Debugging

- See what goes on inside a program as it executes
- Information about specific data structures is part of the code in the form of comments, or in the README file
- Debugging pertains only to the values contained within variables at different points of time during execution

Interactive debugging

- Based on a source code, statement-level debugger
- Allows to discover values of variables by using their names in the source program, tracing their execution one statement at a time
  - Allows you to see what is going on inside a process during execution, or when the process crashes
- Debugger allows you to
  - Start a program with any specified set of arguments
  - Make the process stop on specified conditions
  - Examine what has happened when the process is stopped/suspended
  - Change variable values in your program to correct some bugs and see the effect on other statements
- Different debuggers
  - The native Unix debugger was called dbx
  - It has given way to the Gnu Debugger called gdb that is now ubiquitous with all the other Gnu tools, including compilers
  - A nice front end to gdb is provided by xxgdb and ddd

Working with gdb

- Starting and stopping
  - Start gdb by typing the command gdb
  - Stop gdb by typing quit or ^D
- Invoking gdb
  - Any of the following works:
    gdb
    gdb prog_name
    gdb --args prog_name arg1 arg2 ...
    gdb -h

gdb commands

- Command syntax
  - Each command is a single line of input of arbitrary length
  - The first word is the command, followed by [optional] arguments
  - Commands may be truncated if they are unambiguous
Debugging

- A blank line (just hitting return) repeats the last command
- Any text starting with a # to the end of line is a comment

- Command completion
  - Pressing tab key after some characters will complete the command or give you a list of commands starting with that string
  - Also works with command arguments or symbolic names
    
    ```
    info bre<TAB>
    ```
  - Pressing <TAB> twice gives you all the possibilities starting with the entered string

Running programs under gdb

- Must generate debugging information when the code is compiled
  - Debugging information involves data types of different variables
  - Need to have access to symbolic information along with their line numbers and address in memory

- Compiling for debugging
  - The C (or C++) files are compiled with the –g flag in effect
    ∗ Allows the inclusion of extra symbol table information in the executable
    · Names and locations of all variables
    · Names of all functions and their arguments
    · Data types of all objects declared in the program
    · Path names of the source code files used to compile the program
  - If you want to see your macros inside gdb, you should compile the code using the flag –g3

- Compiling for delivery
  - The code to be delivered to customers does not need to have debugging information
  - It should avoid any overheads and aim for optimized [fast] execution
  - Compile it using the flag –O
  - The flags –g and –O are mutually exclusive, though the compiler gcc allows the use of both, implying that you can debug optimized code

- Running code
  - If a program is not loaded, or to change an existing program, use the command file with argument specifying the program name
  - Run the loaded program using the command run or abbreviation r
    ∗ run creates an inferior process and makes that process run the loaded code
  - You can run the loaded code by using the command start
    ∗ This will be equivalent to stopping the code at the first executable statement (main)
  - The arguments to the program can be specified as arguments to either run or start
    ∗ The arguments will be reused if the code is executed again using run or start within the same session of gdb
  - The I/O can be redirected using the shell redirection operators
    ```
    run > outfile
    ```

Stopping and Continuing

- Breakpoints