Debugging and GDB

- Debugging
- Debugging Search Strategy
- Examining Program State
- C Program Execution
- Normal Program Execution
- Program Execution with Debugger
- Debuggers
- **gdb** Sessions
- Basic **gdb** Commands
- **gdb** Status Commands
- **gdb** Execution Commands
- Using a Debugger
- Other Debugging Ideas
- Laws of Debugging
- Possibly Untrue Assumptions

❖ Debugging

Debugging needed when software "does not behave as expected".

I.e. when results are different to those implied by the specification.

Typically:

- this *is not* due to the software being full of errors
- this *is* due to a small incorrect fragment of code (i.e. a bug)

Bug: fragment of code that does not satisfy its specification.

❖ ... Debugging

Consequences of bugs:

- compiler gives syntax/semantic error (if you're very lucky)
- program halts with run-time error (if you're lucky)
- program never halts (not lucky, but at least you know)
- program completes, but gives incorrect results (unlucky)

Simple example of buggy program:

```
int main(int argc, char *argv[]) {
   int i, sum;
   for (i = 1; i <= argc; i++)
      sum += atoi(args[i]);
   printf(sum);
}
```

❖ ... Debugging

Debugging has three aspects:

- find the code that is causing the problem
- understand why it's causing the problem
- fix the code to eliminate the problem

Generally ...

- understanding a bug is relatively easy once you've found it
- fixing a bug is very easy once you've found and understood it

(Although sometimes fixing a bug might require substantial re-development)

❖ ... Debugging

To understand a bug, once a small buggy region of code is isolated

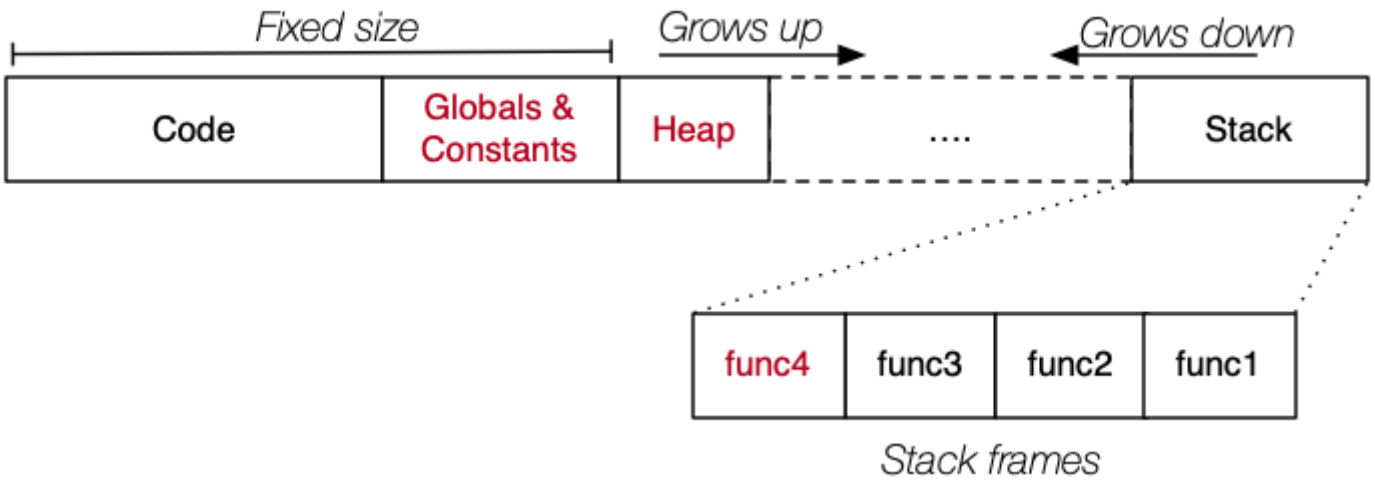
- see what the program state is before the code executes
- see what the program state is after the code executes
- compare this to what you expected from the spec.
- trace the code to see where it changed the state incorrectly

So what is the program state?

Essentially, it's the names/values of all *active* variables.

❖ ... Debugging

Model of state/memory of executing C program:



Active regions are shown in red.

Each function call adds a new *stack frame*:

- contains functions parameters and local variable
- only "top" frame is active, but others can be observed

❖ ... Debugging

The *real* difficulty of debugging is locating the bug.

A bug is a code fragment with unintended action

To repair it, you need to know:

- in detail, what the code *should* do
- in detail, what the code *actually* does

In any real program, the sheer amount of *detail* is the problem.

Trick to effective debugging: narrowing the focus of attention.

i.e. use a search strategy that enables you to zoom in on the bug.

❖ Debugging Search Strategy

When you run a buggy program, initially everything is ok.

At some point, the buggy statement is executed ...

- the program state becomes incorrect
- but the program won't necessarily crash at this point

Aim: identify a region of code where state becomes "corrupted".

Requires you to know the state* at points during execution.

(* not the whole state (too big), just the interesting bits)

❖ ... Debugging Search Strategy

A simple search strategy for debugging is as follows:

1. program prints incorrect result
2. which variable contains the incorrect value
3. what was the last statement that set its value
4. print values of other variables in that statement
5. restrict attention to variables with wrong values
6. repeat the above, from step 2

❖ ... Debugging Search Strategy

A more general strategy:

- display all major data structures just initialisation
  (typically requires a **`show`** function for each data structure)
- display major data structures at "strategic points"
- display major data structures at end of execution

How to determine strategic points? e.g.

- after the first, second, n^{th} iterations of the main program loop
- after the keystroke that causes an interactive program to crash
- after the point where the last output from the program occurred

❖ Examining Program State

A vital tool for debugging is a mechanism to display state.

E.g. diagnostic **`print`** statements of "suspect" variables.

Problems with this approach:

- it changes the program
  (so you're not debugging quite the same thing)
- you may guess wrong about what the suspect variables are
- generally too much is printed
  (e.g. printing to trace execution of a loop)

(The last problem can be handled by writing yet another program to process the diagnostic output.)

❖ ... Examining Program State

An alternative for obtaining access to program state:

- a tool that lets you stop program execution at a certain point
- then allows you to inspect the state (preferably selectively)

This is exactly the functionality provided by debuggers (e.g. **gdb**).

One useful special case ...

- inspect the state of a program that crashed due to a run-time error
- often, this takes you straight to the point where the bug occurred.

❖ C Program Execution

Under Linux, a C program executes either:

- to completion, producing results (correct?)
- until the program detects an error and calls **`exit()`**
- until an **`assert()`** check fails (triggers run-time error)
- until a run-time error halts it (e.g. **`Segmentation fault`**)

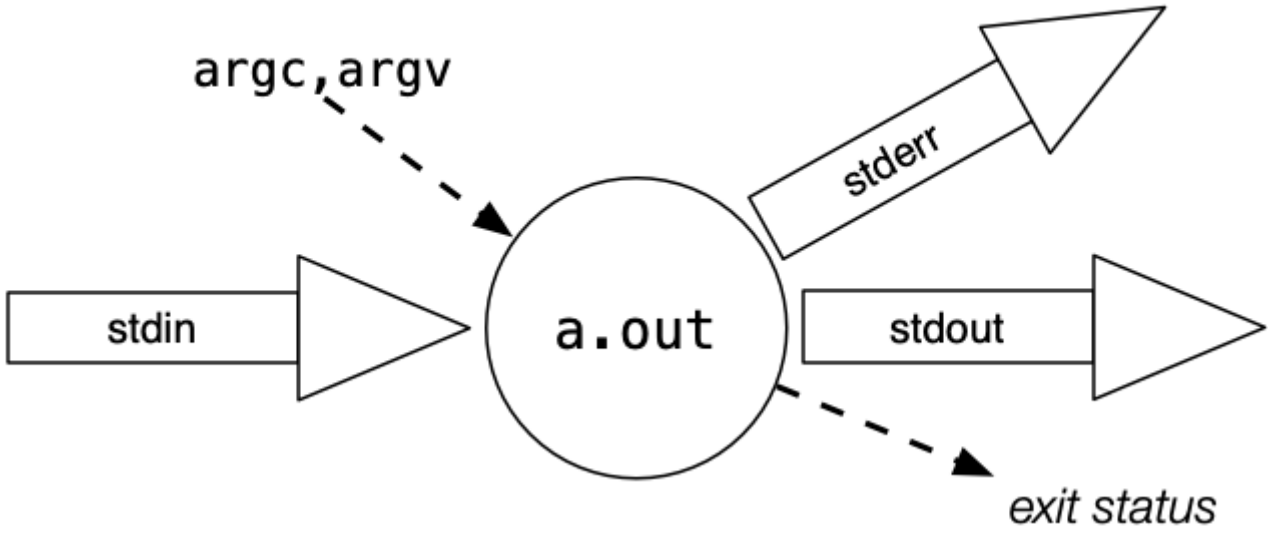
Executing Java and Python programs produce similar behaviour.

However, because they are interpreted, they may give informative error messages from run-time errors.

Note that **`dcc`** tries to achieve something similar for C programs

❖ Normal Program Execution

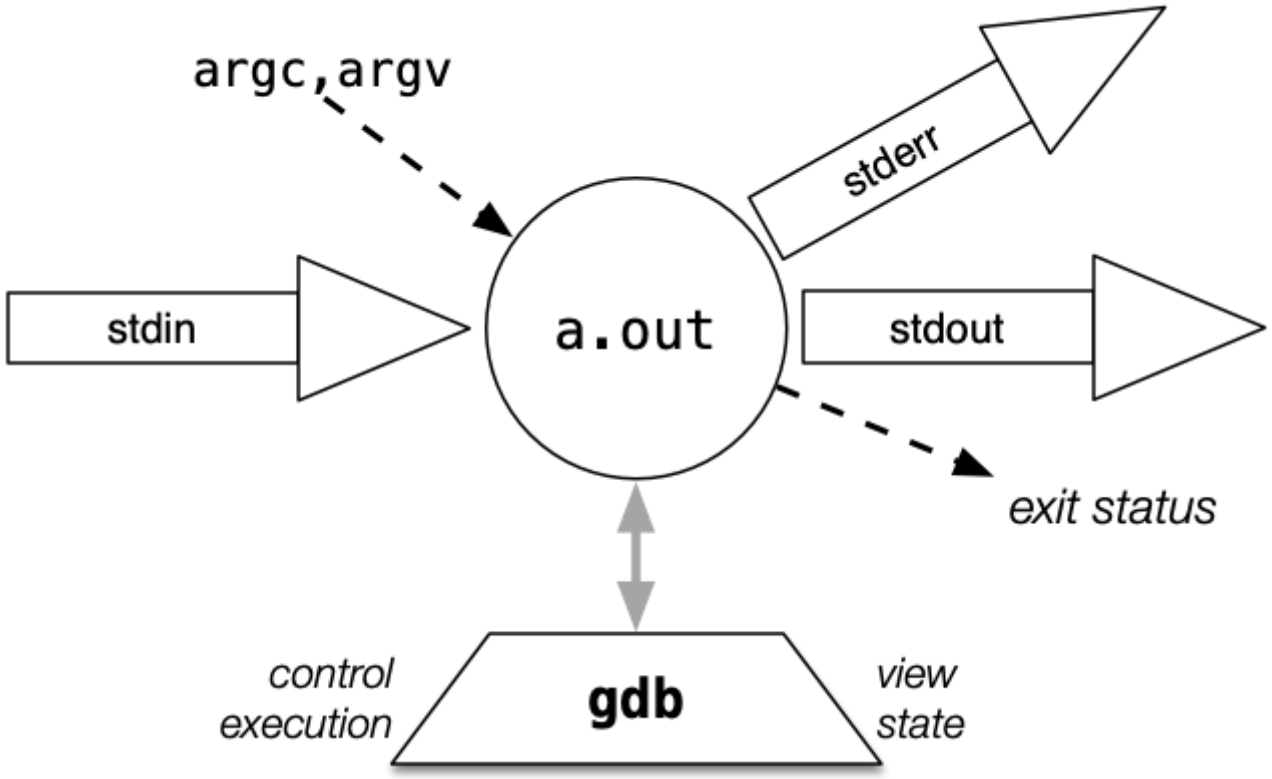
An executing program has multiple ways of getting data in and out.



The program runs autonomously until it finishes execution.

❖ Program Execution with Debugger

GDB adds ability to control execution and observe state interactively.



❖ Debuggers

A debugger gives control of program execution:

- normal execution (**`run`**, **`cont`**)
- stop at a certain point (**`break`**)
- one statement at a time (**`step`**, **`next`**)
- examine program state (**`print`**)

gdb command is a command-line-based debugger for C,C++ ...

There are GUI front-ends available (e.g. **xxgdb**, **ddd**, ...).

GUI development environments typically have a built-in debugger.

❖ ... Debuggers

To use programs with **gdb**, must be compiled with the **-g** flag.

```
$ gcc -g -o myprog myprog.c
```

gdb takes one or two command line arguments:

```
$ gdb executable [core]
```

E.g.

```
$ gdb  a.out  core
$ gdb  myprog
```

The **core** argument is optional.

❖ gdb Sessions

gdb is like a shell to control and monitor an executing C program.

Example session:

```
$ gcc -g -o prog prog.c
$ gdb -q prog
Reading symbols from prog ...done.
(gdb) break f
Breakpoint 1 at 0x1082c: file prog.c, line 32.
(gdb) run
Starting program: ..../prog
Enter a b c: 1 2 3
Breakpoint 1, f (i=1, j=2) at prog.c:32
32              a = i + j;
(gdb) next
33              b = i*i + j*j;
(gdb) next
34              return a*b;
(gdb) print a
$1 = 3
```

```
(gdb) print b
$2 = 5
(gdb) cont
...
```

❖ Basic gdb Commands

`quit` -- quits from **`gdb`**

`help [CMD]` -- on-line help

Gives information about **`CMD`** command.

`run` ***`ARGS`*** -- run the program

`ARGS` are whatever you normally use, e.g.

```
$ xyz  42  <  data
```

is achieved by:

```
(gdb) run  42  <  data
```

❖ gdb Status Commands

`where` -- stack trace

Find which function the program was executing when it crashed.

Stack may also have references to system error-handling functions.

`up [N]` -- move down the stack

Allows you to skip to scope of particular procedure in stack.

`list [LINE]` --- show code

Displays five lines either side of current statement.

❖ ... gdb Status Commands

`print` **EXPR** -- display expression values

EXPR can be (almost) almost any expression

```
print x
print a[5]
print a[i]
print stu[2].name
print cur->val
print cur->next->val
```

EXPR may use (current values of) variables.

Special expression **a@1** shows all of the array **a**.

❖ gdb Execution Commands

`break [PROC|LINE]` - set break-point

On entry to procedure **`PROC`** (or reaching line **`LINE`**), stop execution and return control to **gdb**.

`next` - single step (over procedures)

Execute next statement; if statement is a procedure call, execute entire procedure body.

`step` - single step (into procedures)

Execute next statement; if statement is a procedure call, go to first statement in procedure body.

For more details see **gdb**'s on-line help.

❖ Using a Debugger

Most common time to invoke a debugger: after a run-time error.

If this produces a **`core`** file, try

```
$ gdb -q prog core
```

The **`where`** command can tell you where the program crashed

Where the program crashed ...

- may be your first clue to the location of the bug.
- may be sometime after the buggy statement

❖ ... Using a Debugger

Once you have an idea where the bug might be:

- set breakpoints slightly earlier in the code
- run the program again (supplying the same data)
- single-step through the suspect region of code
- check the values of suspect variables after each step

This will eventually reveal a variable with an incorrect value.

❖ ... Using a Debugger

Once you find that the value of a given variable (e.g. **x**) is wrong, the next step is to determine *why* it is wrong.

Two possibilities:

- the statement that assigned a value to **x** is wrong
- the values of other variables used by that statement are wrong

Example: `if (c > 0) x = a + b;`

If we know that

- **x** is wrong after this statement
- the condition and expression are correct

then we need to find out where **a**, **b** and **c** were set.

❖ Other Debugging Ideas

Debugging is an example of applying a kind of scientific method:

- you develop a hypothesis
- then collect data to verify your hypothesis
- then change your hypothesis on the basis of new evidence

A potential hypothesis in the context of debugging:

"I think the bug is this statement here ..."

❖ Laws of Debugging

Courtesy of Zoltan Somogyi, Melbourne University

Before you can fix it, you must be able to break it (consistently).

(non-reproducible bugs ... Heisenbugs ... are extremely difficult to deal with)

Bug not where you're looking? You're looking in the wrong place.

(taking a break and resuming the debugging task later is generally a good idea)

It takes two people to find a subtle bug, but only one of them needs to know the program.

(the second person asks questions to challenge the debugger's assumptions)

(In fact, sometimes the second person doesn't have to do or say anything! The process of explaining the problem is often enough to trigger a Eureka event.)

❖ Possibly Untrue Assumptions

Debugging can be extremely frustrating when you make assumptions about the problem which turn out to be wrong.

Some things to be wary of:

- the executable comes from the code you're reading
- the problem *must* be in this source file
- the problem *cannot* be in this source file
- functions are never given unexpected arguments
  (e.g. "this pointer will never be **`NULL`**; why would anyone pass **`NULL`**?")
- library functions never return an error status
  (e.g. "**`malloc`** will always give the memory I ask for")
- the system libraries and tools are buggy (VERY UNLIKELY)

Produced: 2 Jun 2020