Debug I

• debug: take the bugs out of a program

• almost every IDE (Integrated Development Environment) has a builtin debugger, e.g.
  – Xcode from Apple, Visual C++ from Microsoft

• other debuggers:
  – gdb (the GNU Project debugger)
  – ddd (Data Display Debugger)

• we are only going to talk about gdb
Useful \texttt{gdb} Commands I

- suppose your program, i.e., the executable is called \texttt{myProg} and it takes some command line arguments:

  > ./myProg -a onlyArg

- outside Emacs:
  - issue the following command at your command prompt:
    > gdb myProg

- inside Emacs:
  - use the following:
    \texttt{M-x RET gdb}
Useful \texttt{gdb} Commands II

- to run-n-debug the program type the following in \texttt{gdb} prompt (run):
  \begin{verbatim}
  (gdb) r -a onlyArg
  \end{verbatim}

- to set a break point just before you see the program crash in \texttt{gdb} (break):
  - say, the program crashed in file \texttt{foo.c} line 123, do this:
    \begin{verbatim}
    (gdb) b foo.c:123
    \end{verbatim}
  - say, the program crashed during the function call \texttt{bar( )}, do this:
    \begin{verbatim}
    (gdb) b bar
    \end{verbatim}

- you could set up multiple break points the same way as above

- to get a list of breakpoints, do this (\texttt{info breakpoints}):
  \begin{verbatim}
  (gdb) i b
  \end{verbatim}

- to skip a breakpoint, do this (\texttt{ignore}):
  \begin{verbatim}
  (gdb) ig <NUM> <TIMES>
  \end{verbatim}
  where \texttt{<NUM>} is the break point number and \texttt{<TIMES>} is how many times to skip it
Useful `gdb` Commands III

- to continue beyond the current break point, do this (`continue`):
  
  ```
  (gdb) c
  ```

- to delete a break point, do this (`delete`):
  
  ```
  (gdb) d <NUM>
  ```

  where `<NUM>` is the break point number

- to run up to a line, e.g., till the end of a `for` loop do (`until`):
  
  ```
  (gdb) u <LINENUM>
  ```

  where `<LINENUM>` is the line number up to which you want to run the program

- to go to the end of a function, use (`finish`):
  
  ```
  (gdb) fin
  ```
Useful *gdb* Commands IV

- when you stop at a break point, to go to the next line, type (`next`):
  
  ```
  (gdb) n
  ```

- when you stop at a break point, to go inside a function which is called from the next line, type (`step`):

  ```
  (gdb) s
  ```

- to print a value of a variable (`print`):
  - a non-pointer variable:
    
    ```
    (gdb) p non_pointer_var
    ```
  - a pointer variable (could be a pointer to a structure):
    
    ```
    (gdb) p *non_pointer_var
    ```
  - a pointer "array":
    
    ```
    (gdb) p *an_array@20
    ```
  
  this will print `an_array[0]`, ..., `an_array[19]`
Useful `gdb` Commands V

- to see exactly which sequence of function calls caused the error, do this (backtrace):
  `gdb` bt

- to select one of the (function) stack frames, do this (frame):
  `gdb` f <NUM>

- to go up/down stack frames, do this (up/down):
  `gdb` u
  `gdb` d

- within a frame to look at the value of all the local variables, use (info locals):
  `gdb` i lo

- within a frame to look at the value of all the argument variables, use (info args):
  `gdb` i ar
Useful `gdb` Commands VI

- if you want to know more about some command, use (help):
  
  (gdb) help print
  (gdb) h p

- if you want to know what's out there, use (help):
  
  (gdb) h

  and then do help on specific categories/commands

- if you want to end `gdb` session, use (quit):
  
  (gdb) q

- some useful commands not covered:
  
  watch, list, set print pretty, set print array,...
A printf Debugging Trick I

- consider the following macro, we saw a version of it earlier:

```c
#define DEBUG(_x) \
do { \
    fprintf(stdout, "===== BEGIN: DEBUG block =====\n", \"
    file: %s, line: %d\n", \n    __FILE__, __LINE__); \
    _x \
    fprintf(stdout, "===== END: DEBUG block =====\n"); \
    fflush(stdout); \
} while (0)
```

- it’s very informative, use it whenever printing out some information may help debug

- now you don’t have to manually delete those `DEBUG()` statements after you are done debugging as newbie’s (gopi in 2001!) do!

- this manual deletion is discouraged, because later if some new bug(s) show up, old debug statements might help you out, you don’t have to remember-n-retype them
A printf Debugging Trick II

• instead for each `.c` or `.C` file do the following:
  – say your file is called `foo.c`
  – put the following lines in `foo.c` after all your `#include` statements and
    before any code:
    ```
    #ifndef FOO_DEBUG
    #define FOO_DEBUG 0
    #endif
    #if (!FOO_DEBUG)
    #define DEBUG(_x) ((void) 0)
    #endif
    ```
    – now add `-DDEBUG_UTILS=1` to the makefile to your `gcc`/`g++`
      compilation options:
      ```
      CC = gcc -pedantic -ansi -g -Wall -DDEBUG_UTILS=1
      ```
    – this will turn printf debugging on
    – to turn it off, just take the option `-DDEBUG_UTILS=1` out:
      ```
      CC = gcc -pedantic -ansi -g -Wall
      ```