# HY255 Εργαστήριο Λογισμικού – L15

## Debugging

HY255 Εργαστήριο Λογισμικού
Άνοιξη 2021
Άγγελος Μπίλας

# Debugging 1/2: Γράψτε το πρόγραμμα σας με στόχο να μην χρειαστεί

- Η διαδικασία εύρεσης λαθών σε ένα σύστημα
- Αρχές / βήματα στο debugging
- **(1) Κατανόηση και σαφήνεια**
- Καταλάβετε τον κώδικά σας όταν σχεδιάζετε/γράφετε το πρόγραμμά και όχι εκ των υστέρων όταν το κάνετε debug
  - Με άλλα λόγια, ΜΗΝ ΓΡΑΦΕΤΕ ΚΩΔΙΚΑ ΠΟΥ ΔΕΝ ΕΙΣΤΕ ΣΙΓΟΥΡΟΙ ΤΙ ΚΑΝΕΙ ΜΕ ΤΟ ΣΚΕΠΤΙΚΟ ΌΤΙ ΘΑ ΤΟΝ ΔΙΟΡΘΩΣΕΤΕ ΑΡΓΟΤΕΡΑ, κατά το debugging
- Οι κύριοι μηχανισμοί που έχουμε για να το πετύχουμε αυτό (με σειρά αυξανόμενων απαιτήσεων από τρίτους) είναι
  - (a) Σκεφτείτε τι κάνει ο κώδικας που γράφετε
  - (b) Διαβάστε (κάντε review) τον κώδικά σας offline
  - (c) Εξηγήστε τον σχεδιασμό ή/και τον κώδικά σας σε κάποιο τρίτο
  - (d) Δώστε σε κάποιον τρίτο να διαβάσει (κάνει review) τον κώδικά σας offline
- Μην αναβάλετε το (1) για αργότερα – τα πράγματα μόνο πιο πολύπλοκα θα γίνουν

# Debugging 2/2: Αν κάτι δεν είναι σωστό…

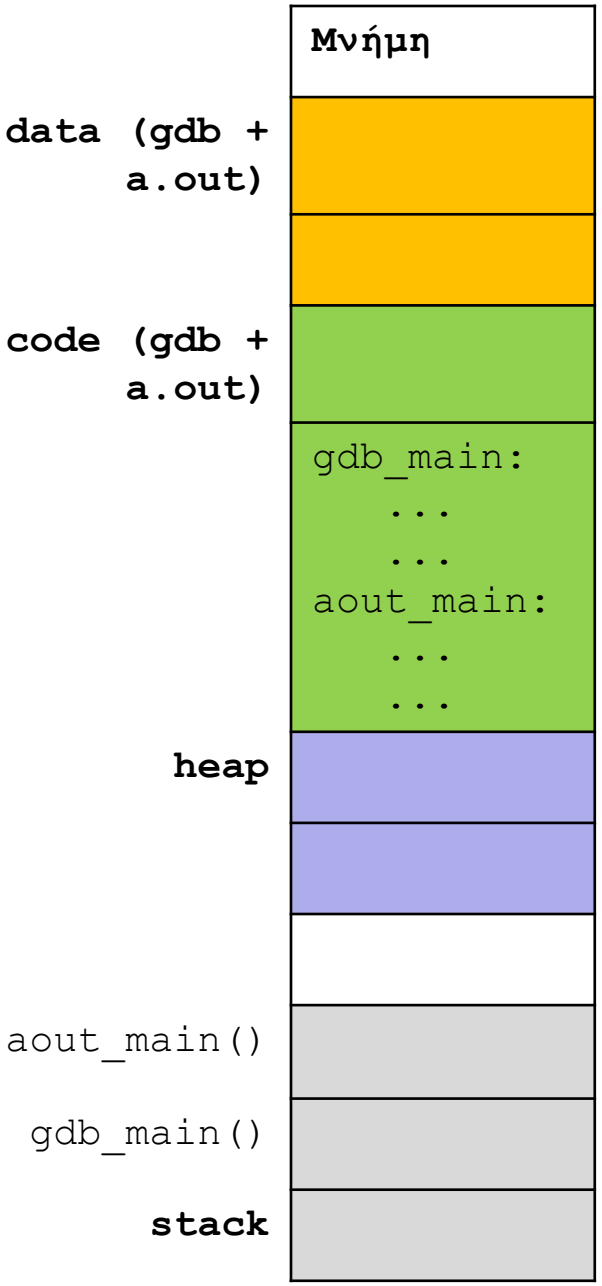
- **(2) Reproducibility**
  - Αφού γράψαμε τον κώδικα με σαφήνεια, αν κάτι δεν εκτελείται σωστά τότε
  - (a) Κάνουμε το λάθος reproducible: βρίσκουμε το input/configuration που οδηγεί πάντα στο λάθος αυτό
  - (b) Στη συνέχεια, κάνουμε το λάθος να εμφανίζεται όσο πιο γρήγορα γίνεται: βρίσκουμε το μικρότερο δυνατό input/configuration που προκαλεί το (a) όσο πιο γρήγορα γίνεται
- **(3) Assertions**
  - Στο σημείο αυτό θεωρούμε ότι έχουμε σχεδιάσει/υλοποιήσει το πρόγραμμά μας σωστά (βήμα 1) αλλά έχουμε ένα reproducible λάθος (βήμα 2)
  - (α) Προσθέτουμε επιπλέον assertions ώστε να ελέγξουμε τι πρέπει να ισχύει και να βρούμε το κομμάτι του προγράμματος που είναι προβληματικό
  - (β) Αν κάποιο από τα assertions κάνει fail, τότε μάλλον έχουμε μια ένδειξη του τι δεν γίνεται σωστά
- **(4) Tracing**
  - Αν τα assertions δεν μας δείξουν κάτι, τότε υπάρχει κάπου λάθος κατανόηση του τι κάνει ο κώδικας και τι ισχύει στο πρόγραμμά μας – οπότε τυπώνουμε (printf) σε διάφορα σημεία την κατάσταση (μεταβλητές, ροή, etc) ώστε να διαπιστώσουμε τι είναι αυτό που δεν έχουμε κατανοήσει ή εκφράσει σωστά
  - Στο tracing, μερικές φορές μας βολεύει να χρησιμοποιήσουμε έναν debugger
  - Σε όλη αυτή την διαδικασία, ο debugger είναι ένα εργαλείο που μας βοηθάει μόνο στο βήμα (4)
- Σε όλη τη διαδικασία του debugging, είναι σημαντικό να κρατάμε σημειώσεις από κάθε πείραμα που δοκιμάζουμε και τα αποτελέσματα γιατί μετά από λίγο αυξάνεται πολύ ο αριθμός τους και είναι δύσκολο να θυμόμαστε όλα τα σενάρια που εξετάσαμε

# Debuggers - gdb

- Ο debugger είναι ένα πρόγραμμα που διαβάζει τη μνήμη του προγράμματός μας και μπορεί να κάνει τα εξής:
- (a) Να σταματήσει την εκτέλεση του προγράμματος μας σε συγκεκριμένα σημεία
- (b) Να μας δείξει περιεχόμενα μεταβλητών, θέσεων μνήμης, καταχωρητών
- (c) Να μας δείξει την εικόνα της στοίβας και την ακολουθία κλήσης των συναρτήσεών μας
- (d) Να μας δείξει τον κώδικα μας στη μορφή που έχει στη μνήμη κατά την εκτέλεσή του (code segment)

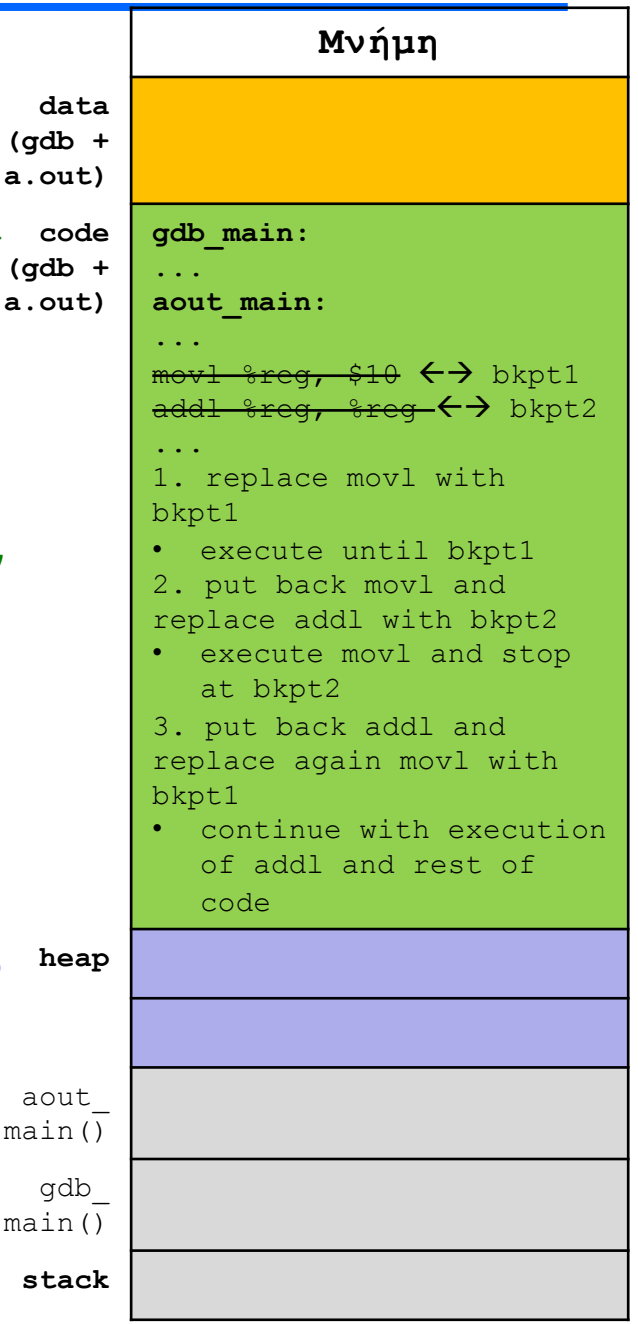
# Πως οι debuggers βλέπουν την μνήμη του προγράμματός μας;

- Για την δική μας περίπτωση, ας απλοποιήσουμε την πραγματικότητα και ας πούμε ότι ο gdb είναι ένα πρόγραμμα που διαβάζει το δικό μας πρόγραμμα σαν input και επομένως ο debugger βρίσκεται στο ίδιο address space με το πρόγραμμα
- Όταν ζητήσουμε τον gdb να εκτελέσει το πρόγραμμά μας, τότε απλά ξεκινά την εκτέλεση της main
- Στην πραγματικότητα ο debugger θέλουμε να είναι ένα χωριστό πρόγραμμα για να μην μπορεί εύκολα να επηρεαστεί από λάθη στο δικό μας πρόγραμμα
- Οπότε για να διαβάσει και να αλλάξει πράγματα στη μνήμη του προγράμματός μας χρησιμοποιεί την βοήθεια του λειτουργικού συστήματος με ένα ειδικό (system) call που ονομάζεται ptrace



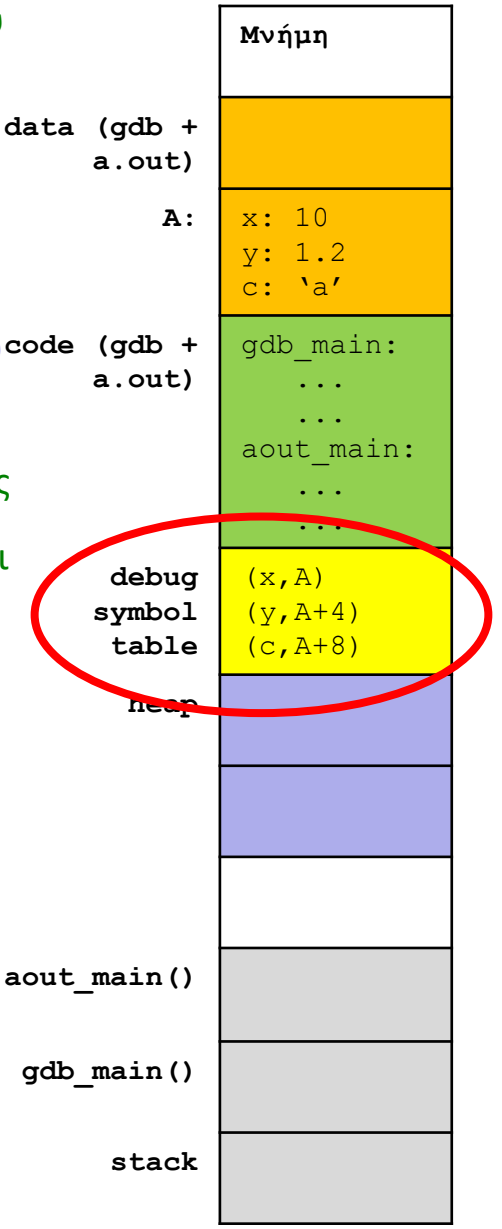
# breakpoints

- Breakpoint είναι ένα σημείο του προγράμματος (μια διεύθυνση στο code segement) στο οποίο θέλουμε να σταματήσει η εκτέλεση του προγράμματος την επόμενη φορά που θα περάσει από το σημείο αυτό
  - Τα breakpoints μπορεί να τα ορίζει ο χρήστης κατά βούληση
- Πως ο debugger σταματά την εκτέλεση του προγράμματος σε μια οποιαδήποτε εντολή;
- Σε κάθε επεξεργαστή, υπάρχει μια ειδική εντολή, ας την ονομάσουμε BKPT που όταν εκτελεστεί αλλάζει την ροή του προγράμματος σε ένα προκαθορισμένο σημείο (handler), τον κώδικα του debugger
  - Κατόπιν, ο debugger μας δείχνει το prompt και μας δίνει την δυνατότητα να εισάγουμε εντολές
- Όταν ορίσουμε ένα breakpoint σε κάποιο σημείο του κώδικά μας, ο debugger αντικαθιστά την εντολή που βρίσκεται σε εκείνο το σημείο με την BKPT1 και εξασφαλίζει έτσι ότι αν το πρόγραμμα περάσει από αυτό το σημείο θα οδηγηθεί στον κώδικα του debugger → (1)
- Τι γίνεται όμως με την εντολή του προγράμματος που αντικαταστάθηκε από την BKPT1;
- Θα πρέπει να εκτελεστεί και αυτή για να έχουμε μια σωστή εκτέλεση του προγράμματος
- Για να το πετύχει αυτό ο debugger χρησιμοποιεί είναι επιπλέον (εσωτερικό) breakpoint που δεν το βλέπουμε εμείς, το οποίο τοποθετεί στην επόμενη εντολή, σαν BKPT2 → (2)
- Στη συνέχεια επαναφέρει την αρχική εντολή στο BKPT1, την εκτελεί και σταματά (κατά ανάγκη) στο BKPT2
- Τότε, επαναφέρει το BKPT1 ενώ αφαιρεί το εσωτερικό BKPT2 και το αντικαθιστά με την αρχική εντολή στη θέση αυτή → (3)
- Με αυτό τον τρόπο μπορεί να τοποθετεί BKPTs σε οποιοδήποτε σημείο του code segment
- Την θέση ενός breakpoint μπορούμε να την ορίσουμε κυρίως με δύο τρόπους
  - Απ'ευθείας σε μια διεύθυνση μνήμης (gdb> b 0xabce0000 → breakpoint στη διεύθυνση 0xabce0000)
  - Μέσω μια γραμμής του source προγράμματός μας (gdb> b main.c:14 → breakpoint στη γραμμή 14 του αρχείου main.c)
  - Σε κάθε περίπτωση ο debugger βρίσκει την θέση μνήμης που αντιστοιχεί στο BKPT και εισάγει στο σημείο αυτό την εντολή BKPT
- Εντολές που χρησιμοποιούμε συχνά με breakbpoints για να εκτελέσουμε μια ή περισσότερες εντολές μετά το breakpoint είναι οι:
  - continue: συνέχισε μέχρι το επόμενο breakpoint
  - next, step: εκτέλεσε μια εντολή (statement) του C προγράμματος περνώντας τις συναρτήσεις (next) ή μπαίνοντας μέσα (step)
  - nexti, stepi: όπως και οι next,step αλλά για εντολές assembly



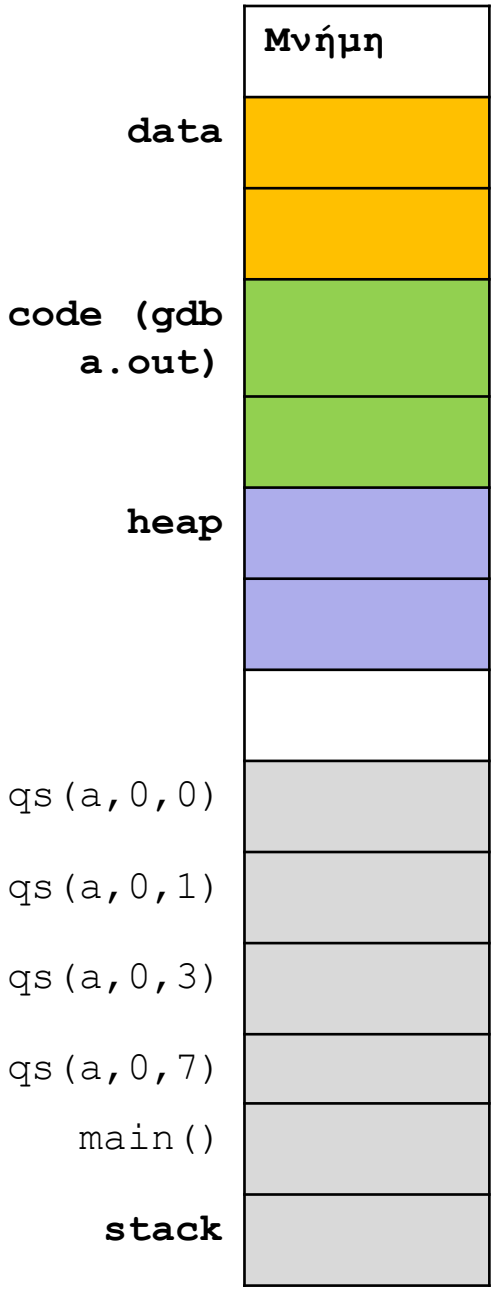
# Περιεχόμενα Μεταβλητών/Μνήμης

- Δεδομένου ότι ο debugger βλέπει την μνήμη του προγράμματός μας, μπορούμε να του ζητήσουμε να τυπώσει τα περιεχόμενα οποιασδήποτε διεύθυνσης μνήμης, σαν να ήταν το ίδιο το πρόγραμμα, π.χ.
  - Η εντολή print μας δείχνει την τιμή μιας έκφρασης που μπορεί να περιέχει μεταβλητές
  - gdb> print x
  - gdb> print a[i]
  - Και οποιαδήποτε άλλη έκφραση της C που γνωρίζει το πρόγραμμά μας
  - Η εντολή examine (x) μας δείχνει τα περιεχόμενα μιας διεύθυνσης: x/count format address
  - gdb> x/10b 0xabcd0000 → δείξε 10 bytes ξεκινώντας από την διεύθυνση μνήμης 0xabcd0000
- Πως γνωρίζει ο debugger τα ονόματα των μεταβλητών (τύπων, κ.ο.κ.) του προγράμματός μας;
- Το a.out περιέχει έναν πίνακα συμβόλων (ονομάτων), symbol table, που περιέχει αυτές τις πληροφορίες. Π.χ. για την μεταβλητή x περιέχει την διεύθυνση A που βρίσκεται η x. Οπότε όταν εμείς ζητήσουμε από τον gdb να τυπώσει το x, βρίσκει την διεύθυνση A και τυπώνει τα περιεχόμενά της
- Αυτός ο πίνακας συμβόλων χρησιμοποιείται για λόγους debugging και δεν είναι απαραίτητος για την εκτέλεση του a.out
- Για τον λόγο αυτό, προστίθεται στο a.out μόνο αν πούμε στον compiler ότι θέλουμε να κάνουμε debug το πρόγραμμά μας, με την παράμετρο "–g" στη φάση της μετάφρασης
  - Π.χ. gcc –ansi –Wall –pedantic –g main.c
- Αν δεν χρησιμοποιήσουμε το –g πάλι μπορούμε να κάνουμε debug το πρόγραμμά μας, αλλά δεν μπορούμε να χρησιμοποιήσουμε ονόματα μεταβλητών, παρά μόνο διευθύνσεις που κάνει την διαδικασία πιο άβολη
- Επιπλέον ο gdb μας δίνει τη δυνατότητα να τυπώσουμε και τιμές καταχωρητών (θα δούμε την επόμενη φορά ποιοι είναι οι καταχωρητές των συστημάτων x86)
  - Πως μας δείχνει ο debugger τις τιμές των καταχωρητών την στιγμή που διακόπηκε το πρόγραμμα (νωρίτερα) ενώ τώρα τρέχει ο debugger?
- Επίσης, εκτός από το να δούμε τα περιεχόμενα μνήμης/μεταβλητών, μπορούμε και να τα αλλάξουμε (εντολή set, π.χ. set x=3)



# stack

- Ο debugger μας επιτρέπει να δούμε την μορφή της στοίβας σε ότι αφορά
  - Την ακολουθία των κλήσεων μέχρι το τρέχον σημείο του προγράμματος
  - Τις τιμές των τοπικών μεταβλητών σε οποιοδήποτε stack frame στην ακολουθία κλήσεων
- gdb> backtrace (bt)
  - Δείχνει την ακολουθία κλήσεων από την main μέχρι το τρέχον σημείο
  - Ουσιαστικά αυτό είναι το path του δέντρου των δυναμικών κλήσεων του προγράμματος από την main (ρίζα) μέχρι το τρέχον breakpoint
  - > bt
    qs(a,0,0)
    qs(a,0,1)
    qs(a,0,3)
    qs(q,0,7)
    main()
  - Μπορούμε να δούμε με τον debugger ένα προηγούμενο path του δέντρου αυτού; Όχι με τους απλούς/συνηθισμένους debuggers. Πιο προχωρημένα εργαλεία μπορούν να το κάνουν και αυτό αλλά δυστυχώς με πολύ μεγάλο (impractical) κόστος σε μνήμη και χρόνο.
- gdb> up/down
  - Οι εντολές αυτές μας επιτρέπουν να μετακινηθούμε ένα frame πάνω/κάτω στη στοίβα και να βρεθούμε στην κλήση της αντίστοιχης συνάρτησης, όπου και μπορούμε να δούμε τις τιμές των αντίστοιχων μεταβλητών (print/examine)
- Πως γνωρίζει ο debugger που ξεκινά και τελειώνει το κάθε stack frame;
  - Το μέγεθος του stack frame κάθε συνάρτησης είναι μέρος της πληροφορίας που αποθηκεύεται στον symbol table του a.out μαζί με κάθε συνάρτηση



# Machine/assembly code

- Το πρόγραμμα a.out που εκτελείτε βρίσκεται στην μνήμη, στο code segment, σε μορφή machine code
- Ο debugger μπορεί να μας δείξει με την εντολή examine (x) και τα περιεχόμενα αυτής της μνήμης (του προγράμματος)

```
gdb> x/100i main
0x804857b <main>:    lea    0x4(%esp),%ecx
0x804857f <main+4>:  and    $0xfffffff0,%esp
0x8048582 <main+7>:  pushl  -0x4(%ecx)
0x8048585 <main+10>: push   %ebp
0x8048586 <main+11>: mov    %esp,%ebp
0x8048588 <main+13>: push   %ecx
0x8048589 <main+14>: sub    $0x14,%esp
0x804858c <main+17>: movl   $0x0,-0xc(%ebp)
0x8048593 <main+24>: sub    $0x8,%esp
0x8048596 <main+27>: push   $0x1
```

  - Η εντολή list, π.χ. gdb> list main.c μας δείχνει τον source κωδικά μας από τα αρχεία στο δίσκο και όχι το πρόγραμμα όπως εκτελείται στη μνήμη
- Ένα συνώνυμο της εντολής x/_i addr είναι η εντολή disassemble addr

# Παρατηρήσεις για τους debuggers

- Χρησιμοποιούμε τον debugger για να εξετάσουμε κάτι που θεωρούμε ότι δεν πρέπει να συμβαίνει
  - Όχι για να καταλάβουμε τι κάνει το πρόγραμμά μας
  - Όταν χρησιμοποιούμε τον debugger πρέπει να έχουμε ένα συγκεκριμένο σενάριο κατά νου που θα έπρεπε να συμβαίνει ή όχι
  - Για να φτάσουμε σε αυτό το σενάριο θα πρέπει να έχουμε αρκετά assertions στο πρόγραμμα μας
- Οι debuggers δυστυχώς παρ΄ότι είναι σημαντική βοήθεια, είναι εξαιρετικά περιορισμένοι
  - Δεν μπορούν να μας μεταφέρουν πίσω στο χρόνο, σε όποιο σημείο της εκτέλεσης θέλουμε (μόνο σε συναρτήσεις που έχουν ακόμη κάποιο active stack frame)
  - Δεν μπορούν να μας βοηθήσουν πολύ με παράλληλα/concurrent/distributed προγράμματα
  - Δεν μπορούν να μας βοηθήσουν πολύ με κώδικα μέσα στο λειτουργικό σύστημα (αν και τα virtual machines έχουν βοηθήσει σε αυτό το σημείο τα τελευταία χρόνια)

# Reading

- Appendix F of Programming from the Ground Up, Jonathan Bartlett 2004
- GDB Internals: Breakpoint handling
- Chapter 14, (online) Programming with GNU Software. Gary V. Vaughan, Akim Demaille, Paul Scott, Bruce Korb, and Richard Meeking. Edition 2, 2002. [pdf]
- Everything about gdb: https://sourceware.org/gdb/documentation