Advanced Programming

Debugging

Advanced Programming

Debugging: Types of Bugs

- Debugging is the process of identifying and fixing errors in code
- Almost impossible to avoid, so must know how to identify and fix them
- Common types of error in C++ code are:
    - Compile-time errors
        - Syntax error (eg. missing `;` or `)`) or type error (eg. `int x = “hello”;`)
        - These are usually flagged during compilation
        - The program will not compile until the error is fixed
        - Usually easy to identify

Advanced Programming

Debugging: Types of Bugs

- Linker error
  - Occurs during the ‘linking’ stage of compilation, program will not compile
  - Can occur if eg.
    - Included header file is not found
    - `main()` is incorrectly written as eg. `Main()`
    - A function or class is declared, but never defined, eg.

```
int func();

int main() { int x = func(); }
```

Advanced Programming

Debugging: Types of Bugs

- Runtime error
  - The code will compile and run, but will crash or give an unexpected result, such as:
    - Segmentation fault - the program tries to access a memory location that is not allowed, eg. accessing `array[10]` when we have only defined `array[5]`
    - `NaN` - this stands for ‘not a number’ and occurs most commonly when the code has attempted to divide a number by zero
  - Can be tricky to find, experience often helps

Advanced Programming

Debugging: Types of Bugs

- Logic or other semantic error (type of runtime error)

  - Again, the code will usually compile and run, but will give a result that we know is wrong, eg.:

```
int add(int x, int y){ return x-y; }
int main() {
      cout << “5+3=“ << add(5,3) << ‘\n’;
      return 0;
 }
```

  - These can be very difficult to identify, and can cause subtly incorrect results

  - Can mitigate against these by comparing with output from another code, or using own physics/maths knowledge

Advanced Programming

Debugging: Types of Bugs

Example: Compile `data_reader_class_buggy.cpp`

First, some setup (some of this will be used later) - this is to enable us to easily edit files within the container:

- Pull the latest commit from the course repository
- Open Docker on your machine (laptop)
- Install ‘Remote Development’ extension pack in VSCode

Advanced Programming

Debugging: Types of Bugs

Running the Docker container:

- Rebuild the Docker image in VSCode (View -> Command Palette -> Docker Images: Build Image)

- ‘Run interactive’ Docker container

Next, we want to be able to use VSCode tools from within our container:

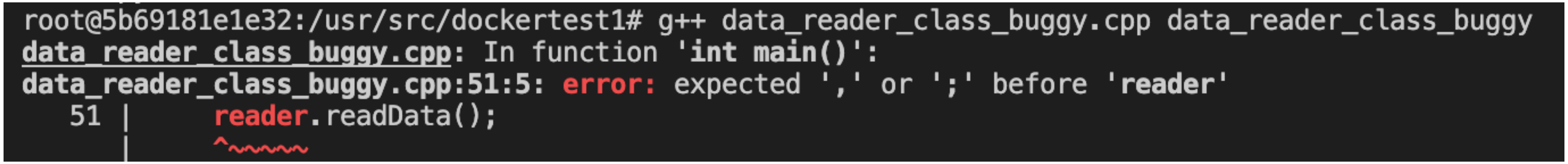
- View -> Command Palette… -> Attach to running container

- Open in new window `/usr/src/dockertest1/data_reader_class_buggy.cpp`

Advanced Programming

Debugging: Types of Bugs

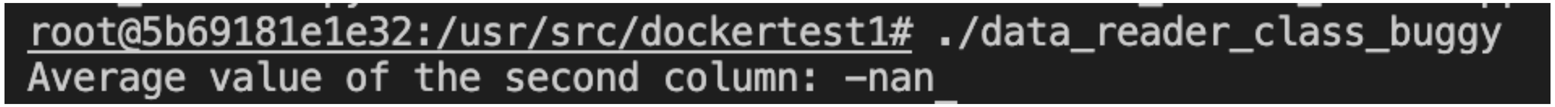
- From within the container, we can now compile the example
- We first see a *syntax error* at compile time - how do we trace it?



```
root@5b69181e1e32:/usr/src/dockertest1# g++ data_reader_class_buggy.cpp data_reader_class_buggy
data_reader_class_buggy.cpp: In function 'int main()':
data_reader_class_buggy.cpp:51:5: error: expected ',' or ';' before 'reader'
   51 |     reader.readData();
      |     ^~~~~~
```

Example: Fix the syntax error, recompile and run

- The code now complies, but running gives a runtime error



```
root@5b69181e1e32:/usr/src/dockertest1# ./data_reader_class_buggy
Average value of the second column: -nan
```

- How do we determine the cause of this bug?

Advanced Programming

Debugging: Methods

- There are two primary `code-based’ methods of debugging:
    - Print debugging - coder adds print statements at strategic points to check whether the code runs to those points
    - Using a debugger program - coder steps through the source code to find sources of error
- Even better than debugging is to add in your own error messages in advance to anticipate potential problems
    - Note that we included these in the `data_reader_class.cpp` example - would this have helped us catch the runtime error above?

Advanced Programming

Debugging: Methods

- Another very useful method is ‘rubber duck debugging’
- Essentially, this means using your own brain rather than outsourcing the solution to the computer
- Go through the section of code that you suspect contains the bug, line by line, explaining what each line does
- ‘Rubber duck’ refers to any object that you can pretend you are explaining to
- Unless you are lucky, real humans tend not to want to listen to the minutiae of your code implementation 😇 (unless you are really stuck, which is where RSEs come in)

Advanced Programming

Debugging: Methods

- Finally, we can debug programs by checking that the output makes sense in and of itself, as well as in comparison to previous output

- This can be done by:

  - Visualising the output (we will come to this later, especially useful with 3D grid output)

  - Plotting global variables, e.g. some physical quantity, such as total energy density

  - Comparing the above with output before the latest change (can use the Linux command `diff`)

  - If running in parallel, compare parallel to serial output

Advanced Programming

Debugging: Print Debugging

Advantages of print debugging:

- Easy to add print statements into code
- No need to learn to use other tools
- Especially useful in serial codes - it is clear where the program has run to
- If you have an idea of what might be going wrong, you can print out specific variables to double check

Advanced Programming

Debugging: Print Debugging

Disadvantages of print debugging:

- In compiled codes, must recompile and run each time you want to move the print statements
- Can take time if you have to queue your simulation on a large machine
- Doesn’t always catch errors
- Need to go back through at the end and remove the statements that you have added
    - This can be avoided by using a specific debug flag in your Makefile

Advanced Programming

Debugging: Print Debugging

Example: Compile `data_reader_class_buggy.cpp` *with additional print statement(s) to help locate the bug*

- We notice that `calculateAverage()` is the function that returns the error
- Might want to print out the variables that it uses in its calculation
- See that `columnData.size()` returns zero

If we are still stuck, we can use a debugger program…

Advanced Programming

Debugging: Debuggers

Advantages of debuggers:

- Can obtain detailed information about variables and memory locations while it executes
- Can step through code line by line
- Can stop execution at any point using a *breakpoint*
- Some integrated with code editors for easy use
- Some debuggers designed for parallel code

Advanced Programming

Debugging: Debuggers

Disadvantages of debuggers:

- Have to learn to use (especially if using from command line)
- Setup and use can be complex
- Can lead to focussing too much on minute details, rather than the bigger picture
- Doesn’t always catch errors

Advanced Programming

Debugging: Debuggers

Some examples of debuggers:

- GDB (`GNU Project Debugger’ - https://www.sourceware.org/gdb/)
    - Main debugger used by VSCode (C/C++ Extension Pack)
    - Debugs serial applications
- Valgrind (https://valgrind.org/)
    - For debugging and profiling Linux programs
    - Can use for parallel programs
- Linaro DDT (formerly Allinea DDT) (DDT - distributed debugging tool)
    - Graphical debugger for parallel programs

Advanced Programming

Debugging: Debuggers

Example - Debugging in Docker Container in VSCode:

- Click ‘Run and Debug’ in tab on left
- Choose preferred compiler from drop down menu and ‘debug active file’
- This allows us to use GDB to examine the code
- At first, the code runs and no obvious bug is identified - we must add *breakpoints* to examine variables which we suspect
- In VSCode, click on the left of a line number where you would like the execution to stop