EECS 2031

Software Tools

Module 10 – Debugging

YORK UNIVERSITÉ UNIVERSITY

Debugging

- Finding the source of logic errors in a complex software system can be very hard
- One can use `printf` to trace the state of the program but this can be very tedious
- Debuggers are tools that help programmers examine the state of the program as it is running

gdb

- **gdb** is a command-line debugger that can be used with C or C++ programs
- To use **gdb**, the program must be compiled with the **-g** flag

```
gcc -g main.c -o main.exe
```

- This adds extra information in the executable, so the debugger can trace your program

gdb

- To launch **`gdb`** with the specially-created executable: **`gdb main.exe`**
- This opens something like a shell, where you can enter commands interactively
  - You can recall commands with the arrow keys, use TAB for auto-completion etc.
  - **`help [command]`** prints information on a command
  - **`apropos [word]`** prints all commands whose description contains **`word`**

`gdb` commands

- **`run`**: Runs the program normally
- If the program crashes, you will get information, such as the line of code that caused the crash, parameter values at the time of crash, a stack trace etc.
- If the program does not crash but contains logic errors, you want to stop at important points and observe the state of the program

Breakpoints

- Debuggers use breakpoints to decide when to stop execution
- Any line of code can be chosen as a breakpoint
- ***If the execution of the program gets to that line***, the debugger will stop the execution and allow the user to continue one line at a time

`gdb` commands

- **`break`**: Adds a breakpoint

```
break main.c:42
```

- Execution will stop if it ever reached line 42 in **`main.c`**

```
break func1
```

- Execution will stop if function **`func1`** gets called

Conditional breakpoints

- We are often interested in stopping execution at a given line only if certain conditions hold
- Can set a conditional breakpoint with

```
break main.c: 42 if i > 9
```

`gdb` commands

- Once execution has stopped at a break point:
- **`continue`**: Continues execution until the next breakpoint
- **`step`**: Execute one more line of code
- **`next`**: Execute one more line of code but treat function calls as one instruction
- **`print var`**: Print the value of variable **`var`**

Watchpoints

- It is also possible to stop the execution of the program every time the value of a particular variable is changed
- Set a watchpoint for a variable **`var`** with

```
watch var
```

- Output gives you the previous and the new value of variable **`var`**
- See **`debugging.c`**

More `gdb` commands

- **`where`**: Gives the stack trace to the current point of execution
- **`finish`**: Continue to the end of the current function
- **`info break`**: Print all breakpoints and watchpoints
- **`delete 3`**: Delete breakpoint #3 (as listed by **`info break`**
- **`quit`**: Exit the debugger

valgrind

- While **gdb** is great for debugging logic errors, it can only help with basic memory management issues
- To detect memory overruns and leaks, run the specially created executable under **valgrind**
- See **memcheck.c**

```
gcc -g memcheck.c -o main.exe

valgrind main.exe
```

`valgrind`

- Produces a lot of output
- Focus on lines of code that produce errors, such as
- `Use of uninitialised value`
- `Invalid write`
- `40 bytes are definitely lost...`
- See link to quick start guide on course website