Unit 4
Advanced C programming

Davydov V.

Multifile project structure

- How it worked before:
  - main.c contains:
    1. includes for all system libraries' header files
    2. Function prototypes + doxygen
    3. main() function
    4. Implementation of functions in p.2
- Why it is bad?
  - Single file being grown much. Maintenance decreases correspondly
  - (spoiler) Next to impossible to cover code with unit tests

Multifile project structure

- How it should look like:
  - lib.h
    - Double include protection
    - includes
    - Functions prototypes, doxygen, constants, structures
  - lib.c
    - #include “lib.h”
    - Function implementation
  - main.c
    - #include “lib.h”
    - Include header files used only in this file
    - main() function

```
|-- Doxyfile
|-- Makefile
|-- README.md
|-- doc
|   `-- lab00.md
`-- src
    |-- lib.c
    |-- lib.h
    `-- main.c
```

- Peculiarities:
  - Files pair like lib.c/lib.h can be more than 1. They should be common sense named according to domain
  - Performing strict functionality delimitation by files. This allows to extend functionality without maintenance decrease

Multifile project structure

- Double include protection
  - #pragma once
  - Macrosafe:
    - #ifndef __STUDENT__H_
    - #define __STUDENT__H_
    - …
    - #endif
- Assembly
  - Step-by-step:
    - Compile each module seperately. Main should have zero-knowledge about implementation. But we promise that it will be during linkage stage
      - clang **-c** src/main.c **-I**./src -o dist/main.o
      - clang **-c** src/lib.c **-I**./src -o dist/lib.o
    - Module Linkage. If there is a function that’s declared in header file, being in use and has no implementation -> error.
      - clang dist/main.o dist/lib.o -o dist/main.bin
  - Single command
    - clang src/main.c src/lib.c **-I**./src -o dist/main.bin

Software Testing

- Books:
  - Vladimir Khorikov / Unit Testing: Principles, Practices, and Patterns
  - Kent Beck: *Test-Driven Development: By Example* (Addison-Wesley Professional, 2002).
  - *Growing Object- Oriented Software, Guided by Tests* (Addison-Wesley Professional, 2009)
  - P.S. Don't rely on language specific books. Testing is a common approach!
- Goal:
  - Verify a small piece of code (also known as a *unit*),
  - Do it quickly
  - Do it in isolate manner (one test shouldn't rely on another ones' data / state / context)
  - (ideally) One test should cover one unit/case block
- Why do we need this?
  - tests reduce the degree of uncertainty of the problem being solved
  - Eugene Borisov, Spring Patterns: https://www.youtube.com/watch?v=61duchvKI6o (timeframe: 15:10 - 18:30)
- Types:
  - Unit Tests
  - Integration Tests

4 pillars of Unit tests

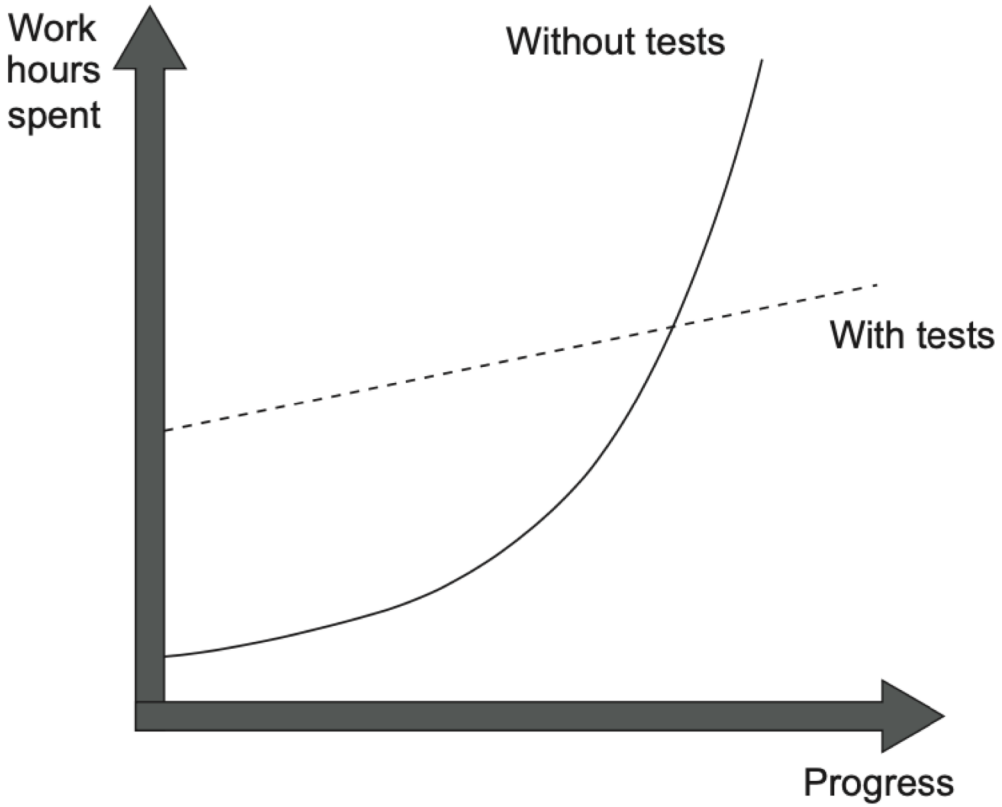
- Protection against regressions
- Resistance to refactoring
- Fast feedback
- Maintainability

Table 4.1 The pros and cons of white-box and black-box testing

	Protection against regressions	Resistance to refactoring
White-box testing	Good	Bad
Black-box testing	Bad	Good

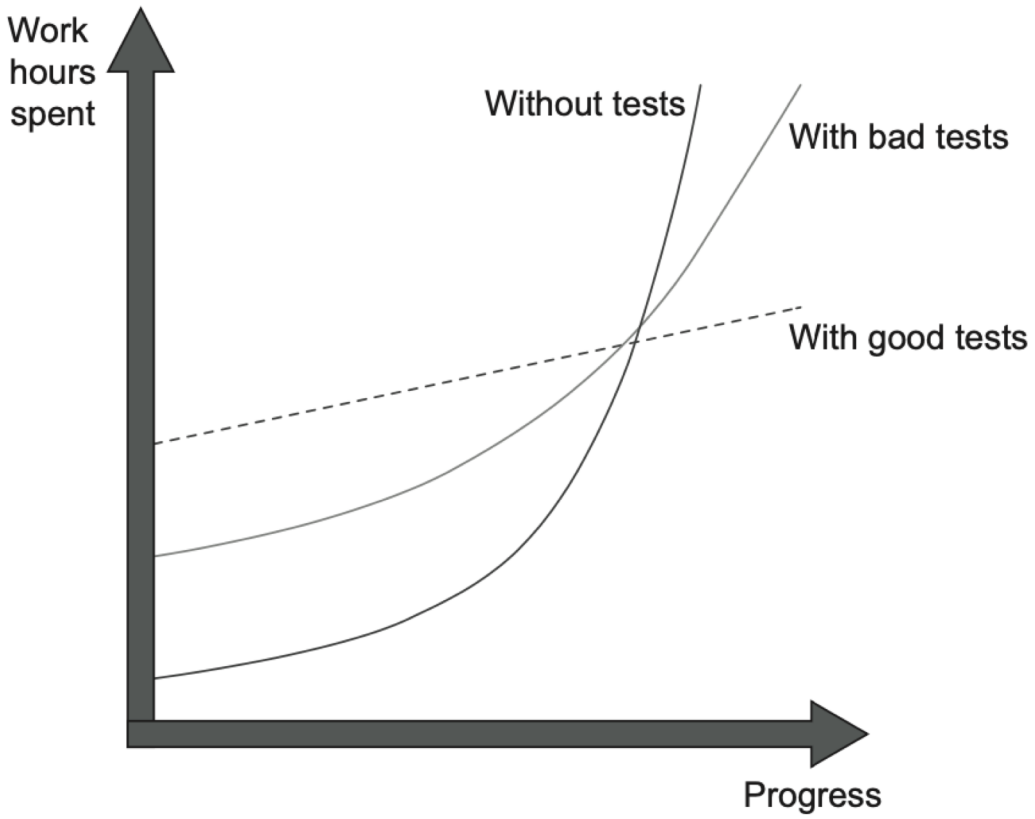
- Other:
  - There is no ideal test, as *protection against regressions*, *resistance to refactoring*, and *fast feedback*— are mutually exclusive
  - BDD – Behavior Driven Development
  - TDD – Test Driven Development

Software testing



This phenomenon of quickly decreasing development speed is also known as *software entropy*. Entropy (the amount of disorder in a system) is a mathematical and scientific concept that can also apply to software systems.

Good tests vs bad tests



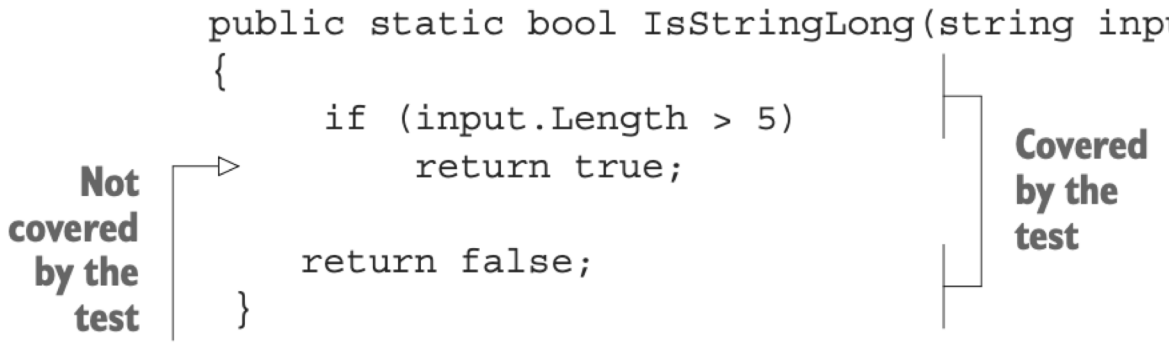
Remember, *not all tests are created equal*. Some of them are valuable and contribute a lot to overall software quality. Others don't. They raise false alarms, don't help you catch regression errors, and are slow and difficult to maintain. It's easy to fall into the trap of writing unit tests for the sake of unit testing without a clear picture of whether it helps the project.

You can't achieve the goal of unit testing by just throwing more tests at the project. You need to consider both the test's value and its upkeep cost. The cost component is determined by the amount of time spent on various activities:

- Refactoring the test when you refactor the underlying code
- Running the test on each code change
- Dealing with false alarms raised by the test
- Spending time reading the test when you're trying to understand how the underlying code behaves

Code coverage

$$\text{Code coverage (test coverage)} = \frac{\text{Lines of code executed}}{\text{Total number of lines}}$$

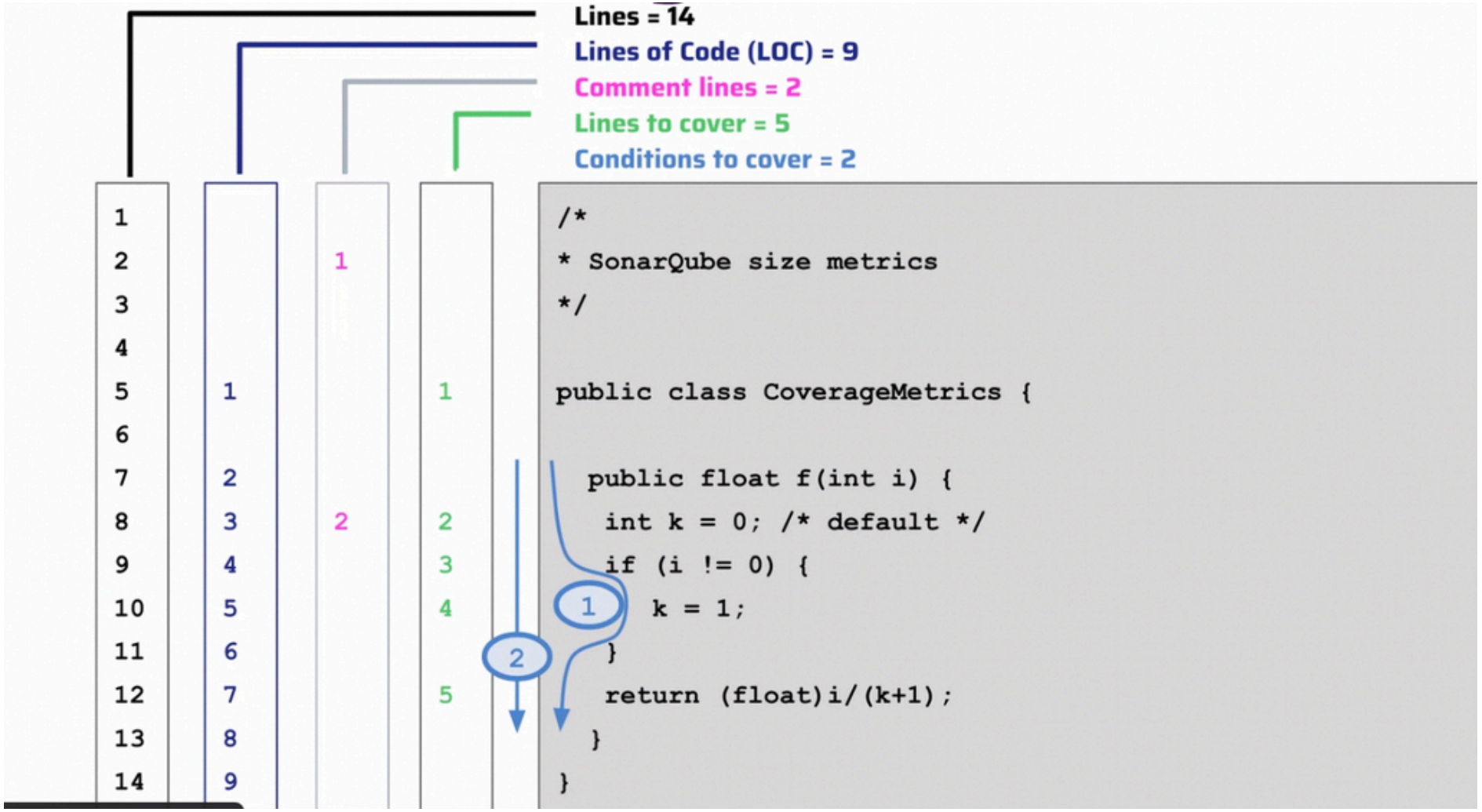



```
public void Test()
{
    bool result = IsStringLong("abc");
    Assert.Equal(false, result);
}
```

What cases we should cover:

- Long string
- Short string
- Empty / null string
- Some wrong string (if we deal with parsing to numbers)

Line and coverage metrics (by Sonar Cube)



Unit Test lifecycle

- Arrange: Create data setup.
  - Ideally, we should avoid randomize data
  - Ideally, we should avoid external data
  - We should avoid user interaction
- Act: Execute `victim` method/unit -> retrieve `actual` data
- Assert: Compare actual results with expected results
  - If test failed – information about testcase (function; expected data, actual data; line) should be shown
- TearDown : release all open resources (if any)
- Other:
  - In result – collect list of passed and failed unit tests
    - If failed unit tests found – we need either to adopt tests (5% cases) or update main code (95%)
  - The smaller unit test – the better unit test

Project Structure

- File test/test.c should be created
- Makefile file should have targets for compile test.bin
- You have to create 2 binaries: test and main
  - Main can interact with user and/or external data.
  - Test shouldn't use external data; Tests _HAVE_NOT_ to interact with user – they should be 100% automatized

```
|-- Doxyfile
|-- Makefile
|-- README.md
|-- dist
|   |-- main.bin
|   `-- test.bin
|-- doc
|   `-- lab00.md
|-- src
|   |-- lib.c
|   |-- lib.h
|   `-- main.c
`-- test
    `-- test.c
```

Necessary software

- Unit testing: https://github.com/libcheck/check
  - sudo apt-get install check
  - $(CC) $(C_OPTS) test/test.c -o ./dist/test.c **-lcheck** -lm *-lrt -lpthread -lsubunit*

- Code coverage: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html
  - sudo apt-get install llvm-cov
  - test.bin: test/test.c

    $(CC) $(C_OPTS) $< **-fprofile-instr-generate -fcoverage-mapping** -o ./dist/$@ -lcheck -lm -lrt -lpthread –lsubunit
  - test: clean prep compile

    LLVM_PROFILE_FILE="dist/test.profraw" ./dist/test.bin

    llvm-profdata merge -sparse dist/test.profraw -o dist/test.profdata

    llvm-cov report dist/test.bin -instr-profile=dist/test.profdata src/*.c

    llvm-cov show dist/test.bin -instr-profile=dist/test.profdata src/*.c --format html > dist/coverage.html

Simple test sample

// lib.h

```
int sum(int a, int b);
```

// lib.c

```
#include “lib.h”
int sum(int a, int b) {
   return a + b;
}
```

// test.c

```
#include "lib.h"
#include <check.h>

START_TEST(test_sum)
{
     int data_a = 4;
     int data_b = 9;
     int expected = 13; /* don't use 4+9 */
     int actual = sum(data_a, data_b);
     ck_assert_int_eq(actual, expected);
}
END_TEST
```

// test.c (continuation)

```
int main(void)
{
     Suite *s = suite_create("Programing");
     TCase *tc_core = tcase_create("Lab-??");

     tcase_add_test(tc_core, test_sum);
                              /* test_sum - method we created */
     suite_add_tcase(s, tc_core);

     SRunner *sr = srunner_create(s);
     srunner_run_all(sr, CK_VERBOSE);
     int number_failed = srunner_ntests_failed(sr);
     srunner_free(sr);

     return (number_failed == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Tests results

- Success

Running suite(s): **Programing**

100%: Checks: 1, Failures: 0, Errors: 0

~/sample_project/lab00/test/test.c:14:P:**Lab-??**:**test_sum**:0: Passed

Process finished with exit **code 0**

- Failure

Running suite(s): **Programing**

0%: Checks: 1, Failures: 1, Errors: 0

~/sample_project/lab00/test/test.c:14:F:**Lab-??**:test_sum:0: Assertion 'actual == expected' failed: actual == 12, expected == 13

Process finished with exit **code 1**

Assertion functions

- ck_assert_{int,uint,float,double,ldouble,str}_{eq,ne,lt,le,gt,ge} (actual, expected)

- ck_assert_{float,double,ldouble}_{eq,ne,lt,le,gt,ge}**_tol** (actual, expected, tolerance)

Set tests (table data)

```
// Classic way
START_TEST(test_sum_set)
{
#define N 4
	int data_a[N] = { 4, 10, -10 , -5};
	int data_b[N] = { 10, -10, 0, 25};
	int expected[N] = { 14, 0, -10, 20};

	for (int i = 0; i < N; i++) {
		int actual = sum(data_a[i], data_b[i]);
		ck_assert_int_eq(actual, expected[i]);
	}
}
END_TEST
```

Note: libcheck uses `Looping tests` for this case. @see tcase_add_loop_test

Arrays assert sample

- Ensure size of both arrays are equal (especially for strings or unknown sized data)
- Ensure that both arrays not NULL (for dynamic allocated data)
- Loop through one array and assert i-th element from both arrays
- P.S. Not recommended (but not restricted) to use data sets

```
START_TEST(test_sort_asc)
/* sort prototype: int sort(int array[], int size) */
{
#define N 10
    int initial_actual[N] = { -1, 2, 3,-5, 3, 2, -6, 1, 0, 4};
    int expected[N] = { -6,-5,-1,0,1,2,2,3,3,4};

    sort(initial_actual, N);
    for (int i = 0; i < N; i++) {
        ck_assert_int_eq(initial_actual[i], expected[i]);
    }
}
END_TEST
```

llvm-cov sample result

```
28      2   default:
29      2     result = "N/A";
30     16   }
31     16   return result;
32     16 }
33
34        void generate_animal(struct animal *entity)
35     10 {
36     10   entity->height = (unsigned int)random() % INT8_MAX;
37     10   entity->weight = (unsigned int)random() % INT8_MAX;
38     10   entity->type = (unsigned int)random() % ANIMAL_TYPE_COUNT;
39     10 }
40
41        void show_animals(struct animal animals[], unsigned int count)
42      0 {
43      0   for (unsigned int i = 0; i < count; i++) {
44      0     printf("Інформація про тварину №%02u: ", i + 1);
45      0     printf("%s: зріст = %u см, маса = %u гр. \n", get_animal_type_name(animals[i].type), animals[i].hei
46      0   }
47      0 }
```

Filename	Lines	Missed Lines	Cover
lab00/src/lib.c	31	6	80.65%
TOTAL	31	6	80.65%

How to test?

- int rand();
- void abrakadabra() {

```
        int i;
        int result;
        scanf("%d", &i); // read `i` value
        for (int k = 0; k < i; k++) {
                result *= i;
        }
        printf("%d", result); // print `result` value
}
```

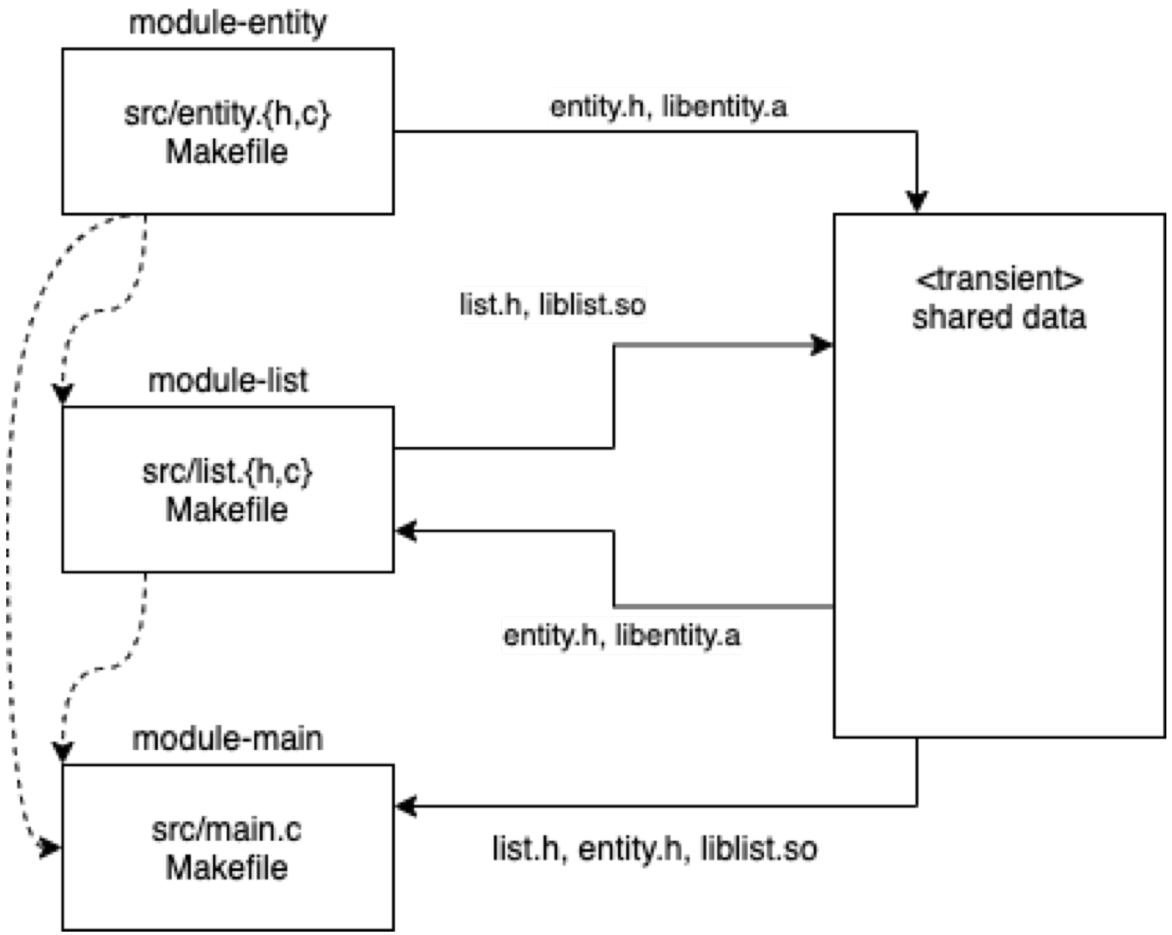
Other soft

- Recommend:
  - Text editor -> IDE (e.g. CLion, Eclipse)
  - lldb -> IDE Debugger
  - llvm-cov -> IDE Coverage Report
  - Git has GUI Manager -> Tortoise git, SourceTree, https://linuxhint.com/best_git_gui_clients_ubuntu/
- Allowed, not recommend:
  - make -> cmake

Libraries

- What is library?
  - A *library* in C is a collection of header files, exposed for use by other programs. The library therefore consists of an *interface* expressed in a .h file (named the "header") and an *implementation* expressed in a .c file. This .c file might be precompiled or otherwise inaccessible, or it might be available to the programmer.
- Compilation stages:
  - Compile
  - Linking
- Types
  - Dynamic Libraries / Shared libraries
  - Static Libraries
- **Library contains No main function**

Project Structure



Static library compile

- A static library is basically a set of object files that were copied into a single file with the suffix `.a`.
- struct Point {
  double x;
  double y;
  };
  void print(struct Point *point);
  void generate(struct Point *stub);
- $(CC) -c $(C_OPTS) src/entity.c -o ./dist/entity.o
  ar rcs ./dist/**lib**entity**.a** ./dist/entity.o
  cp ./dist/libentity.a ../shared
  cp ./src/entity.h ../shared

Compile main with static linkage

- #include "entity.h"
  int main() {
  struct Point point;
  generate(&point);
  print_point(&point);
  return 0;
  }
- `$(CC) -I../shared src/main.c -lentity -L../shared -o dist/main.bin`

Analyze main binary

- With only static library `entity`
  - nm main.bin
  - 0000000100008010 d __dyld_private
  - 0000000100000000 T __mh_execute_header
  - **0000000100003f00 T _generate**
  - 0000000100003e80 T _main
  - **0000000100003ec0 T _print_point**
  - U _printf
  - U _random
  - U dyld_stub_binder

Compile shared library

- Object files for the shared library need to be compiled with the -fPIC flag (PIC = position independent code) because they are mapped to any position in the address space.
- #include "entity.h"
  struct List {
  struct Point points[10];
  size_t size;
  };
  extern void print_list(struct List* list);
  extern void add_to_end(struct List* list, struct Point *point);
- `$(CC) -c $(C_OPTS) -fPIC -I../shared src/list.c -o dist/list.o`
  `$(CC) -shared dist/list.o ../shared/libentity.a -o dist/liblist.so`
  `cp dist/liblist.so ../shared`
  `cp src/list.h ../shared`

Compile main with dynamic linkage

- #include "list.h"
  int main() {
  struct List list;
  list.size = 0;
  struct Point point;
  generate(&point);
  add_to_end(&list, &point);
  print_point(&point);
  print_list(&list);
  return 0;
  }
- `$(CC) -I../shared src/main.c -llist -L../shared -o dist/main.bin`
- **`cp ../shared/liblist.so ./dist`**
  `./dist/main.bin`

Analyze main binary

- With only dynamic library `list`
  - nm main.bin
  - 0000000100008020 d __dyld_private
  - 0000000100000000 T __mh_execute_header
  - **U _add_to_end**
  - **U _generate**
  - 0000000100003f00 T _main
  - **U _print_list**
  - **U _print_point**
  - U dyld_stub_binder

Conclusion

- **Static libraries:**
  - **+** are faster than the shared libraries because a set of commonly used object files is put into a single library executable file which enables you to build multiple executables without the need to recompile the file.
  - **(+)** No dependencies
  - **(-)** Higher memory usage, as the OS can no longer use a shared copy of the library.
    (-) If the library needs to be updated, your application needs to be rebuilt.
- **Dynamic libraries:**
  - **(+)** Consumes disk space and memory only once.
  - **(+)** If a **.so** is already loaded, the executable may load faster because it is a smaller file
  - **(-)** If a **.so** is not already loaded, the executable's remaining undefined symbols must be resolved at start-up.
  - **(-)** There is a chance to break executables without relinking.