Code style, Tests, Debugging

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

Code style

- In most programming languages **whitespace is ignored**
  - Leaves many options for styling
- The exact style is not important, no need to be “dogmatic” about it
- But it is very important to be **consistent**
  - Good style makes the code **readable**

Naming conventions

- Similarly, how we **name** things is important
  - variables
  - modules
  - functions
  - types
  - etc.
- Consistent naming greatly improves code quality

In this class

- The following slides present some style & naming choices
- The code used in the lectures follow this style
- You are **not** required to follow it
- But you **are** required to consistently follow a specific style

Comments

```
// C++ style
// στα Ελληνικά

void foo() {
    int a = 1;      // μικρά comments στην ίδια γραμμή
}
```

- Makes it easy to toggle comments (`Ctrl-/` in VS Code)
- Don't over-use comments
  - they should not explain **what** the code does
  - but **how/why**
- **Don't** leave old garbage code in comments (Git keeps the history!)

Brackets

```
// Στην ίδια γραμμή με την εντολή που τα ανοίγει

void foo() {
    for (unsigned int i = 0; i < ...; i++) {
        ...
    }

    // Για μικρές εντολές το παρακάτω είναι ok (χωρίς κατάχρηση)
    if (condition)
        do_something();
}
```

Indentation

- One **tab** for each level
  - allows each developer to configure the tab size differently
- Alternative option: **4 spaces**
  - appears the same in all editors
- **Don't** mix the two

Pointer types

```
// Το * κολλητά με τον τύπο (όχι με το όνομα)

int* foo(char* param) {
    int* pointer = &var;
    ...
}
```

Conceptually, `int*` is a type.

Variable declarations

```
// Μία δήλωση ανά γραμμή, επαναλαμβάνουμε τον τύπο
// Επίσης, δηλώνουμε μεταβλητές στο σημείο και το scope που χρειάζοντ

void foo() {
    int var1 = 1;
    int var2 = 3;

    ...

    int var3 = 3;                    // δε χρειάζεται πιο πάνω

    if (condition) {
        int var4 = 4;                // var4 ορατό μόνο μέσα στο if
        ...
    }

    for (unsigned int i = 0; i < N; i++) {   // i ορατό μόνο μέσα στο
        int var5 = 5;
        ...
    }
}
```

Names

- **Functions, variables, parameters**: `lowercase_with_underscores`
- **Types**: `CamelCase`
- **Constants**: `UPPERCASE`
- Choose **readable** names (not `a`, `b`, `c`, …)
- In **modules**: prefix with name of module (or abbreviation)
  - avoids conflicts

Names

```c
// ADTList.h

// Οι σταθερές αυτές συμβολίζουν εικονικούς κόμβους στην αρχή/τέλος.
#define LIST_BOF (ListNode)0
#define LIST_EOF (ListNode)0

// Λίστες και κόμβοι αναπαριστώνται από τους τύπους List και ListNode
typedef struct list* List;
typedef struct list_node* ListNode;

// Δημιουργεί και επιστρέφει μια νέα λίστα.

List list_create(DestroyFunc destroy_value);

// Προσθέτει έναν νέο κόμβο __μετά__ τον node με περιεχόμενο value.

void list_insert_next(List list, ListNode node, Pointer value);

// Αφαιρεί τον __επόμενο__ κόμβο από τον node.

void list_remove_next(List list, ListNode node);
```

How to test our code

- For simple code, we typically test it in `main`
  - often with input from the user
- This does not work for larger programs
  - Time consuming
  - Easy to miss edge cases
  - No automation
  - We tend to assume that fixes remain forever

Unit Tests

- A **test** is a piece of code that tests some other code
  - e.g. tests a **module**
- It calls some functions of the module, then checks the result
- Each test should be **independent**
- It should test some basic functionality
  - especially edge cases

Unit Tests

Advantages

- Re-run on every change
- Detect regressions
- Test different implementations of the same module
- Run in automated scripts (e.g. on `git push`)
- Write specifications even before writing the actual code
  - test-driven development

A simple test for stats.h

```c
#include "acutest.h"            // Απλή βιβλιοθήκη για unit testing

#include "stats.h"

void test_find_min(void) {
    int array[] = { 3, 1, -1, 50 };

    TEST_ASSERT(stats_find_min(array, 4) == -1);
    TEST_ASSERT(stats_find_min(array, 3) == -1);
    TEST_ASSERT(stats_find_min(array, 2) == 1);
    TEST_ASSERT(stats_find_min(array, 1) == 3);
    TEST_ASSERT(stats_find_min(array, 0) == INT_MAX);
}
```

A simple test for stats.h

```
void test_find_max(void) {
    int array[] = { 3, 1, -1, 50 };

    TEST_ASSERT(stats_find_max(array, 4) == 50);
    TEST_ASSERT(stats_find_max(array, 3) == 3);
    TEST_ASSERT(stats_find_max(array, 2) == 3);
    TEST_ASSERT(stats_find_max(array, 1) == 3);
    TEST_ASSERT(stats_find_max(array, 0) == INT_MIN);
}

// Λίστα με όλα τα tests προς εκτέλεση
TEST_LIST = {
    { "find_min", test_find_min },
    { "find_max", test_find_max },
    { NULL, NULL } // τερματίζουμε τη λίστα με NULL
};
```

Test coverage

- How to know if the tests cover all functionalities of the code?
- Simple solution: check **which lines** are executed
- `lcov`: a test coverage tool for C
- Try the following in `sample-project`

```
cd tests
make coverage
firefox coverage/index.html
```

Valgrind

- Tool to check memory access
- Finds memory **leaks**
- Also detects access of **deallocated** memory
- Simple use:

```
valgrind ./program
```

Debugging

- Fixing bugs is an art that needs **experience**
- Often more difficult than writing the code
- But following some **concrete steps** can help

Debugging

Step 1: **reproduce** the problem

- In the **simplest** possible way
  - Simplest code, smallest input, number of steps, etc
- Ideally: with a simple **automated test**
- Sometimes this is the hardest part!
  - Keep in mind when reporting bugs to others

Debugging

Step 2: **isolate** the bug

- Which parts of the code are affected by the bug?
- Use the **debugger** (sometimes)
  - Pause at **segmentation faults**
  - Set **breakpoints** and conditional breakpoints
- Add useful logs (`printf`) and `asserts`
- **Comment out** parts of the code to see if behaviour changes

Debugging

Step 3: find the **root cause**

- Often the code that breaks is not the real root
- Follow the code flow **backwards**
  - Examine the debugger's **call stack**
  - Add logs
- Compare the flow in buggy and non-buggy executions

Debugging

Step 4: **understand** and **fix** the bug

- Don't do random changes
- A fix that you don't **understand** is usually not a correct fix!
- Add **documentation**
  - Usually code that fixes a tricky bug needs explanation

Debugging

Step 5: **test** the fix

- Add **tests** if you don't have them already
- Important so that the bug does not **reappear** in the future
- Ideally the new tests should **only pass when the fix is applied**