Testing

- Unit Testing
  - Think through at the design stage how the overall program can be built from smaller units
  - And how each of those units can be tested in isolation
  - And how to test the interfaces/interactions between these units in combination
- Be systematic!
  - Document the tests, inputs & outputs.
  - Random/ad-hoc testing is very suspect
  - Try to ensure all the code is covered in your testing! Analysis and testing tools can help here – more in later lectures.
  - Regression testing – make sure any new additions do not affect earlier functionality
- Modular approach to design, coding & testing

Validation

- What simple tests can you apply to validate a given unit? Or whole program?
  - Some simple theoretical limits (e.g. energy conservation in MD program)?
  - Specified inputs that give a known result? Also useful for checking later modifications!
- Check robustness of code to compiler flags
  - Do all initial testing with maximum debugging & compiler support and NO optimisation (e.g. `f90 -g -O0`).
  - Only turn on compiler optimisation once code is correct and check it does not change anything significantly!

Debugging

- If testing shows up a problem, how do you fix it? Just as with testing, need to be systematic – use the scientific method!
  1. Gather information and form a hypothesis
  2. Test your hypothesis
  3. Iterate until proven hypothesis is found
  4. Propose and test solution
  5. Iterate until solution is proven correct
  6. Regression test – has your fix broken something else?!

Common Types of Bug

- Memory/Resource leaks
  - Make sure that everything allocated is always deallocated (once) when finished
  - Some other resources are very limited (e.g. window handles, or file unit numbers, etc)
- Logic Errors
  - Syntactically correct but code does not perform as expected. Impossible to catch with automated tools.
  - Needs rigorous testing of different sets of inputs to show up an odd/unexpected behaviour
- Coding Errors
  - E.g. parameter type and/or number mismatch, or exceeding valid input range of a routine, etc.

More Common Bugs …

- Memory Overruns
  - Beloved of hackers! Basically, trying to access a bit of memory that does not belong to you. E.g. copying too long a string or going beyond an array bound.
  - Compiler flags can help catch this at runtime with some languages.
  - Common symptom – a strange numerical result or crash that goes away when you insert a print statement to try to see in more detail what is going on! Or when add an additional variable definition, etc.
  - This is an anti-fix – it masks the real problem and makes it appear to go away but in fact it makes the real problem even harder to find!

… and some more …

- Loop Errors
  - Infinite loops – make sure loop has a guaranteed exit strategy
  - Off-by-one loops – does your loop or array index go from 1 to N or 0 to N-1? Exit condition?
- Conditional Errors
  - Boolean logic mangled, e.g. testing for $x<0.0$ or $x\leq0.0$?
  - Beware nested ‘if-blocks’ – easy to get a missing ‘else’ clause. Always make sure that there is a catch-all clause.
  - Case statements better – add a ‘case default’
- Pointer Errors
  - Memory style – beware uninitialised pointers (F90 as well as C-style languages although situation better in F95), pointers to deallocated blocks of memory, or pointers to wrong location
  - Also other situations, e.g. integers used for array indices?

… and there's more …

- Integration Errors
  - i.e. when units pass tests OK but fail when put together
  - Usually due to inappropriate assumptions in one unit, e.g. will only work with data within certain ranges
- Storage Errors
  - What if your program wrote out an intermediate state of the calculation, in order to read it later to continue executing?
  - E.g. trying to create a file with an invalid filename, or file system becoming full, or file being locked by another application, or wrong access permissions, etc.
- Conversion Errors
  - SI vs. imperial units led to loss of NASA Mars Climate Orbiter in 1998. Converting 64-bit floating point variable to 16-bit signed integer led to loss of ESA Arianne 5 in 1996.
  - E.g. integer division, 12/24=0, but 12.0/24.0=0.5 …

… and finally …

- Hard-coded Lengths/Sizes
  - Magic numbers at random points in code makes changes hard to localise and code hard to maintain.
  - Much better to have assigned to variables or parameters with meaningful names and use name everywhere not value
- Versioning Bugs
  - Make file formats robust to future developments and build in versioning info – otherwise will get strange results when using old data in newer program version
- Inappropriate Reuse Bugs
  - Must always carefully unit test code, even when reusing old “trusted and reliable” code as it may be being used in a new way not originally planned & tested for!