Development Tools 5: Debugging and Tracing

1. Intro
2. Text Editors
3. IDE's
4. Compiling
5. **Debugging and Tracing**
   - **GDB**
   - **DDD**
   - **ltrace and strace**
   - **valgrind**
6. Advanced Development

Debuggers: GDB

The basic Linux debugger is GNU's **GDB** (GNU DeBugger).

GDB is a command-line tool.

Be sure to compile using GCC's `-g` option, as this will allow GDB to associate machine instructions with source code statements.

You invoke GDB on executable `myprog` like: `gdb myprog`

This gives you a "`(gdb)`" prompt to enter GDB commands.

Once you have started GDB, you can use commands to *set breakpoints*, start the program running, step through the next program line, print out the values of expressions, and so forth.

Debuggers: GDB (contd.)

Detailed use of GDB is beyond the scope of these slides, but is covered in some course texts, and in many web tutorials.

Some key example commands:

- `break` *function/linenum* – set breakpoint
- `info breakpoints` – list breakpoints
- `cotinue` – go on executing after break
- `step` – execute next line of code
- `print` *variable* – print value stored in *variable*
- `bt` – show stack frame backtrace
- `run` – execute with current arguments
- `run` *arguments* – execute with command-line *arguments*

Debuggers: GDB (contd.)

A particularly handy use for GDB is to find out where a program is crashing due to an error like a **segmentation fault** (illegal memory access).

Your shell session must be configured to create **core files** (memory dumps) when such crashes occur:

`ulimit -c MAX_CORE_SIZE` (e.g., set `MAX_CORE_SIZE` to `unlimited`)

Now, when e.g., `myprog` crashes, a core file named `core` or `core.`PID gets created.

You can determine what happened by starting GDB as:

`gdb myprog core`

(make sure you have compiled `myprog` with the `-g` option)

At the gdb prompt, use the `bt` command to show you the sequence of calls that led to the segfault, explore variable values, etc.

Debuggers: GDB (contd.)

GDB is a command-line tool, but there are several GUIs that can be used as front ends:

- GNU's **DDD** (Data Display Debugger)
- KDE's **KDbg**
- **Xxgdb**

Debuggers: DDD

GNU's **DDD** (Data Display Debugger) is the most commonly used GDB GUI.

It allows you to easily set breakpoints, mouse over variables for their current values, and so forth.

Invoke DDD on executable `myprog` like: `ddd myprog&`

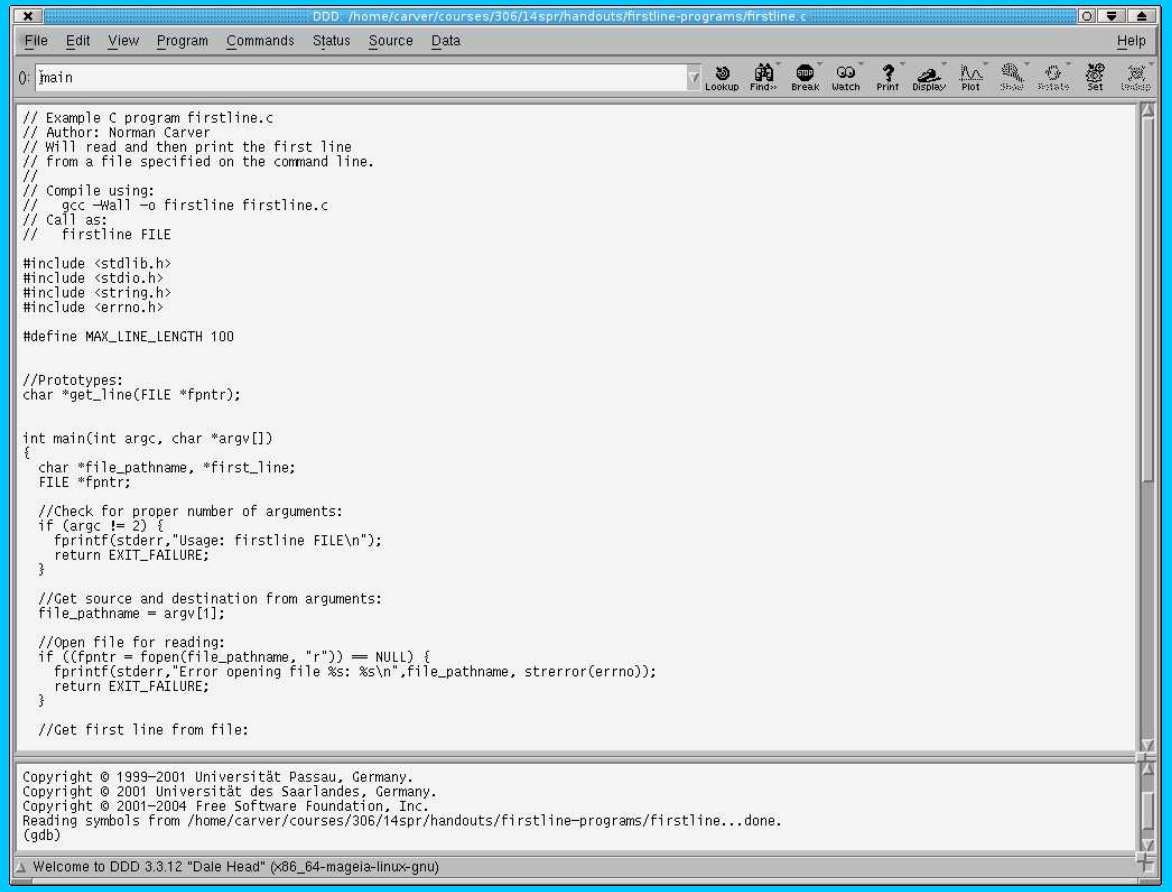
You must have compiled using GCC's `-g` option.

The menu entry *Program* $\Rightarrow$ *Run* allows you to supply command line arguments.

Right-clicking (e.g., at the start of lines) brings up context-sensitive menus of actions.

Debuggers: DDD (contd.)

DDD after startup with `firstline`:



Execution Tracing Tools

There are two tools that allow you to trace calls in the execution of a program while it is run:

- **ltrace** – shows calls to *C library functions*
- **strace** – shows calls to *system calls*

These functions permit you to see the calls that are being made, their arguments, and the return values from the calls.

You invoke each with the command line you want executed, e.g.,:

```
ltrace ./myprog arg1 arg2
```

Note that these tools require only the executable being run, so they can even be used to figure out what might be wrong when you do not have source code.

Execution Tracing Tools (contd.)

The output of `strace` can be a bit confusing at first, as it shows everything that is being done to load libraries and setup memory as part of normal program execution; just ignore the output until you start to see calls that are in your code.

Key options:

- `-f`: follow any *child processes* created
- `-o`$file$: write trace's output to $file$
  (cannot redirect output nor pipe through `more`)
- `-S`: also show system calls with `ltrace`
- `-e`$expr$: modify the tracing accoring to $expr$
  (e.g., "`-e trace=open,close`" to trace only `open`/`close` calls)

Memory debugging tools

One of the most common sources of programming errors in C programs is the requirement for **manual memory management**.

Four common errors:

- **running off array ends** – C does not maintain array size nor check array accesses, plus since **C strings** are arrays, invalid C strings cause problems with library functions
- **illegal memory accesses** – the ability to manipulate memory addresses using **pointers** can lead to **segmentation faults** (segfaults)
- **memory leaks** – **dynamic memory** is allocated using `malloc` but never `free`'d
- **dangling pointers** – memory that has been `free`'d continues to be accessed

Memory debugging tools: Valgrind

One of the most useful tools to help in debugging memory errors is **Valgrind**.

Valgrind effectively runs a program in a **virtual machine** and analyzes memory accesses and allocations/deallocations.

It can catch all of the four errors listed above.

Valgrind includes multiple tools (see the man page), but the default is **memcheck**.

You invoke Valgrind with your command line, such as:

```
valgrind ./myprog arg1 arg2
```

Be sure to compile using GCC's `-g` option.

Memory debugging tools: glibc

The GNU C library, **glibc**, also includes the ability to detect some memory allocation errors.

This is controlled by the value of the **environment variable** `MALLOC_CHECK_`:

- 0: do no checking
- 1: detect errors and print messages on stderr
- 2: call `abort()` on error

To set the environment variable you can run the following command before you run your executable:

```
export MALLOC_CHECK_=1
```

or you can set it for the call like this:

```
MALLOC_CHECK_=1 ./myprog ...
```