

## GCC and tips

- GNU Compiler Collection consists of many language front ends to the gnu compiler
- here we will look at some of the common options to gcc and g++
- these slides are simply a taster and huge simplification of how GCC might be used

## GCC debugging

- all front ends (in our case: gcc, g++ and gm2) accept `-g -O0` which tell the compiler not to optimize and emit debugging information for gdb

## GCC debugging

- turn on all warnings by: `-Wall`
- so our command line to compile `hello.c` is:
- ```
$ gcc -g -O0 -Wall -c hello.c
```
- notice that this compiles `hello.c` but does not link it
- to link this we can:

- ```
$ gcc -g hello.o
```

## GCC debugging

- we could combine the last two steps by:

- ```
$ gcc -g -O0 -Wall hello.c
```

## Debugging your code

```
$ gdb a.out
(gdb) break exit
(gdb) run
(gdb) quit
```

- set break points, single step code, finish functions, invoke functions as necessary

```
(gdb) print t
(gdb) break pf
(gdb) print pf(t)
(gdb) next
(gdb) step
(gdb) finish
```

## Valgrind

- no excuse for not using this program!
- it requires no effort to run your executable in valgrind

```
$ valgrind ./a.out
```

- valgrind is a memory mismanagement detector, it can detect using memory which has not been allocated or has been freed

## What is wrong with this code?

```
#include <stdlib.h>

void myfunc (int n)
{
    int* a = malloc(n * sizeof(int));

    a[n] = 0;
}

int main ()
{
    myfunc(3);
    return 0;
}
```

## Valgrind gives you a huge hint

```
$ valgrind ./a.out
==30984== Command: ./a.out
==30984==
==30984== Invalid write of size 4
==30984==    at 0x400511: myfunc (bad.c:7)
==30984==    by 0x400526: main (bad.c:12)
==30984== Address 0x518b04c is 0 bytes after a block of
```

## Making your program go faster

- firstly profile your code to check if there are any obvious inefficiencies

```
$ gcc -g -O0 -pg -c foo.c
$ gcc -g -pg foo.o
```

## Making your program go faster

- again we could combine these two commands with

```
$ gcc -g -O0 -pg foo.c
```

- most large projects will involve a discrete compile and link step

## Making your program go faster

- run your program as before

```
$ ./a.out
```

- now invoke the profiler

```
$ gprof a.out
```

## Making your program go faster

```
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls s/call s/call name
35.38 20.64 20.64 4771989280 0.00 0.00 IN
25.59 35.56 14.93 1049864543 0.00 0.00 scan
15.22 44.44 8.88 345772285 0.00 0.00 makemove
8.52 49.41 4.97 868833748 0.00 0.00 IN
7.34 53.69 4.28 10274416 0.00 0.00 evaluate
```

- we could choose to rewrite the functions IN, scan or makemove

## Making your code go even faster

- use options: `-O1` or `-O2` or `-O3` on the command line to `gcc`
- these optimizations may vary according to architecture

## Making your code go even faster

- for detail as to which optimizations they turn on use:

```
$ gcc -c -O -O3 --help=optimizers | grep enabled
```

- to see the difference between `-O2` and `-O3` use:

```
$ gcc -c -O -O3 --help=optimizers > /tmp/O3-opts
$ gcc -c -O -O2 --help=optimizers > /tmp/O2-opts
$ diff /tmp/O2-opts /tmp/O3-opts | grep enabled
```

## Making your code go even faster

- if you don't need full compliant math code, you could use the `-ffast-math` option (which will inline `sin`, `cos`, `tan` etc)

```
$ gcc -O3 -ffast-math -c foo.c
$ gcc -O3 -ffast-math foo.o
```

## Size of code generated

- you can always check the size of your code via:

```
$ size a.out
```

- also optimize for space via the option `-Os`

```
$ gcc -Os -c foo.c
$ gcc -Os foo.o
```