CSCI251/CSCI851 Autumn-2020
Advanced Programming (**S3d**)

Getting organised IV:
Debugging and profiling

Debugging …

- The syntax for debugging and profiling is non-examinable, the concepts are examinable.
- In some of the labs you have used the compiler to pick up problems through compilation time errors and warnings.
- That's useful but having a program compile is usually only part of the battle, we should expect there to be run time problems too.

- As part of testing the state of our program at different points, we could put output statements into our code.
- Those additional lines of code may introduce new errors though, so using an external tool is better.
- A debugger is used to step through our programs as they run, and help us pick up errors in our programs.
- Knowing how to use a debugger will likely help at some point…

Using gdb

- The debugger gdb is available on capa, generally on Ubuntu.
  - GNU Project Debugger.
- You need to compile a program with –g flag.
  - This pretty much just stores ties to the original.
- So for Test.cpp,

```
$ g++ –ggdb Test.cpp –o Test
```

- Having compiled our test programs with the debugger information turned on we can run `gdb` for debugging.

```
$ gdb Test
```

- This loads the program into the debugger, ready for working on.
- To run …

```
(gdb) run
```

- If you run `gdb` without the command line argument you can use …

```
(gdb) file Test
```

```
struct Test {
   string name;
   int number;

   void setTest(string, int);
   void showTest();
};

void Test::setTest(string TestName, int TestNumber) {
   name = TestName;
   number = TestNumber;
}


void Test::showTest() {
  cout<<"Test string " << name << endl;
  cout<<"Number for this  "<< number << endl;
}
```

```
int main()
{
     Test myTest;
     myTest.setTest("Bob", 19);
     myTest.showTest();
}
```

```
int main()
{
     Test myTest;
     myTest.name="Bobby";
     myTest.number=15;
     myTest.showTest();
}
```

Test.cpp

- We can get the debugger to step through our program, for example stopping when we get to a particular function…

```
(gdb) break showTest
```

Or give a code line.

```
Breakpoint 1 at 0xdfa: file Test.cpp, line 19.
(gdb) run
Starting program: /home/lukemc/251/Test

Breakpoint 1, Test::showTest (this=0x7fffffffe900) at Test.cpp:19
cout<<"Test string " << name << endl;

(gdb) where
#0  Test::showTest (this=0x7fffffffe900) at Test.cpp:19
#1  0x000055555554ef6 in main () at Test.cpp:28
(gdb)
```

- At those breakpoints we can ask for variable values using `print`, as in the following example for our Test.cpp executable…

```
(gdb) print name
$1 = "Bob"
(gdb) print name[0]
$2 = (__gnu_cxx::__alloc_traits<std::allocator<char> >::value_type &) @0x7fffffffe910: 66 'B'
(gdb) print &name
$3 = (std::__cxx11::string *) 0x7fffffffe900
```

Checking memory …

- The debugger on Banshee included built in functionality for checking memory leaks.
- Unfortunately `gdb` doesn't.
- It is possible to use other tools, such as `valgrind`, to find memory leaks.
- The tool `valgrind` is not on capa ☹.
- We are going to initially use C++ code that doesn't clear dynamic memory correctly to see that `valgrind` can pick up leaks.
- If you have your own Ubuntu you can install `valgrind` using…

```
$ sudo apt install valgrind
```

MemTest.cpp

```
#include<iostream>
using namespace std;

int main()
{

        int *ptr = new int(3);
        cout << ptr << endl;

       // delete ptr;
}
```

`valgrind` for Memory leaks …

```
lukemc@laptop:~/temp$ valgrind ./a.out
==5217== Memcheck, a memory error detector
==5217== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5217== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5217== Command: ./a.out
==5217==
0x5b7dc80
==5217==
==5217== HEAP SUMMARY:
==5217==     in use at exit: 4 bytes in 1 blocks
==5217==   total heap usage: 3 allocs, 2 frees, 73,732 bytes allocated
==5217==
==5217== LEAK SUMMARY:
==5217==    definitely lost: 4 bytes in 1 blocks
==5217==    indirectly lost: 0 bytes in 0 blocks
==5217==      possibly lost: 0 bytes in 0 blocks
==5217==    still reachable: 0 bytes in 0 blocks
==5217==         suppressed: 0 bytes in 0 blocks
==5217== Rerun with --leak-check=full to see details of leaked memory
==5217==
==5217== For counts of detected and suppressed errors, rerun with: -v
==5217== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Profiling …

- Another tool that can be used to help our coding is a profiler.
- It allows us to determine how much time we are spending in different parts of our program.
- On capa we have `gprof`.
- We will initially look at this in the context of our `Test.cpp` struct test program.
- For compilation we use the flag `-pg` to prepare for using gprof …

```
$ g++ -pg Test.cpp -o Test
```

- When you run that executable …

```
$ ./Test
```

- … you get an file `gmon.out`, for use with `gprof`.
- Then run

```
$ gprof Test
```

- The output is likely initially not going to make a lot of sense.
  - Up the top there is a table with the usage of functions.
- Using the man pages in Linux can help see understand or see command options…

```
$ man command
```

- Let's look at a clearer illustration of using `gprof`.

```
void funA(){for (int x=0;x<100;x++){}};
void funB(){for (int x=0;x<1000;x++){}};
void funC(){for (int x=0;x<10000;x++){}};

int main()
{
        srand(time(0));
        for (int x=0; x< 1000; x++)
        {
                switch ( rand() % 3 ){
                        case 0: funA();
                                break;
                        case 1: funB();
                                break;
                        case 2: funC();
                                break;
                        default:
                                break;
                }
        }
        return 0;
}
```

calling.cpp

Why?
Functions with different costs…

- Let's look at the output from `gprof` …

```
$ g++ -pg calling.cpp -o call
$ ./call; gprof call | head
```

  - The semi-colon is used to chain multiple commands to one input line.
    - Here it makes it easier to repeat both commands together.
  - The | is a pipe, here so only the top part of the gprof output is displayed.

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
 100.46      0.01     0.01      325    30.91    30.91  funC()
  0.00      0.01     0.00      341     0.00     0.00  funA()
  0.00      0.01     0.00      334     0.00     0.00  funB()
  0.00      0.00     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z4funAv
  0.00      0.00     0.00        1     0.00     0.00
__static_initialization_and_destruction_0(int, int)
```