
ASTRODAT
AstroSTatistics and Research-Oriented DATa analysis
esa academy
UKRI Science and Technology Facilities Council
Durham University
DEBUGGING YOUR CODE
WITHOUT LOSING YOUR MIND
SARAH JOHNSTON
SHE/THEY | PHD STUDENT @ DURHAM UNIVERSITY
SARAH.C.JOHNSTON@DURHAM.AC.UK

In this session:

- What is debugging?
- Debugging Techniques
  - Rubber duck debugging
  - Print debugging
- Debugging Tools
  - C & Fortran (gdb, compilers)
  - Python (pdb, IDE-based)
  - Valgrind & Address Sanitizer
- Debugging Large Shared Projects
  - Testing
  - Minimal, reproducible examples
  - Using remote machines

WHAT IS DEBUGGING?

Debugging is the process of identifying, isolating, and fixing errors in code.

Debugging is important because it:

- makes our code work!
- ensures scientific accuracy
- keeps our software stable
- reduces chance of hardware errors

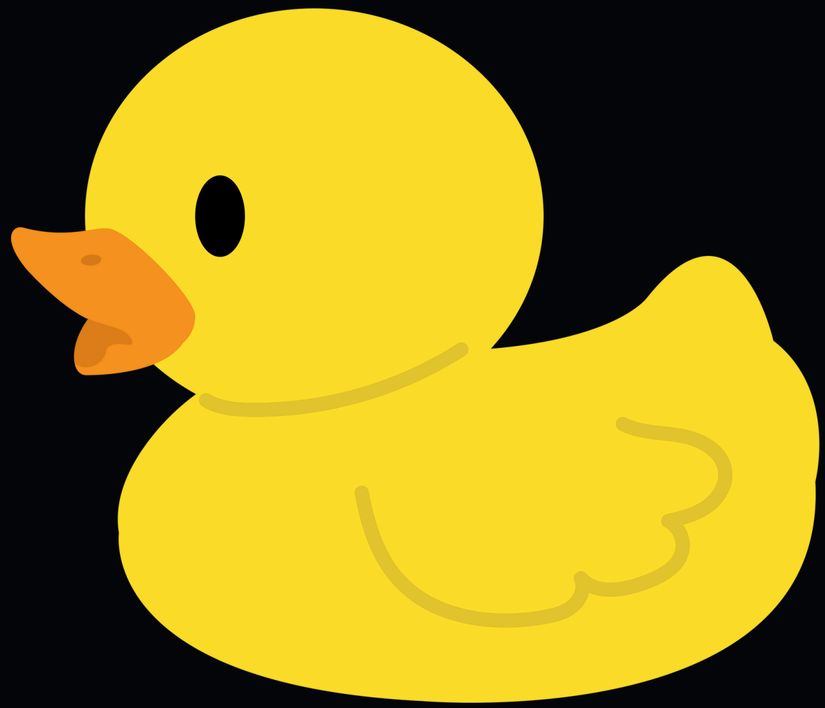

BASIC
TECHNIQUES

RUBBER DUCK DEBUGGING

Explain the problem with your code out loud!

Detatch yourself from your code and explain:

- what is happening
- where its happening in the code
- what evidence (if any) you have of what the fault could be
- what quantities and operations are involved



Sometimes just speaking through your problems can trigger ideas.

PRINT DEBUGGING

Include print statements in your code to show what is going on inside your code

Print statements are often used to show:

- where you are in the code
- what value a variable currently has
- how many iterations of a loop have been performed

Choose useful values to print!

Consider formalising this to assert() or error() statements


DEBUGGING
TOOLS

GDB

GNU debugger for unix-like systems on languages like C, C++, Fortran, Open CL, and Rust.

Breakpoint-based tool

Compile code with the -g flag

Open a session by typing gdb then use the commands to navigate your code

Command	Description
run or r	Executes the program from start to end.
break or b	Sets a breakpoint on a particular line.
disable	Disables a breakpoint
enable	Enables a disabled breakpoint.
next or n	Executes the next line of code without diving into functions.
step	Goes to the next instruction, diving into the function.
list or l	Displays the code.
print or p	Displays the value of a variable.
quit or q	Exits out of GDB.
clear	Clears all breakpoints.
continue	Continues normal execution

```c
#include <stdio.h>

int main()
{
        int x = 1;
        int a = 2 * x;
        int b = x + 3;
        int c = a + b;
        printf("%d\n", c);
        return 0;
}
```

```
gcc -std=c99 -g -o test test.c
```

```
gdb ./test
```

```c
#include <stdio.h>

int main()
{
        int x = 1;
        int a = 2 * x;
        int b = x + 3;
        int c = a + b;
        printf("%d\n", c);
        return 0;
}
```

```
(gdb) b 5
Breakpoint 1 at 0x40112e: file test.c, line 5.
(gdb) b 8
Breakpoint 2 at 0x401146: file test.c, line 8.
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x000000000040112e in main at test.c:5
2       breakpoint     keep y   0x0000000000401146 in main at test.c:8
```

```
(gdb) r
Starting program: /cosma/home/dp004/dc-john7/astrodat_debugging/test
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 1, main () at test.c:5
5               int x = 1;
```

```c
#include <stdio.h>

int main()
{
        int x = 1;
        int a = 2 * x;
        int b = x + 3;
        int c = a + b;
        printf("%d\n", c);
        return 0;
}
```

```
(gdb) continue
Continuing.

Breakpoint 2, main () at test.c:8
8               int c = a + b;
(gdb) p a
$1 = 2
(gdb) p b
$2 = 4
(gdb) p c
$3 = -1
(gdb) continue
Continuing.
6
[Inferior 1 (process 61465) exited normally]
```

COMPILERS

Compilers have flags for warnings and bugs (e.g. -Wall and -Wextra)

You can also try compiling with different compilers - some issues will flag in other compilers that may not flag in the one you are using

Consider verbose output

FOR PYTHON: traceback is the closest thing for compiler-style debugging functionality

PDB

Python interactive debugger

Similar breakpoint and stepping capabilities as gdb

Post-mortem debugging functionalities

Startup and Help

`python -m pdb <name>.py [args]`	begin the debugger
`help [command]`	view a list of commands, or view help for a specific command

within a python file:

```
import pdb
...
pdb.set_trace()
```

begin the debugger at this line when the file is run normally

Navigating Code (within the Pdb interpreter)

`l(ist)`	list 11 lines surrounding the current line
`w(here)`	display the file and line number of the current line
`n(ext)`	execute the current line
`s(tep)`	step into functions called at the current line
`r(eturn)`	execute until the current function's return is encountered

Controlling Execution

`b [#]`	create a breakpoint at line [#]
`b`	list breakpoints and their indices
`c(ontinue)`	execute until a breakpoint is encountered
`clear [#]`	clear breakpoint of index [#]

Changing Variables / Interacting with Code

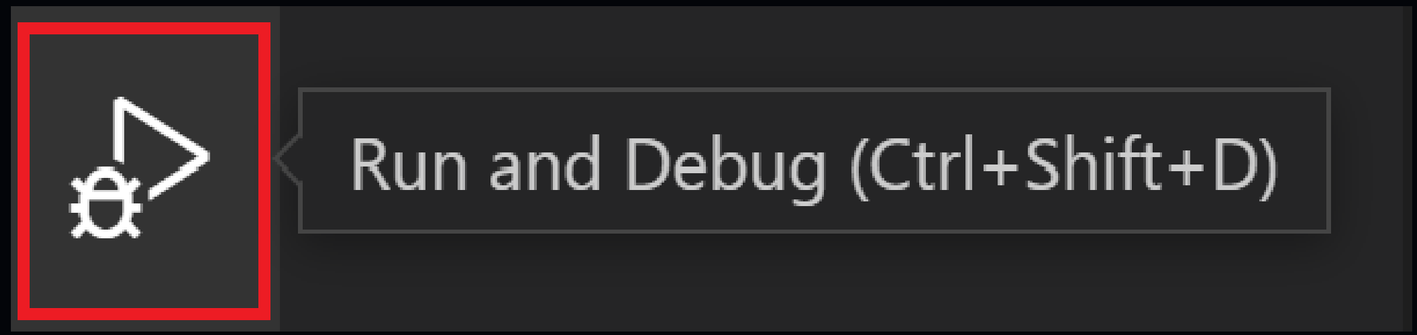
`p <name>`	print value of the variable <name>
`!<expr>`	execute the expression <expr> NOTE: this acts just like a python interpreter
`run [args]`	restart the debugger with sys.argv arguments [args]
`q(uit)`	exit the debugger

IDE DEBUGGERS

Popular IDEs with debugging features include:

- PyCharm
- Visual Studio Code (python extension)

- PyDev
- Komodo



VALGRIND & ASAN

Valgrind is a tool that looks for memory corruptions and leaks (however it slows your code down A LOT!)

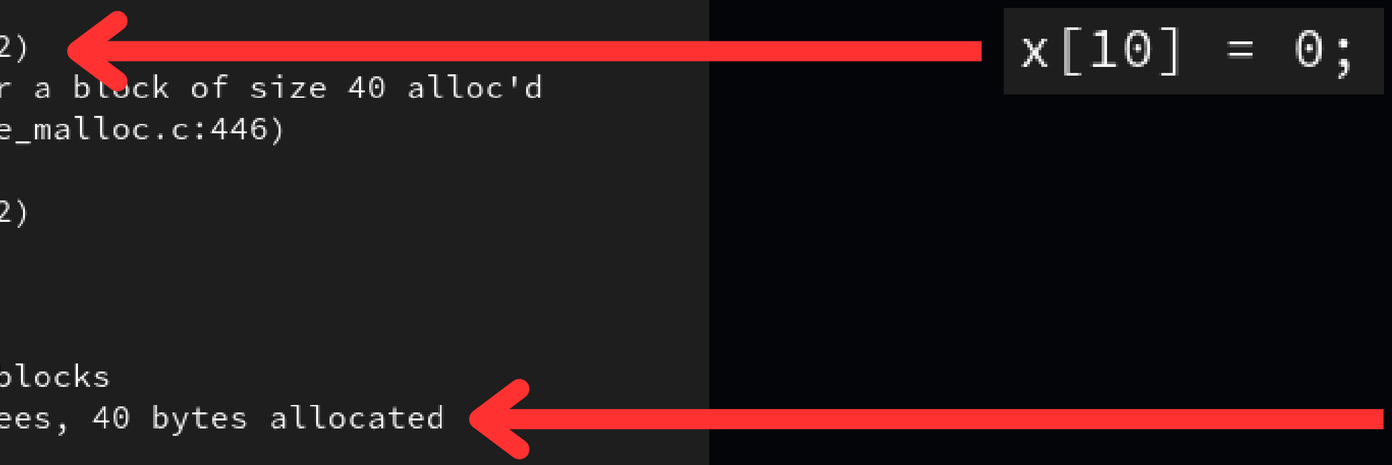
Address Sanitizer (ASAN) can also be used for memory debugging. It has fewer features than Valgrind but runs much faster

```c
#include <stdio.h>
#include <stdlib.h>

  void f(void)
  {
     int* x = malloc(10 * sizeof(int));
     x[10] = 0;          // problem 1: heap block overrun
  }                      // problem 2: memory leak -- x not freed

  int main(void)
  {
     f();
     return 0;
  }
```

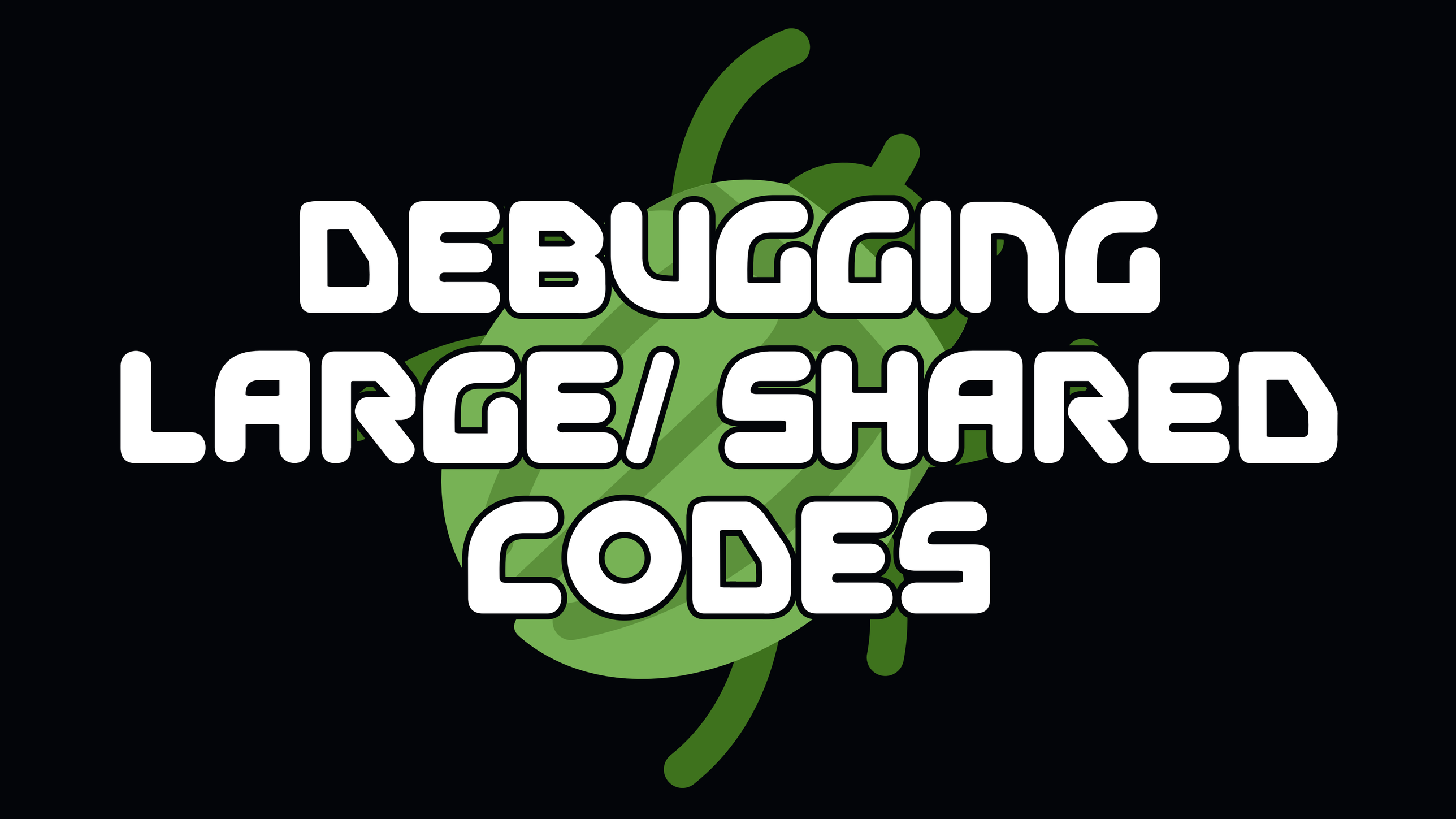
```
[dc-john7@login8a astrodat_debugging]$ valgrind --leak-check=yes ./memcheck.out
==150522== Memcheck, a memory error detector
==150522== Copyright (C) 2002-2024, and GNU GPL'd, by Julian Seward et al.
==150522== Using Valgrind-3.23.0 and LibVEX; rerun with -h for copyright info
==150522== Command: ./memcheck.out
==150522==
==150522== Invalid write of size 4
==150522==    at 0x401140: f (memcheck.c:7)
==150522==    by 0x401151: main (memcheck.c:12)
==150522==  Address 0x4a77068 is 0 bytes after a block of size 40 alloc'd
==150522==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==150522==    by 0x401133: f (memcheck.c:6)
==150522==    by 0x401151: main (memcheck.c:12)
==150522==
==150522==
==150522== HEAP SUMMARY:
==150522==     in use at exit: 40 bytes in 1 blocks
==150522==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==150522==
==150522== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==150522==    at 0x484482F: malloc (vg_replace_malloc.c:446)
==150522==    by 0x401133: f (memcheck.c:6)
==150522==    by 0x401151: main (memcheck.c:12)
==150522==
==150522== LEAK SUMMARY:
==150522==    definitely lost: 40 bytes in 1 blocks
==150522==    indirectly lost: 0 bytes in 0 blocks
==150522==      possibly lost: 0 bytes in 0 blocks
==150522==    still reachable: 0 bytes in 0 blocks
==150522==         suppressed: 0 bytes in 0 blocks
==150522==
==150522== For lists of detected and suppressed errors, rerun with: -s
==150522== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```



```
[dc-john7@login8a astrodat_debugging]$ gcc -g -fsanitize=address -static-libasan memcheck.c -o memcheck.out
[dc-john7@login8a astrodat_debugging]$ ./memcheck.out
=================================================================
==198378==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x604000000078 at pc 0x000000506b5c bp 0x7ffc402fa5a0 sp 0x7ffc402fa598
WRITE of size 4 at 0x604000000078 thread T0
    #0 0x506b5b in f /cosma/home/dp004/dc-john7/astrodat_debugging/memcheck.c:7
    #1 0x506b6d in main /cosma/home/dp004/dc-john7/astrodat_debugging/memcheck.c:12
    #2 0x14ff362295cf in __libc_start_call_main (/lib64/libc.so.6+0x295cf) (BuildId: fc46bc419367003d0e4e399cbe22aade4a1ee7be)
    #3 0x14ff3622967f in __libc_start_main@GLIBC_2.2.5 (/lib64/libc.so.6+0x2967f) (BuildId: fc46bc419367003d0e4e399cbe22aade4a1ee7be)
    #4 0x407364 in _start (/cosma/home/dp004/dc-john7/astrodat_debugging/memcheck.out+0x407364)

0x604000000078 is located 0 bytes after 40-byte region [0x604000000050,0x604000000078)
allocated by thread T0 here:
    #0 0x4b6acf in __interceptor_malloc ../../../../libsanitizer/asan/asan_malloc_linux.cpp:69
    #1 0x506b18 in f /cosma/home/dp004/dc-john7/astrodat_debugging/memcheck.c:6
    #2 0x506b6d in main /cosma/home/dp004/dc-john7/astrodat_debugging/memcheck.c:12
    #3 0x14ff362295cf in __libc_start_call_main (/lib64/libc.so.6+0x295cf) (BuildId: fc46bc419367003d0e4e399cbe22aade4a1ee7be)
```

```
SUMMARY: AddressSanitizer: heap-buffer-overflow /cosma/home/dp004/dc-john7/astrodat_debugging/memcheck.c:7 in f
Shadow bytes around the buggy address:
  0x603fffffffd80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x603fffffffe00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x603fffffffe80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x603ffffffff00: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
  0x603ffffffff80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
=>0x6040000000000: fa fa 00 00 00 00 00 00 fa fa 00 00 00 00 00[fa]
  0x6040000000080: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x6040000000100: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x6040000000180: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x6040000000200: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
  0x6040000000280: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
  Addressable:           00
  Partially addressable: 01 02 03 04 05 06 07
  Heap left redzone:       fa
  Freed heap region:       fd
  Stack left redzone:      f1
  Stack mid redzone:       f2
  Stack right redzone:     f3
  Stack after return:      f5
  Stack use after scope:   f8
  Global redzone:          f9
  Global init order:       f6
  Poisoned by user:        f7
  Container overflow:      fc
  Array cookie:            ac
  Intra object redzone:    bb
  ASan internal:           fe
  Left alloca redzone:     ca
  Right alloca redzone:    cb
```


DEBUGGING
LARGE/ SHARED
CODES

TESTING

The best form of debugging is catching your bugs before they are even a problem!

Unit testing - small units of your code (functions or classes especially!)

System testing - predefined tests of your whole code that you can compare to known values

Continuous integration - tests that happen at every update e.g. through GitHub

MINIMAL REPRODUCIBLE EXAMPLE

If your code takes a long time to run then you need smaller sections to debug

- Isolate and extract just the code that fails and its inputs
- Put it into a separate program
- Run and debug the smaller program

Keep trying to shrink the code in on the problem area

When fixed comment or document any bugs so others don’t spend their time doing the same thing

LARGE COLLABORATIONS

- Keep an error Wiki
- Have help channels
- Onboarding guide or FAQ documentation
- Ask other people as a first point of call
- Keep accountability for errors and bugs!

USING REMOTE MACHINES

Ask system admins about installing debugging tools in an accessible way for all users rather than just building your own version

Keep an eye on system use metrics e.g. through top or smi

Check error logs

Connect up code editors with debugging help inbuilt (e.g. VS Code)


HANDS ON EXERCISES:
https://github.com/
sarahcjohnston/
ASTRODAT_debugging

SARAH JOHNSTON

SHE/THEY | DURHAM UNIVERSITY

SARAH.C.JOHNSTON@DURHAM.AC.UK