Lecture A for foreign students

Lecture 1

Norbert Pozar (npozar@se.kanazawa-u.ac.jp)

Apr 19, 2018

Today's plan

- Basic `gcc` toolchain usage
- Using the command line efficiently
- Shell scripts for automation

gcc toolchain

- The GNU Compiler Collection
- Website: https://gcc.gnu.org/
- Current version: 7.3
- Supports C (`gcc`), C++ (`g++`), Fortran (`gfortran`), ...
- Alternative: clang (used on macOS by default)

Basic usage

Compile and link:

```
$ gcc main.c -o main
$ g++ main.cpp -o main
```

Run the resulting binary:

```
$ ./main
```

`main.c`, `main.cpp` input source file (`.c` for C and `.cpp` for C++)

`-o main` compiler flag `-o` with a parameter for the name of the produced executable[1]

[1] On Unixy systems (Linux, macOS), executables do not have an extension by convention. On Windows it would be `main.exe`.

Optimizations

- By default, the compiler **does not perform any optimizations**.
- Important for development and debugging $\rightsquigarrow$
  - Original structure of the code is preserved.
  - It takes time to optimize code.
- Optimization can **significantly speed up** the code: Use optimized builds for running computations!

```
$ gcc main.c -o main -O3
```

-O3 Turns on the highest level of optimizations[2].

[2] See gcc manual: 3.10 Options That Control Optimization

Example: Sum $1 + \cdots + 10^9$ (optimization1.c)

```c
#include <stdio.h>

int main()
{
    double sum = 0.;
    for (int i = 0; i < 1000000000; i ++) {
        sum += i;
    }
    return 0;
}
```

Build it and run it (with timing)

```
$ gcc optimization1.c -o optimization1
$ time ./optimization1
```

Run with optimizations enabled

Build it and run it (with timing)

```
$ gcc optimization1.c -o optimization1 -O3
$ time ./optimization1
```

Optimization surprises

- Optimized binary should produce the same *output* as the nonoptimized one.
- Code that does not influence the output can be safely removed: This is one of the most important optimizations!

Let's make sure that we print the result and try again.

```c
#include <stdio.h>

int main()
{
    double sum = 0.;
    for (int i = 0; i < 1000000000; i ++) {
        sum += i;
    }

    printf("%f\n", sum);

    return 0;
}
```

More optimizations: `-march=native`

- Instructs the compiler to use any instructions that your processor supports, including the modern ones.
- A good idea to always use this. (It is unlikely that you will try to run the same binary on another machine without recompiling.)

```
$ gcc optimization2.c -o optimization2 -O3 \
        -march=native
```

More optimizations: `-ffast-math`

Numbers on the computer behave differently from real numbers.

Example. Addition is not associative. On a computer

$$(a + b) + c \neq a + (b + c)$$

With double precision, try $a = 10^{16}$, $b = -a$, $c = 1$.

How is $a + b + c$ computed?
Compiler follows the order of operations and emits code for $(a + b) + c$.

More optimizations: `-ffast-math`

But using $a + (b + c)$ might be more efficient:

- Maybe $b + c$ was already computed.
- Modern processors are **superscalar**: Can execute multiple **independent** instructions per cycle.

  $$a + b + c + d$$

  may be possibly computed faster as

  $$(a + b) + (c + d)$$

  since $a + b$ and $c + d$ can potentially be computed **at the same time**.

  $\rightarrow$ **instruction-level parallelism**

More optimizations: `-ffast-math`

Unfortunately, changing the order of floating point operations can change results.

⇒ Compiler cannot reorder operations by default.

`-ffast-math` Flag that tells compiler that it is **OK** to reorder floating point operations[3] to produce a more efficient code, even if it changes the result.

```
$ gcc optimization2.c -o optimization2 -O3 \
        -ffast-math
```

`-O3 -ffast-math` can be replaced by `-Ofast`

[3] And a few other optimizations.

More optimizations: `-ffast-math`

Use with great care!
Most of the time OK but some numerical algorithm are very sensitive to the order of operations. Test if `-ffast-math` changes your results.

`-ffast-math` also assumes that all floating point results are finite numbers, so no NaNs and no infinities. Make sure your code does not rely on these.

It also cheats a bit on some operations like division, using faster but less accurate methods.

General optimization tips for faster code

- Start with choosing an **efficient algorithm for your problem**.
  - bubble sort VS quicksort
  - Gaussian elimination VS iterative methods
- **Only ever optimize “hot code”**: code that actually is run the most in the program.
- Use timing (or even profiling[4]) tools to test if changes are actually beneficial and to find which code to focus on.

---

[4] `perf` on Linux: See `perf` Examples for more info. This is quite advanced topic.

Quick tip for numerical code

Division is much slower than multiplication.

If you need to divide by a constant in a hot loop, precompute the reciprocal $c = 1/b$ outside of the loop and replace a/b with $a * c$.

Useful Unix commands

`time command args...` Run `command args...` and print the time used.

`top`, `htop` Show CPU and memory usage of running processes.

`perf` Advanced profiling on Linux. Very detailed information down to which instructions in your code take the most time.

Enable more warnings

`-Wall -Wextra` flags enable many useful warnings that help identify serious bugs in your code early. **Always use**.

```c
#include <stdio.h>
int main() {
    double x;
    printf("%f\n", x);  // unitialized variable
}

int n(int i) {
    i;      // forgotten return + no effect
}
```