Embedded Software

CS 145/145L

Caio Batista de Melo
Recap

- Project 1 was extended until Wednesday (2022-04-20);

- You should have all the tools to work on Project 2
  - Keypad
  - LCD

- If your LCD randomly resets at times…
  - https://edstem.org/us/courses/20963/discussion/1412547
  1. Replace libraries (avr/lcd) with new ones: https://caiobatista.com/courses/uci/s22/cs145-project-2/;
  2. Where you used lcd_puts before, replace it with lcd_puts2;
  3. Add avr_init() before your lcd_init() call.
Interpreter vs Compiler

https://www.learningelectronics.net/vol_5/chpt_7/3.html
Compiler

Source Code

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
}
```

Preprocessor

- substitutes #include directives with content of included files

Compiler

- generates binary machine code

Linker

- combines binary machine code and connects function calls

Binary Executable

```
... cout = ...
...
```

https://hackingcpp.com/cpp/hello_world.html
Available Compilers

- gcc is the most popular one
  - have different versions, e.g., xc8, avr-gcc, arm-gcc

- clang is an alternative that you might have used (default on Macs)

- usually a compiler works for a specific architecture (e.g., Windows)

gcc -Flags

-Wall: shows all warnings

-Wextra: shows more warnings

-Werror: treats warnings as errors

-ansi: uses the ANSI version of C (universally compatible)

-pedantic: turns off even more features than ansi

-o: output file

-On: optimization level (O0…3, Ofast, Os)
## gcc -On

<table>
<thead>
<tr>
<th>option</th>
<th>optimization level</th>
<th>execution time</th>
<th>code size</th>
<th>memory usage</th>
<th>compile time</th>
</tr>
</thead>
<tbody>
<tr>
<td>-O0</td>
<td>optimization for compilation time (default)</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-O1 or -O</td>
<td>optimization for code size and execution time</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>-O2</td>
<td>optimization more for code size and execution time</td>
<td>--</td>
<td>+</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>-O3</td>
<td>optimization more for code size and execution time</td>
<td>---</td>
<td>+</td>
<td>+++</td>
<td></td>
</tr>
<tr>
<td>-Os</td>
<td>optimization for code size</td>
<td>--</td>
<td>-</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>-Ofast</td>
<td>O3 with fast none accurate math calculations</td>
<td>---</td>
<td>+</td>
<td>+++</td>
<td></td>
</tr>
</tbody>
</table>

https://www.rapidtables.com/code/linux/gcc/gcc-o.html
Testing

After you’ve written your code and it compiles, you need to make sure it works. Testing helps setting up a framework that verifies it’s working as intended. Also helps make sure that changes will not break previous versions.

gcov: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
  Included in the GNU toolchain (if you have gcc, you have gcov)
  Calculates the coverage of the code

unity: https://github.com/ThrowTheSwitch/Unity
  Lightweight framework to create unit tests for C embedded software

googletest: https://github.com/google/googletest
  C++ test framework (can be used for C code as well)
```
#ifndef __DATETIME_H__
#define __DATETIME_H__

typedef struct {
    // Other things...
    int seconds;
} DateTime;

void advance_one_second(DateTime *dt);
void advance_hours(DateTime *dt, const int hours);
#endif // __DATETIME_H__

#include "datetime.h"

void advance_one_second(DateTime *dt) {
    ++dt->seconds;
}

void advance_hours(DateTime *dt, const int hours) {
    int total_seconds = 60 * 60 * hours;
    while (total_seconds--) {
        advance_one_second(dt);
    }
}
```
#include "datetime.h"
#include "unity.h"

void setUp (void) {}
void tearDown (void) {}

void test_advance_one_second (void) {
    DateTime dt;
    dt.seconds = 0;
    TEST_ASSERT_EQUAL(0, dt.seconds);
    advance_one_second(&dt);
    TEST_ASSERT_EQUAL(1, dt.seconds);
}

int main(void) {
    UNITY_BEGIN();
    RUN_TEST(test_advance_one_second);
    RUN_TEST(test_advance_hours);
    return UNITY_END();
}
Testing Example (unity + gcov)

$ gcc testDatetime.c datetime.c unity.c -fprofile-arcs -ftest-coverage
$ ./a.out

testDatetime.c:29:test_advance_one_second:PASS
testDatetime.c:30:test_advance_hours:PASS

-----------------------
2 Tests 0 Failures 0 Ignored
OK

$ gcov a-datetime.c
File 'datetime.c'
Lines executed:100.00% of 8
Creating 'datetime.c.gcov'

flags for gcov
all tests passed!
covered all lines!

These results are only as good as our tests...
We might still have problems in the code!
What if hours < 0?
If your code isn’t working as intended (maybe a test failed?), how can you fix it? Debugging helps you pinpoint what’s going wrong.

**gdb:** [https://www.sourceware.org/gdb/](https://www.sourceware.org/gdb/)
- Most popular C debugger
- Let’s you pause execution and check values, step-by-step
Debugging Example (gdb)

- https://onlinegdb.com/E4M3dnhv7

After finding out the bug (negative hours -> really long loop), we can fix that.
Profiling

After you make sure your code works as intended, you should make it faster! Profiling helps you find bottlenecks and points in your code where you can improve.

gprof: [https://users.cs.duke.edu/~ola/courses/programming/gprof.html](https://users.cs.duke.edu/~ola/courses/programming/gprof.html)
Show time spent of functions, number of calls, other stats.

valgrind: [https://valgrind.org/](https://valgrind.org/)
Help analyze memory management (e.g., find leaks).

gperftools: [https://github.com/gperftools/gperftools](https://github.com/gperftools/gperftools)
Can profile CPU and memory.
Profiling Example (gprof)

```bash
gcc main.c datetime.c -pg
./a.out
32400
gprof
```

```
0.00 0.00 1/1   main [8]
[2]  0.0 0.00 0.00  1   advance_hours [2]
0.00 0.00 32400/32400   advance_one_second [1]
```

Maybe we don't need to call this function that many times?
C Topics
int x;
int *y;
x = 2;
y = &x;     // st #200, 173
*y = 5;     // st (#200), 5 (an indirect store)

Attributes in the compiler:
(x: 173, 2, Integer)  
(y: 200, 4, Pointer)
Pointers... why?!?!

```c
void init_dt(DateTime *dt) {
    dt->year = 2022;
    ...
    dt->second = 0;
}
```

Side-effects: change things in other scopes.

```c
void print_dt(const DateTime *dt) {
    char buf[17];
    // Print date on top row.
    lcd_pos(0, 0);
    sprintf(buf, "%04d-%02d-%02d",
            dt->year,
            dt->month,
            dt->day)
    lcd_puts(buf);
    // Do similar thing to print time on bottom row.
}
```

Efficiency: only pass one address (2 bytes) instead of the whole DateTime (7 bytes) to the function. (ATmega32)
Pointers and consts

Regular pointer to a regular variable, both ops ok!

Regular pointer to a const variable, second op is bad!

test-const-pointers.c:10:6: error: read-only variable is not assignable
*p2 = 2;

Const pointer to a regular variable, first op is bad!

test-const-pointers.c:13:4: error: cannot assign to variable 'p3' with const-qualified type 'int *const'
p3++;

Const pointer to a const variable, both ops bad!

test-const-pointers.c:17:4: error: cannot assign to variable 'p4' with const-qualified type 'const int *const'
p4++;;

Shifts

Left Logical Shift

MSB: 1 0 1 1 0 0 1 1
LSB: 0 1 1 0 0 1 1 0

Right Logical Shift

MSB: 1 0 1 1 0 0 1 1
LSB: 0 1 0 1 1 0 0 1

Left Arithmetic Shift

MSB: 1 0 1 0 0 1 1 1
LSB: 0 1 0 0 1 1 1 0

Right Arithmetic Shift

MSB: 1 0 1 0 0 1 1 1
LSB: 1 1 0 1 0 0 1 1

Retains the sign!
Shifts

● Type of the operand:
  ○ Unsigned -> logic shift
  ○ Signed -> arithmetic shift

● You can use casting to force the one you want:

```c
int x = -1;
x = (unsigned) x >> 1;
printf("%d\n", x);
```
See you next time :)