (displayed by where)
- Syntax: *frame <frame number>*

up

- move one position up to the previous stack frame

down

- moves one position down to the next stack frame

backtrace (bt) bt full

- Syntax: backtrace <full>|<count>
- prints a backtrace of the entire stack starting with the current executing frame

Viewing assembly code

disassemble

- disassemble /s <function-name>

Modifying the value of a program variable

set <symbol>=<new-value>

Convenience variables

set $<varname>=<value>

- set $i=0

$<number> value of output <number>

- p $3

$ contains the value of the last output

Macros

define

- user-defined commands. A sequence of gdb commands
- define <command-name>

Commands

commands [list]

- breakpoint/watchpoint commands to execute when program stops

Logging

set logging <on> <off>

- enable/disable logging

set logging file <file>

- change the name of the current logfile

executing scpits

- gdb -x script.txt <program-name>
- at the gdb prompt: source script.txt

Debugging Techniques

Types of defects

- Simple errors:
  - Precedence errors, wrong order of operations
  - Unexpected assignments, using = instead of ==
  - Passing invalid parameters to functions.
- Implementation errors
  - memory overruns
  - incorrect memory allocations
  - wrong loop indeces
  - wrong use of pointers
- Errors in the logic (HARD)
  - faulty algorithms
  - missed special cases
  - wrong logic
  - core runs but results are wrong

Debugging Techniques

Unit Testing, Functional Testing, Regression testing

- Test each components as you develop them
- If you have previous correct results, check periodically against those results. Stop development immediately if you see diferences and debug.
- Run against multiple input combinations
- Test components separately. Validate components.
- Remember, testing only indicates that the program is faulty, but not necessarily that it is correct.

Incremental Development

- Define intermediate goals, establish checkpoints. Test.

Type of bugs

- Repeatable bugs (bad situation).
- Sporadic bugs (much worse).

Approach

- Oftentimes the bug happend way before the place where its effects are noticed,
- Try to localize the bug.
- Blind search doesn't help. Try to understand the bug. Use analytical thinking.
- For instance, it's a memory overwrite. Probably, this was caused in a loop.

Bracketing

- Try to find boundaries or regions where the code is well behaved and regions that are untrusted.
- The smaller the region, the better
- Comment out regions of the code that you think may not affect the bug. If the bug disappears, well, that may be a clue.

Bag of tricks

- Reduce the size of the input. Try to come up with the smallest and simplest input that still produces the bug.
- The goal of the previous step is to reduce the time taken by each debugging cycle. Debugging is and iterative process.
- Use a good debugger, set watch points, monitor variables.
- Find invariants. Monitor them
- Use other tools, for instance valgrind is effective to detect memory overwrites
- Create buffer areas around dynamically allocated regions. See if the problem goes away. Check those buffer areas
- Use malloc debugging tools.
- Use your analytical skills. Come up with your own approach.
- And sometimes... Take a break, go for a walk, do something different for a while. When you spend many hours debugging you may not see the forest for the trees.