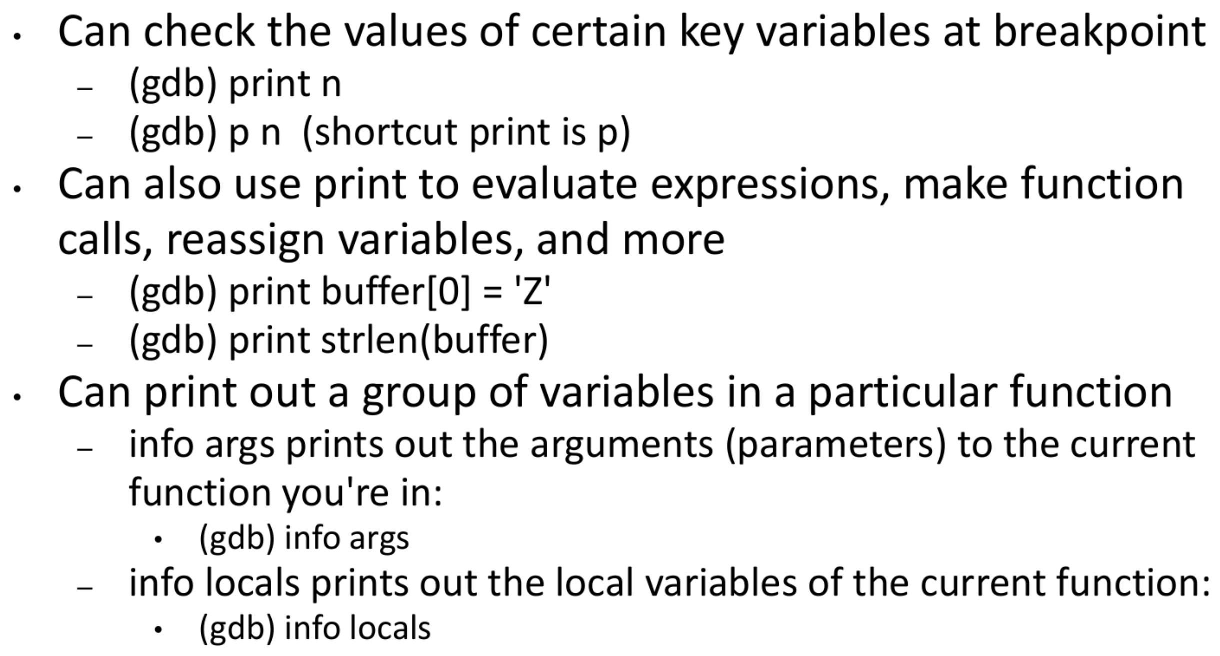
SOME NICE PTS YOU MAY LOOK OTHERWISE BUT CAN COME IN QS



- Can check the values of certain key variables at breakpoint
  - (gdb) print n
  - (gdb) p n (shortcut print is p)
- Can also use print to evaluate expressions, make function calls, reassign variables, and more
  - (gdb) print buffer[0] = 'Z'
  - (gdb) print strlen(buffer)
- Can print out a group of variables in a particular function
  - info args prints out the arguments (parameters) to the current function you're in:
    - (gdb) info args
  - info locals prints out the local variables of the current function:
    - (gdb) info locals

- (Can also use print to evaluate expressions, make function calls, reassign variables,and more.)
- backtrace/bt command prints the current functions on the stack.

FULL NOTES

- Debugging is process of finding compiling and run-time errors.(You know what each mean im not writing that)
- g++ myprogram.cpp 2>&1 | more (self explanatory command)(to help in compile errors viewing and resolution)
- Alternative to gdb: print statements. Send debugging information to standard err. Comment out when not needed. Or instead set a compile time symbol DEBUG, and use #ifdef debug statements. Then while compiling you can add option -DDEBUG to define it and hence don debugging.
- Assertions: Program stops with an informational msg whenever an assertion condition fails. If you do not wish to apply the assertions while compiling then you can add option to define NDEBUG during compiling, this will compile code while treating assertions as comments.
- GDB:
- Using: While compiling add, -g compiler flag. Then gdb filename will start the debugger for that program.
- help: Prints out a list of possible topics you can ask help for.
- run: It runs the program.
- To stop the program at various points in its execution, can use breakpoints.

  - To specify breakpoint on entry to a function: break function OR break filename:function or b function(b is shortcut for break).
  - Set a breakpoint by specifying a file and line number: break 26 OR break filename:26.
  - TO list where the current breakpoints are set: info breakpoints.
  - To delete the breakpoint numbered 2: delete 2.
  - We can also set breakpoints to only trigger when certain conditions hold true. Eg. break 23 if i==count-1
  - If we wish to continue from a breakpoint, we can use “continue”. Contrarily, run will execute the program from the beginning while continue will take us to the next breakpoint.
- Can check the values of certain variables at breakpoints(print):
  - Eg. print n OR p n(p is shortcut for print)
  - Can also use print to evaluate expressions, make function calls, reassign variables and more. Eg. print buffer[0]=’Z’.
  - Can print out a group of variables in a particular function: info args prints out the arguments to the current function you’re in. info locals prints out the local variables of the current function.
- At a breakpoint we can ask the debugger to print out the contents of the code file via list: list. We can also list the contents of files by name and line number: list filename:1.
- There are 2 commands to step through:
  - Next steps over function calls.
  - Step steps into function calls.
- To see the value of a specific variable whenever we reach a breakpoint or are stepping through our program, we can use the “display” command.
  - Display result.
  - Print vs display: Print displays the value of an expression once. Display automatically prints the value of an expression every time execution stops.
- Can examine the contents of the stack via backtrace which prints the current functions on the stack. Backtrace OR bt(shortcut for backtrace). It also prints the arguments to the functions on the call stack.
- You can use the commands up and down for moving up and down the stack frames. (up moves you up one stackframe i.e. to current function’s caller) and down moves down one stackframe i.e. to it’s callee. Helpful if you're stuck at a segfault and want to know the arguments and local vars of the faulting function's caller or callee.

PROFILING

- Profilers allow us to examine the program’s overall use of system resources. Focused primarily on memory use and CPU consumption. Can determine parts of code that are time-consuming and need to be re-written.
- For enabling profiling while compiling use -pg tag.
- Execute the program code to produce the profiling data. Generates gmon.out(a binary file, if alr present its overwritten, program must exit normally dont use ctrl C).
- Run gprof tool on the data as gprof programname gmon.out > analysis.txt.

- Analysing the output:
    - Flat profile: Shows the total amt of time your program spent executing each function.
    - Call graph: Shows how much time was spent in each function and it’s children.