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

Similarly, how we name things is important.
- variables
- modules
- functions
- types
- etc

Consistent naming greatly improves code quality

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

- 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

- In the same line as the command that opens it
- For small commands (without exaggeration)

Identation

- 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

- Conceptually, int* is a type.
  // The * sticks to the type (not the name)
  int* foo(char* param) {
    int* pointer = &var;
    ...
### Variable declarations

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

### 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

### 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 xes remain forever
Unit Tests

A test is a piece of code that tests other parts of the code
- eg. 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

Advantages
- Re-run on every change
- Detect regressions
- Test different implementations of the same module
- Run in automated scripts (eg 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);
}

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
  ```