

# COMP1917: Computing 1

## 12. Debugging

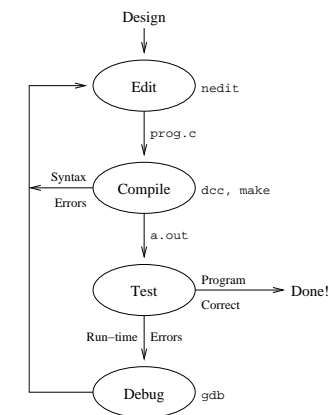
## Developing Programs

1. Understand the problem
2. Design data structures and algorithms
3. Implement design as a C program
4. Execute the program on test data
5. Check whether it gives correct results
6. **Repeat** above steps until it works ok

## Overview

- Programming cycle
- Do-it-yourself debugging
- Debugging with gdb
- Nastier bugs
- Memory leaks

## The Programming Cycle



## Compiler Errors

The C compiler will pick up superficial errors, and give warnings when it thinks a program “looks wrong”:

- mis-spelling of function or variable names
- forgetting to include a header file
- missing brackets, semicolons, etc.

## Linker Errors

The compiler produces object code for each module independently.

It assumes that all references to external names will be resolved later.

If you refer to an undefined name:

Undefined	first referenced
symbol	in file
<code>_print</code>	<code>program.o</code>

```
ld fatal: Symbol referencing errors.
No output written to a.out
```

Often caused by misspelling function names, or forgetting to include an external declaration or function prototype in your header file.

## Program Debugging

**Debugging:** process of **locating** and **fixing** bugs.

**Bug:** piece of program that does not do what you intended.

Consequences of bugs:

- run-time error (if you're lucky)
- incorrect results (if you're unlucky)

To perform debugging, you need to know:

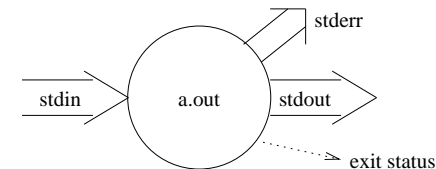
- in detail, what the code should do
- in detail, what it actually does

A **debugger** (e.g. `gdb`) can assist with the latter.

## Program Execution

Under Unix, a program executes either:

- to completion, producing results (correct?)
- until it detects an error and `exits`
- until a run-time error halts it (e.g. Segmentation violation - core dumped)



## Do-it-yourself Debugging

place `printf()`'s in your code to print intermediate values of variables

The `assert()` function can also be used to check assumptions about what should be true at certain points in your code.

```
#include <assert.h>

...
assert( expression );
```

if the asserted expression evaluates to 0 (False) the program will print a diagnostic message and halt.

## Making your own DEBUG mode

You must remember to remove debugging `printf()` statements from the final version of your program. `assert()` statements should also be removed, because they may slow down execution of the code.

Pre-processor directives can be used to switch debugging on and off:

```
#define DEBUG 1

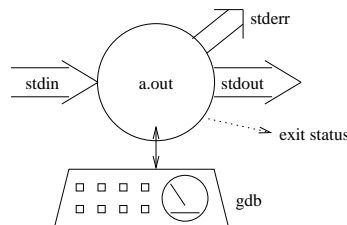
...
#ifdef DEBUG
    printf(" x = %d, y = %d\n", x, y );
#endif
```

When you think the bugs are all fixed, comment out the top line and all the debugging code will be suppressed.

## Debuggers

A debugger gives you 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`)



## The gdb debugger

`gdb` – a line-based interactive debugger.

`ddd` – an X-windows-based interface for `gdb`.

To use programs with `gdb` (or `ddd`), they must be compiled with the `gcc` compiler.

`gdb` (`ddd`) takes two arguments:

```
% gdb executable core
```

E.g.

```
% gdb a.out core
```

```
% gdb myprog
```

(The `core` argument is used when the program has already crashed.)

## gdb sessions

A [session](#) with gdb is a sequence of commands to control and observe the executable.

Command = sequence of words on single line.

```
% gcc -Wall -g -o prog prog.c
% gdb prog
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
(gdb) break f
Breakpoint 1 at 0x1082c: file prog.c, line 32.
(gdb) run
```

## gdb sessions

```
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.

```
% ./prog < data
```

is achieved by:

```
(gdb) run < 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 “up” the stack

Allows you to skip to “scope” of a particular function in stack.

- `list [LINE]` — show code

Displays five lines either side of current statement.

- `print EXPR` – display expression values

EXPR may use (current values of) variables.

Special expression `aat1` shows all of the array `a`.

## **gdb execution commands**

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

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

- **next** - single step (over functions)

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

- **step** - single step (into functions)

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

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

## **Nastier Bugs**

When a program really makes a mess of things, the debugger may get confused due to corruption of system memory.

There are other compilers like `dcc`, which are similar to `gcc` but do more checking. If the program crashes, or an array is overrun, etc., the compiler will (hopefully) give you some useful information:

```
*****
mudflap violation 47 (check/write): time=1161558399.248071
ptr=0xbfbf1aa8 size=4
pc=0xb7ee6347 location='overrun.c:10 (main)'
Segmentation fault
```

## **dcc error messages**

```
*****
mudflap violation 1 (check/write): time=1161558716.559908
ptr=0xbfb84fb0c size=4
pc=0xb7e44347 location='overrun.c:10 (main)'
  /usr/lib/libmudflap.so.0(__mf_check+0x37) [0xb7e44347]
  ./a.out(main+0x79) [0x8048761]
  /usr/lib/libmudflap.so.0(__wrap_main+0x176) [0xb7e44f66]
Nearby object 1: checked region begins 1B after and ends 4B after
mudflap object 0x80cb318: name='overrun.c:6 (main) a'
bounds=[0xbfb84fae4,0xbfb84fb0b] size=40 area=stack check=0r/10w
liveness=10
alloc time=1161558716.559778 pc=0xb7e44d97
number of nearby objects: 1
```

## **Memory Leaks and valgrind**

Sometimes a program eventually crashes due to a “memory leak” – a slow accumulation of memory blocks which are `malloc`'ed but never `free`'d.

Tools like `valgrind` can help you to track these memory leaks.

```
% gcc -g -o xyz xyz.c
% valgrind xyz
```

When the program terminates, `valgrind` will give you a summary of memory usage and report any memory blocks which have not been `free`'d.

Warning: `valgrind` makes the program run **very slowly** because it does so much checking.

## valgrind messages

---

```
==31180== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 11 from 1)
==31180== malloc/free: in use at exit: 64 bytes in 8 blocks.
==31180== malloc/free: 8 allocs, 0 frees, 64 bytes allocated.
==31180== For counts of detected errors, rerun with: -v
==31180== searching for pointers to 8 not-freed blocks.
==31180== checked 59,376 bytes.
==31180==
==31180== LEAK SUMMARY:
==31180==    definitely lost: 64 bytes in 8 blocks.
==31180==    possibly lost: 0 bytes in 0 blocks.
==31180==    still reachable: 0 bytes in 0 blocks.
==31180==    suppressed: 0 bytes in 0 blocks.
==31180== Use --leak-check=full to see details of leaked memory.
```