Using the GNU debugger (gdb)

A debugger is a very useful tool for finding bugs in a program. You can interact with a program while it is running, start and stop it whenever, inspect the current values of variables and modify them, etc.

If your program runs and crashes, it will produce a 'core dump'. You can also use a debugger to look at the core dump and give you extra information about where the crash happened and what triggered it.

Some debuggers (including recent versions of gdb) can also go *backwards* through your code: you run your code forwards in time to the point of an error, and then go backwards looking at the values of the key variables until you get to the start of the error. This can be slow but useful sometimes!

To use a debugger effectively, you need to get the compiler to put extra 'symbol' information into the binary, otherwise all it will contain is machine code level – it is much more useful to have the actual variable names you used. To do this, you use:

```
gfortran –g –O0 mycode.f90 –o mybinary
```

where '-g' is the compiler option to include extra symbols, -O0 is no optimization so the code is compiled exactly as written, and the output binary is called 'mybinary'.

If the source files and executable file is in the same directory, then you can run the binary through the debugger by simply doing:

```
gdb ./mybinary
```

This will then put you into an interactive debugging session. Most commands can be shortened (eg 'b' instead of 'break') and pressing 'enter' will repeat the last command. Useful gdb commands include:

gdb command	Example	Explanation
break	break 20	Sets a breakpoint which will cause the execution to pause when it reaches line number 20
list	list 10,20	List source code lines
run	run	Starts program executing
print	print var	Print current value of local variable 'var' (lowercase)
	print modname::var	Print current value of variable 'var' that is defined in module 'modname' and used here
	p *((double *)x)@10	Print first 10 elements of double precision allocatable array 'x'
	P *((integer *)x+1)	Print x(2) where x is integer allocatable array
step	step 20	Execute next 20 lines
reverse-step	reverse-step 20	Step backwards 20 lines
next	next 10	Execute next 10 lines, stepping over any function calls
continue	cont	Continue from here until next breakpoint (or end)
where	where	Give current position in code (aka backtrace)
display	disp var	Give current value of 'var' every time the program stops. Can also use expression instead of 'var'.
watch	watch var	Sets a watchpoint, which will cause the execution to pause whenever the value of 'var' changes. Can also use expression instead of 'var'.
set	set var=value	Change current value of 'var' to 'value'
delete	delete 2	Deletes the breakpoint number 2
help	help step	Built-in help function – can get help on specific commands (eg step) or generic (eg help data). There are many more gdb commands than these few!!!
quit	quit	Stop debugging, kill current process and end gdb session

GUI

Using gdb can be a bit intimidating. As it is a general debugger for the entire GNU Compiler Collection (gcc) it includes support for C, C++, etc. and has only recently adopted Fortran, so some of the syntax is very C-biased e.g. accessing elements of an array is done via pointers and offsets, and assumes that arrays start with element zero etc.

One way to make it easier to use is to access gdb via a GUI. There is one built-in GUI:

```
gdb -tui ./my_binary
```

which is a 'Text User Interface' and so will work with any terminal. This gives a split screen display and make it easy to see the current line, where the breakpoints are, etc. If you use this, you should be careful to not write anything to the screen when running the code, so you will need to use redirection with gdb, i.e. start your program running using

```
run > a.log
```

so that any output is sent to the file a.log and will not interfere with the TUI.

Other 'proper' GUIs include `ddd` and `eclipse` and `kdbg` and `emacs` …

Using emacs with gdb

Emacs is a very powerful text editor with many features for programmers. It has a built-in F90 mode that does syntax highlighting, auto-indenting, etc. It also has built-in support for gdb! If you are editing your file in emacs, then at any time you can type '`Alt+x gdb`' to get a gdb command line. The menus at the top of the screen then switch to being gdb specifc. You may want to split the window (File/Split or '`Ctrl+x 2`') and set one window to be your code source (Buffers/<mycode.f90>). You can then easily switch between source and gdb by clicking in the appropriate window. If you now create a breakpoint in your gdb window, it shows in your source window as a red dot next to the appropriate line. You can then use gdb comments to step through your code (or click the appropriate toolbar icons) and you will see an arrow that moves along with the code pointer. The GUD menu makes it very easy to access many gdb functions …

Coredumps

If your program crashes then it may write a coredump – a binary file that is the content of memory at the time of the crash. You can use gdb to see what went wrong:

```
gdb -c core
```

which will show you where in your program the crash happened. You can also use gdb commands like `backtrace` to see the call stack, which shows the current function at the top, and the list of calling functions in order underneath. This is useful to find out how your program got to the point of the crash.

MIJP 09/06/2017