13/2/2006 (1) bilas@csd.uoc.gr | HY-255

1) Διαλέξεις 11) Ασκήσεις (6)
3) Πρόοδος και τελικό

- Introduction
- Review of C concepts (variables & types)
- Character I/O in C

- Σημειώσεις - Πληροφορίες για κάθε άσκηση
- C Programming: a modern approach King

Στόχος του μαθήματος είναι να καταναόσουμε πως πρέπει να γράφουμε προσεγμένα προγράμματα, και στο βαθμό που αυτό είναι δυνατό, ανεξάρτητα από τη γλώσσα και το σύστημα. Χρειαζόμαστε όλο κ περισσότερα "προγράμματα" και δεν υπάρχει "αυτόματος" τρόπος να γράφουμε προγράμματα. (π.χ άλλο προσε) Θέματα a) σχεδιασμού προγραμμάτων (π.χ. πως γράφουμε κώδικα, χωρίζουμε σε πολλά αρχεία) β) αντιστοιχία των μηχανισμών γλωσσών προγρ. με τους μηχανισμούς που παρέχουν τα συστήματα γ) εργαλεία που χρησιμοποιούμε σε αυτή τη διαδικασία.

Γλώσσα Προγραμματισμού C. Κάθε υπολογιστής έχει επεξεργαστή, μνήμη, περιφερειακά. Καθε γλώσσα προγραμματισμού για διευκολύνει στο χειρισμό αυτών των resources παρέχει αντίστοιχες δομές. a) Μεταβλητές / Variables που αναπαριστούν/αντιστοιχούν στις μνήμη του υπολογιστή. b) Εντολές / Statements -"- στις πράξεις που μπορεί να εκτελεί κάποιος επεξεργαστής c) Βιβλιοθήκες που χρησιμοποιούμε για να χειριστούμε περιφερειακές συσκευές.

a) Variables : 1) απλές μεταβλητές απλού τύπου μεταβλητών / basic data types

τύπος κάθε μεταβλητής ορίζει το μέγεθος της μεταβλητής στη μνήμη (δηλ πόσα bytes είναι) και τις "πράξεις" που επιτρέπεται να συμμετέχει η μεταβλητή.	int	short	char	float	double	long
	4 bytes (συνήθως)	2 bytes (συνήθως)	1 byte (πάντα)	4 bytes	8 bytes	συνήθως 4/8 bytes
	αριθμητικές πράξεις σύγκριση			αριθμητικές πράξεις "fixed point"		

*Κάθε ένας από αυτούς τους τύπους έχει signed κ unsigned εκδόσεις

2) Σύνθετοι τύποι μεταβλητών - array π.χ. int a[10], το "a" είναι ένα array, μιας διάστασης και αποτελείται από 10 στοιχεία που το κάθε ένα είναι τύπου int.
• το μέγεθος του "a" είναι 40 bytes • πράξεις : indexing με τον τελεστή [ ]
Επίσης έχουμε και πίνακες πολλαπλών διαστάσεων * Στη C τα indices ξεκινούν πάντα από 0.

- Structures Μια μεταβλητή τύπου struct είναι μια συλλογή μεταβλητών άλλων τύπων π.χ. ο τύπος struct coord { int x, int y }; struct coord c1; • το μέγεθος είναι το άθροισμα των μεγεθών των επιμέρους μεταβλητών • πράξεις : πρόσβαση στα πεδία "." c1.x π.χ.
- Enumerated types - ένας τύπος enum είναι ένα σύνολο από σταθερές τιμές. π.χ. enum color_t { red, black, white }; πράξεις : π.χ. σύγκριση, ανάθεση
enum color_t c; c = red; ... = c; αντί του απλού int c; c = 3; /* 3 = red */
μας βοηθούν να εκφράσουμε με ακρίβεια τι θέλουμε (μειωμένη πιθανότητα λαθών)
* Επίσης υπάρχουν τα unions, pointers

b) Statements 1) assignment 2) αλλαγής ροής του προγράμματος (if then else, switch case, for, while)
+ Operators 1) αριθμητικούς : +, *, /, -, % 2) λογικούς : &&, ||, 3) σύγκρισης ==, !=, >, <
4) indexing : [] 5) πεδία struct: "." 6) τέλος εντολής : ";"

c) Input / Output σε περιφερειακά. Στη C όλα τα περιφερειακά φαίνονται ως μια ακολουθία χαρακτήρων (char) την οποία μπορούμε να γράφουμε/διαβάσουμε και η οποία τελειώνει με έναν ειδικό χαρακτήρα. (stdout • stdin)
Όταν ένα πρόγραμμα ξεκινάει να εκτελείται έχει πάντα ορισμένα 3 streams : stdout, stdin, stderr. Παράδειγμα : ⟶

```
#include <stdio.h>
int main(void) {
 fprintf(stdout, "This is a test \n");
 exit(0);
}
```

fprintf, fscanf: συνώνυμα/διαβάζουν πολλούς χαρακτήρες από κάποιο stream

fputc, fgetc: συνώνυμα/διαβάζουν ένα μόνο χαρακτήρα.

π.χ. `char c = 'a';`

`fputc(c, stdout); fputc('b', stdout); c = fgetc(stdin);`

Για να ξέρει ένα πρόγραμμα πότε τελειώνει ένα stream, ο τελευταίος χαρακτήρας του stream είναι πάντα ένας ειδικός χαρακτήρας που ονομάζεται end-of-stream ή end-of-file (EOF) ορισμένος στη γλώσσα.

#Θέλουμε να γράψουμε πρόγραμμα που τυπώνει το μέγεθος των ειδών μεταβλητών της C σε κάποιο σύστημα.

```
#include <stdio.h>
int main(void){
    printf("Bytes per char: %d\n", sizeof(char));
    printf("Bytes per int: %d \n", sizeof(int));
    ⋮
                ... coord ... , sizeof(struct coord));
}
```

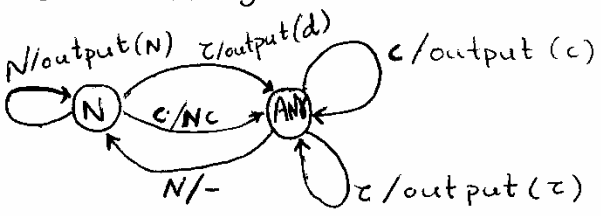
ή `int i; sizeof(i);`

Η C διαθέτει μεταβλητές global: είναι ορατές από οποιοδήποτε σημείο του προγράμματος
local: δηλώνονται σε συναρτήσεις και είναι ορατές μόνο από τις εντολές των συναρτήσεων. δυναμικές/η μεταβλητές / μνήμη: ———

Άσκηση 1 - Γράφουμε ένα πρόγραμμα "translate" το οποίο παίρνει input σε ελληνικούς χαρακτήρες (iso-8859-7) κ το γράφει στο stdout σε αγγλικούς χαρακτήρες (iso 8859-1). 1) αντιστοιχίες 1→1 π.χ. ν→n, β→v
2) ά → a' 1→2 3) ντ→d : 2→1, `translate < test.in > test.out`

`main(void){ while ((c=getchar())!EOF) { output(c); } return 0; }`

FSM



input: N, τ, c

state \ input	N	τ	C
ANY	-/N	τ/ANY	C/ANY
N	N/N	d/ANY	Nc/ANY

```
int main (void) {

    state = ANY;
    while ((c=getchar()) != EOF) {

        switch (state) {

            case ANY: if (c=='N') {_; state=N;}
                      else {output(c); state=ANY;}
                      break;   ↑?

            case N:  ...

            default: printf("error"); exit(-1); }

    }exit(0); }
```

```
ANY_f (char c) {
  if (c==N) {...}
  else {...}
}
N_f (char c) {...}
```

actions

②

```
int main (void) {
    state = ANY
    while ((c=getchar()) != EOF) {
        (actions[state])(c);
    }
}
```

```
enum state_t {ANY, N, ...};
int main (void) {
    enum state_t [redacted] state;
    char c;
    while ((c=getchar())!EOF){
        (action[state])(c);
    }
    exit(0);
}
```

```
enum
void (actions[])(char c)={ANY_f,
                          N_f,
                          :
                          };
```

translate.c

```
gcc -ansi -pedantic -Wall
    -o translate translate.c
./translate < test1.7 > test1.1
```

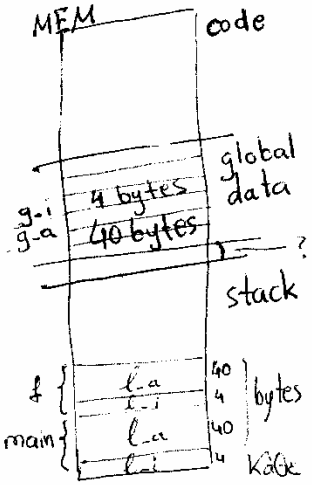
20/2/2006 Έστω το πρόγραμμα:

```
# include <stdio.h>
int g_i;
int g_a[10];

void f(int a) {
    int l_i;
    int l_a[10];
    :
}

int main(void) {
    int l_i;
    int l_a[10];
    :
}
```

global: μεταβλητές ορατές από όλο το πρόγραμμα
local: ορατές μόνο στη συνάρτηση που είναι ορισμένες



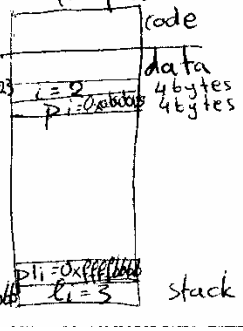
Οι global μεταβλητές τοποθετούνται στη μνήμη του συστήματος στο "global data" segment (σήμερα) (κ υπάρχει ένα μόνο αντίγραφο κάθε μεταβλητής)

Για τις local μεταβ. κάθε συνάρτηση έχει το δικό της τοπικό αντίγραφο. Γι' αυτό χρησιμοποιούμε ένα επιπλέον σήμα της μνήμης που ονομάζεται stack (segment).

Κάθε συνάρτηση τοποθετεί τις τοπικές μεταβλητές της σε ένα σήμα του stack που ονομάζεται stack frame της συνάρτησης.

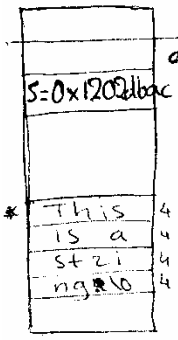
- Ανεξάρτητα από το αν η μεταβλητή είναι global ή local, το μέγεθος της στη μνήμη καθορίζεται από τον τύπο της μεταβλητής.
- Οπότε η δήλωση κάθε μεταβλητής καθορίζει τις πράξεις που μπορούμε να κάνουμε κ τον χώρο που χρειάζεται στη μνήμη.
- Ο τύπος μεταβλητής pointer ορίζει τις πράξεις &, * και δεσμεύει χώρο αρκετό για μια διεύθυνση μνήμης (4 bytes). *: επιστρέφει την τιμή μιας μεταβλητής
  &: δημιουργεί έναν ptr από μια μεταβλητή τύπου ptr

```
int i=2;
int *pi = &i;
int main(void) {
    int li=3;
    int *pli = &li;
    *pi = *pli;
}
```



i=li

Στη C τα strings είναι pointers σε char που ο τελευταίος χαρακτήρας είναι '\0'. π.χ. char *s = "This is a string.";



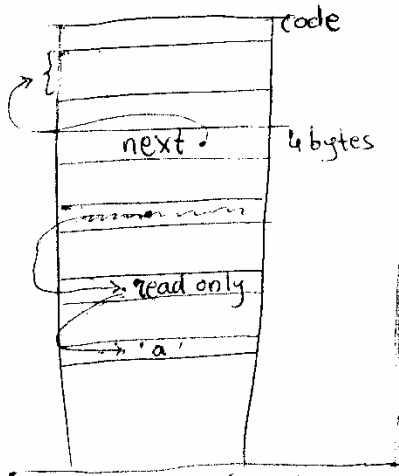
Δηλώσεις στη C Κανόνες "αποκωδικοποίησης" δηλώσεων:

1. Πήγαινε στο "leftmost" όνομα (identifier) της δήλωσης. Πες "ο identifier ... είναι μεταβλητή τύπου".
2. Κοίταξε το σύμβολο δεξιά αν είναι:
   "[" : για κάθε ζευγος [] πες "array of"
   "(" : διάβασε μέχρι το ")" και πες "function που επιστρέφει:"
   · goto 2
3. αν το σύμβολο στα αριστερά είναι "(" τότε διάβασε μέχρι το ")" go to βήμα 2.
4. Εάν το σύμβολο στα αριστερά είναι *, const, volatile
   · Συνέχισε να διαβάζεις όλα αυτά τα σύμβολα και για κάθε ένα πες: * → pointer σε
   const → read-only
   volatile → volatile
   · goto βήμα 3
5. Τα υπόλοιπα σύμβολα στα αριστερά είναι ο βασικός τύπος της δήλωσης

π.χ. char *const *(*next)();

1. πήγαινε next το next είναι μεταβλητή τύπου
2. x βήμα2 4. "*" pointer σε
3. x βήμα3 5. "(" → διαβάζαμε)
6. βήμα2 διαβάζαμε ")" function να επιστρέφει
7. βήμα4 στα αριστερά "*" pointer σε
   read-only, pointer σε char

" Το next είναι μεταβλητή τύπου pointer σε συνάρτηση που επιστρέφει pointer σε read-only pointer σε char."

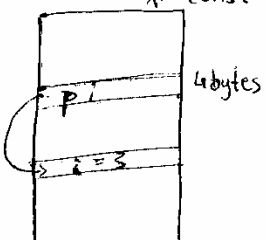


int a[A][B]; το a είναι τύπου array of /*size A*/ (π.χ. 2) array of /*size B*/ int

(α) int i=2; → μπορούμε στο πρόγραμμα να διαβάσουμε και να γράψουμε το i;

const int j=2; → μπορούμε να διαβ. το j αλλά όχι να αλλάξουμε την τιμή του. π.χ. const int WEEK_DAYS = 7;

(β) int i=3;
const int *pi = &i;
int *const pi = &i;

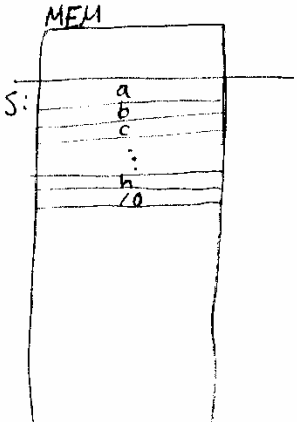


22/2/2006 (α) Ποια είναι η σχέση ανάμεσα σε arrays, strings κ pointers

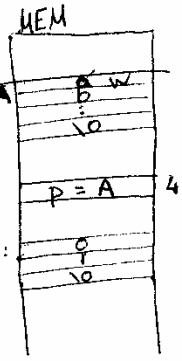
i) Έστω οι δηλώσεις: char s[9] = "abcdefgh"; char c = 'a';

Το s είναι ένα array 9 χαρακτήρων (char). Τι πράξεις μπορούμε να κάνουμε με τη μεταβλητή "s". · s[i] ✓ · s = ?

Η δήλωση ενός array δεσμεύει χώρο για τα στοιχεία του array. Το όνομα του array είναι 'κάτι' που δείχνει στο πρώτο στοιχείο του array. Για να βρούμε το στοιχείο s[i] πρέπει: - να πάρουμε την έτσιμη (θέση μνήμης) που "αντιστοιχεί" το s - προσθέσουμε το i - το στοιχείο s[i] βρίσκεται στη θέση μνήμης που προέκυψε.



ii) `char *p = "abcdefgh"; char c = 'a';` ③ Το p είναι μεταβλητή τύπου pointer που δείχνει στο string "abcdefgh".



Το p είναι μια μεταβλητή που παίρνει χώρο στη μνήμη και η αρχική της τιμή είναι η διεύθυνση της μνήμης όπου βρίσκεται το string.

Πράξεις με το p: `*p = 'w'; *(p+i) = ␣; p = B;`

Στη C τα ονόματα των arrays και οι pointers αντίστοιχου τύπου είναι (σε ορισμένες περιπτώσεις) ισοδύναμα. p[i]

a) ένα όνομα array σε μια έκφραση (expression) είναι πάντοτε ισοδύναμο με ένα ptr.
β) ένας δείκτης σε πίνακα είναι ισοδύναμος με το offset από την αρχή του πίνακα. c) τα ονόματα των arrays και οι αντίστοιχοι ptrs είναι ισοδύναμα στα περισσότερα διπλώσεων συναρτήσεων.

π.χ. `int a[10], *p, i;`

$$\left[\begin{array}{l} p = a; \\ = p[i]; \\ = a[i]; \end{array}\right] \equiv \left[\begin{array}{l} p = a; \\ = *(p+i); \\ = *(a+i); \end{array}\right] \equiv \left\{\begin{array}{l} p = a+i; \\ = *p; \end{array}\right.$$

- Χώρος στη μνήμη για arrays, ptrs

① `char s[3] = "ab"; s[1] = 'c';`

② `char *p; *p = 'a';`

① sizeof(s) → 3
② sizeof(p) → 4

- C preprocessor

Ένα πρόγραμμα C έχει την εξής δομή:

`#include < > / <declarations> / <functions> /`

Όλες οι γραμμές του .c προγράμματος που ξεκινούν με "#" (hash) είναι οδηγίες του C preprocessor (cpp)

- Ο cpp είναι ένα πρόγραμμα που: (1) αφαιρεί όλα τα σχόλια από το input του (2) επεξεργάζεται όλες τις οδηγίες (που ξεκινούν με "#") (3) ενώνει τις (φυσικές) γραμμές που χωρίζονται στο input με "\"
- Ο cpp παράγει το output file. Ο cpp εξ ενώς δεν έχει σχέση με τη γλώσσα C. Δηλ. θα μπορούσαμε να το χρησιμοποιήσουμε με οποιοδήποτε input (π.χ. any text file). Ωστόσο ιστορικά ο cpp έχει καταλήξει να είναι το πρώτο βήμα κατά τη μετάφραση ενός προγράμματος C.

① `#include <file>` / `"file"` αντικαθίσταται το `#include` με τα περιεχόμενα του file. Συνήθως χρησιμοποιείται για να συμπεριλάβουμε ορισμούς μεταβλητών κ.α. συναρτήσεων. [Κάνουμε include μόνο .h αρχεία. Ποτέ .c αρχεία]

② `#ifdef STRING` Αν το STRING έχει οριστεί προηγουμένως (με #define) τότε το output θα περιέχει μόνο το εμπίπτα (a) διαφορετικά μόνο το (b)

```
#ifdef STRING
  <a>
#else
  <b>
#endif
```

```
#if (0)
  ...
#endif
```
— παραλείπεται π.χ. κώδικας με σχόλια

π.χ.
```
#ifdef LINUX
  ...
#else   /* WINDOWS */
  ...
#endif
```

③ `#define NAME expression` - Ορίζει το NAME να αντιστοιχεί στο expr. και αντικαθιστά κάθε εμφάνιση του NAME στο τρέχον αρχείο από το σημείο του #define εως το τέλος με expr. `#define KBYTE 1024` → `a = 10 * KBYTE` → `a = 10*1024`

```
#define KB 1024
#define MB KB*KB
int a;
a = 3*MB;
```
1024 1024

→
```
int a;
a=3*1024*1024;
```

Προσοχή:
```
#define MAX(x,y) (x<=y ? y : x)
#define MULT(x,y) x*y  ή  ((x)*(y))

x = MAX(3,7);  → x = (3<=7 ? 7 : 3)
x = MULT(4,3,7); → x = 4+3*7  ή x = ((4+3)*(7))
```

④ • Operator s1##s2 ενώνει τα string s1, s2

π.χ. #define TEMP(x) temp##x / TEMP(1) = 3; → TEMP1 = 3;

• Μεταβλητές _FILE_ , _LINE_

π.χ. #define PRINTMSG(s) printf("file:%s \
line:%d \
%s \n", \
FILE,
LINE,
s)

PRINTMSG("memory alloc error");

→ file:test.c line:105 memory alloc.

Θέματα στυλ προγραμματισμού

A) Ονόματα : τα ονόματα που χρησιμοποιούμε πρέπει :
- να δίνουν χρήσιμη πληροφορία - να είναι εύκολα σε ανάγνωση και απομνημόνευση - να είναι συνεπή στον συμβολισμό.

1) Γενικοί κανόνες για ονόματα : - Χρησιμοποιούμε όλο CAPITALS για σταθερές (defines, consts)
- FirstCapital για global μεταβλητές
- firstLowercase για local μεταβλητές
- κάποιο πρόθεμα π.χ. i, f, d, .. για να δηλώσουμε τον τύπο π.χ. inelts ή iNumElements

2) Προσπαθούμε να είμαστε "συνεπείς" με τα ονόματα
- σχετικές μεταβλητές ⇒ σχετικά ονόματα - αποφυγή πλεονασμών

3) Για συναρτήσεις χρησιμοποιούμε ενεργά ονόματα π.χ. put, get

4) Να είμαστε ακριβείς στα ονόματα π.χ. if (checkOctal(x)) → if(isOctal(x))

B) Εκφράσεις και εντολές

1) Χρησιμοποιούμε στοίχιση που αντιστοιχεί στη δομή του προγράμματος

2) Όρω από τελεστές +, *, &, |, χρησ. παρενθέσεις και δεν βασιζόμαστε στην προκαθορισμένη προτεραιότητά τους.

3) Χρησιμοποιούμε τη θετική μορφή των εκφράσεων !(x<y) → (x ≥ y)

⟶ Το πρόγραμμα/εργαλείο indent • σύμφωνα με κάποιους κανόνες (που αλλάζουν) κάνει format ένα .c/.h αρχείο

4) Χωρίζουμε πολύπλοκες εκφράσεις σε απλούστερες

5) Προσοχή στο side effect των εκφράσεων / side effect: όταν μια έκφραση εκτός από το να επιστρέφει μια τιμή, έχει και κάποιο άλλο αποτέλεσμα.

π.χ. i = x++; / i=0; str[i++] = str[i++] = '\0'; / if (cond1 && cond2)
? → str[0] = str[0] = '\0', i++, i++; / false x=a ?

char s1[] = "123";
char s2[] = "12345";
scanf("%d %d", &yr, &profit[yr]); profit[?] ← 10000
printf("%d %s %s %d %s",
sizeof(s1), s1, strcpy(s1,s2), sizeof(s1), s1);
Λάθος δεν χωράει

Γ) 1) Όταν θέλουμε να ορίσουμε συναρτήσεις χρησ. συναρτήσεις της γλώσσας και όχι macros (defines). Στα macros ο compiler δεν μπορεί να ελέγξει αν οι παράμετροι κατά την κλήση της συνάρτησης αντιστοιχούν στις παραμέτρους κατά τον ορισμό της συνάρτησης

2) Δεν χρησιμοποιούμε αριθμούς (αλλά τους δίνουμε ονόματα) με defines ή καλύτερα constants int const N-CHARS = 256;

Δ) Σχόλια

1) Δεν αναλύουμε το προγραμα
2) Προσθέτουμε σχόλια σε αρισμούς συναρτήσεων:
   - Τι είναι η κάθε input param της συνάρτησης
   - Τι κάνει η συνάρτηση (χρησ. τα ονόματα των local var)
   - Τι επιστρέφει η συνάρτηση
   - Τι άλλα side-effects έχει η συνάρτηση εκτός από το input/output

π.χ.
```
/* return an integer in
   the range [0..(r-1)] */
int random (int r) {
    return (int) floor(random() * r);
}
```

3) προσθέτουμε σχόλια σε global vars π.χ.
```
struct State { /* prefix + suffix list */
    char *prefix[NPREF]; /* possible prefix words */
    char *sufix; /* list of suffixes */
    State *next; /* next element in hash table */
}
```

4) Δεν σχολιάζουμε κακογραμμένο κώδικα αλλά τον διορθώνουμε.

1/3/2006 a) Η stdlib περιέχει ένα αρχείο string.h που ορίζει ένα σύνολο από συναρτήσεις για την επεξεργασία μεταβλητών τύπου string. Η stdlib περιέχει εκτός από τους ορισμούς και κάποια υλοποίηση των συναρτήσεων. Η ασκ. 2 μας δίνει τα string.h και ζητάει δύο διαφορετικές υλοποιήσεις των συναρτήσεων → string.h → pstring.c, astring.c

b) assert.h header file της stdlib που ορίζει μια συνάρτηση "assert(cond)" η οποία ελέγχει αν cond==TRUE αν όχι τυπώνει ένα μήνυμα κ κάνει exit (όλο το πρόγραμμα)
π.χ. assert(x > 0); 1) έλεγχος συνθήκης 2) debugging

→ Τι είναι "μεγάλα" προγράμματα?
Οποιοδήποτε πρόγραμμα κάνει κάποιες εκτενείς λειτουργείες, χρησιμοποιεί "αρχετές" δομές/μεταβλητές και συναρτήσεις θέλουμε να χωρίσουμε τις μεταβ/συναρτήσεις σε πολλαπλά αρχεία: 1) μεγάλα αρχεία παίρνουν πολύ χρόνο για μετάφραση / compilation. 2) Ο προγραμματιστής πρέπει να ψάχνει ένα μεγάλο αρχείο για να εντοπίσει το κώδικα που πρέπει να αλλάξει.
3) ~~Για~~ για να ξαναοχεδιάσουμε κάποιο τμήμα του προγράμματος, πρέπει να επιρεάσουμε και άσχετα τμήματα που είναι στο ίδιο αρχείο.

→ Άρα πρέπει να χωρίζουμε το προχ. σε πολλαπλά αρχεία.
Πρόβλημα: Ⓐ Συναρτήσεις Έστω ότι έχουμε μια συνάρτηση void f(void){printf} και θέλουμε να την χρησιμοποιήσουμε σε 2 αρχεία: f1.c κ f2.c
(a) αν η f είναι μόνο στο f1.c αν την καλέσουμε στο f2.c τότε ο compiler δεν θα γνωρίζει την υπαρξή της (για το f2.c)
(b) αν ορίσουμε την f και στα δύο αρχεία, τότε ο compiler (linker) θα παραπονεθεί για διπλό ορισμό.

- Για να λύσουμε αυτό το πρόβλημα, σε κάθε γλώσσα έχουμε
  a) ένα μηχανισμό για "δήλωση" συναρτήσεων (declaration)
  b) ένα μηχανισμό για "ορισμό" συναρτήσεων (definition)

  a) στη γλώσσα C : function prototype
  b) f.h —"— : ο κώδικας της συνάρτησης

f1.c
```
void f(void);
{
 f();
}
void f(void) {
    printf(" ");
}
```

f.h
```
void f(void);
```

f2.c
```
void f(void);
{
 f();
}
```

ορισμός (1 φορά)

b) Global μεταβλητές

Έστω ότι το προγ. χρησιμοποιεί τη global μεταβ. int x;
και κάθε ένα από τα δύο αρχεία f1.c, f2.c έχει κώδικα που χρειάζεται το x.

Πρόβλημα:
a) αν το x είναι δηλωμένο στο ένα από τα δύο αρχεία ως int x; αλλά στο άλλο ο compiler θα παραπονεθεί ότι δε γνωρίζει το x. β) αν ορίσουμε το x και στα δύο αρχεία θα έχουμε 2 μεταβλητές με το ίδιο όνομα. Γιατί?

f1.c
```
int x;
```

f2.c
```
int x;
```

κάθε δήλωση στη C 1. ορίζει τον τύπο 2. δεσμεύει χώρο

Για κάθε μεταβλητή χρειαζόμαστε: a) ένα και μόνο ορισμό (definition) που δεσμεύει χώρο για τη μεταβλητή. β) μια ή περισσότερες δηλώσεις που απλά ορίζουν τον τύπο της μεταβλητής.
a) στη C όλες οι δηλώσεις μεταβλητών
β) δήλωση μεταβλητής που προηγείται το keyword "extern"
Όπως και σε συναρτήσεις μπορούμε να έχουμε ένα .h αρχείο που να περιλαμβάνει μόνο δηλώσεις και στη συνέχεια κάνουμε #include το .h αρχείο όπου χρειάζονται οι μεταβλητές. Ένα από τα .c αρχεία πρέπει να περιέχει τον ορισμό κάθε μεταβλητής.

f.h
```
extern int x;
```

f1.c
```
#include "f.h"
int x;
```

f2.c
```
#include "f.h"
```

Σε κάθε γλώσσα προγ. τα αρχεία του προγρ. χωρίζονται σε 2 κατηγορίες. a) αρχεία που περιέχουν μόνο δηλώσεις .h (στη C)
β) αρχεία που περιέχουν ορισμούς (κώδικα) .c

- Χωρισμός προγράμματος σε αρχεία

(1) Τοποθετούμε κάθε ~~συναρτ~~ σύνολο από συναρτήσεις σχετικές μεταξύ τους σε .c αρχεία.

(2) Δημιουργούμε για κάθε .c αρχείο ένα .h που περιέχει τις δηλώσεις που είναι απαραίτητες για να χρησιμοποιήσει κάποιος το .c

(3) Το .h το κάνουμε include στο αντίστοιχο .c και σε οποιοδήποτε άλλο αρχείο (.c, .h) που χρειάζεται τις σχετικές δηλώσεις.

(4) Η συνάρτηση main ορίζεται σε κάποιο από τα .c αρχεία του προγράμματος (συνήθως ~~από~~ το αρχείο • που έχει το όνομα του προγράμματος)

- Τι είναι ένα <u>module</u> / <u>βιβλιοθήκη</u> ? ένα .h και κάποια .o αρχεία
Για να χρησιμοποιήσουμε τον κώδικα κάποιου άλλου χρειαζόμαστε:
  - δηλώσεις συναρτήσεων (<u>interface</u>, .h)
  - μια υλοποίηση των συναρτήσεων (<u>object file</u>, .o)

gcc s1.c(ή)s2.c test.c -o test gcc -c (δεν κάνει executable)

Ένα module είναι ένα σύνολο από services (π.χ. συναρτήσεις, μεταβλητές / δομές δεδομένων) τα οποία μπορούν να χρησιμοποιήσουν άλλα προγράμματα. Κάθε module αποτελείται από (1) ένα <u>interface</u>, που συνήθως είναι ένα .h file, και το οποίο δηλώνει τα services που παρέχει το module (π.χ. prototypes, κτλ) και (2) από μια υλοποίηση, που είναι ένα ή περισσότερα .c αρχεία υλοποιούν/ορίζουν τις συναρτήσεις κ μεταβλητές που "παρέχει" το module. Κάθε module έχει <u>ένα</u> μόνο interface, αλλά μπορεί να έχει πολλές υλοποιήσεις. Γιατί χρειαζόμαστε modules ?

(1) Επαναχρησιμοποίηση κώδικα (code re-use)

(2) Πιο εύκολο να γράψουμε άλλα κομμάτια κώδικα (όπου το interface ενός module είναι το μόνο που χρειάζεται να γνωρίζουμε).

(3) Είναι πιο εύκολο να κάνουμε debug ένα κομμάτι, γιατί το interface του module αποτελεί ένα "όριο" για το που μπορεί να βρίσκονται τα λάθη.

Π.χ. module που υλοποιεί μια στοίβα

α) interface: stack.h

```
void empty_stack(void); /* αρχικοποιεί τη στοίβα του module */
int is_empty(void); /* επιστρέφει TRUE αν η στοίβα είναι άδεια, FALSE διαφορετικά */
void push(int i); /* τοποθετεί το i στην κορυφή. Αν η στοίβα είναι γεμάτη
                     τυπώνει λάθος και τερματίζει το πρόγραμμα */
int pop(void); /* επιστρέφει το στοιχείο στην κορυφή. Αν άδεια */
```

→ extern const int STACK_SIZE;

Stack.c υλοποιεί stack.h

```
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"

const int STACK_SIZE = 100;
static int contents[STACK_SIZE];
static int top = 0;

void empty_stack(void) { top = 0; }
int is_empty(void) { return (top == 0); }
static int is_full(void) { return (top == STACK_SIZE); }
int pop(void) {
    if (is_empty()) {
        printf("Error in pop\n");
        exit(EXIT_FAILURE);
    }
    return (contents[--top]);
}
void push(int i) {
    if (is_full()) {
        printf("Error in push\n");
        exit(EXIT_FAILURE);
    }
    contents[top++] = i;
}
```

[static: keyword στη C (ορίζει storage class μιας μεταβλητής) λέει ότι η μεταβλητή (global) "φαίνεται" μόνο στο .c αρχείο που δηλώνεται.]

f.h
```
#include "g.h"
```
⟷ πρόβλημα ⟷
g.h
```
#include "f.h"
```

λύση

f.h
```
#ifndef _F_H_
#define _F_H_
#include "g.h"
#endif
```

g.h
```
#ifndef _G_H_
#define _G_H_
#include "f.h"
:
#endif
```

Abstract Data Types (ADTs) Ένας ADT είναι ένα module που μας επιτρέπει να ορίζουμε και να χρησιμοποιούμε πολλά αντίγραφα ενός δομής (του τύπου data type) που παρέχει.

stack.h
```
const int STACK_SIZE = 100;
struct stack_s {
    int contents[STACK_SIZE];
    int top;
};
void make_empty(struct stack_s *);
int is_empty(struct stack_s *);
void push(int i, struct stack_s *);
int pop(struct stack_s *);
```
stack_T

Π.χ.
```
#include "stack.h"
:
struct stack_s *Name_Stack;
struct stack_s *Salary_Stack;
is_empty(Name_Stack);
```

stack_T
```
void make_empty(struct stack_s *s){
    s = malloc(sizeof(struct stack_s));
    s->top = 0;
    return;
```

Στους ADTs πολλές φορές θέλουμε να ορίσουμε στο interface του ADT το όνομα ενός τύπου, χωρίς να περιγράφουμε τη δομή του. Τότε οι τύποι ονομάζονται αδιαφανείς - opaque.

```
typedef struct stack_s {
   ...
} stack_T;
```

```
stack_T *Name_Stack;
stack_T *Salary_Stack;
```

stack.h : typedef struct stack_s stack_T;

stack.c : struct stack_s { int contents[STACK_SIZE];
int top; };

Το typedef είναι ένα keyword της C που μας επιτρέπει να ορίσουμε νέα ονόματα για τύπους. typedef type-specification name ;
Όταν το typedef, όταν το χρησιμοποιούμε σε ένα interface (.h), μπορούμε να παραλείπουμε τον ορισμό του τύπου, εφ' όσον δεν χρειάζεται στο υπόλοιπο interface.

Compilation modules & ADTs

Μέχρι τώρα : gcc ——→ -o test1 test.c mystr.c
Αυτή η διαδικασία έχει 2 βήματα. ① Compilation ② Linking
① Μεταγράφουμε κάθε .c αρχείο σε .o (object file) που περιέχουν κώδικα μηχανής και κλήσεις σε συναρτήσεις που ορίζονται αλλού (όχι στο .o)
Άρα τα .o αρχεία δεν είναι εκτελέσιμα.
② Από "πολλά" .o αρχεία, δημιουργούμε ένα εκτελέσιμο που περιέχει κώδικα μηχανής, και περιλαμβάνει όλες τις συναρτήσεις που χρειάζεται το εκτελέσιμο.

Αν έχουμε ένα module ή ADT μπορούμε να διαχωρίσουμε τα δύο αυτά βήματα. (1) gcc ——→ -c stack.c, παράγει το stack.o
Αν κάποιο πρόγραμμα χρειάζεται μια στοίβα, του δίνουμε τα εξής:
1) stack.h 2) stack.o ——→ test.c και παράγουμε το
εκτελέσιμο : (1) gcc —— -c test.c ——→ test.o
(2) gcc —— -o test1 test.o stack.o ——→ #./test1
Δεν έχει σχέση με τη γλώσσα

10/3/2006 HY-255

Makefiles

→ Τα πιο πολλά προγράμματα είναι αρκετά μεγάλα (πολλαπλά files) και χρειάζονται πολύ χρόνο για να τα μεταφράσει κανείς πλήρως.

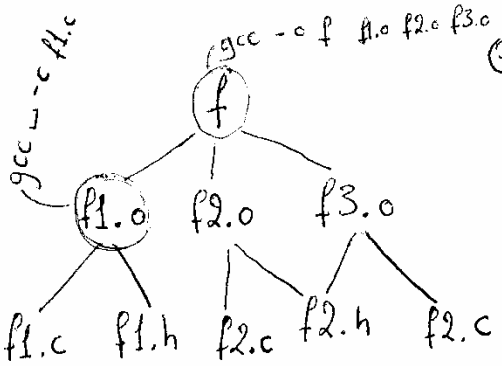
→ Άρα θέλουμε να μπορούμε να κάνουμε "μερική" μετάφραση (partial compilation)

π.χ. Έστω ότι έχουμε το πρόγραμμα f που αποτελείται από τα αρχεία f1.c, f2.c, f3.c και f1.h, f2.h (*)

Στη C έχουμε dependencies (εξαρτήσεις) → ένα .o αρχείο εξαρτάται από το αντίστοιχο .c → επιπλέον ένα .o αρχείο εξαρτάται και από όλα τα .h αρχεία που γίνονται include στο .c → Το executable (εκτελέσιμο) εξαρτάται από όλα τα .o αρχεία που το αποτελούν.

(*) →

f1.c	f2.c	f3.c
#include "f1.h"	#include "f2.h"	#include "f3.h"



① → Από τα source files ενός προγράμματος και με βάση τις εξαρτήσεις μπορούμε να κατασκευάσουμε ένα γράφο εξαρτήσεων, όπου κάθε κόμβος έχει ως παιδιά του τα αρχεία από τα οποία εξαρτάται. - Σε αυτό το γράφο (δέντρο) συνήθως η ρίζα είναι το τελικό exec.

② Συμπληρώνουμε κάθε κόμβο του γράφου με την εντολή που χρειαζόμαστε για να παράγουμε τον κόμβο από τα παιδιά του. ③ Κάθε makefile είναι η αναπαράσταση ενός τέτοιου γράφου.

→ Κάθε κόμβος του γράφου που έχει τη μορφή 

παριστάνεται ως

```
f1 : c1 ... cn
<tab> cmd 1
<tab> cmd 2
```

Το πρόγραμμα make διαβάζει το Makefile και 1) να βρει ποιά αρχεία έχουν αλλάξει 2) να εκτελέσει τις αντίστοιχες εντολές για κάθε κόμβο μέχρι τη ρίζα. Το make συγκρίνει την "ώρα" αλλαγής του κάθε κόμβου με την ώρα αλλαγής κάθε εξάρτησης και αν είναι μικρότερη εκτελεί τις εντολές που έχουν σχέση με αυτό τον κόμβο. Ο έλεγχος αυτός γίνεται bottom-up

Makefile

```
f : f1.o f2.o f3.o
    gcc -o f f1.o f2.o f3.o
f1.o : f1.c f1.h
    gcc ⌊⌋ -c f1.c
f2.o : f2.c f2.h
    gcc ⌊⌋ -c f2.c
f3.o : f3.c f2.h
    gcc ⌊⌋ -c f3.c
clean: rm *.o
```

10/3/2006

Hash Tables

• Ένας hash table είναι μια δομή που μας επιτρέπει να βρίσκουμε "πράγματα" (strings) γρήγορα. • Έστω ότι έχουμε ζευγάρια (string, value) και θέλουμε, όταν μας δίνουν το string, να επιστρέφουμε την τιμή π.χ. ("red", 3), ("while", 1)
"Αν αντιστοιχίσαμε int - σε οποιδήποτε το int θα ήταν index κτλ."
Στον hash table ~~παρουσιάζε~~ μετατρέπουμε το string σε κάποιον integer. Χρησιμοποιούμε μια συνάρτηση hash function που παίρνει σαν input το string και μας επιστρέφει τον αριθμό, που βρίσκεται σε κάποιο δεδομένο εύρος. Έστω π.χ. 0 - 1023.

string $\xrightarrow{\text{hash function}}$ hash (int ∈ [0, 1023])

→ Πάντοτε, αν ο αριθμός των strings είναι πολύ μεγάλος σε σχέση με το εύρος του hash τότε θα υπάρχουν "συγκρούσεις". Σύγκρουση: 2 strings απεικονίζονται από τη hash function στην ίδια θέση στον πίνακα (έχουν το ίδιο hash)

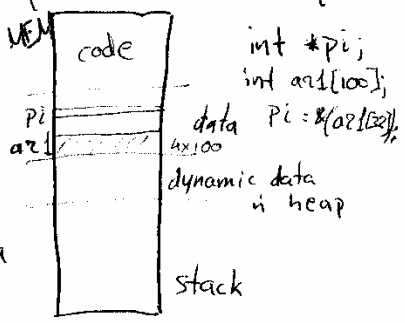
• Για να λύσουμε το πρόβλημα των συγκρούσεων, κάθε θέση του πίνακα δείχνει σε μια λίστα που περιέχει όλα τα ζευγ. που συγκρούονται.

Υποστηρίζει τις λειτουργίες: (1) insert (string, value), (2) lookup (string) → value
(3) delete (string)

(1) a) βρίσκουμε το hash(string) b) πηγαίνουμε στην αντίστοιχη θέση και βάζουμε το ζεύγος στην αρχή της λίστας.
(2) a) βρίσκουμε το hash (string) b) ψάχνουμε την list
(3) a) — " — b) αφαιρούμε το στοιχείο από τη list.

→ Οι hash tables δουλεύουν πολύ καλά (γρήγορα) αν οι λίστες είναι μικρές. Τότε ο χρόνος για όλες τις λειτουργίες είναι O(1).
→ Επομένως ο ρόλος της hash function είναι πολύ σημαντικός.
→ Οι hash tables έχουν και 2 ακόμη λειτουργίες: shrink, expand.

13/3/2006 Μέχρι τώρα αναφέραμε σε σχέση με τη μνήμη, αφορά σε στατική χρήση μνήμης
- Όλα τα συστήματα δίνουν τη δυνατότητα στα προγράμματα να παίρνουν και να ελευθερώνουν μνήμη δυναμικά. Οι pointers στα προγράμματα δεν έχουν απαραίτητα σχέση με δυναμική ανάθεση μνήμης. Αντίστροφα αν χρειαζόμαστε δυναμική μνήμη τότε οι μεταβλητές τύπου pointer είναι απαραίτητες για να "ονομάζουμε" και να προσπελαύνουμε τα διάφορα τμήματα δυναμικής μνήμης. → Η μνήμη του προγράμματος περιέχει ένα τμήμα που χρησιμοποιείται αποκλειστικά για να τρέχει το πρόγραμμα (address space) κατά την εκτέλεσή του. Αυτό το τμήμα ονομάζεται dynamic data segment ή heap → Επιπλέον, η C μας παρέχει ένα module (stdlib) το οποίο μπορούν να χρησιμοποιούν τα προγ. για να χειρίζονται το heap. Αυτό το module περιέχει τις εξής συναρτήσεις: malloc, free, calloc, realloc



→ Χρήση των συναρτήσεων → malloc: παίρνει ως παράμετρο το μέγεθος της μνήμης που θέλουμε να δεσμεύσουμε (bytes) και επιστρέφει έναν pointer στο τμήμα της μνήμης που δέσμευσε → void *malloc(size_t); / pi=(int *)malloc (sizeof (int));
- Δεν γνωρίζουμε που βρίσκεται η μνήμη που θα επιστρέφει η malloc
- Ο ptr που επιστρέφει η malloc δείχνει σε ένα κομμάτι συνεχόμενης μνήμης που έχει μέγεθος τουλάχιστον όσο ζητήσαμε → Αν η malloc δεν βρει όσο μνήμη ζητήσαμε επιστρέφει τον NULL pointer value → Πάντοτε ελέγχουμε το αποτέλεσμα της malloc → Για να ελευθερώσουμε ένα τμήμα μνήμης, που είχε allocate με την malloc χρησιμοποιούμε τη συνάρτηση free (void *ptr). Το ptr πρέπει να είναι ο ptr που επέστρεψε η malloc. → Μετά την κλήση της free δεν έχουμε "έλεγχο" στα περιεχόμενα του block που ελευθερώσαμε → calloc: σαν τη malloc + initialize το block μνήμης.
→ realloc: μεγαλώνει ένα υπάρχον block μνήμης

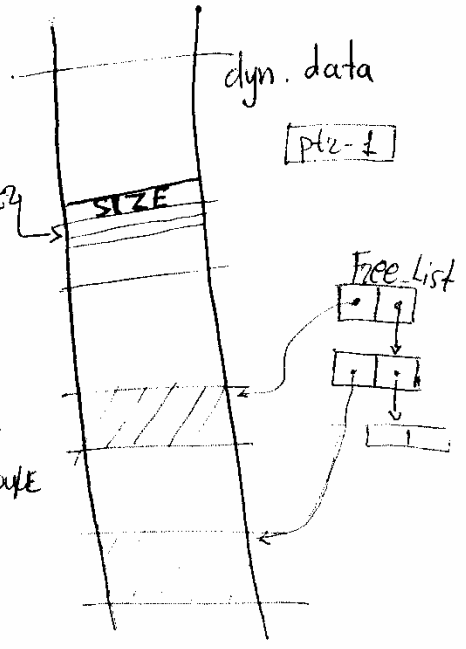
Υλοποίηση malloc/free → Έστω ότι έχουμε μια λίστα free_list που περιέχει τα διαστήματα/block μνήμης του heap που είναι ελεύθερα.
- Όταν ξεκινάει το πρόγραμμα η free_list περιέχει ένα block που είναι όλο το heap π.χ. freeList : (HEAP_START, HEAPSIZE)
- malloc: (1) διατρέχει τη FreeList και βρίσκει ένα block το οποίο έχει μέγεθος τουλάχιστον όσο η παράμετρός της (2) χωρίζει αυτό το block σε δύο τμήματα, το ένα το επιστρέφει (και το αφαιρεί από τη Free_List) ενώ το υπόλοιπο το αφήνει στη Free_List.
- free (1) τοποθετεί το block που δείχνει ο ptr στη Free_List.

→ Το μέγεθος της Free_List έχει σχέση με το πόσα ελεύθερα blocks έχουμε (κ όχι πόσο ελεύθερη μνήμη συνολικά).

→ Για να αποφύγει η malloc/free να έχα κάποια δομή δεδομένων για τα blocks που έδωσε στο χρήστη πριν από την αρχή κάθε block αποθηκεύει το μέγεθος του block.

→ Άρα κατά την εκτέλεση της free(ptr), γνωρίζει που να βρει το μέγεθος size κ προσθέτει στη free list το (ptr, size)
* Σημαντικό το ptr να είναι αυτό που επέστρεψε η malloc

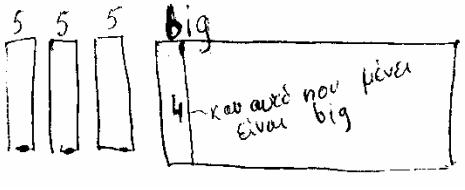
→ Μέγεθος της Free_List. Όταν ελευθερώνουμε ένα block υπάρχει περίπτωση να πρέπει να δημιουργήσουμε έναν επιπλέον κόμβο (αν δεν υπάρχει κάποιο block που είναι συνεχόμενο στη μνήμη) ή να μπορούμε να ενωμματώσουμε το block μαζί με κάποιο άλλο.



→ tradeoff στην απόδοση malloc/free (και στο μέγεθος της λίστας)

→ allocation κατά την malloc

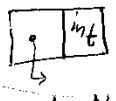
Η καλύτερη δυνατή πολιτική για να επιστρέψουμε μνήμη στο χρήστη, εξαρτάται από τη σειρά και το μέγεθος του block, που ζητάει το πρόγραμμα



Πολιτικές ανάθεσης είναι: best fit, largest (worst) fit first fit

Προβλήματα χρήσης malloc / free: 1. χρήση μνήμης που δεν κάναμε malloc
2. χρήση μνήμης που την κάναμε malloc αλλά και free
3. να κάνουμε malloc μνήμη αλλά να μην την κάνουμε free.
1. (pointers που δεν είναι αρχικοποιημένοι) 3. (memory leak) 2. (dangling pointer problem)

→ Υλοποίηση διασυνδεδεμένης λίστας με δυναμική μνήμη.



Χρειαζόμαστε a) δηλώσεις b) πράξεις - insert - delete - search

a) struct node{
    int value;
    struct node *next;
};

e.g. struct node *head;

b) insert (list, int value) {
    struct node *n; ~~struct node *~~
    n = (struct node *) malloc (sizeof(struct node));
    n → value = value; [n (*n). value = value;]
    n → next = head;
    head = n;
    return h;
}

e.g. insert (head, N);
→ struct node *head

Στις συναρτήσεις στη C η κλήση των συναρτήσεων γίνεται με "call by value" δηλ. οι τιμές των μεταβλητών που περνιούνται ως παράμετροι στη συνάρτηση αντιγράφονται στη στοίβα ως input params στη συνάρτηση). 1) πρώτη λύση [head = insert (head, N);]

2) δεύτερη λύση [insert (&head, N);] [insert (struct node **h, int value)]
*h = n;

15/3/2006 → Έλεγχος συνθηκών (assertions) → Ποιά είναι η "κατάσταση" που αλλάζει ένα οποιοδήποτε κομμάτι προγράμματος → Σε μια procedure, η "input" και "output" κατάσταση είναι οι μεταβλητές ● input/output και οι global μεταβλητές. → Ένα κομμάτι κώδικα (π.χ. μια procedure) για να είναι σωστό πρέπει: (1) η input κατάσταση του να είναι "σωστή" (με βάση τους κανόνες που λειτουργεί το πρόγραμμα) (2) ο κώδικας θα πρέπει να κάνει τις σωστές λειτουργίες (3) Η output κατάσταση θα πρέπει να είναι σωστή (με βάση τους κανόνες του προγράμματος).

π.χ. Έστω η συνάρτηση

```
char *strcpy (char *s1, const char *s2){
    char *s = s1; [assert(s2!=NULL)]
    while ((*(s++) = *(s2++)) != '\0')
        ;
    return s1;
}
```
[assert(s1!=NULL)]

a) Πρέπει τα s1, s2 να μην είναι NULL
b) Πρέπει τα s1, s2 να δείχνουν σε σωστή μνήμη
c) η τιμή επιστροφής θα πρέπει να είναι μη NULL

→ Τέτοιες συνθήκες, μπορούμε στη C να τις ελέγξουμε με την assert(). Η assert ορίζεται στο <assert.h> ως εξής:

```
#ifndef NDEBUG
#define assert(cond) { if((cond)!=TRUE){
        printf("assertion failure, file %,
                line %d\n", __FILE__,
                __LINE__);
        exit (ASSERTION_FAIL);
    }
#else
#define assert(cond) {;}
#endif
```

• Στα πιο πολλά προγράμματα υπάρχουν συνθήκες που πιθανόν δεν είναι δυνατόν να ελεγχθούν

→Έστω ότι ένα πρόγραμμα χρησιμοποιεί μια διατεταγμένη λίστα.

• Το insert πρέπει να τοποθετήσει την τιμή value στη διατεταγμένη list l στη σωστή θέση.

```
insert(l, value);
```

```
void insert (list *l, int v){
→ η list πρέπει να υπάρχει και να είναι διατεταγμένη
  assert(is_sorted(l));
  ⋮
  assert(is_sorted(l));
→ η l να είναι διατεταγμένη
```

→Γράφουμε μια συνάρτηση is_sorted(l) που επιστρέφει TRUE if l is sorted, FALSE otherwise

→Για να ελέγξουμε τις συνθήκες που πρέπει σε ένα πρόγραμμα χρειάζεται:
(1) να έχουμε ξεκάθαρες τις συνθήκες (2) να μπορούμε να τις εκφράσουμε
(3) να τις ελέγξαμε στα σωστά σημεία. → Σε ένα πρόγραμμα δεν χρειάζεται να ελέγχουμε τετριμμένες συνθήκες. Ελέγχουμε: input params, return values, global

π.χ. `enum cond_T { ONE, TWO, THREE } cond;`

```
switch (cond) {
  ONE: ....
       break;
  TWO: ....
       break;
  THREE: ....
       break;
  default:
       assert(FALSE);
}
```

```
main() {
    while(1){
        get_request();
        process_request();
    }
    assert(FALSE);
}
```

(*)
```
p = ( ) malloc( );
  — ? —
```

Πότε ελέγχουμε συνθήκες με assertions και πότε με κώδικα "if( ) error"

• Τα assertions δεν ελέγχονται όταν η σταθερά NDEBUG είναι ορισμένη.

• Ένα πρόγραμμα επιθυμούμε πρέπει να "εκτελείται" σωστά με ή χωρίς τα assertions.

•Άρα τα assertions είναι ένας μηχανισμός που μας βοηθάει στο να γράψουμε (και να κάνουμε debug) ένα σωστό πρόγραμμα. Σε production code τα assertions αφαιρούνται για λόγους ταχύτητας.

(*) Σε αυτή την περίπτωση η malloc στην κανονική λειτουργία της μπορεί να επιστρέψει είτε έναν valid pointer είτε NULL. Άρα όποιος την καλεί πρέπει να χειριστεί και τις δύο περιπτώσεις. Άρα ο έλεγχος πρέπει να γίνεται με if (cond) και κατά την κανονική λειτουργία του προγράμματος.

π.χ.
```
void strcpy(s1, s2){
    while( )

    return s1;
}
```

Ο ορισμός (περιγραφή) κάθε τμήματος προγράμματος καθορίζει αν μια συνθήκη είναι επιτρεπτή κατά την κανονική εκτέλεση του προγράμματος. Για αυτές τις συνθήκες χρησιμοποιούμε ελέγχους if (cond) error, οι οποίοι εκτελούνται πάντοτε. (runtime checked errors)

Αν η περιγραφή της insert ήταν ότι δέχεται μια λίστα, και αν η λίστα είναι διατεταγμένη τότε τοποθετεί το v στη σωστή θέση, διαφορετικά...

Για όλες τις υπόλοιπες συνθήκες που πρέπει να ισχύουν, αλλά δεν πρέπει να εμφανίζονται κατά την κανονική λειτουργία του προγράμματος, χρησιμοποιούμε asserts.

```
count--; //assert((count--)!=0) ΛΑΘΟΣ
if (count == MIN) {
}
```

Πότε τα assertions δεν περιλαμβάνουν κώδικα που είναι "χρήσιμος".

<errno.h>

• Οι συναρτήσεις της stdlib στη C όταν υπάρχει κάποιο λάθος επιστρέφουν μια τιμή που είναι ενδεικτική του λάθους. • Υπάρχουν όμως συναρτήσεις, όπου δεν μπορούμε να δεσμεύσουμε κάποια από τις τιμές επιστροφής για να δηλώσουμε την ύπαρξη λάθους π.χ. τέτοιες συναρτήσεις είναι αυτές που κάνουν μαθηματικές πράξεις (π.χ. sqrt, pow, ...) και οι οποίες βρίσκονται στη βιβλιοθήκη <math.h>. → Για να μπορούν αυτές οι συναρτήσεις να επιστρέφουν <u>λάθος</u> υπάρχει το <errno.h> που ορίζει μια <u>global</u> μεταβλητή <u>errno</u> της οποίας η τιμή μετά την κλήση κάποιας συνάρτησης δείχνει το λάθος που έχει συμβεί π.χ.

```
#include <errno.h>
.
.
.
errno = 0;
x = sqrt(y);
if (errno != 0) { /* errno ≠ 0
                     => υπάρχει error */
   printf("Error in
           sqrt\n");
}
```

Το errno.h ορίζει και μηνύματα που εξηγούν το κάθε λάθος. Επιπλέον ορίζει την συνάρτηση perror(errno) που τυπώνει το αντίστοιχο μήνυμα.

17/3/2006

Μετατροπές τύπων

Οι μετατροπές τύπων στην C χωρίζονται σε 2 κατηγορίες, άμεσες και έμεσες

(1) <u>Έμεσες μετατροπές τύπων</u> (implicit type conversion) αυτόματα

Είναι οι μετατροπές που τις ορίζει η γλώσσα και τις υλοποιεί ο compiler. Συμβαίνουν στις εξής περιπτώσεις (a) σε αριθμητικές και λογικές <u>πράξεις</u> όταν οι τελεστέοι (operands) δεν έχουν τον ίδιο τύπο. (β) σε <u>εντολές ανάθεσης</u> (assignment statements) όταν οι παραστάσεις δεξιά και αριστερά του "=" έχουν διαφορετικούς τύπους. (c) στο <u>πέρασμα παραμέτρων</u> ανάμεσα στις τυπικές και πραγματικές παραμέτρους της συνάρτησης (d) στις <u>τιμές επιστροφής</u> των συναρτήσεων

<u>Έμμεσες μετατροπές σε αριθμητικούς τύπους</u>: Οι αριθμητικοί τύποι χρειάζονται για να αναπαριστούμε αριθμούς διαφορετικής ακρίβειας π.χ. ακεραίους με 32 ή 64 bit. Οι CPU σήμερα μπορούν να κάνουν πράξεις μόνο όταν τα operands είναι του ίδιου τύπου. π.χ. add $reg, $op1, $op2 Επομένως σε μεικτές αριθμητικές εκφράσεις ο compiler μετατρέπει τα operands των τελεστών για να εκτελεστούν οι αντίστοιχες πράξεις. Ο γενικός κανόνας για τις μετατροπές αυτές είναι ότι: "Τα operands μετατρέπονται στον "στενότερο" δυνατό τύπο, που μπορεί να αναπαραστήσει όλες τις δυνατές τιμές και των δύο operands. i) Έστω ότι ένα από τα δύο operands ενός τελεστή (π.χ. +, *, ..) είναι float ή double. Τότε η μετατροπή γίνεται με βάση τον κανόνα float → double → long double

Αν ένα από τα args είναι long double το άλλο γίνεται long double

——— " ——— double — " ——— " — double

ii) αν κανένα operand δεν είναι τύπος double ή float:

Οι μετατροπές γίνονται με βάση τους κανόνες

unsigned long int
↑
long int
↑
unsigned int
↑
int
↑ ↖
short char

π.χ. unsigned int u=3; int i=-10;

int: = s + c (short char)

unsigned: ? = u + i (↳ μεγάλος θετικός)

Προσέχουμε όταν αναμιγνύουμε unsigned και signed integers.

Έμμεσες μετατροπές στις αναθέσεις

Στις αναθέσεις "=", πάντοτε ο τύπος στα δεξιά του = μετατρέπεται στον τύπο στα αριστερά του =. π.χ. float x = 3; ✓ int x = 3.5; warning

Στις αναθέσεις όταν η μετατροπή γίνεται από πιο στενό τύπο σε πιο ευρύ τότε είναι εντάξει, αν όμως η μετατροπή είναι από ευρύ σε στενό, το οποίο αντιβαίνει τους κανόνες, θα υπάρξει warning και πρέπει να το διορθώσουμε.

(2) Άμεσες μετατροπές τύπων

(a) Σε ορισμένες περιπτώσεις είναι βολικό να μετατρέψουμε εμείς άμεσα τον τύπο κάποιων operands. π.χ. float r; int x, y; r = x/y;

Σε αυτή την περίπτωση παρ' ότι θέλουμε πραγματική διαίρεση, το σύστημα με βάση τους ορισμούς κάνει ακέραια διαίρεση οπότε μετατρέπουμε άμεσα τον τύπο ενός από τα operands για να λειτουργήσει "σωστά" η διαίρεση (type casting) r = x/(float) y;

(b) Άμεσες μετατροπές τύπων pointer

Έστω η συνάρτηση malloc: void *malloc(u_int);

Επειδή η malloc πρέπει να επιστρέφει pointer σε διάφορους τύπους χρειαζόμαστε μια επιπλέον δεσμευμένη λέξη που μπορεί να περιγράψει ένα σύνολο από τύπους. (στη C όλους τους άλλους τύπους) και είναι η void.

Ο τύπος "void *" δηλώνει τιμές που είναι pointers, άρα δείχνουν σε ένα κομμάτι μνήμης αλλά δεν γνωρίζουμε τι τύπο έχει η μεταβλητή που είναι αποθηκευμένη σε αυτό το κομμάτι μνήμης. Για να χρησιμοποιήσουμε αυτή τη μεταβλητή (στην οποία δείχνει ο pointer void *) πρέπει να την μετατρέψουμε άμεσα σε κάποιον δηλωμένο τύπο. Άρα:

void *p = malloc(52); π.χ. *(int *)p = 3; (struct Person_S *)p->name =
((int *)p)[3] = ...

Χρήσεις του void (1) σε συναρτήσεις δηλώνει ότι δεν παίρνουν args ή δεν επιστρέφουν τιμές. void f(void); (2) "void *" δηλώνει μεταβλητή pointer σε οποιονδήποτε τύπο.

Χρειαζόμαστε την άμεση μετατροπή τύπων σε μεταβλητές τύπου pointer ώστε να μπορούμε να υλοποιήσουμε συναρτήσεις (& μεταβλητές) που έχουν πολυμορφικά χαρακτηριστικά, δηλαδή arguments ή/και τιμές επιστροφής που παίρνουν διαφορετικούς τύπους. → Εισάγουμε τον τύπο (void *) για να περιγράψουμε μεταβλητές pointer που δείχνουν σε μεταβλητές διαφορετικών τύπων. → Οπότε οι άμεσες μετατροπές τύπων pointer που επιτρέπονται είναι πάντα από & προς (void *).

Χρήση άμεσης μετατροπής χωρίς πολυμορφισμό (προτείνουμε μετατροπή από ασθενέστερο σε ισχυρότερο τύπο)

```
char (όχι pointer) str[] = "aaa";   | c2 = c1; → warning
char const *c1 = str;               | *c1 = 'b'; ✗
char *c2;                           | *c2 = 'b'; ✓
```

Επιλογές: (a) αγνοούμε το warning
(b) άμεση μετατροπή `c2 = (char *)c1;`

"είναι όνομα για το str που δεν επιτρέπει την αλλαγή του str
"είναι όνομα για το str που επιτρέπει την αλλαγή του str

Οι έμεσες μετατροπές τύπων ονομάζονται & type coersion
Οι άμεσες " " " type conversions
και τις υλοποιούμε με type casting.

Πολυμορφικές συναρτήσεις (με input args πολλαπλών τύπων)

→ Έστω ότι θέλουμε να γράψουμε συνάρτηση που διατάσσει τα στοιχεία ενός πίνακα (in-place) χωρίς να γνωρίζουμε ποιός είναι ο τύπος των στοιχείων

→ Στη C lib η συνάρτηση αυτή είναι: `qsort(void *base, size_t nelt, size_t eltsz, int (*compare)(const void *, const void *))`

→ compare ~~το όνομα~~ -1 αν πρώτο < δεύτερο, 0 ==, >0 . . >.

π.χ.
```
struct arnode { int value, char *str; } struct arnode salary_ar[ARSIZE];
int compare (const void *e1, const void *e2){
    struct arnode *t1 = (struct arnode *)e1;
    struct arnode *t2 = (struct arnode *)e2;
    if (t1->value < t2->value) return -1;
    if (t1->value == t2->value) return 0;
    return 1;
```

`qsort(void *base, size_t nelts, size_t eltsz, int (*compare)(const void *, const void *))`

→ base: αρχή του array στη μνήμη, → nelts: αριθμός στοιχείων στο array
→ eltsz: το μέγεθος του κάθε στοιχείου
→ Κλήση: [redacted] `qsort((void *)salary_ar, ARSIZE, sizeof(struct arnode), compare);`

Program building process

Δημιουργία exe:

```
gcc -ansi -pedantic -Wall -o exefile *.c -l libname
```

libname: Αν η βιβλιοθήκη ονομάζεται libx.a τότε το libname = x.

π.χ.: Στη C η stdlib ονομάζεται libc.a, Η math lib ονομάζεται libm.a. Για να τις χρησ. προσθέτουμε -lc ή -lm
(default)

Βήματα: Έστω ότι το πρόγραμμα test χρησιμοποιεί τα αρχεία (source) f1.c, f2.c, test.c, και τις βιβλιοθήκες libc.a και libm.a

(1) f1.c → f1.i, f2.c → f2.i, test.c → test.i   [gcc -E : κλήση του C preprocessor]
← περιέχουν μόνο C εντολές και εκφράσεις

(2) f1.i → f1.s, f2.i → f2.s, test.i → test.s   [gcc -S : για κλήση του compiler]

(3) f1.s → f1.o, f2.s → f2.o, test.s → test.o   [gcc -c : assembler]

(4) f1.o, f2.o, test.o, libc.a →, libm.a → test (executable)   [linking]   [gcc –]

↓   [loader: load + execute]

copy το executable στη μνήμη και ξεκινάει την εκτέλεση

```
gcc -v -o exefile *.c
```
(δείχνει όλα τα βήματα)

1) Ο C preprocessor παράγει τα .i αρχεία από τα .c αντικαθιστώντας: (i) τα σχόλια με κενά (ii) τις '#' εντολές με το αποτέλεσμα της εκτέλεσής τους (iii) ενώνοντας τις πολλαπλές λογικές ~~αρχικές~~ γραμμές με μια φυσική γραμμή (\ ).

2) Ο compiler μετατρέπει ένα .i σε ένα .s αρχείο ως εξής:

.i		.s
C definitions & declarations	⟷	.data directive ⋮

C functions
void f() {
⋮
}
	⟷	.text directive
		⋮
		f: [prepare stack frame]
		[code]
		[free stack frame]

(3) Ο assembler μεταγράφει το .s σε .o
(4) Ο linker παράγει από τα .o το exe που έχει το ίδιο format
- Το ELF είναι ένα (από τα μερικά) format για να περιγράψουμε:
(i) .o files (ii) exe files (iii) libx.a + libx.so files
- Executable and Linking Format

i) Το .data κομμάτι των .s αρχείων μεταγράφεται σε δεσμεύσεις μνήμης για global μεταβλητές. ii) Το .text του .s μεταγράφεται σε εντολές μηχανής, αντικαθιστώντας το όνομα και παραμέτρους κάθε εντολής assembly με την ακολουθία 0, 1 που ορίζει η κάθε CPU.

ELF format

περιγράφει αν το αρχείο είναι lib, .o ή exe

ELF file header
Program table header (exe)
.text → εντολές σε κώδικα μηχανής
.data → δεσμευση χώρου για global μεταβλητές με αρχικοποίηση
.bss → (βάθωση χωρίς αρχικοποίηση σαν μείωση του μεγέθους του file)
.symtable — linker
relocation text — loader
relocation data — loader
debug — debugger

.bss
Έστω οι δηλώσεις:
int ar1[100] = {0, 1, 2, ..};
int ar2[100];
Για το ar1 πρέπει να "θυμόμαστε" στο .o αρχείο και ποιές είναι οι αρχικές τιμές κάθε στοιχείου (κ επομένως χρειαζόμαστε ανεξάρτητο χώρο).
→ Ωστόσο για το ar2 μπορούμε να "θυμόμαστε" μόνο το μέγεθος.

Linking ονομάζουμε την διαδικασία δημιουργίας ενός εκτελέσιμου από .o και lib αρχεία. Υπάρχουν 2 μορφές linking: (1) static. το εκτελέσιμο αρχείο περιέχει όλο τον κώδικα (συναρτήσεις) που είναι απαραίτητες για την εκτέλεση του προγράμματος. (2) dynamic Το εκτελέσιμο αρχείο περιέχει μόνο τον δικό μας κώδικα και όχι τις (shared) βιβλιοθήκες του συστήματος → Οι βιβλιοθήκες προστίθενται (φορτώνονται) στη μνήμη δυναμικά πριν την εκτέλεση.
→ Το static linking απαιτεί κάθε πρόγραμμα να έχει στη μνήμη το δικό του αντίγραφο βιβλιοθηκών ⇒ επειδή οι κοινές βιβλιοθήκες είναι συνήθως μεγάλες, σπαταλούμε πολύ κύρια μνήμη. → Σε δυναμικό linking όλα τα προγράμματα χρησιμοποιούν το ίδιο αντίγραφο των κοινών βιβλιοθηκών του κώδικα

Στατικό linking

• Ο assembler/compiler στο .o αρχείο τοποθετούν στο κομμάτι του symbol table μια αναφορά σε κάθε global μεταβλητή και συνάρτηση που χρησιμοποιείται ή ορίζεται σε αυτό το αρχείο.

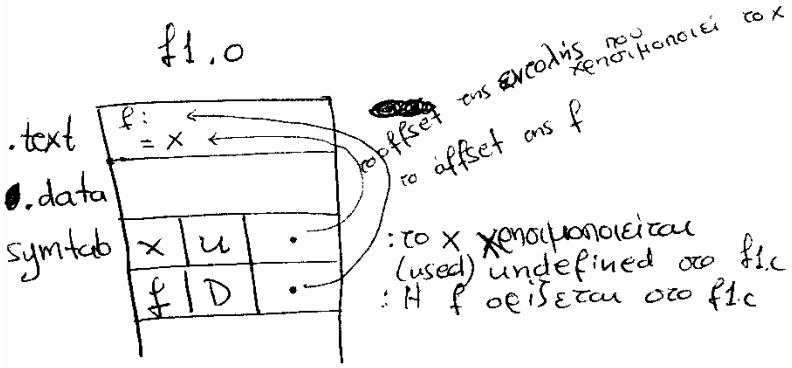
π.χ. f1.c

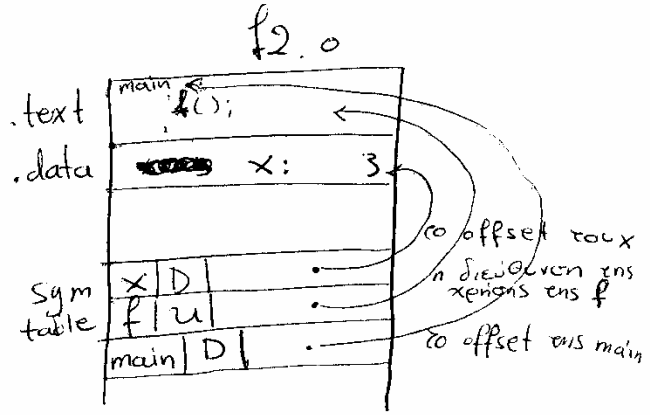
```
extern int x;
void f(void){
    --- = x;
}
```

(χρήση του x)

f2.c

```
int x=3;   (ορισμός)
:
main(){
    f();
}
```

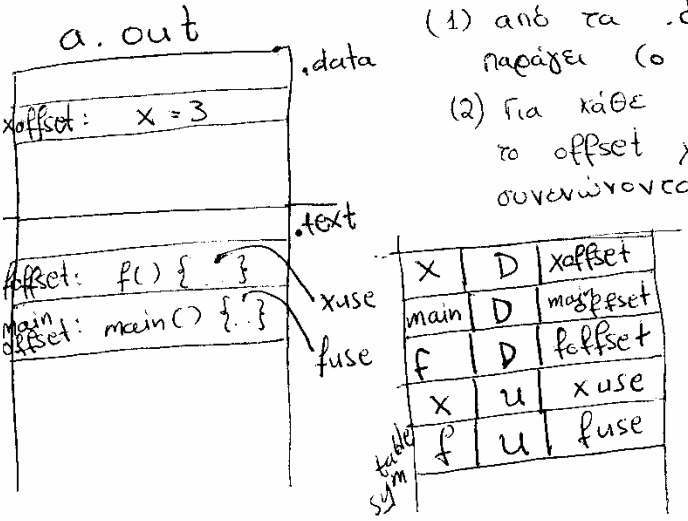




3/4/2006

Για κάθε όνομα σημειώνουμε αν αυτό χρησιμοποιείται στο συγκεκριμένο αρχείο ή αν ορίζεται (D, δηλ. καταλαμβάνει μνήμη).
Επιπλέον για κάθε σύμβολο αποθηκεύουμε (i) αν D, τη θέση στο .data ή .text segment που ορίζεται ή (ii) αν U, τη θέση στο .text όπου χρησιμοποιείται.

Το linking είναι η διαδικασία με την οποία από ένα ή περισσότερα .o αρχεία παράγουμε ένα εκτελέσιμο (a.out) το οποίο δεν περιέχει σύμβολα που χρησιμοποιούνται αλλά δεν ορίζονται.



(1) από τα .data και .text του κάθε αρχείο παράγει (ο linker) ένα ενιαίο .data κ .text

(2) Για κάθε σύμβολο στο νέο αρχείο προσαρμόζει το offset χρήσης ή ορισμού ανάλογα με το πως συνενώνονται τα επιμέρους .data, .text εμπίπτα

(3) Στον καινο symbol table (a.out) υπάρχουν όλες οι διευθύνσεις του .text όπου χρησιμοποιούνται σύμβολα που αντιστοιχούν σε μεταβλητές ή συναρτήσεις και οι διευθύνσεις του .data που ορίζονται μεταβλητές και οι διευθύνσεις του .text που ορίζονται συναρτήσεις.

(4) Ο linker αντικαθιστά στο .text διευθύνσεις που αντιστοιχούν σε ορισμούς (symbol resolution) (x use → x offset)

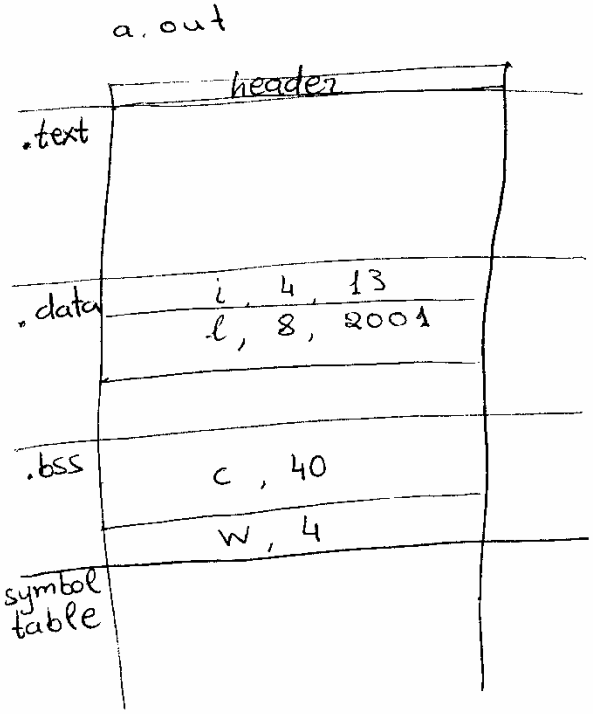
Κατά την διαδικασία του linking τα 2 λάθη που μπορεί να συμβούν είναι 1) να υπάρχουν δυο ορισμοί για το ίδιο σύμβολο 2) για κάποιο σύμβολο που χρησιμοποιούμε να μην υπάρχει ορισμός σε ~~κάθε~~ κανένα .o αρχείο.

→ Το πρόγραμμα "nm -s" τυπώνει το symbol table κάθε .o , .exe, lib αρχείου.

e.g.
```
char c[40];
static double w;

int i=13;
static long l=2001;

main () {
    int j=3, k, *jp;
    jp = malloc (sizeof(j));
    c[5] = j;
    w = 2.0 * l;
}
```

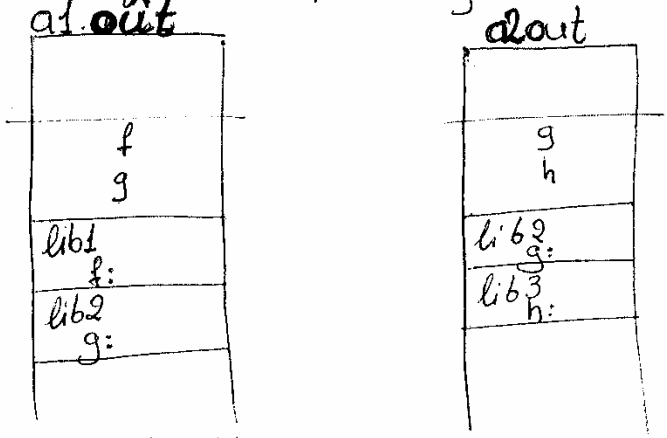


index	Name	size	type	bind	segment
29	w	8	OBJT	LOCL	.bss
42	c	40	OBJT	GLOB	.bss
34	i	4	OBJT	GLOB	.data
2	l	4	OBJT	LOCL	.data
9	main	88	FUNC	GLOB	.text
36	malloc	56	FUNC	GLOB	UNDEF

10/4/2006

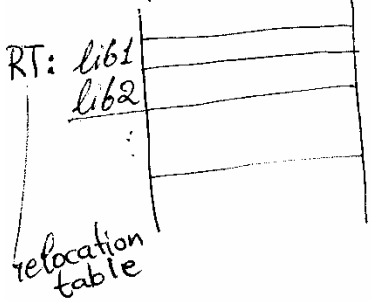
(1) Το πρόβλημα με το static linking είναι ότι κάθε executable περιέχει ένα αντίγραφο των βιβλιοθηκών που χρησιμοποιεί και επομένως χρειάζεται αρκετή μνήμη για να εκτελεστεί. (π.χ. 10s MBytes) Αν όμως σε ένα σύστημα έχουμε πολλούς χρήστες (processes) τότε ίσως να μην έχουμε αρκετή μνήμη.

→ Η λύση θα ήταν να έχουμε ένα αντίγραφο για όλα τα προγράμματα Αυτό επιτυγχάνεται με το dynamic linking.



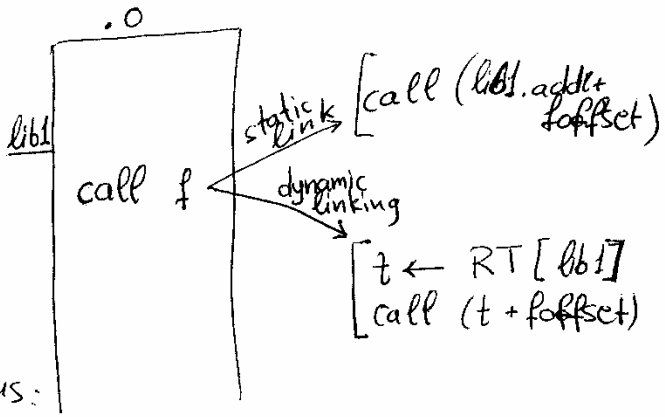
f() → lib1
g() → lib2
h() → lib3

Πρέπει να βρούμε κάποιον τρόπο ώστε να μην χρειάζεται να γνωρίζουμε κατά το linking τις διευθύνσεις/θέσεις στη μνήμη των βιβλιοθηκών. Λύση 1



Προσθέτουμε στο executable έναν πίνακα ο οποίος όταν το πρόγραμμα τοποθετηθεί στη μνήμη, θα περιέχει την αρχική διεύθυνση στη μνήμη κάθε βιβλιοθήκης. (Relocation table)

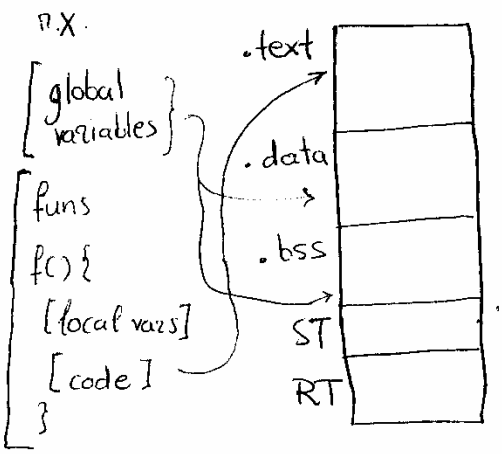
→ Όταν ένα πρόγραμμα φορτώνεται στη μνήμη, το σύστημα τοποθετεί στον πίνακα αυτό την αρχική διεύθυνση κάθε βιβλ. που υπάρχει στη μνήμη του συστήματος εκείνη τη στιγμή. Ο 'dynamic linker' παράγει έναν a.out αρχείο όπου:
Κάθε προσπέλαση ενός global συμβόλου μετατρέπεται σε δύο προσπελάσεις:
(i) διάβασε από τον RT την αρχική διευθ. ενός lib. που περιέχει το σύμβολο.
(ii) χρησιμοπ. τη διευθ. αυτή και το offset του συμβόλου για να διαβάσεις/γράψεις/εκτελέσεις το σύμβολο.
- Οπότε στο δυναμικό linking κάθε αναφορά σε εξωτερικό σύμβολο μετατρέπεται σε περισσότερες αναφορές μνήμης.



(2) Στοίβα + Αναδρομή

a.out

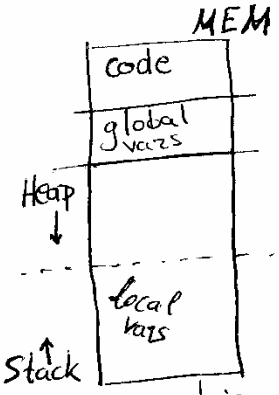
π.χ.



- Οι local μεταβλητές υπάρχουν μόνο όσο εκτελείται μια συνάρτηση.
- Οι τοπικές μεταβλητές είναι "private" για κάθε εκτέλεση μιας συνάρτησης.

→ Για να τηρήσουμε αυτούς τους περιορισμούς οι τοπικές μεταβλητές, βρίσκονται, κατά την εκτέλεση του προγράμματος, σε ένα ειδικό τμήμα της μνήμης που ονομάζεται στοίβα. (Η στοίβα δεν εμφανίζεται παθενά στο a.out.)

- Όταν ένα πρόγραμμα αρχίζει να εκτελείται το σύστημα του δίνει ένα τμήμα μνήμης όπου τοποθετούνται οι local variables. Κατα σύμβαση η στοίβα βρίσκεται στο τέλος της μνήμης (μεγάλες διευθύνσεις) και μεγαλώνει προς την ανάποδη κατεύθ. που μεγαλώνει το Heap.

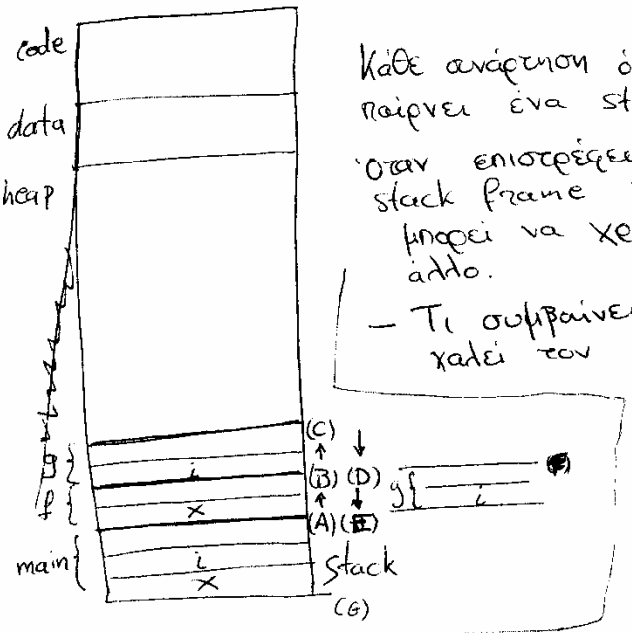


- Μεταβολή του stack κατά την εκτέλεση

```
main() {
    int x, i;
(A) f();
(E) g();
    return;
}(G)

f() {
(B) int x;
(D) g();
    return;
}

g() {
(C)(F) int i;
    return;
}
```



Κάθε ανάρτηση όταν αρχίσει να εκτελείται παίρνει ένα stack frame στη στοίβα.

Όταν επιστρέφει μια συνάρτηση το stack frame της ελευθερώνεται και μπορεί να χρησιμοποιηθεί από κάποιον άλλο.

— Τι συμβαίνει αν μία συνάρτηση καλεί τον εαυτό της (αναδρομή)?

Άμεση `f(){ f(); }`

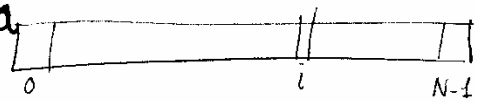
Έμμεση `f(){ g(); }` `g(){ f(); }`

π.χ. υλοποίηση quicksort: Ο αλγόριθμος έχει τα εξής βήματα:

Έστω ότι θέλουμε να διατάξουμε ένα array a με 0 . . . N-1 στοιχ.

① Χώρισε το a σε δύο υποarrays a_{low}, a_{high} με βάση κάποιο στοιχείο e του a, έτσι ώστε -κάθε στοιχείο του $a_\ell < e$, $a_h \geq e$

② Διάταξε το a_ℓ ③ Διάταξε το a_h




```
void quicksort(int a[], int low, int high)
{ if(high ≤ low) return;
  i = split(a, low, high, a[low])
  quicksort(a, 0, i-1)
④ quicksort(a, i+1, N-1);
  return;
}
```

Η στοίβα, σε κάθε χρονική στιγμή περιέχει ένα stack frame για κάθε συνάρτηση που είναι "ενεργή". Η στοίβα ταυτίζεται κάθε στιγμή με ένα μονοπάτι του δέντρου.

Άσκηση 5

(1) Γράφουμε συναρτήσεις που διαβάζουν και κάνουν output ένα puzzle
(i) read / print

* 1-9 ή 0 "άδεια θέση"

* ␣ * ␣ * ␣ * ␣ * ␣ * ␣ * ␣ * ␣ * ␣ \n

(ii) **check**

rules: (α) ένας αριθμός δεν εμφανίζεται 2 φορές σε κάποια γραμμή & κάνενα 0
(β) στήλη.. (γ) grid..

(2) solve

array [81][9] to counter

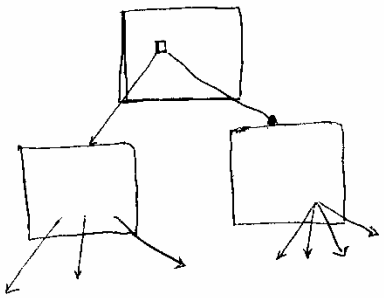
init ∀ elt ≠0 επιλογές = αμή. ∀ elt = 0 επιλογές = {1,9}
∀ elt ≠0 eliminate_element (g, i, j)

(i) Υπάρχει κάποιο στοιχείο που έχει μία μόνο επιλογή :

main () { read; init; solve; }
solve () { while (next) { updat_puzzle; eliminate_element; } }

(1) διαλέγουμε ένα τέτοιο στοιχείο
(2) το προσθέτουμε στο grid
(3) eliminate_element
• επαναλαμβάνουμε εως ότου ολοκληρώσουμε το puzzle (ή δεν έχουμε επιλογές).

(ii) (backtracking) δεν υπάρχει κανένα στοιχείο που έχει μία μόνο επιλογή

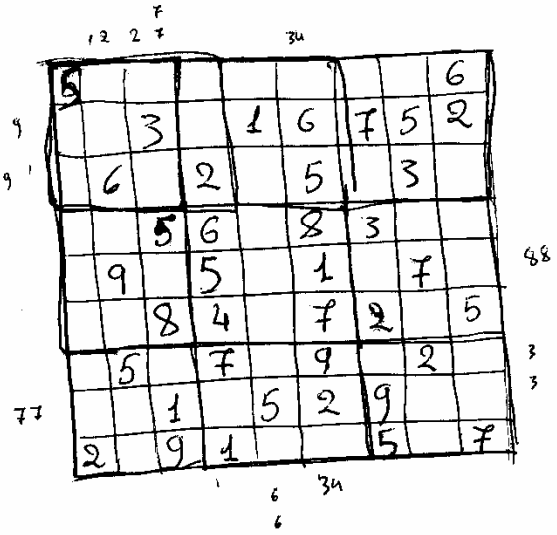


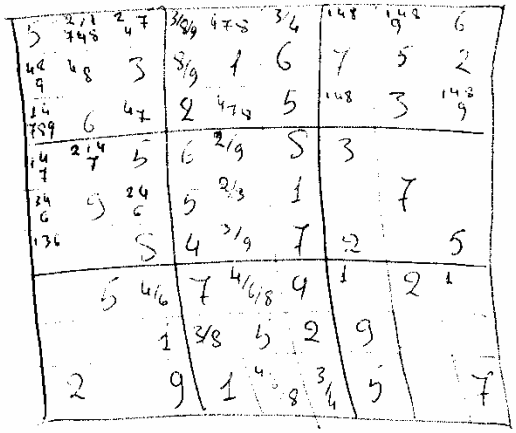
```
sudoku_solve ( g ) {
    (i,j,n) = next (g);
    g' = g + (i, j, n);
    sudoku-solve ( g' );
    if ( g' correct )
        return g';
    else
        return g;
}
```

} while(next)

---

void srand (int) → getpid()
int rand ()





3/5/2006 - debugging
- gdb

(1) Πάντοτε πρέπει να καταλαβαίνουμε τον κώδικα που γράφουμε.
(i) σκεφτόμαστε τον κώδικα που γράφουμε (ii) offline code review (διαβάζουμε εκτύπωση του κώδικα)
(iii) εξηγούμε τον κώδικα σε κάποιον άλλο.
(2) Αν δεν καταναοούμε κάτι, δεν το αναβάλουμε για αργότερα.
(3) Έστω ότι προσπαθούμε να βρούμε κάποιο λάθος: (i) κάνουμε το σφάλμα reproducible δηλ. κάθε φορά που τρέχουμε το πρόγραμμα να συμβαίνει το λάθος.
(ii) προσπαθούμε να βρούμε την απλούστερη περίπτωση που συμβαίνει το λάθος, (input, αφαιρώντας κώδικα κλπ.) (iii) προσθέτουμε assertions τα οποία ελέγχουν συνθήκες που νομίζουμε ότι τηρεί το πρόγραμμα μας.
(iv) → χρησιμοποιούμε printf για να δούμε ~~τιμές~~ μεταβλητές, τη ροή ελέγχου κλπ.
→ χρησιμοποιούμε κάποιο debugger για τον ίδιο σκοπό
↳ όχι για να καταλάβουμε το πρόγραμμα αλλά για να βρούμε γιατί δεν ισχύουν οι συνθήκες που έχουμε στο μυαλό μας.
(v) κρατούμε μια λεπτομερή λίστα με τις περιπτώσεις / τεκμήρια που έχουμε δοκιμάσει.

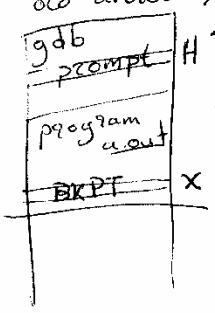
Τι κάνουν οι debugger Είναι προγράμματα που μας βοηθούν:

(α) να σταματήσουμε την εκτέλεση του προγράμματος σε ~~ε~~ οποιοδήποτε σημείο.
(β) να εξετάσουμε τα περιεχόμενα των μεταβλητών / καταχωρητών
(γ) να εξετάσουμε τα περιεχόμενα ενός στοίβας
(δ) να δούμε το πρόγραμμα στη μνήμη σε μορφή assembly.

(α) breakpoints Σε κάθε debugger υπάρχει η δυνατότητα να σταματήσουμε την εκτέλεση σε οποιαδήποτε εντολή του source κώδικα. π.χ. >gdb a.out

Για την υλοποίηση των breakpoints κάθε CPU διαθέτει μια ειδική εντολή (έστω BKPT) η οποία όταν εκτελείται προκαλεί ένα trap/interupt/exception. Το αποτέλεσμα του trap είναι να αρχίσει να εκτελείται ένα άλλο κομμάτι κώδικα από το πρόγραμμα που έτρεχε εκείνη τη στιγμή. Αυτό το κομμάτι κώδικα ανήκει στον debugger. Για κάθε breakpoint που θέλει ο χρήστης ο debugger εισάγει μια εντολή BKPT στο αντίστοιχο σημείο του κώδικα.

```
gdb> list fname  π.χ. main
1 main()
2 ...
3 ...
brkpoint (bl 1
```



→ ο debugger διαβάζει το prog στη μνήμη και ορίζει ότι ο κώδικας που εκτελείται για την εντολή BKPT είναι στην διεύθυνση H (trap handler)
→ Όταν ο χρήστης βάζει ένα bkpt στη διεύθυνση X, ο debugger αντικαθιστά την εντολή στην X με bkpt, και αποθηκεύει την πραγματική εντολή του προγράμματος.
→ Όταν συνεχίζουμε την εκτέλεση του προγράμματος (μέχρι το επόμενο bkpt) ο debugger (handler) επιστρέφει από το trap που έχινε αρχικά.

β) Εκτύπωση μεταβλητών
Αν το πρόγραμμα έχει σταματήσει σε κάποιο σημείο της εκτέλεσης ο debugger μας επιτρέπει να εξετάσουμε τις τιμές των global + local μεταβλητών
π.χ. gdb> print (p) x
i
a[i]
a[0]
(int)(int)(&x)

```
>gdb a.out
gdb> b X
gdb> run
gdb> continue (c)  b Y
```

i) global vars Το a.out περιέχει στο symbol-table πληροφορίες (size, address) για κάθε global var.
Άρα ο gdb διαβάζει το ST το (size, address) και τυπώνει τα περιεχόμενα της μνήμης.

ii) local vars Όταν θέλουμε να κάνουμε debug ένα πρόγραμμα πρέπει να το κάνουμε compile με μια ειδική παράμετρο (για τον gcc το "-g") που λέει στον compiler να τοποθετήσει στο a.out πληροφορίες για debugging, π.χ. (size, addr, scope) για τοπικές μεταβλητές. Αυτές οι πληροφορίες βρίσκονται στο DEBUG segment του a.out.

Εκτύπωση καταχωρητών

gdb > p $pc
      $sp

} δείχνουν τις τιμές των καταχωρητών τη στιγμή που εκτελέστηκε η εντολή BKPT
⇓
ο debugger σε κάθε BKPT κρατάει τις τιμές των καταχ.

gdb > set x = 5;
      set *($pc) = 7

γ) Περιεχόμενα της στοίβας

gdb > bt (backtrace) τυπώνει σε ποια συνάρτηση ανήκει κάθε stack frame που βρίσκεται στη στοίβα.

gdb > up / down : αλλάζουν το τρέχον stack frame
το μέγεθος του stack frame είναι πάντα το ίδιο για όλες τις κλήσεις μιας συνάρτησης και ο gdb μπορεί να το διαβάσει από το debug segment.

δ) disassemble source code - η εντολή list μας δείχνει τον κώδικα του προγράμματος από το source αρχείο (σε C)
- Στη μνήμη του συστήματος, υπάρχει το πρόγραμμα σε machine code.
gdb > disassemble main : 0x ffcc0000 add...

>step >next (δεν μπαίνει μέσα σε συναρτήσεις)
>step i >next i (assembly εντολή)

5/5/2006

Όταν εκτελείται ένα πρόγραμμα που βρίσκεται στη μνήμη σε μορφή machine code είναι βολικό να το εξετάζουμε σε μορφή assembly. Αυτό μπορεί να το κάνει για μας ο debugger.

assembly για x86

Τα βασικά στοιχεία σε κάθε γλώσσα assembly είναι:
(1) ονόματα των εντολών (2) η μορφή των παραμέτρων (3) Χειρισμός της στοίβας.
(4) πρόσβαση στη μνήμη κ ίσως ιδιαιτερότητες της αρχιτεκτονικής.

a) εντολές σε x86. Η x86 υποστηρίζει εκατοντάδες εντολές.
Μερικές από τις βασικές είναι:
movl : μετακινεί δεδομένα από κάτι σε κάτι άλλο.
addl : προσθέτει τα δύο args.
call : καλεί υπορουτίνα
ret : επιστρέφει από υπορουτίνα
jump : αλλάζει την ροή του προγράμματος

κάθε εντολή μπορεί να έχει παράμετρο από τους εξής τύπους:
(1) έναν αριθμό/σταθερά (π.χ. $1, $726)
(2) μια θέση μνήμης (1 → θέση μνήμης με addr 1)
(3) ένα καταχωρητή ειδικές θέσεις μνήμης μέσα στη CPU
γενικοί: %eax, %ebx, %ecx, %edx, %esi, %edi
ειδικοί: %esp, %ebp (στοίβας)

π.χ. movl : δύο arguments + μεταφέρει το πρώτο στο δεύτερο

```
movl $0, %eax   # ο καταχ. eax θα πάρει την τιμή 0
movl 0, %eax    # eax ← MEM[0]
movl %eax, %ebx # ebx ← eax
movl 3, 7       # MEM[7] ← MEM[3]
addl %eax, %ebx # arg2 ← arg2 + arg1   %ebx ← %eax + %ebx
addl $3, %eax   # %eax ← %eax + 3
addl 5, 7       # MEM[7] = MEM[5] + MEM[7]
```

cmpl : 2 arguments, συγκρίνει, αποθηκεύει κάπου το αποτέλεσμα της σύγκρισης (σε ένα εσωτερικό καταχωρητή)
je : 1 arg που είναι η διευθ. προορισμού ανάλογα με το αποτέλεσμα της τελευταίας σύγκρισης (χρονικά), εκτελεί την αντίστοιχη εντολή

jg / jge . jmp addr # αλλάζει τη ροή ανεξάρτητα από συνθήκες.

π.χ.

C προγρ. που βρίσκει τον μέγιστο δυο αριθμών

```
int a = 10;
int b = 12;
int main (void){
    int t;
    if (a ≥ b)
        t = b;
    else
        t = a;
    printf ("max is %d \n", t);
    return 0;
}
```

`gcc -S max.c ⇒ max.s`

`gcc -S -O3 max.c`

-n.x. optimization level

```
    .section .dat
a:  .long 10
b:  .long 12
msg: .ascii "max is %d\n\0"
    .globl main
# t is in %ecx
main:
    movl a, %eax
    movl b, %ebx
    cmpl %eax, %ebx
    jge  Lb≥a
    movl %eax, %ecx
    jmp  Lprint
Lb≥a: movl %ebx, ecx
Lprint: pushl %ecx
    pushl msg
    call  printf
    popl  %ecx
    popl  %ecx
    movl  $0, %eax
    ret
```

→ εάν το δεύτερο arg είναι ≥ του πρώτου

→ Χρήση gdb για να εξετάσουμε προγράμματα κ δεδομένα στη μνήμη

(1) Για να δούμε τον κώδικα του προγράμματος

(gdb) x/i addr

εξέτασε τη θέση addr και δείξε το αποτέλεσμα σαν εντολή assembly

x/??i # ??: πόσες θέσεις μνήμης
10

x/20i main

0x . . . :

(2) x/??b addr # εξέτασε ?? θέσεις μνήμης από τη διευθ. addr και εύπωσέ τες ως bytes

0x . . . : ab
0x . . . . : 03
0x . . . . . : 07

(3) print *(int *)a

8/5/2006 HY-255

(1) x86 stack

Έστω η απλή κλήση συνάρτησης:

```
main(){          void f(int x, int y){
   .                .
   .                .
   f(1,2);          return;
   .                }
   .
}
```

Η στοίβα σε x86 περιλαμβάνει 2 ειδικούς καταχωρητές: τους %ebp, %esp

esp: δείχνει στην τελευταία (κορυφή) θέση της στοίβας που χρησιμοποιείται (μικρότερη διευθ.)

ebp: δείχνει στη βάση του stack frame της τρέχουσας συνάρτησης (μεγαλ. διευθ)

- Κατά την κλήση μιας συνάρτησης έχουμε τις εξής φάσεις / βήματα:

(1) κλήση της f από τη main

πρέπει να:

(1.1) περάσουμε τις παραμ. της f στη στοίβα | παράμετροι περνιούνται με τις εντολές push l, pop l

push l arg : αντίγραφο στο arg στην κορυφή της στοίβας και αύξηση του esp. (movl arg, (esp) / esp--)

pop l arg : αφαιρεί την κορυφή της στοίβας και αυξάνει τον esp

Άρα όταν η main καλεί την f : main:

```
.
pushl $1
pushl $2
call f      → label
```

(1.2) να αποθηκ. την διευθ. επιστροφής } call addr

(1.3) να πραγματοποιήσουμε την κλήση

Τα βήματα 1.2, 1.3 γίνονται με την εντολή "call addr": Η return αποθ. στην κορυφή της στοίβας (και προσαρμόζεται ο stack pointer)

(2) Αρχή εκτέλεσης της f. Οι λειτουργίες είναι:

(2.1) αποθήκευση του ebp στην κορυφή της στοίβας και τον αλλάζει ώστε να δείχνει στη νέα θέση.

(2.2) δεσμεύει χώρο για τις τοπικές της μεταβλητές αλλάζοντας τον esp

(2.1) pushl %ebp # ο ebp στη στοίβα + αλλαγή του esp

(2.2) Αν π.χ. η f χρειάζεται 8 bytes subl $8, %esp

(3) επιστροφή της f

(3.1) απελευθέρωση χώρου: addl $8, %esp

(3.2) επαναφορά ebp: popl %ebp # ο ebp παίρνει την παλιά τιμή και προσαρμόζεται ο esp.

(3.3) return: ret # jump στη διευθ. στην κορυφή της στοίβας και pop την retaddr

(4) στη main μετά την επιστροφή της f. - ελευθερώνουμε τον χώρο των παραμέτρων

```
popl %ecx   1η
popl %ecx   2η

main:                    f: pushl %ebp
   push $1                  subl $8, %esp
   push $2                  .
   call f                   addl $8, %esp
   popl %ecx                popl %ebp
   popl %ecx                ret  → movl $retval, %eax
```

Όταν βρισκόμαστε σε μια συνάρτηση μπορούμε να προσπελάσουμε:

a) τις local vars ως esp + offset

b) τις input params, τον παλιό ebp, κ raddr ως ebp + offset

(2) Buffer Overrun Attacks

Internet Worm, end of 1980's
- ένα μικρό πρόγραμμα, που "μετέδιδε" τον εαυτό του από σύστημα σε σύστημα
- δεν έκανε τίποτε επικίνδυνο, απλά (α) βρες ένα host + transmit (infect)
  (β) spin (κρατά τη CPU)

(1) Ιδέα μηχανισμός. Έστω ότι έχουμε ένα πρόγραμμα που διαβάζει input.
- Έστω ότι το input το διαβάζει κάποια συνάρτηση f.
  - η f τοποθετεί το input σε κάποιον τοπικό buffer buf
- Έστω ότι η f δεν εξετάζει το max size του input
- Φροντίζουμε ώστε το input που δίνουμε να είναι ένα σωστό πρόγραμμα σε εντολές μηχανής.
- Φροντίζουμε να κάνουμε overwrite την raddr με μια άλλη διεύθυνση
- η f returns

→ Κάθε (unix) σύστημα τρέχει ένα finger-deamon

```
finger_deamon(){  char login[8];
    while (1) {
        read_login_from_net(login);
        r=lookup(login);
        send_reply(r);
    }
}
```

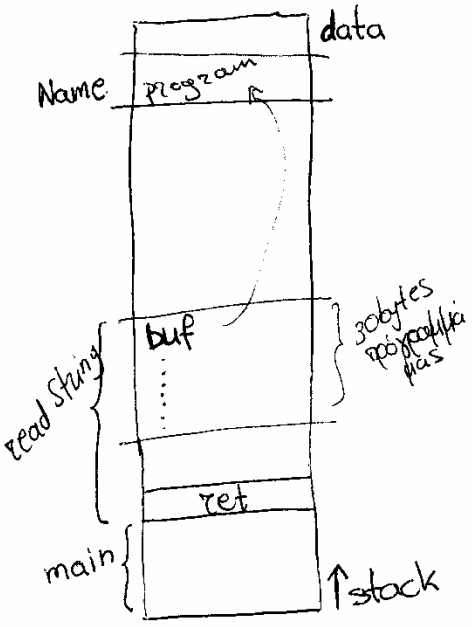
η read_login_from_net διάβασε ένα login που ήταν ένα πρόγραμμα >8 bytes
Έτσι return άρχισε να εκτελείται το πρόγραμμα.

15/5/2006 Άσκηση 1

```
char Name[30];
int main(void) {
    ⋮
    read_String(Name);
    ⋮
    return 0;
}
```

```
void read_String(char *s){
    char buf[80];
    int i;
    char c;

    while(1){
      c=getchar();
      if ((c==EOF)||(c=='\n'))
          break;
      buf[i] = c;
      i++;
    }
    buf[i]='\0';
    for(i=0; i<80; i++) { s[i]=buf[i];}
    return;
}
```



(1) "Κανονικό πρόγραμμα"
- δεν υπάρχει έλεγχος για buf size
- χρησιμοποιούμε την local buf μέν περάσουμε το input στη global Name/s

(2) Πρέπει να γράψουμε ένα σωστό πρόγραμμα σε κώδικα μηχανής.

→ Θέλουμε να γράψουμε ένα πρόγραμμα που:
add %eax, %ebx —compile→ exec που το περιέχει —gdb→

gdb → exec στη μνήμη —x/i→ βρίσκουμε τη θέση στη μνήμη —x/b→ βρίσκουμε την αναπαράσταση σε machine code.

(3) μέρη του προγράμματος

<30b { (i) η λειτουργεία που κάνει το πρόγραμμα
- να αλλάξουμε τιμές μεταβλητών
- να κάνουμε jump σε "σωστά" σημεία του προγρ.
(ii) να αρχίσει η εκτέλεση του προγράμματός μας.

Χειρισμός Λαθών

• Μέχρι τώρα, οι συναρτήσεις, επιστρέφουν είτε την τιμή που ορίζει η λειτουργία της συνάρτησης είτε κάποια τιμή λάθους.

```
int f() { ... if(malloc( ) == NULL) {
```

→ λάθος <(1) τυπώνουμε μήνυμα + exit(ERROR_FAILURE);
<(2) επιστρέφουμε στον caller της f κάποιο κωδικό λάθους κ αποφασίζει αυτός για το τι θα γίνει.

→ Η πιο ρεαλιστική είναι η επιλογή (2). Δηλαδή, οι συναρτήσεις επιστρέφουν κωδικούς ανάλογα με το τι έχει συμβεί κ σε κάποιο σημείο του προγράμματος (π.χ. main) ελέγχουμε κ' χειριζόμαστε όλους τους κωδικούς λάθους.

π.χ. αν έχουμε κώδικα ενός προγραμ. που επεξεργάζεται requests.

```
while(1){
    ret = process_request();
    switch(ret){
    case ...
    :
    case ...
    default ...
}
```

→ Παρατηρούμε ότι αφιερώνουμε πολύ κώδικα για το χειρισμό "λαθών" → μη κοινές περιπτώσεις, π.χ. έλλειψη μνήμης.

→ Έστω ότι έχουμε και nested συναρτήσεις

```
f(){ f1(); }
f1(){ f2(); }
f2(){ f3(); }
f3(){ f4(); }
:
```

Όταν έχουμε nested συναρτήσεις (και ειδικά σε μεγάλο βάθος) • i) ελέγχουμε αν κάθε κλήση δούλεψε σωστά ii) αν όχι επιστρέφουμε στο δικό μας caller.

```
f(){
  ret = f1();
  if (ret) {
    return;
  }
}
```

Τα exceptions είναι ένας μηχανισμός που μας επιτρέπει να υλοποιήσουμε τα Α, Β με πιο απλό τρόπο, από το προηγούμενο παράδειγμα.

→ Ένα exception είναι ισοδύναμο με ένα error case

→ Τα exceptions υποστηρίζουν τους εξής 3 μηχανισμούς.

a) try/catch exception : ορίζουμε κώδικα που χειρίζεται το λάθος.

b) raise exception : Στο σημείο που συμβαίνει το λάθος "σηκώνουμε" ένα exception που χαρακτηρίζει το αντίστοιχο λάθος.

→ Οπουδήποτε συνάρτηση μπορεί :

```
f3() {
  f4();
}
f4() {
  if(error1) raise exception OUT_OF_MEM;
  :
}
```

```
while(1){
 try {
   process_request();
 }
 catch (except 1) {
 }catch (OUT_OF_MEM){
   :
 }
```

π.χ. διαβάζουμε από το stdin με exceptions

```
void read_input() {
        try {
            while(1) {
                c = getchar();
            }
        } catch (EOF) {
            /* reached EOF ; do nothing ; return */
        }
        return;
}
```

"ΛΑΘΟΣ" = Εδώ στην συνηθισμένη μέρος του προγράμματος * όχι! δίδει το EOF είναι μέρος διαδικασίας