

# Testing – Έλεγχος Λειτουργίας

Το Υλικό από το βιβλίο

*The Practice of Programming*

(Kernighan & Pike)

Chapter 6

# Σφάλματα Παντού

“On two occasions I have been asked [by members of Parliament!], ‘Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?’ I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.”

– Charles Babbage

“Program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.”

– Edsger Dijkstra

“Beware of bugs in the above code; I have only proved it correct, not tried it.”

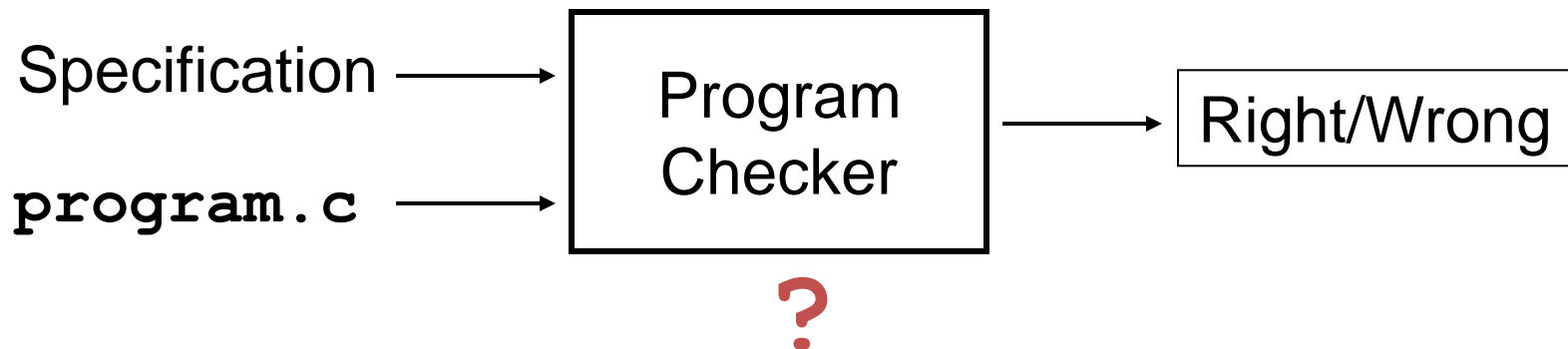
– Donald Knuth

# Στόχοι

- Παρουσιάζουμε
  - External testing (Εξωτερικά)
  - Internal testing (Εσωτερικά)
  - General testing strategies (Γενικές Στρατηγικές)
- Οι λόγοι
  - Hard to know if a large program works properly
  - When developing a large program, a power programmer expends **at least as much effort writing test code** as he/she expends writing the program itself
  - A power programmer is comfortable with a wide variety of program testing techniques and strategies

# Program Verification - Απόδειξη

- Ιδανικά: Prove that your program is correct
  - Can you **prove** properties of the program?
  - Can you **prove** that it even terminates?!!!



# Program Testing

- Πραγματιστικά: Convince yourself that your program probably works



# External vs. Internal Testing

- Types of testing

- External testing

- Designing data to test your program

Καθώς γράφουμε το πρόγραμμα σχεδιάζουμε τα δεδομένα με τα οποία θα το ελέγχουμε

- Internal testing

- Designing your program to test itself

Αναπτύσσουμε κώδικα για αυτό-έλεγχο του προγράμματος καθώς εκτελείται

# External Testing

External testing: Designing data to test your program

- External testing taxonomy
  - (1) Boundary testing (οριακά δεδομένα)
  - (2) Statement testing (έλεγχος εντολών)
  - (3) Path testing (πιθανές ροές εκτέλεσης)
  - (4) Stress testing (μεγάλα και πολλά δεδομένα)
- Θα τα εξετάσουμε ένα – ένα ...

# Boundary Testing

## (1) Boundary testing (έλεγχος οριακών δεδομένων)

- “A testing technique using input values at, just below, and just above, the defined limits of an input domain; and with input values causing outputs to be at, just below, and just above, the defined limits of an output domain.”

— Glossary of Computerized System and Software Development Terminology

- Almost all bugs occur at boundary conditions
- If a program works for boundary conditions, it probably works for all others



# Boundary Testing Example

- Code to get line from stdin and put in character array s (example ex1)

```
int i;  
char s[MAXLINE];  
for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++);  
s[i] = '\0';  
printf("String: |%s|\n", s);
```

- Boundary conditions
  - Input starts with '\n' (κενή γραμμή)
    - Prints empty string (“\0”), so output is “| |”

# Boundary Testing Example (cont.)

- Code to get line from stdin and put in character array

```
int i;  
char s[MAXLINE];  
for (i=0; (s[i]=getchar()) != '\n' && i < MAXLINE-1; i++);  
s[i] = '\0';  
printf("String: |%s|\n", s);
```

- Boundary conditions
  - Line exactly MAXLINE-1 characters long
    - Output is correct, with '\0' in s[MAXLINE-1]
  - Line exactly MAXLINE characters long
    - Last character on the line is overwritten, and newline never read
  - Line more than MAXLINE characters long
    - Some characters, plus newline, not read and remain on stdin

# Boundary Testing Example (cont.)

- Rewrite the code

```
int i;  
char s[MAXLINE];  
for (i=0; i<MAXLINE-1; i++)  
    if ((s[i] = getchar()) == '\n')  
        break;  
s[i] = '\0';
```

- Another boundary condition: EOF

```
for (i=0; i<MAXLINE-1; i++)  
    if ((s[i] = getchar()) == '\n' || s[i] == EOF)  
        break;  
s[i] = '\0';
```

- What are other boundary conditions?
  - Nearly full
  - Exactly full
  - Over full

Λάθος;

# Boundary Testing Example (cont.)

```
for (i=0; ; i++) {  
    int c = getchar();  
    if (c==EOF || c=='\n' || i==MAXLINE-1) {  
        s[i] = '\0';  
        break;  
    }  
    else s[i] = c;  
}
```

- There's still a problem...MAXLINE=9

Input:

**Four**

score	and	seven
-------	-----	-------

**years**

Output:

FourØ
-------

score anØ
-----------

sevenØ
--------

yearsØ
--------

# Ασάφειες στις Προδιαγραφές

- If line is too long, what should happen?
  - Keep first MAXLINE characters, discard the rest?
  - Keep first MAXLINE-1 characters + '\0' char, discard the rest?
  - Keep first MAXLINE-1 characters + '\0' char, save the rest for the next call to the input function?
- Probably, the specification didn't even say what to do if MAXLINE is exceeded
  - Probably the person specifying it would prefer that unlimited-length lines be handled without any special cases at all
  - Moral: testing has uncovered a design problem, maybe even a specification problem!
- Define what to do
  - Truncate long lines?
  - Save the rest of the text to be read as the next line?

# Διδάγματα

- Πολύπλοκες οριακές περιπτώσεις είναι συχνά συμπτώματα κακού σχεδιασμού ή κακών προδιαγραφών.
- Διευκρινίστε τις προδιαγραφές, αν μπορείτε.
- Αν δεν μπορείτε να διευκρινίσετε τις προδιαγραφές τότε διορθώστε το πρόγραμμα, όπως νομίζετε.

# Statement Testing

## (2) Statement testing

- “Testing to satisfy the criterion that each statement in a program be executed at least once during program testing.”

– Glossary of Computerized System and Software Development Terminology

# Statement Testing Example

- Example pseudocode:

Statement testing:

Should make sure both “if” statements and all 4 nested statements are executed

```
if (condition1)  
    statement1;  
else  
    statement2;  
...  
if (condition2)  
    statement3;  
else  
    statement4;  
...
```

- Requires two data sets; example:
  - *condition1* is true and *condition2* is true
    - Executes *statement1* and *statement3*
  - *condition1* is false and *condition2* is false
    - Executes *statement2* and *statement4*



# Path Testing

## (3) Path testing

- “Testing to satisfy coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes. One path from each class is then tested.”

– Glossary of Computerized System and Software Development Terminology

- Much more difficult than statement testing
  - For simple programs, can enumerate all paths through the code
  - Otherwise, sample paths through code with random input

# Path Testing Example

- Example pseudocode:

Path testing:

Should make sure all logical paths are executed

```
if (condition1)  
    statement1;  
else  
    statement2;  
...  
if (condition2)  
    statement3;  
else  
    statement4;  
...
```

- Requires four data sets:

- condition1* is true and *condition2* is true
- condition1* is true and *condition2* is false
- condition1* is false and *condition2* is true
- condition1* is false and *condition2* is false

- Realistic program => combinatorial explosion!!!

- Έλεγχος σε μικρότερα τμήματα ( $N+M < N*M$ )

# Stress Testing

## (4) Stress testing

- “Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements”
  - Glossary of Computerized System and Software Development Terminology

- What to generate
  - Very large inputs
  - Random inputs (binary , ASCII)
- Use computer to generate inputs

# Stress Testing Example 1

- Example program:

Stress testing: Should provide **random** (binary , ASCII) inputs

```
#include <stdio.h>
int main(void) {
    char c;
    while ((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

- Intention: Copy all characters of stdin to stdout; but note the bug!!!
- Works for typical (human-generated) ASCII data sets
- Random (computer-generated?) data set containing byte 255 (decimal), alias 11111111 (binary), causes loop to terminate before end-of-file

# Stress Testing Example 2

- Example program:

Stress testing: Should provide **very large** inputs

```
#include <stdio.h>
int main(void) {
    short charCount = 0;
    while (getchar() != EOF)
        charCount++;
    printf("%hd\n", charCount);
    return 0;
}
```

- Intention: Count and print number of characters in stdin
- Works for reasonably-sized data sets
- Fails for (computer-generated?) data set containing more than 32767 characters

# The assert Macro `assert(expr)`

`#include <assert.h>`

- The **`assert`** macro
  - One actual parameter, which should evaluate to true or false
  - If true (non-zero):
    - Do nothing
  - If false (zero):
    - Print message to stderr “assert at line x failed”
    - Exit the process

# Uses of assert

- Typical uses of **assert**

- Validate formal parameters («προ» συνθήκη)

```
size_t Str_getLength(const char *str) {  
    assert(str != NULL);  
    ...  
}
```

- Check for “impossible” logical flow

```
switch (state) {  
    case START: ... break;  
    case COMMENT: ... break;  
    ...  
    default: assert(0); /* Never should get here */  
}
```

- Make sure dynamic memory allocation requests worked
  - x=malloc(...); assert(x); (Described later in course)

# Disabling asserts

- Problem: **asserts** can be time-consuming
  - Want them in code when debugging, but...
  - Might want to remove them from released code
- Bad “solution”:
  - When program is finished, delete asserts from code
  - But **asserts** are good documentation
  - And in the “real world” no program ever is “finished”!!!
- Solution: Define the **NDEBUG** macro
  - Place **#define NDEBUG** at top of .c file, before all calls of **assert**
  - Makes the **assert** macro expand to nothing
  - Essentially, disables asserts



# Disabling asserts (cont.)

- Problem: Awkward to place **#define NDEBUG** in only released code
- Solution: Define **NDEBUG** when building
  - -D option of gcc defines a macro
  - **gcc -DNDEBUG myfile.c**
    - Defines **NDEBUG** macro in myfile.c, just as if myfile.c contains **#define NDEBUG**
- Controversy: Should **asserts** be disabled in released code?
  - **Asserts** are very time consuming => yes
  - **Asserts** are not very time consuming => sometimes unclear
    - Would user prefer (1) exit via assert, or (2) possible data corruption?

# Internal Testing

- Internal testing: Designing your program to test itself
- Internal testing techniques
  - (1) Testing invariants
  - (2) Verifying conservation properties
  - (3) Checking function return values
  - (4) Changing code temporarily
  - (5) Leaving testing code intact
- Εξετάζουμε τις τεχνικές μία-μία...

# Testing Invariants (Έλεγχος Αναλοΐωτων Συνθηκών)

## (1) Testing invariants

- testing pre-conditions and post-conditions
- Some aspects of data structures should not vary
- A function that affects data structure should check those invariants at its leading and trailing edges (προ- και μετά- αλλαγής)
- Example: “doubly-linked list insertion” function
  - At leading and trailing edges
    - Traverse doubly-linked list
    - When node x points forward to node y, does node y point backward to node x?
- Example: “binary search tree insertion” function
  - At leading and trailing edges
    - Traverse tree
    - Are nodes still sorted?

# Testing Invariants (cont.)

- Convenient to use **assert** to test invariants

```
#ifndef NDEBUG
int isValid(MyType object) {
    ...
    Test invariants here.
    Return 1 (TRUE) if object passes
    all tests, and 0 (FALSE) otherwise.
    ...
}
#endif

void myFunction(MyType object) {
    assert(isValid(object));
    ...
    Manipulate object here.
    ...
    assert(isValid(object));
}
```

Can use NDEBUG  
in your code, just  
as **assert** does

# Verifying Conservation Properties (Πιστοποίηση Σταθερών Ιδιοτήτων)

## (2) Verifying conservation properties

- Generalization of testing invariants
- A function should check affected data structures at leading and trailing edges
- Example: **Str\_concat()** function
  - At leading edge, find lengths of two given strings; compute sum
  - At trailing edge, find lengths of resulting string
  - Is length of resulting string equal to sum?
- Example: List insertion function
  - At leading edge, find old length of list
  - At trailing edge, find new length of list
  - Does new length equal old length + 1?

# Checking Return Values

## (3) Checking function return values

### – In Java and C++:

- Method that detects error can “throw a checked exception”
- Calling method must handle the exception (or rethrow it)

### – In C:

- No exception-handling mechanism
- Function that detects error typically indicates so via return value
- Programmer easily can forget to check return value
- Programmer (generally) **should** check return value

# Checking Return Values (cont.)

## (3) Checking function return values (cont.)

- Example: **scanf()** returns number of values read

Bad code

```
int i;  
scanf("%d", &i);
```

Good code

```
int i;  
if (scanf("%d", &i) != 1)  
    /* Error */
```

- Example: **printf()** can fail if writing to file and disk is full; returns number of characters (not values) written

Bad code???

```
int i = 100;  
printf("%d", i);
```

Good code, or overkill???

```
int i = 100;  
if (printf("%d", i) != 3)  
    /* Error */
```

# Changing Code Temporarily

## (4) Changing code temporarily

- Temporarily change code to generate artificial boundary or stress tests
- Example: Array-based sorting program
  - Temporarily make array very small
  - Does the program handle overflow?
- Μέγεθος στοίβας για έλεγχο υπερχείλισης
- Example: Program that uses a hash table
  - Temporarily make hash function return a constant
  - All bindings map to one bucket, which becomes very large
  - Does the program handle large buckets?



# Leaving Testing Code Intact

## (5) Leaving testing code intact

- Leave important testing code in the code
- Maybe surround with **#ifndef NDEBUG ... #endif**
- Control with **-DNDEBUG** gcc option
  - Enables/disables assert macro
  - Also could enable/disable **your** debugging code (see “Testing Invariants” example)
- Beware of conflict:
  - Extensive **internal testing** can lower maintenance costs
  - **Code clarity** can lower maintenance costs
  - But... Extensive **internal testing** can decrease **code clarity!**

# General Testing Strategies

- General testing strategies
  - (1) **Testing incrementally**
  - (2) Comparing implementations
  - (3) Automation
  - (4) Bug-driven testing
  - (5) Fault injection
- Ας τις εξετάσουμε μια-μια...

# Testing Incrementally

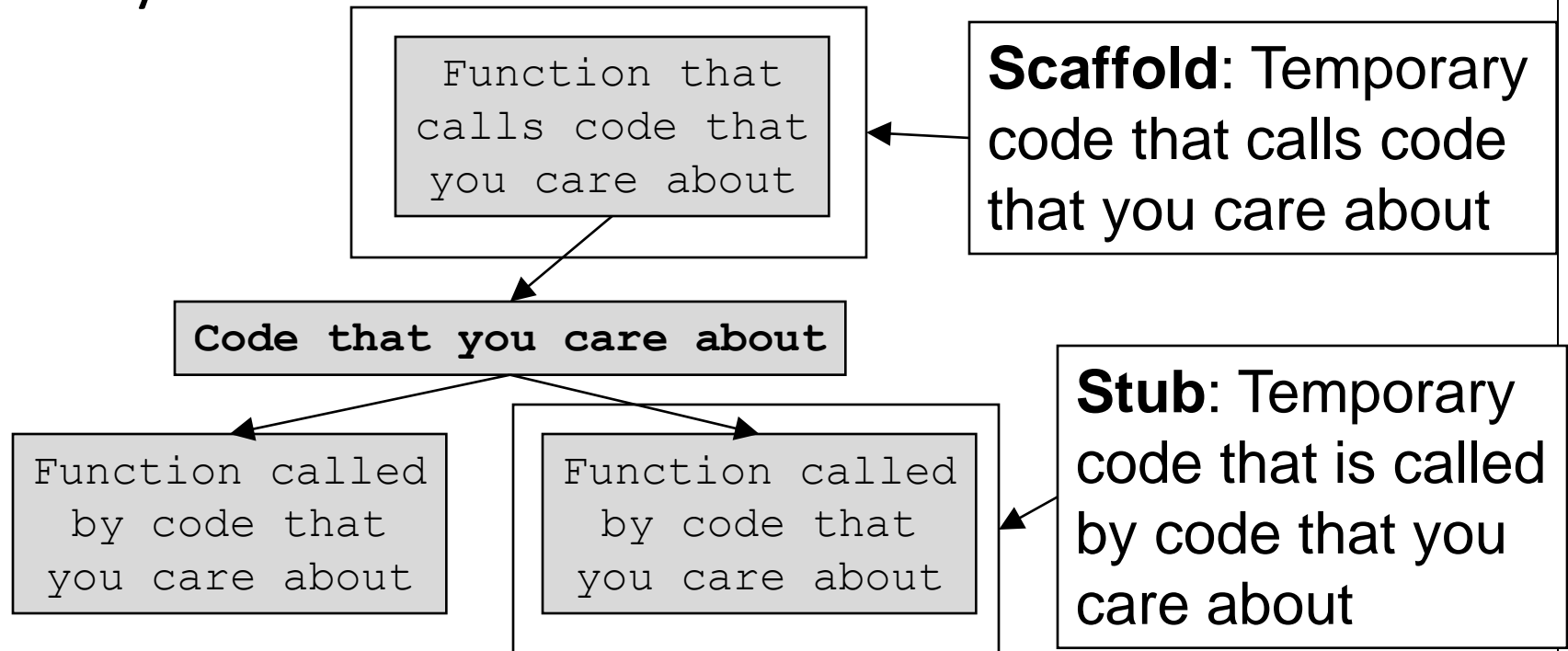
## (1) Testing incrementally

- Test as you write code
  - Add tests as you create new cases
  - Test simple parts before complex parts
  - Test units (i.e., individual modules) before testing the system
- Do regression testing (έλεγχος προηγούμενων)
  - A bug fix often creates new bugs in a large software system, so...
  - Must make sure system has not “regressed” such that previously working functionality now is broken, so...
  - Test all cases to compare the new version with the previous one

# Testing Incrementally (cont.)

## (1) Testing incrementally (cont.)

- Create scaffolds and stubs to test the code that you care about



# Comparing Implementations

## (2) Compare implementations

- Make sure that multiple independent implementations behave the same
- Example: Compare behavior of your `str.h` functions vs. standard library `string.h` functions

# Automation (κώδικας ελέγχου)

## (3) Automation

- Testing manually is tedious and unreliable, so...
- Create testing code
  - Scripts and data files to test your programs (recall decomment program testing)
  - Software clients to test your modules (recall Str module testing)
- Know what to expect
  - Generate output that is easy to recognize as right or wrong
  - Example: Generate output of **diff** command instead of raw program output
- Automated testing can provide:
  - **Much** better coverage than manual testing
  - Bonus: Examples of typical/atypical use for other programmers

# Bug-Driven Testing

## (4) Bug-driven testing

- Find a bug => immediately create a test that catches it
- Facilitates regression testing

# Fault Injection

## (5) Fault injection

- Intentionally (temporarily) inject bugs!!!
- Then determine if testing finds them
- Test the testing!!!



# Who Tests What

- Programmers
  - White-box testing
  - Pro: An implementer knows all data paths
  - Con: Influenced by how code is designed/written
- Quality Assurance (QA) engineers
  - Black-box testing
  - Pro: No knowledge about the implementation
  - Con: Unlikely to test all logical paths
- Customers
  - Field testing
  - Pros: Unexpected ways of using the software; “debug” specs
  - Cons: Not enough cases; customers don’t like “participating” in this process; malicious users exploit the bugs

# Περίληψη

- External testing taxonomy
  - Boundary testing
  - Statement testing
  - Path testing
  - Stress testing
- Internal testing techniques
  - Checking invariants
  - Verifying conservation properties
  - Checking function return values
  - Changing code temporarily
  - Leaving testing code intact

# Περίληψη (συνέχεια)

- General testing strategies
  - Testing incrementally
    - Regression testing
    - Scaffolds and stubs
  - Automation
  - Comparing independent implementations
  - Bug-driven testing
  - Fault injection
- Test the code, the tests – and the specification!