Advanced Programming

Debugging: Debuggers

Example - Debugging in Docker Container in VSCode:

- Click 'Run and Debug' again - the code will stop at the breakpoint, we can inspect the variables in the panel on the left
- Can use the icons at the top to step over/into/out
- If you add a breakpoint, two new windows will appear on the left:
  - Call Stack - shows which functions have been called to reach the line with the breakpoint
  - Watch - allows you to type in variable names that you want to watch throughout the execution

Advanced Programming

Debugging: GDB

- We can also run GDB from the command line (installed via Dockerfile):
    - `cd /usr/src/dockertest1/`
    - `gdb data_class_reader_buggy`
- We then add breakpoints/watchpoints and control the program flow using command line
    - The sample session (https://sourceware.org/gdb/current/onlinedocs/gdb.html/Sample-Session.html#Sample-Session) is a useful starting point

Advanced Programming

Debugging: GDB

- Add breakpoint with eg. `break data_reader_class_buggy.cpp:47`
- Run with `run` (or `r`)
- Use `n` for next, `s` to step into a subroutine, `c` to continue, `p` to print a value
- `bt` gives a backtrace to see where we are in the stack

```
(gdb) break data_reader_class_buggy.cpp:47
Breakpoint 1 at 0x26b7: file /usr/src/dockertest1/data_reader_class_buggy.cpp, line 48.
(gdb) run
Starting program: /usr/src/dockertest1/data_reader_class_buggy
warning: Error disabling address space randomization: Operation not permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at /usr/src/dockertest1/data_reader_class_buggy.cpp:48
48          CSVReader reader(filename);
(gdb)
```

Advanced Programming

Debugging: GDB

Other useful features are:

- Calling functions from command line eg. `call calculateAverage()`
- Adding watch points eg. watch `average`
  - Program must be running and stopped at a breakpoint
- Inspecting the code without running eg. look at specific memory address, `l *0x8000000000000`
- Inspect core dumps eg. to find out which line caused a seg fault
- Define your own commands via `~/.gdbinit`

Advanced Programming

Debugging: Valgrind

- Valgrind is open source `instrumentation framework’ for building dynamic analysis tools
- This includes profiling tools (see later) and debugging
- For debugging, Valgrind is most useful for detecting memory-related errors and threading errors for parallel applications
- Most popular (and default) Valgrind tool is `memcheck`
- To use Valgrind, first compile your program with `-g` flag to include debugging information
  - This will ensure that error messages include exact line numbers
- Do not compile with any optimisation flags higher than `-O0`

Advanced Programming

Debugging: Valgrind

- *Example: run memory check using*

  `valgrind --leak-check=yes ./data_reader_class_buggy`

  (this enables the detailed memory leak detector)

- To change from default tool `memcheck`, add `--tool=`
- Program is run on a `synthetic' CPU provided by Valgrind core
- `Memcheck` adds code to check every memory access and computed value
- Note: program will run 20-30x slower and use a lot more memory

Advanced Programming

Debugging: Valgrind

- Example output for `data_reader_class_buggy`:

```
root@526151d9bd1f:/usr/src/dockertest1# valgrind --leak-check=yes ./data_reader_class_buggy
==4655== Memcheck, a memory error detector
==4655== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4655== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4655== Command: ./data_reader_class_buggy
==4655==
Average value of the second column: -nan
==4655==
==4655== HEAP SUMMARY:
==4655==     in use at exit: 0 bytes in 0 blocks
==4655==   total heap usage: 3 allocs, 3 frees, 74,200 bytes allocated
==4655==
==4655== All heap blocks were freed -- no leaks are possible
==4655==
==4655== For lists of detected and suppressed errors, rerun with: -s
==4655== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@526151d9bd1f:/usr/src/dockertest1#
```

Advanced Programming

Debugging: Valgrind

- For more detailed debugging, Valgrind can interface with gdb

Example: run the command

```
valgrind --vgdb-error=0 /.data_reader_class_buggy
```

then follow instructions provided in the terminal

- Start gdb in another shell
- Tunnel to vgdb (output from first command is input to second command)
- Can debug with gdb commands and combine with eg. the command `monitor` to use Valgrind tools

Advanced Programming

Profiling

Advanced Programming

Profiling

- Profiling is the measuring of the performance of a program
- Can identify where the program spends its time/memory
- Usually used to identify bottlenecks to enable optimisation
- It can also help to identify bugs, if certain functions are being called more or less than expected
- Profiling can be carried out by:
  - Putting timers manually into the program
  - Using software eg. gprof, Valgrind, VTune
    - Note: there is no straightforward profiling tool built into VSCode, as there is for debugging (other than for JavaScript)

Advanced Programming

Profiling

- Performance is primarily measured in two ways:
  - Wall time - total time elapsed while a certain section of the code is running
  - Processor time
    - You will often see the phrase `CPU time’ used as a measurement
    - Nowadays, GPUs are becoming increasingly popular, so `GPU time’ will likely also be widely used
    - The concept behind both is the same - the total time taken by the processor to run part of the code
    - This excludes communication time, I/O and other costs

Advanced Programming

Profiling

- Simple profiling can be done using the Unix `time` command
  - `real` - total time for program to run from start to finish i.e. wall time
  - `user` - CPU time spent in user mode
  - `sys` - CPU time spent in kernel mode (access hardware components etc.)
  - *Example: Try this on one of the examples from the class repo*

```
root@526151d9bd1f:/usr/src/dockertest1# time ./data_reader_class_buggy
Average value of the second column: -nan

real    0m0.020s
user    0m0.013s
sys     0m0.007s
```

Advanced Programming

Profiling

- For more detailed profiling, you will need to use additional tools:
  - Timers - most coding languages will have a timer module that can be used to write timers into the code
  - Profilers - these are programs that can tell the user where the time running code is being spent, eg. functions, wait time for parallel programs
- (There is a parallel here with print debugging vs software debuggers)
- In both cases, timing data can fluctuate - usually a good idea to take an average/median

Advanced Programming

Profiling: Instrumenting with Timers

Advantages:

- Straightforward to implement
- Straightforward to run and interpret
- You can specify exactly which parts you want to time

Disadvantages:

- Takes time to implement in a large code
- Can easily miss sections if implemented sporadically
- Can provide incomplete information eg. Wall time vs CPU time

Advanced Programming

Profiling: Instrumenting with Timers

- Most up to date timer for C++ is `<chrono>` - measures wall time
- Another commonly used library in C++ and C is `time.h` (`<ctime>`)
    - Can measure CPU (`<time.h>`) or wall time (`<sys/time.h>`)
    - Use of this library is sometimes discouraged (out of date, not thread safe)
- Equivalent timers exist in other languages eg. Python `time` and Fortran `CPU_TIME`

Advanced Programming

Profiling: Instrumenting with Timers

- `<chrono>` - main functions used are `high_resolution_clock` and `duration_cast` (converts time into desired measurement, eg. milliseconds)

Example 1: Compile `chrono_timer.cpp` *and run a few times*

- `time.h` - main function is `clock()` and macro `CLOCKS_PER_SEC`

Example 2: Compile `timeh_time.cpp` *and run a few times*

Advanced Programming

Profiling: Profilers

Advantages:

- Can quickly identify bottlenecks, especially useful for parallel code
- Some have user-friendly graphical interfaces

Disadvantages:

- Need to learn how to use
- Sometimes unavailable on large machines, need to arrange installation
- Can add overhead which skews performance data

Advanced Programming

Profiling: gprof

- `gprof` (GNU profiler) is an open source profiler for Unix applications
- It provides the user with (https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html)
  - *Flat profile* - how much time the program spends in each function, and how many times that function is called
  - *Call graph* - which functions called each function, and which other functions they called. This also estimates how much time is spent in the subroutines of each function
  - *Annotated source listing* - copy of program's source code, labelled with number of times each line is executed, `-A` flag

Advanced Programming

Profiling: gprof

- Before profiling with `gprof`, the program must be compiled with the flag `-pg`
  - The `-g` flag can also be useful for line-by-line profiling and basic-block counting
- The program must be run before profiling to generate the information for `gprof`
  - It will run slower due to the time taken to collect and write the profile
- Running a program compiled with the `-pg` flag will generate a file called `gmon.out` - this is the profiling information
- We can then run `gprof` with e.g. `gprof chrono_timer > output`

Example: Run on eg. chrono_timer.cpp and examine output

Advanced Programming

Profiling: gprof

```
^L
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.33% of 3.02 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    3.02                 main [1]
                1.76    0.00       1/1           someFunction() [2]
                0.88    0.00       1/1           anotherFunction() [3]
                0.38    0.00       1/1           yetAnotherFunction() [4]
                0.00    0.00       3/3           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::dur\
ation<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<std::chrono::_V2::system_clock, std::chrono::duration<long, std::rat\
io<1l, 1000000000l> >, std::chrono::duration<long, std::ratio<1l, 1000000000l> > >(std::chrono::time_point<std::chrono::_V2::system_clock, st\
d::chrono::duration<long, std::ratio<1l, 1000000000l> > > const&, std::chrono::time_point<std::chrono::_V2::system_clock, std::chrono::durati\
on<long, std::ratio<1l, 1000000000l> > > const&) [18]
                0.00    0.00       3/3           std::enable_if<std::chrono::__is_duration<std::chrono::duration<long, std::ratio<1l, 1000l> \
> >::value, std::chrono::duration<long, std::ratio<1l, 1000l> > >::type std::chrono::duration_cast<std::chrono::duration<long, std::ratio<1l,\
 1000l> >, long, std::ratio<1l, 1000000000l> >(std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [14]
                0.00    0.00       3/3           std::chrono::duration<long, std::ratio<1l, 1000l> >::count() const [13]
-----------------------------------------------
                1.76    0.00       1/1           main [1]
[2]     58.3    1.76    0.00       1         someFunction() [2]
-----------------------------------------------
                0.88    0.00       1/1           main [1]
[3]     29.1    0.88    0.00       1         anotherFunction() [3]
-----------------------------------------------
                0.38    0.00       1/1           main [1]
[4]     12.6    0.38    0.00       1         yetAnotherFunction() [4]
-----------------------------------------------
                0.00    0.00       3/9           std::chrono::duration<long, std::ratio<1l, 1000l> > std::chrono::__duration_cast_impl<std::c\
hrono::duration<long, std::ratio<1l, 1000l> >, std::ratio<1l, 1000000l>, long, true, false>::__cast<long, std::ratio<1l, 1000000000l> >(std::\
chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [15]
                0.00    0.00       6/9           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::dur\
ation<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<long, std::ratio<1l, 1000000000l>, long, std::ratio<1l, 1000000000l>\
 >(std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&, std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [19]
[11]     0.0    0.00    0.00       9         std::chrono::duration<long, std::ratio<1l, 1000000000l> >::count() const [11]
-----------------------------------------------
                0.00    0.00       6/6           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::dur\
ation<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<std::chrono::_V2::system_clock, std::chrono::duration<long, std::rat\
```

Advanced Programming

Profiling: Valgrind for Profiling

- We can also use Valgrind for profiling
- Run command such as eg.

  `valgrind --tool=callgrind ./chrono_timer`

  (Remember we need to have compiled using `-g` to use valgrind)
- Note that with no optimisation, this is significantly slower than gprof
- The output depends significantly on the level of optimisation - we use `-O0` here
- Outputs `callgrind.out.xxxxx` which can be interpreted using `callgrind_annotate callgrind.out.xxxxx`

Example: try out the above commands

Advanced Programming

Profiling: Valgrind for Profiling

```
--------------------------------------------------------------------------------
-- Auto-annotated source: chrono_timer.cpp
--------------------------------------------------------------------------------
Ir

            .           #include <iostream>
            .           #include <chrono>
            .
            .           // Function to be timed
            .           void someFunction()
            3 ( 0.00%)  {
            .               // Simulating some work
3,000,000,004 (58.80%)      for (int i = 0; i < 1000000000; ++i)
            .               {
            .                   // Do some computation
            .               }
            4 ( 0.00%)  }
            .
            .           // Another function to be timed
            .           void anotherFunction()
            3 ( 0.00%)  {
            .               // Simulating some work
1,500,000,004 (29.40%)      for (int i = 0; i < 500000000; ++i)
            .               {
            .                   // Do some computation
            .               }
            4 ( 0.00%)  }
            .
            .           // Yet another function to be timed
            .           void yetAnotherFunction()
            3 ( 0.00%)  {
            .               // Simulating some work
  600,000,004 (11.76%)      for (int i = 0; i < 200000000; ++i)
            .               {
            .                   // Do some computation
            .               }
            4 ( 0.00%)  }
            .
            .           int main()
```

Advanced Programming

Profiling: Other Advanced Tools

- Advanced profiling tools usually require licenses, eg. Intel VTune
- Provides much more detailed information than open source options
- Particularly useful for parallel code eg. effectiveness of threading and vectorisation, scalability
- Often will have a GUI - these can be tricky to set up remotely, but your system admin should be able to help

Advanced Programming

Profiling: Other Advanced Tools