ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών

Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Ενότητα 5: Τεχνικές Προγραμματισμού

Ιωάννης Κοτρώνης
Σχολή Θετικών Επιστημών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

Σκοποί ενότητας

- Να εξετάσει πλεονεκτήματα και μειονεκτήματα των τριών σχεδιαστικών επιλογών που χρησιμοποιήσαμε
- Να παρουσιάσει τεχνικές ελέγχου προγραμμάτων
- Να προσδιορίσει κανόνες καλής χρήσης των ενοτήτων
- Να εξετάσει την τεχνική της αναδρομής

Περιεχόμενα ενότητας

- Επιλογές Σχεδιασμού ΑΤΔ
- Έλεγχος Προγραμμάτων
- Χαρακτηριστικά Ενοτήτων
- Αναδρομή

Επιλογές Σχεδιασμού

Επιλογές και Κριτήρια Σχεδιασμού ΑΤΔ Ανεξαρτήτως από Γλώσσα Υλοποίησης

Κύκλος (Ζωής) Λογισμικού (ΑΤΔ)

Γενικά

- Ορισμός ΑΤΔ (Προδιαγραφές)
    - Οργάνωση Δεδομένων
    - Τι κάνει
- Υλοποίηση
    - Σχεδιασμός (ανεξάρτητος από γλώσσα υλοποίησης)
        - Πώς το κάνει (πίνακες, συνδεδεμένες λίστες, κλπ)
    - Ανάπτυξη (σε Γλώσσα)
        - Ενότητες στην C, java class
- Δοκιμή και πάλι …

Παράδειγμα

- Ορισμός ΑΤΔ Στοίβα
    - Γραμμική Διάταξη
    - Δημιουργία, κενή, ώθηση, εξαγωγή
- Υλοποίηση
    - Σχεδιασμός
        - πίνακα + κορυφή
        - Συνδεδεμένη Λίστα
    - Ανάπτυξη σε C
        - Ενότητες της C με πλήρη Απόκρυψη
- Δοκιμή και πάλι …

Τρεις Επιλογές Σχεδιασμού ΑΤΔ

1. Με Πίνακα: Αντιστοιχούμε Στοιχεία του ΑΤΔ σε Πίνακα.
    1. TyposStoixeiou List[N]; // ο πίνακας με τα δεδομένα
    2. Συναρτήσεις πρόσβασης στα στοιχεία του πίνακα
2. Με Συνδεδεμένους Κόμβους

    typedef struct Node {TyposStoixeiou data; TyposLink next} Node; // τύπος κόμβου

    TyposLink Start; // η Αρχή των κόμβων

a) Σύνδεση/διαχείριση με pointers-Κόμβοι από heap

- i. typedef Node * TyposLink ; // η σύνδεση-δείκτης
- ii. Κόμβοι από Heap // διαθέσιμος χώρος
- iii. New=malloc(sizeof(Node)); free(New); // διαχείριση κόμβων

b) Σύνδεση/διαχείριση με θέσεις πίνακα-Κόμβοι στοιχεία πίνακα

- i. typedef int TyposLink; // η σύνδεση-int
- ii. Node List[N]; // όλοι οι κόμβοι σε πίνακα
- iii. πάρε-κόμβο και αποδέσμευση // διαχείριση

Σχεδιασμός με Διάταξη Πίνακα (ΔΠ)

Πλεονεκτήματα

- Άμεση Πρόσβαση σε κάθε στοιχείο, π.χ. List[i]
- Οικονομία Μνήμης σε επίπεδο στοιχείου, το List[i] περιέχει μόνο δεδομένα
- Εισαγωγές και Διαγραφές στην αρχή ή τέλος υποστηρίζονται αποδοτικά σε σταθερό χρόνο O(1)

Μειονεκτήματα

- Συγκεκριμένο μέγεθος (συντηρητική δήλωση δεν είναι αρκετή ή υπερβολική δήλωση με σπατάλη χώρου)
- Πολλά Ενδιάμεσα Κενά Στοιχεία, αραιές δομές (π.χ. πολυώνυμο, πίνακες)
- Εισαγωγές + Διαγραφές από την μέση κοστίζουν (Λίστα O(n))

Σχεδιασμός Με Συνδεδεμένους Κόμβους (ΣΚ)

Πλεονεκτήματα

- Μόνο χρήσιμα στοιχεία, δεν έχουμε κενά στοιχεία (πολυώνυμο)
- Εισαγωγές/Διαγραφές σε σταθερό χρόνο σε/από οποιαδήποτε θέση

Μειονεκτήματα

- Επιπλέον μνήμη σε επίπεδο στοιχείου λόγω σύνδεσης (π.χ. μονής ή διπλά συνδεδεμένης λίστας)
- Σειριακή Πρόσβαση στα στοιχεία. Η τυχαία πρόσβαση κοστίζει O(n).

Σχεδιασμός Με Συνδεδεμένους Κόμβους (ΣΚ) – Struct+Pointers (SP)

Πλεονεκτήματα

- Όλα τα γενικά των ΣΚ
- Δυναμικό μέγεθος δομής, όσοι κόμβοι απαιτούνται

Μειονεκτήματα

- Όλα τα γενικά των ΣΚ
- Διαχείριση Heap με malloc + free ευθύνη του προγραμματιστή

Σχεδιασμός Με Συνδεδεμένους Κόμβους (ΣΚ) – Πίνακα + Θέση (ΠΘ)

Πλεονεκτήματα

- Όλα τα γενικά των ΣΚ
- Χρήσιμος σχεδιασμός όταν δεν έχουμε δυναμική ανάκτηση μνήμης
  - Κάρτες SIM με συγκεκριμένο χώρο για επαφές
- Αποφεύγουμε δείκτες
  - σε συστήματα ασφαλείας πολλές φορές οι δείκτες δεν συνιστώνται

Μειονεκτήματα

- Όλα τα γενικά των ΣΚ
- Συγκεκριμένο μέγεθος πίνακα, αλλά με κάποια ευελιξία (π.χ. στα Δένδρα το μέγεθος του πίνακα καθορίζει τον μέγιστο αριθμό των κόμβων ανεξάρτητα βάθους)

Πώς Επιλέγουμε

Διάταξη Πίνακα

- Γνωστό πλήθος στοιχείων
- Διάταξη στοιχείων χωρίς πολλά κενά, π.χ. πλήρη πολυώνυμα
- Όχι μετακινήσεις στοιχείων.
- Αλγόριθμοι με άμεση πρόσβαση στοιχείων (αναζήτηση, ταξινόμηση)
- Συχνές τυχαίες προσβάσεις σε σύγκριση με εισαγωγές και διαγραφές

Συνδεδεμένοι Κόμβοι (ΣΚ-SP)

(ΣΚ-ΠΘ είναι συμβιβασμός)

- Δεν γνωρίζουμε πλήθος στοιχείων
- Διάταξη έχει κενά, π.χ. μη πλήρη πολυώνυμα, αραιοί πίνακες, μη ισοζυγισμένα δένδρα
- Συχνές εισαγωγές και διαγραφές σε σύγκριση με τυχαίες προσβάσεις.

Υλοποιήσεις Στοίβας, Ουράς, Λίστας

	Διάταξη Πίνακα	ΣΚ-ΠΘ	ΣΚ-SP
Στοίβα	ΝΑΙ- Γνωστό μέγεθος	ΌΧΙ-μηδέν όφελος Δεν έχουμε μετακινήσεις, εισαγωγές, διαγραφές από την μέση. Πράξεις με βάσει την κορυφή.	ΝΑΙ- Δυναμικό μέγεθος
Ουρά	ΝΑΙ- Γνωστό μέγεθος	ΌΧΙ. Δεν έχουμε μετακινήσεις, εισαγωγές, διαγραφές από την μέση. Πράξεις βάσει εμπρός και πίσω.	ΝΑΙ- Δυναμικό μέγεθος
Λίστα	ΝΑΙ- Γνωστό μέγεθος, άμεση πρόσβαση στα στοιχεία, χωρίς (πολλές) διαγραφές και εισαγωγές.	ΝΑΙ-Γνωστό μέγεθος. Εισαγωγές, διαγραφές από την μέση. Επιτρέπει παραλλαγές ανάλογα με τις απαιτήσεις (είδαμε μόνο μία παραλλαγή-μονά συνδεδεμένη)	ΝΑΙ-Δυναμικό μέγεθος Εισαγωγές, διαγραφές από την μέση Επιτρέπει παραλλαγές (πόσες;) ανάλογα με τις απαιτήσεις

Παραλλαγές Λίστας με ΣΚ ΠΘ ή SP

- **Απλή ή Διπλά συνδεδεμένη;** (κόστος μνήμης – χρόνου)
  - Απλή: βασική μνήμη σύνδεσης, O(n) για προηγούμενο κόμβο
  - Διπλά: επιπλέον μνήμη σύνδεσης , O(1) για προηγούμενο κόμβο
- **Με κεφαλή ή χωρίς;** (κόστος ανάπτυξης- μνήμης)
  - Χωρίς: απαιτούνται 2 πράξεις εισαγωγής και 2 διαγραφής
  - Με κεφαλή: 1 πράξη εισαγωγής και 1 διαγραφής με κόστος έναν κόμβο επιπλέον
- **Κυκλική ή μη;** (κόστος χρόνου)
  - αν απαιτούνται πράξεις που συνεχίζουμε από το τέλος.
- **Με ένδειξη Τέλους Λίστας ή μη;** (κόστος χρόνου – μνήμης)
  - Αν γίνονται συχνά εισαγωγές στο τέλος (π.χ. ουρά)

2x2x2x2=16 παραλλαγές (μόνο;) x2 (ΠΘ-SP)= **32** Όλες χρήσιμες υλοποιήσεις. Με πόσες διεπαφές; (1-2-16-32;)

Έλεγχος Προγραμμάτων

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

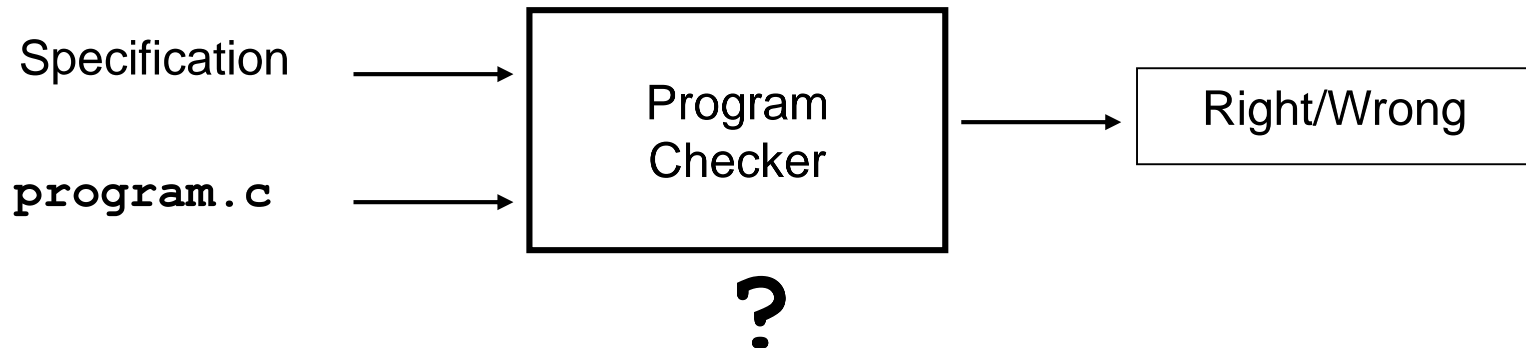
“Beware of bugs in the above code; I have only proved it correct, not tried it.”
– Donald Knuth

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 ελέγξουμε
  - 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';
```

Λάθος;

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

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:

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

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

- 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:

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

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

- 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 Αναλλοίωτων 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

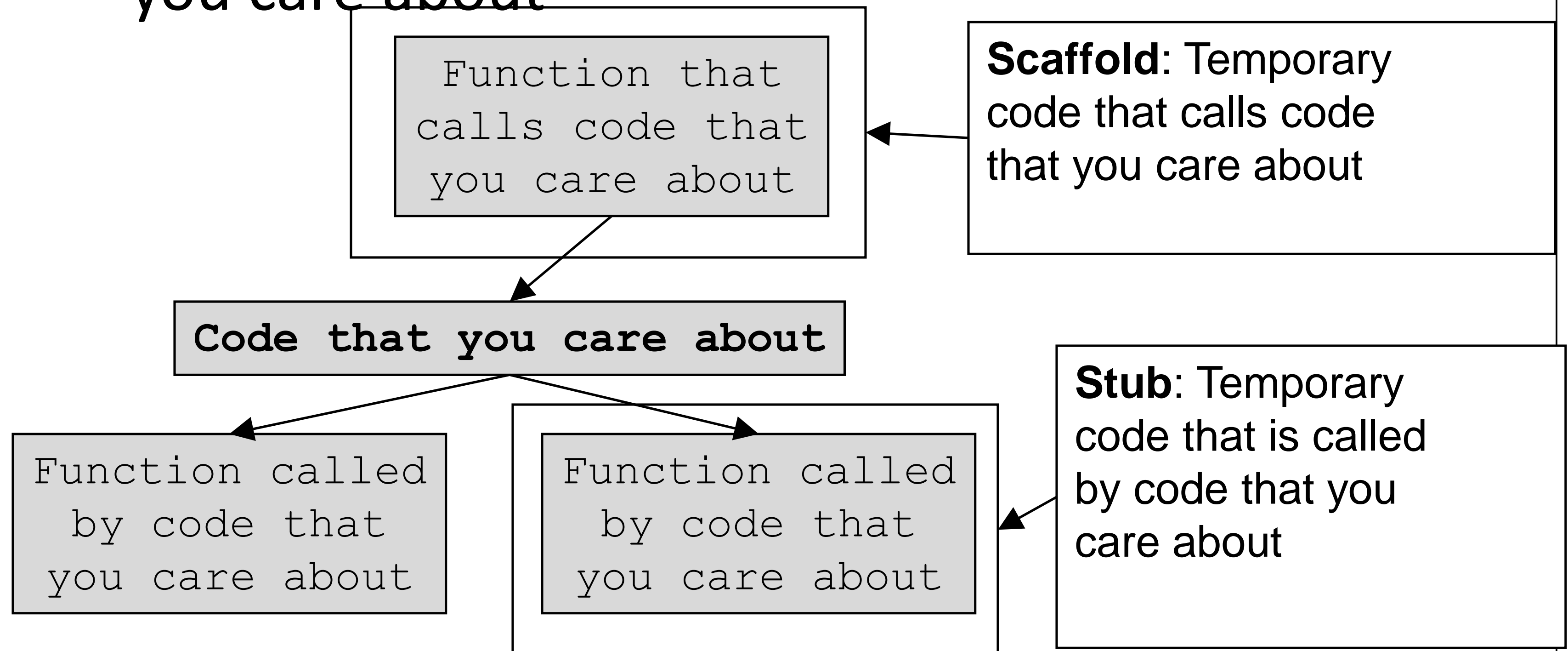
(1) Testing incrementally

- 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 προηγουμένων)
  - 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

Summary (cont.)

- 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!

Χαρακτηριστικά Ενοτήτων

Στόχοι

- Πώς να δημιουργήσεις ενότητες υψηλής ποιότητας στην C.
- Γιατί χρειάζεται;
  - Η αφαίρεση είναι απαραίτητη τεχνική για την ανάπτυξη και κατανόηση μεγάλων, πολύπλοκλων συστημάτων
  - Ο καλός προγραμματιστής γνωρίζει πώς να βρει κατάλληλες αφαιρέσεις σε ένα μεγάλο πρόγραμμα
  - Ο καλός προγραμματιστής γνωρίζει πώς να παρουσιάζει τις αφαιρέσεις σε ένα μεγάλο πρόγραμμα μέσω των ενοτήτων.

Χαρακτηριστικά Ενοτήτων

- Μια καλά σχεδιασμένηενότητα
  1. Διαχωρίζει διεπαφή και υλοποίηση (μερική απόκρυψη)
  2. Ενθυλακώνει δεδομένα (ολική απόκρυψη)
  3. Διαχειρίζεται πόρους με συνέπεια
  4. Κάνει καλή επιλογή Ονομάτων
  5. Έχει ελάχιστη διεπαφή
  6. Αναφέρει λάθη στους πελάτες
  7. Ορίζει συμβόλαια
  8. Έχει υψηλή συνάφεια
  9. Έχει ασθενή εξάρτηση

1. Διαχωρίζει διεπαφή και υλοποίηση

- Αποκρύπτει την υλοποίηση συναρτήσεων από τους πελάτες (υλοποίηση αφαίρεσης)
- Επιτρέπει την ανεξάρτητη μεταγλώττιση

Παράδειγμα

Stack: Στοίβα που διαχειρίζεται strings

- Σχεδιαστική Επιλογή με Δυναμική Συνδεδεμένη Λίστα.
- Πράξεις
  - new: Δημιουργία νέου Stack
  - free: Απελευθέρωση
  - push: ώθηση
  - top: εξαγωγή στοιχείου κορυφής
  - pop: απόρριψη στοιχείου κορυφής
  - isEmpty: Return 1 αν η Stack κενό

Χωρίς απόκρυψη

```
/* stack.c */
struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};
struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
void  Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}
```

```
/* client.c */

#include "stack.c"

/* Use the functions
defined in stack.c. */
```

- Δεν υπάρχει διεπαφή (ένα αρχείο)
- Αλλαγή stack.c => rebuild stack.c και πελάτη
- Χωρίς αφαίρεση, ορισμοί με αχρείαστες λεπτομέρειες είναι ορατοί.

Η Μερική Απόκρυψη (διεπαφή)

Ενότητα Stack αποτελείται από δυο αρχεία

- Το stack.h (the interface) δηλώνει functions and ορίζει data structures

    (ο τύπος στοιχείου είναι char*)

```
/* stack.h */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};
struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Μερική Απόκρυψη (υλοποίηση)

Το stack.c (η υλοποίηση) ορίζει functions

- #include “stack.h”
  - Έλεγχος μεταγλωττιστή
  - Οι συναρτήσεις έχουν πρόσβαση στα δεδομένα

```
/* stack.c */
#include "stack.h"

struct Stack *Stack_new(void) {…}
void  Stack_free(struct Stack *s) {…}
void  Stack_push(struct Stack *s, const char *item) {…}
char *Stack_top(struct Stack *s) {…}
void  Stack_pop(struct Stack *s) {…}
int   Stack_isEmpty(struct Stack *s) {…}
```

Μερική Απόκρυψη (χρήση)

- Πελάτης #include την διεπαφή
- Αν αλλάξει το stack.c => μεταγλώττιση μόνο του stack.c, όχι του πελάτη.
- Καλύτερη αφαίρεση (οι προσβάσεις στην δομή μπορούν να γίνουν μέσω συναρτήσεων)

```
/* client.c */

#include "stack.h"

/* Use the functions declared in stack.h. */
```

2. Ενθυλάκωση Δεδομένων (Ολική Απόκρυψη)

Η πρόσβαση στη δομή αποκλειστικά μέσω συναρτήσεων

- Πληρης απόκρυψη λεπτομερειών υλοποίησης
- Συναρτήσεις για πρόσβαση
- Οι πελάτες δεν έχουν πρόσβαση απευθείας στα δεδομένα.

• Όφελος:

- Ευκρίνεια - μέσω της αφαίρεσης
- Ασφάλεια - οι πελάτες δεν μπορούν να καταστρέψουν αντικείμενα
- Ευελιξία – Επιτρέπει αλλαγές στην υλοποίηση, ακόμα και στα δεδομένα, χωρίς να επηρεάζονται οι πελάτες.

Πρόβλημα με Μερική απόκρυψη

```
/* stack.h */

struct Node {
   const char *item;
   struct Node *next;
};
struct Stack {
   struct Node *first;
};

struct Stack *Stack_new(void);
void  Stack_free(struct Stack *s);
void  Stack_push(struct Stack *s, const char *item);
char *Stack_top(struct Stack *s);
void  Stack_pop(struct Stack *s);
int   Stack_isEmpty(struct Stack *s);
```

Ορισμοί στο .h

– Διεπαφή αποκαλύπτει υλοποίηση με συνδεδεμένη λίστα

– Ο πελάτης μπορεί να αλλάξει/προσπελάσει απευθείας τα δεδομένα, πιθανά να τα αλλοιώσει.

– struct Stack S; … S->item = NULL;

Ολική Απόκρυψη

```
/* stack.h */

typedef struct Stack * Stack_T;

Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

- Μετακίνηση ορισμού στο .c και χρήση δείκτη
  - Αδιαφανής (opaque) δείκτης Stack_T
  - “Stack_T” ο νέος τύπος
  - Ο πελάτης δεν έχει πρόσβαση στα δεδομένα άμεσα; Τα δεδομένα είναι αδιαφανή.
  - Stack_T S; … S->item=NULL (syntax error).

Stdio

```
/* stdio.h */

struct FILE {
   int cnt;     /* characters left */
   char *ptr;   /* next character position */
   char *base; /* location of buffer */
   int flag;    /* mode of file access */
   int fd;      /* file descriptor */
};
…
```

– Μερική απόκρυψη

– Αλλά δεν χρειάζεται να δούμε τους ορισμούς

3. Διαχειρίζεται πόρους με συνέπεια

- Ελευθερώνει πόρους μόνο αν τους έχει δεσμεύσει. Π.χ
    - malloc <=> free
    - opens file <=> closes file
- Δέσμευση και αποδέσμευση σε διαφορετικά επίπεδα έχει κινδύνους
    - Δεν κάνουμε free => memory leak (διαρροή)
    - Δεν κάνουμε malloc => dangling pointer (αιωρούμενος), seg fault
    - Δεν κλείνουμε close file => μη αποδοτική χρήση πόρου
    - Δεν ανοίγουμε to open file => dangling pointer (αιωρούμενος), seg fault

Παράδειγμα

- Stack: Ποιος δεσμεύει και αποδεσμεύει τα strings;
  - Αποδεκτές Επιλογές

    (1) Client allocates and frees strings

    - **`Stack_push()`** δεν δημιουργεί string, δέχεται δείκτη.
    - **`Stack_pop()`** δεν ελευθερώνει το string
    - **`Stack_free()`** δεν ελευθερώνει τα strings

    (2) Stack object allocates and frees strings

    - **`Stack_push()`** δημιουργεί string
    - **`Stack_pop()`** ελευθερώνει το string
    - **`Stack_free()`** ελευθερώνει τα strings
  - Η επιλογή μας η (1) για να έχουμε γενικό ορισμό του Stack.
  - Μη αποδεκτές επιλογές:
    - Ο πελάτης δημιουργεί string , Stack ελευθερώνει
    - Stack δημιουργεί strings, ο πελάτης ελευθερώνει

4. Καλή επιλογή Ονομάτων

Ονόματα με συνέπεια

- Όνομα συνάρτησης περιέχει το όνομα της δομής
  - Βοηθάει στην συντήρηση του προγράμματος
  - Μειώνει πιθανότητα σύγκρουσης ονομάτων
- Οι συναρτήσεις να έχουν συνεπή διάταξη παραμέτρων
  - Βοηθάει στην κλήση τους

Παραδείγματα

- Stack

  (+) Συναρτήσεις ξεκινούν με πρόθεμα “Stack_”

  (+) πρώτη παράμετρος τύπος δομής

- string

  (+) ξεκινούν με “str”

  (+) Προορισμός πρώτη, προέλευση δεύτερη; Μιμείται απόδοση τιμής

- stdio

  (-) Μερικές συναρτήσεις ξεκινούν “f”, άλλες όχι

  (-) Μερικές συναρτήσεις έχουν πρώτη παράμετρο FILE. Άλλες (e.g. **`putc()`**) διαφορετική.

5. Έχει ελάχιστη διεπαφή

– Δήλωση στο .h εφόσον
  - Είναι απαραίτητη ή
  - Βολική
– Περισσότερες συναρτήσεις σημαίνει υψηλότερο κόστος συντήρησης και χρόνο μάθησης

Παράδειγμα Στοίβα

```
/* stack.h */

typedef struct Stack *Stack_T ;
Stack_T Stack_new(void);
void  Stack_free(Stack_T s);
void  Stack_push(Stack_T s, const char *item);
char *Stack_top(Stack_T s);
void  Stack_pop(Stack_T s);
int   Stack_isEmpty(Stack_T s);
```

– Όλες απαραίτητες

Παράδειγμα Stack

- Αν προσθέταμε την

```
void Stack_clear(Stack_T s);
```

  - Ακυρώνει όλα τα στοιχεία
  - Δεν είναι απαραίτητη; Ο πελάτης μπορεί με συνεχόμενα **pop()** να επιτύχει το ίδιο
  - Ίσως βολική

Παράδειγμα String

```
/* string.h */
/* απαραίτητες*/
size_t strlen(const char *s);
char  *strncpy(char *dest, const char *src, size_t n);
char  *strncat(char *dest, const char *src, size_t n);
char  *strncmp(const char *s, const char *t, size_t n);
char  *strstr(const char *haystack, const char *needle);

/* βολικές και αποδοτικές */
char  *strcpy(char *dest, const char *src);
char  *strcat(char *dest, const char *src);
char  *strcmp(const char *s, const char *t);
```

Παράδειγμα stdio

```
/* απαραίτητες */
FILE *fopen(const char *filename, const char *mode);
int   fclose(FILE *f);
int   fflush(FILE *f);
int   fgetc(FILE *f);
int   putc(int c, FILE *f);
int   fscanf(FILE *f, const char *format, …);
int   fprintf(FILE *f, const char *format, …);

/* βολικές */
int   getchar(void);
int   printf(const char *format, …);
int   putchar(int c);
int   scanf(const char *format, …);

/* τίποτα από τα δυο */
int   getc(FILE *f);
…
```

6. Αναφέρει λάθη

Αναφέρει λάθη στους πελάτες

– Η ενότητα διαπιστώνει λάθη, το οποία χειρίζεται ο πελάτης

Τα λάθη τα χειρίζεται καλύτερα ο πελάτης

Αναφορά λαθών στην C

- Εντοπισμός λάθους
  - `if` statement
  - `assert` macro
- Αναφορά στον πελάτη
  - Με καθολική μεταβλητή
    - Ξεχνάμε να ελέγξουμε (δεν φαίνεται άμεσα)
    - Δεν ενδείκνυται για πολυνηματικές εφαρμογές
  - Επιστροφή τιμής
    - Ποια τιμή να επιλέξουμε για λάθος
  - Επιπλέον παράμετρος-σημαία (flag)
    - Μια επιπλέον παράμετρος
  - Κλήση `assert` macro?
    - Εντοπίζει, αλλά δεν ανακάμπτει
- Δεν υπάρχει ιδανική επιλογή

Αναφορά λαθών στην C (συνεχ.)

- Διαφοροποιούμε
- Λάθη χρήστη (User errors)
  - Errors made by human user
    - Example: Bad data in stdin
    - Example: Bad command-line argument
  - Errors that “easily could happen”
  - To detect: Use **`if`** statement
  - To report: Use return value or (by-pointer-value) parameter
- Λάθη Προγραμματιστή (Programmer errors)
  - Errors made by a programmer
  - Errors that “never should happen”
    - Example: Call **`Stack_pop()`** with NULL stack, empty stack
  - To detect and report: Use **`assert`**
- The distinction sometimes is unclear
  - Example: Write to file fails because disk is full

Αναφορά λαθών στην C (συνεχ.)

- Stack

```
/* stack.c */

…

void Stack_push(Stack_T s, const char *item) {
   struct Node *p;
   assert(s != NULL);
   p = (struct Node*)malloc(sizeof(struct Node));
   assert(p != NULL);
   p->item = item;
   p->next = s->first;
   s->first = p;
}
```

  – Stack functions:
    - Consider invalid parameter to be **programmer** error
    - Consider **`malloc()`** failure to be **programmer** error
    - Detect/report no **user** errors

Παραδείγματα

- string
  - No error detection or reporting
    - Example: NULL parameter to **`strlen()`** => probable seg fault
- stdlib
  - Uses return values to indicate failure
    - Note awkwardness of **`scanf()`**
  - Sets global variable “errno” to indicate cause of failure

7. Ορίζει συμβόλαια

- A module should establish contracts with its clients
- Contracts should describe what each function does, esp:
  - Meanings of parameters
  - Valid/invalid parameter values
  - Meaning of return value
  - Side effects

- Οι λόγοι
  - Establishing contracts facilitates cooperation between multiple programmers on a team
  - Establishing contracts assigns blame to violators
    - Catch errors at the door!
    - Better that the boss yells at the programmer who is your client rather than at you!!!

Ορισμός συμβολαίων στην C

- Προτείνουμε
- Με σχόλια
  - Η υλοποίηση ακολουθεί τα σχόλια

Παράδειγμα

- Stack

```
/* stack.h */
…
char *Stack_top(Stack_T s);
/* Return the top item of stack s.
   It is a checked runtime error for s
   to be NULL or empty. */
…
```

- Comment defines contract:
  - Meanings of function's parameters
    - s is the pertinent stack
  - Valid/invalid parameter values
    - s cannot be NULL or empty
  - Meaning of return value
    - The return value is the top item
  - Side effects
    - (None, by default)

8. Συνάφεια

Υψηλή Συνάφεια

- – Οι συναρτήσεις να έχουν σχέση μεταξύ τους

- Βοηθάει την αφαίρεση

Υψηλή Συνάφεια – Παραδείγματα

- Stack
  - (+) All functions are related to the encapsulated data
- string
  - (+) Most functions are related to string handling
  - (-) Some functions are not related to string handling
    - **`memcpy(),memmove(),memcmp(),memchr(),memset()`**
  - (+) But those functions are similar to string-handling functions
- stdio
  - (+) Most functions are related to I/O
  - (-) Some functions don't do I/O
    - **`sprintf(),sscanf()`**
  - (+) But those functions are similar to I/O functions

9. Ασθενής Εξάρτηση

Τι σημαίνει

- Να συνδέεται λίγο με άλλες ενότητες
- Διάδραση εσωτερικά σε ενότητες πιο έντονες από διάδραση μεταξύ ενοτήτων

- Παρατηρήσεις
  - Συντήρηση : Με ασθενή εξάρτηση το πρόγραμμα αλλάζει πιο εύκολα
  - Επαναχρησιμοποίηση: Η ασθενή εξάρτηση βοηθάει στην επαναχρησιμοποίηση σε άλλα προγράμματα
- Εμπειρία
  - Ενότητες έχουν λιγότερα λάθη

Ασθενής Εξάρτηση Παράδειγμα

- Σχεδιασμός

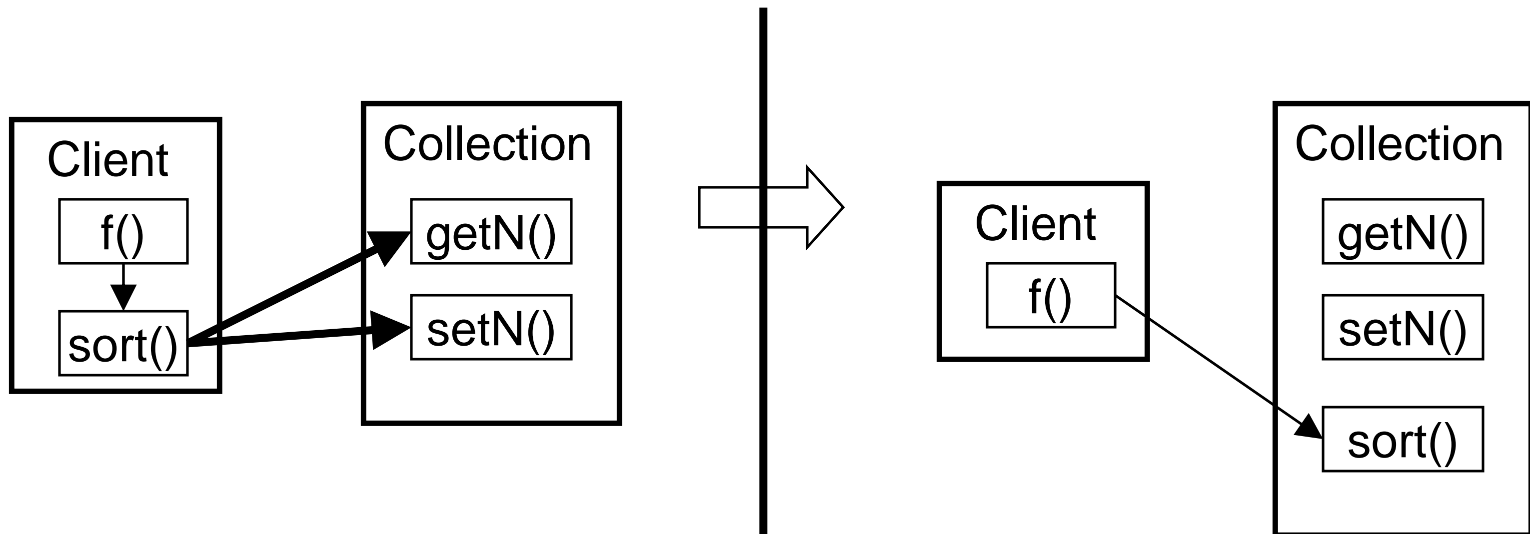
- Πελάτης καλεί πολλές συναρτήσεις
- Δυνατή Εξάρτηση - σχεδιασμός

- Πελάτης καλεί λίγες συναρτήσεις
- Ασθενής εξάρτηση - σχεδιασμός

Ασθενής Εξάρτηση-Παράδειγμα

Many function calls One function call

- Κατά την εκτέλεση-ασθενής εξάρτηση



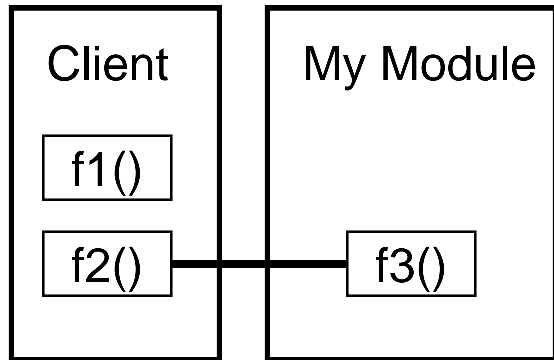
- Πολλές κλήσεις
- Δυνατή Εξάρτηση

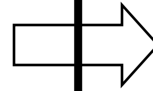
- Λίγες Κλήσεις
- Ασθενής Εξάρτηση

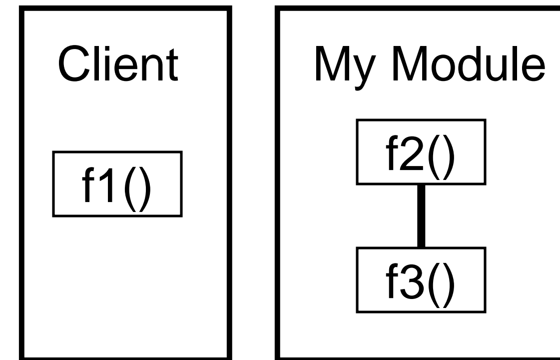
Ασθενής Εξάρτηση-Παράδειγμα

- Συντήρηση-εξάρτηση

Changed together often







- Εξάρτηση συντήρησης
- Δυνατή εξάρτηση στην συντήρηση

- Όχι αλλαγές
- Ασθενής εξάρτηση στην συντήρηση

Επιτυγχάνοντας Ασθενή Εξάρτηση

Μετακίνηση κώδικα

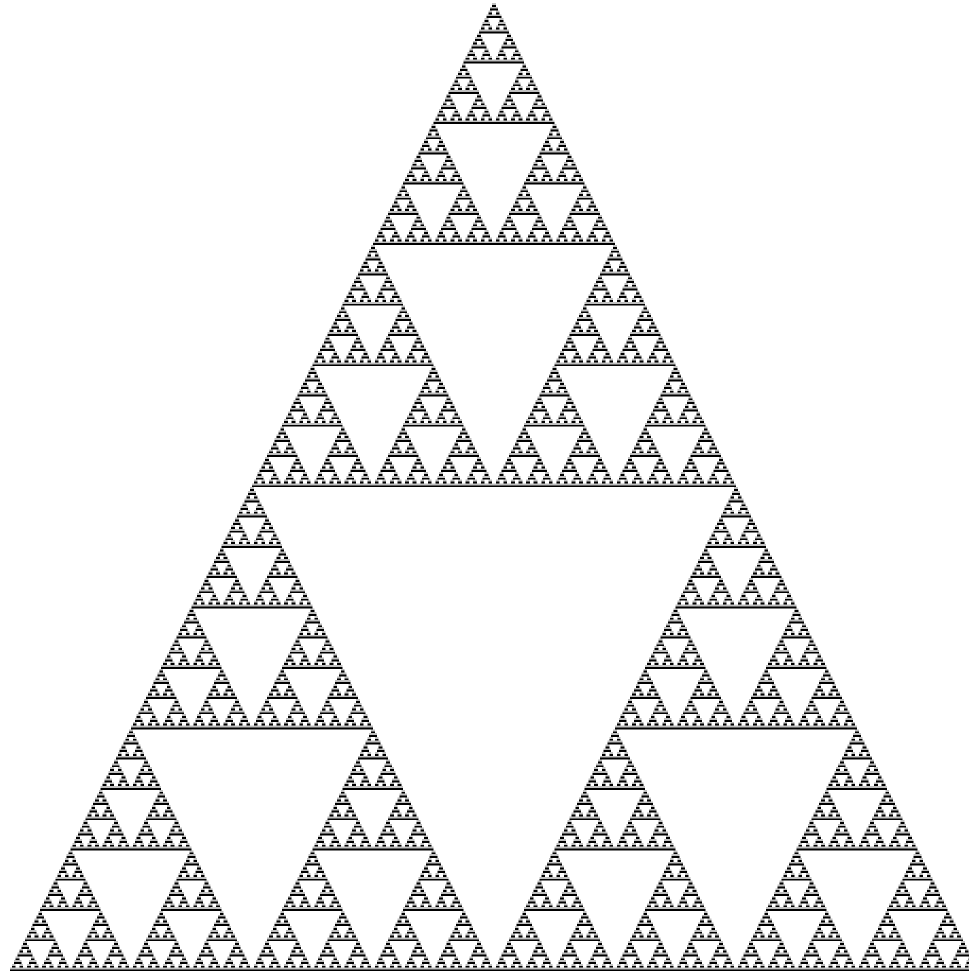
- Από πελάτες σε ενότητα
- Από ενότητα σε πελάτες
- Από πελάτες και ενότητες σε νέες ενότητες

Αναδρομή

Αναδρομή (Recursion)

- Πώς να λύσουμε ένα πρόβλημα κάνοντας λίγη δουλειά και ανάγοντας το υπόλοιπο να λυθεί με τον ίδιο τρόπο.
- Πού χρειάζεται;
  - Πολλές μαθηματικές συναρτήσεις ορίζονται αναδρομικά. Δεν είναι προφανές πώς μπορούν να οριστούν αλλιώς.
  - Πολλά προβλήματα λύνονται εύκολα αναδρομικά και δύσκολα μη αναδρομικά

Τρίγωνο Sierpinski
Μη αναδρομικός ορισμός;



Δύο παρεξηγήσεις

- «Είναι δύσκολο να κατανοηθεί»
  - Όχι, απλά απαιτείται εξάσκηση

- «Δεν είναι αποδοτική (χάσιμο χρόνου, χώρου)»
  - Κριτήριο η ευκολία για τον άνθρωπο, όχι την μηχανή.
  - Η αποδοτικότητα εξαρτάται από την επιλογή αλγορίθμου (καλή και κακή χρήση αναδρομικότητας). Θα δούμε δύο παραδείγματα.

Σκέψου Αναδρομικά

- Χωρίζουμε τη λύση σε τρία μέρη
  i. Κάνε κάποια δουλειά προς την λύση
  ii. Χρησιμοποίησε την μέθοδο για ένα (ή περισσότερα) μικρότερα υπο-προβλήματα σε ένα (ή περισσότερα) υποσύνολο δεδομένων
  iii. Ένωσε το i) και ii)

  Πρέπει να περιλαμβάνει μια τετριμμένη περίπτωση για το i), που να μην απαιτείται η ii) ώστε να σταματάει η αναδρομικότητα

Παράδειγμα: Παραγοντικό

- Αναδρομικός Ορισμός του n!, n>=0

  0! = 1

  n! = n * (n-1)!

- Αναδρομικός Ορισμός Υποπρογράμματος
  - «Λίγη δουλειά»: ένας πολλαπλασιασμός και μια αφαίρεση

    Βήμα Διακοπής (stopping case) το 0!

  -«Υποπρόβλημα»: Αναδρομικό Βήμα (recursive step) (n-1)!

  -Ενώνουμε τα δύο

Αναδρομικός Υπολογισμός του 3!

$3! = 3 * 2!$

$2! = 2 * 1!$

$1! = 1 * 0!$

$0! = 1$

Χρησιμοποιώντας την τιμή 1 του 0! (βήμα διακοπής) είναι δυνατός ο υπολογισμός του 3! επιστρέφοντας στον υπολογισμό του 1!, του 2! και τέλος του 3! όπως φαίνεται στο παρακάτω σχήμα

$3! = 3 * 2! = 3 * 2 = 6$

$2! = 2 * 1! = 2 * 1 = 2$

$1! = 1 * 0! = 1 * 1 = 1$

$0! = 1$ Βήμα Διακοπής

Ο αναδρομικός αυτός ορισμός μπορεί να υλοποιηθεί εύκολα όπως φαίνεται στο παρακάτω υποπρόγραμμα:

```
long par(int n)
{
        long    timi;
        if (n == 0)  /* βήμα διακοπής */
                timi = 1;
        else    /* αναδρομικό βήμα     n! = n * (n-1)!*/
                timi = n * par(n - 1);  // επιστροφή (A)
        return (timi);
}
```

Παρουσιάζεται ο τρόπος λειτουργίας του αναδρομικού υποπρογράμματος, όπου κάθε πλαίσιο σχετίζεται με την **εγγραφή ενεργοποίησης (activation record)**.

paragontiko = par(3);

Το n είναι 3
timi = 3* par(2)
επιστροφή

6

Το n είναι 2
timi = 2* par(1)
επιστροφή

Αποτέλεσμα 2

Το n είναι 1
timi = 1* par(0)
επιστροφή

Αποτέλεσμα 1

Το n είναι 0
timi = 1
επιστροφή

Αποτέλεσμα 1

Υλοποίηση Αναδρομής

```
{ /*κύριο πρόγραμμα*/
    x = par(3); /* program address B */
}
```

παράμετροι

Διεύθυνση επιστροφής συνάρτησης

3	B

1η κλήση της par(3) στο main (B)

2	A
3	B

2η κλήση της par(2) στην par (A)

3η και 4η κλήσεις της par(1) και par(0) στην par (A)

0	A
1	A
2	A
3	B

Τερματισμός 4ης κλήσης της par(0) στην par (A)

Επιστροφή τιμής 0!==1

Επιστροφή ροής προγράμματος στην διεύθυνση A

1	A
2	A
3	B

1	A

Τερματισμός $3^{ης}$ κλήσης της par(1) στην par (A)

Επιστροφή τιμής 1!==1*0!==1

Επιστροφή ροής προγράμματος στην διεύθυνση A

2	A
3	B

1	A

Τερματισμός $2^{ης}$ κλήσης της par(2) στην par (A)

Επιστροφή τιμής 1!==2*1!==2

Επιστροφή ροής προγράμματος στην διεύθυνση A

3	B

2	A

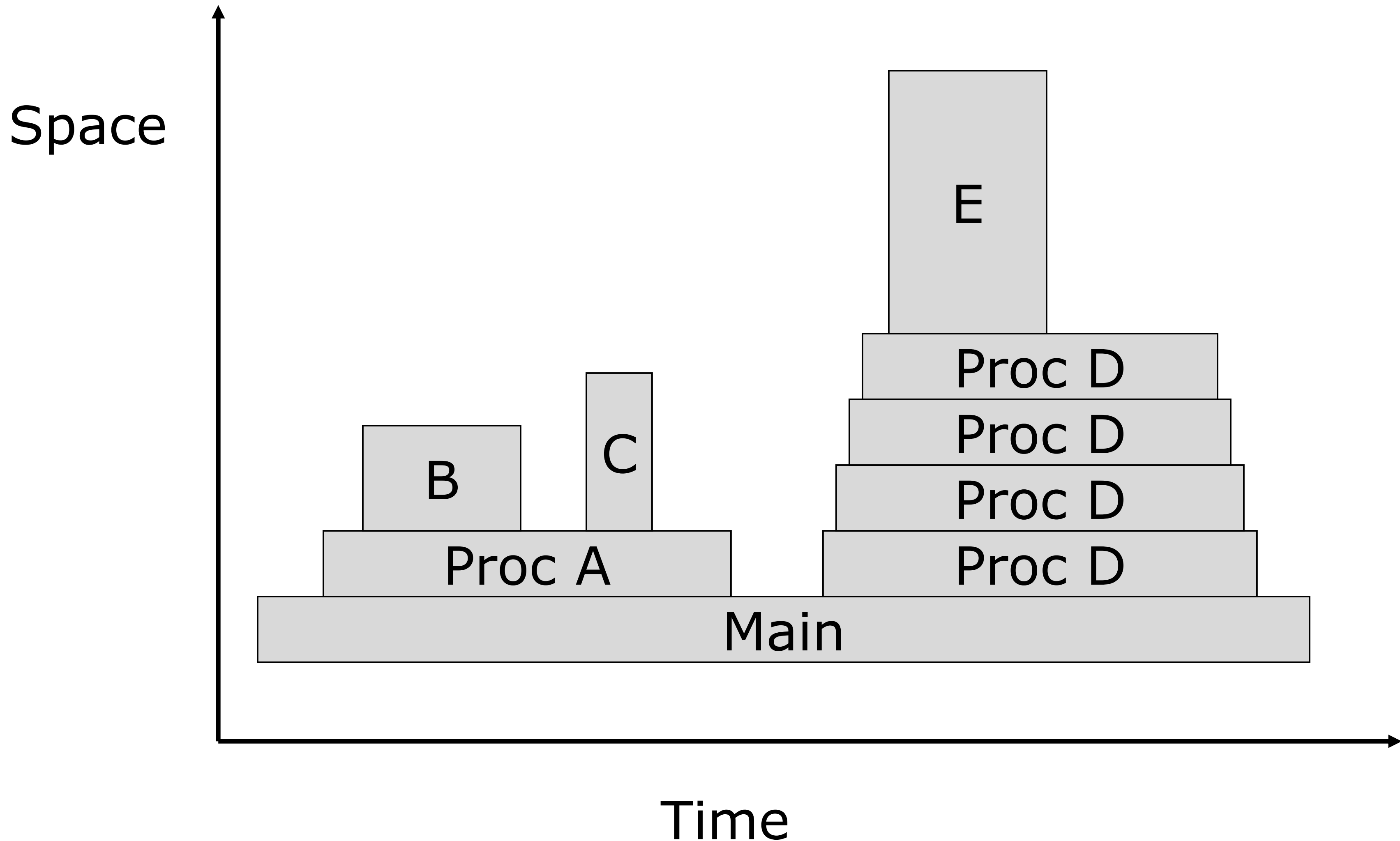
6	B

Τερματισμός $1^{ης}$ κλήσης της par(3) στην main (B)

Επιστροφή τιμής 3!==3*2!=6

Επιστροφή ροής προγράμματος στην διεύθυνση B

Οπτικοποίηση Χρόνου - Χώρου Συναρτήσεων
Χρόνος Κλήσεων, Διάρκεια Εκτέλεσης και Απαιτούμενη μνήμη)



Πολυπλοκότητα O(?) Παραγοντικού Αναδρομή-Επανάληψη Χρήση αναδρομικών συναρτήσεων

A. Με αναδρομή timi = n * par (n - 1) : Δυο κύριες πράξεις *, -

$T_A(n) = 2 + T(n - 1) =$

$2 + 2 + T(n - 2) =$

$2 + 2 + ... + 2 + T(0) =$

$2*n+1$

$T_A(n) = O(n)$

B. Με επανάληψη n! = 1*1 * 2 * 3 * .... * n, μια κύρια πράξη *

$T_E(n) = 1+1+...+1=(n+1)*1=n+1$

$T_E(n) = O(n)$

$T_A(n) = T_E(n) = O(n)$

Αριθμοί Fibonacci

0 1 1 2 3 5 8 13 21 . . . $Fib_n = Fib_{n-1} + Fib_{n-2}$

Αναδρομική Υλοποίηση

```
long  Fib (long n)
        {
                long    Fibnum;
                if (n <= 0)
                        Fibnum = 0;
                else if (n == 1)
                        Fibnum = 1;
                else
                        Fibnum = Fib (n - 1) + Fib (n - 2);

                return (Fibnum);
        }
```

Πολυπλοκότητα Αναδρομικής Υλοποίησης

Fibnum = Fib (n - 1) + Fib (n - 2); **3 βασικές πράξεις -, +, -**

$T(n) = 3 + T(n - 1) + T(n - 2) =$
$3 + [3 + T(n - 2) + T(n - 3)] + T(n - 2) =$
$6 + 2T(n-2) + T(n - 3)$, για $n>=3$

$T(n) >= 2T(n - 2) >= 2^2 T(n - 4) >= \ldots >= 2^{n/2} T(0)$, αν n άρτιο
ή
$T(n) >= 2T(n - 2) >= 2^2 T(n - 4) >= \ldots >= 2^{(n-1)/2} T(1)$, αν n περιττό

Αλλά $T(0) = T(1) = 1$, συνεπώς

$$T_A(n) = O(2^{n/2}) \quad \text{για όλα τα } n >=2$$

Αναδρομικές Συναρτήσεις – Αναλυτικές Λύσεις
(μάθημα Πολυπλοκότητα)

```
long eFib (long n)
 /*        Επαναληπτικό υποπρόγραμμα για τον υπολογισμό του n-ιοστού
                     αριθμού Fibonacci*/
{
          long      Fib1, Fib2, Fib3, i;

          Fib1 = 0;
          Fib2 = 1;
          for (i = 3;i<=n;i++)
          {
                    Fib3 = Fib1 + Fib2;
                    Fib1 = Fib2;
                    Fib2 = Fib3;
          }
          return(Fib2);
}
```

$\mathbf{T_E = 1+1+1+\ldots+1 = 1*(n+1) = n+1 = O(n)}$

The Ackermann function

Η συνάρτηση Ackermann ορίζεται αναδρομικά για μη-αρνητικούς ακέραιους m, n ως εξής

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

Η υλοποίηση απλή

```
long ack(int m, int n)
{       if (m == 0)
                return (n+1) ;
        else if (n = =0)
                return ack(m-1, 1) ;
        else
                return ack(m-1, ack(m, n-1)) ;
}
```

... Αλλά η συμπεριφορά της

A(1, 2) = A(0, A(1,1)) =
A(0, A(0, A(1,0))) =
A(0, A(0, A(0,1))) =
A(0, A(0, 2)) =
A(0, 3) =
4

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3\text{ twos}} - 3$$

A(4, 3)
= A(3, A(4, 2))
= A(3, A(3, A(4, 1)))
= A(3, A(3, A(3, A(4, 0))))
= A(3, A(3, A(3, A(3, 1))))
= A(3, A(3, A(3, A(2, A(3, 0)))))
= A(3, A(3, A(3, A(2, A(2, 1)))))
= A(3, A(3, A(3, A(2, A(1, A(2, 0))))))
= A(3, A(3, A(3, A(2, A(1, A(1, 1))))))
= A(3, A(3, A(3, A(2, A(1, A(0, A(1, 0)))))))
= A(3, A(3, A(3, A(2, A(1, A(0, A(0, 1)))))))
= A(3, A(3, A(3, A(2, A(1, A(0, 2))))))
= A(3, A(3, A(3, A(2, A(1, 3)))))
= A(3, A(3, A(3, A(2, A(0, A(1, 2))))))
= A(3, A(3, A(3, A(2, A(0, A(0, A(1, 1)))))))
= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(1, 0))))))))
= A(3, A(3, A(3, A(2, A(0, A(0, A(0, A(0, 1))))))))
= A(3, A(3, A(3, A(2, A(0, A(0, A(0, 2)))))))
= A(3, A(3, A(3, A(2, A(0, A(0, 3))))))
= A(3, A(3, A(3, A(2, A(0, 4)))))
= A(3, A(3, A(3, A(2, 5)))) = ...
= A(3, A(3, A(3, 13))) = ...
= A(3, A(3, 65533))
= ...

A(3, 65533) επιστρέφει $2^{65536} - 3$, αριθμός μεγαλύτερος από τον αριθμό των ατόμων σε όλο τον ορατό κόσμο.
Κατόπιν, αυτός ο αριθμός *χρησιμοποιείται ως δύναμη* του 2 για το τελικό αποτέλεσμα.

Πιθανές Παγίδες

- Λάθος συνθήκη τερματισμού (do … while) Ατέρμων Βρόχος
- Συνεχής Αναδρομή
  - Δεν γίνεται Έλεγχος τερματισμού
  - Χρησιμοποιούμε αναδρομή με μη αποδεκτές παραμέτρους

Μη τερματίζουσα Αναδρομή

- Δεν ικανοποιείται η συνθήκη τερματισμού
  - Κλήση par(-1) ενώ ελέγχουμε (n==0)
  - Συμπτώματα: συνεχείς κλήσεις έως ότου εξαντληθεί η μνήμη
- Δεν αλλάζουν οι παράμετροι της αναδρομής, N=f(N). Δεν υπάρχει πρόοδος.

Αφαίρεση της Γραμμικής Αναδρομής
(αναδρομή με μια μόνο αναδρομική κλήση)
Με χρήση Στοίβας

```
void grammiki_anadromi (long n)
{
        if (συνθήκη (n))
                μη αναδρομική περίπτωση (n);
        else
        {
                προηγούμενες πράξεις (n);
                grammiki_anadromi (F(n));
                μετέπειτα πράξεις (n);
        }
}
```

```
void epanaliptiki (long n)
{         typos_stoiva   stoiva;
          dimiourgia(stoiva);

          while (!συνθήκη (n))
          {         προηγούμενες πράξεις (n);
                    othisi (stoiva, n);
                    n = F(n);
          }

          μη αναδρομική περίπτωση (n);
          while (!keni(stoiva))
          {
                    exagogi(stoiva, &n);
                    μετέπειτα πράξεις (n);
          }
}
```

Τέλος Ενότητας

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στο πλαίσιο του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Αθηνών**» έχει χρηματοδοτήσει μόνο την αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Σημειώματα

Σημείωμα Αναφοράς

Copyright Εθνικόν και Καποδιστριακόν Πανεπιστήμιον Αθηνών, Κοτρώνης Ιωάννης. «Δομές Δεδομένων και Τεχνικές Προγραμματισμού. Ενότητα 5: Τεχνικές Προγραμματισμού». Έκδοση: 1.01. Αθήνα 2015.

Διαθέσιμο από τη δικτυακή διεύθυνση: http://opencourses.uoa.gr/courses/DI105/.

Σημείωμα Αδειοδότησης

Το παρόν υλικό διατίθεται με τους όρους της άδειας χρήσης Creative Commons Αναφορά, Μη Εμπορική Χρήση Παρόμοια Διανομή 4.0 [1] ή μεταγενέστερη, Διεθνής Έκδοση. Εξαιρούνται τα αυτοτελή έργα τρίτων π.χ. φωτογραφίες, διαγράμματα κ.λ.π., τα οποία εμπεριέχονται σε αυτό και τα οποία αναφέρονται μαζί με τους όρους χρήσης τους στο «Σημείωμα Χρήσης Έργων Τρίτων».

[1] http://creativecommons.org/licenses/by-nc-sa/4.0/

Ως **Μη Εμπορική** ορίζεται η χρήση:

- που δεν περιλαμβάνει άμεσο ή έμμεσο οικονομικό όφελος από την χρήση του έργου, για το διανομέα του έργου και αδειοδόχο
- που δεν περιλαμβάνει οικονομική συναλλαγή ως προϋπόθεση για τη χρήση ή πρόσβαση στο έργο
- που δεν προσπορίζει στο διανομέα του έργου και αδειοδόχο έμμεσο οικονομικό όφελος (π.χ. διαφημίσεις) από την προβολή του έργου σε διαδικτυακό τόπο

Ο δικαιούχος μπορεί να παρέχει στον αδειοδόχο ξεχωριστή άδεια να χρησιμοποιεί το έργο για εμπορική χρήση, εφόσον αυτό του ζητηθεί.

Διατήρηση Σημειωμάτων

Οποιαδήποτε αναπαραγωγή ή διασκευή του υλικού θα πρέπει να συμπεριλαμβάνει:

- το Σημείωμα Αναφοράς
- το Σημείωμα Αδειοδότησης
- τη δήλωση Διατήρησης Σημειωμάτων
- το Σημείωμα Χρήσης Έργων Τρίτων (εφόσον υπάρχει)

μαζί με τους συνοδευόμενους υπερσυνδέσμους.