HY255 Εργαστήριο Λογισμικού – L10

(a) Type conversions
(b) Polymorphic functions

HY255 Εργαστήριο Λογισμικού
Άγγελος Μπίλας

Ισοδυναμία τύπων

- Στις εκφράσεις και στους διάφορους τελεστές μπορούμε γενικά να χρησιμοποιούμε τύπους που είναι “ισοδύναμοι”
  - Οι ίδιοι τύποι είναι ισοδύναμοι και άρα μπορούμε να γράψουμε π.χ. int x = 3 + 4;
- Έστω όμως η έκφραση: float x = 3 + 5.2 + 3 / 7;
  - Πως επιτρέπεται να γράψουμε αυτή την έκφραση αφού οι διάφορες τιμές/μεταβλητές έχουν διαφορετικούς τύπους;
- Ας ορίσουμε πρώτα ποιοι τύποι θεωρούνται ισοδύναμοι
- Γενικά υπάρχουν δύο μορφές ισοδυναμίας τύπων στις γλώσσες
  - Name equivalence (ισοδυναμία κατά όνομα): Δύο τύποι θεωρούνται ισοδύναμοι μόνο όταν έχουν το ίδιο όνομα
    - Στην ισοδυναμία κατά όνομα ο compiler συγκρίνει τα ονόματα των τύπων για να αποφασίσει αν είναι ισοδύναμοι (απλό)
  - Structural equivalence (ισοδυναμία κατά δομή): Δύο τύποι θεωρούνται ισοδύναμοι όταν έχουν την ίδια δομή
    - Στην ισοδυναμία κατά δομή, ουσιαστικά ο compiler κατασκευάζει ένα δέντρο που αναπαριστά τον τύπο και συγκρίνει τα δέντρα (δομή) αντί για τα ονόματα
  - Π.χ. Αν χρησιμοποιήσουμε το typedef με την 2η χρήση του να ορίσουμε ένα άλλο όνομα για τον τύπο int *:

```
typedef int *intp_T;
```

  - Τότε μπορούμε να γράψουμε:

```
int *(a[10]);
intp_T b[10];
```

  - Οι μεταβλητές a, b έχουν ισοδύναμους τύπους με βάση την ισοδυναμία κατά δομή, αλλά όχι με βάση την ισοδυναμία κατά όνομα
- Η C υποστηρίζει ισοδυναμία κατά δομή, με εξαίρεση τα structs που υποστηρίζει ισοδυναμία κατά όνομα, αλλά μας επιτρέπει να ορίζουμε συνώνυμα με το typedef, π.χ.

```
struct s1 {int x; int y;} v1;
struct s2 {int x; int y;} v2;
```

  - Τα struct s1 και struct s2 δεν είναι ισοδύναμοι τύποι στη C, επειδή έχουν διαφορετικό όνομα, παρ΄ότι έχουν την ίδια δομή. Επομένως δεν μπορούμε να γράψουμε π.χ. v1 = v2;
  - Αλλά μπορούμε να πούμε: typedef struct s1 structS1_T;
  - Και μετά, τα struct s1 και structS1_T, είναι ισοδύναμοι τύποι, παρότι έχουν διαφορετικά ονόματα.

Μετατροπές τύπων

- Στην έκφραση 3 + 5.2 + 3 / 7 ωστόσο, δεν έχουν ισοδυναμία τύπων. Οι ints είναι διαφορετικός τύπος από τους floats.
- Η γλώσσα, ορίζει λοιπόν ότι συγκεκριμένοι τύποι μπορούν να μετατρέπονται αυτόματα και έμμεσα/implicitly (χωρίς να πει τίποτε ο προγραμματιστής ή το πρόγραμμα) σε έναν άλλο τύπο
- Πότε γίνονται έμμεσες μετατροπές τύπων; Σε τέσσερις περιπτώσεις:
  - (1) Όταν τα operands σε λογικές ή αριθμητικές εκφράσεις δεν έχουν ισοδύναμους τύπους
  - (2) Σε assignments όταν η έκφραση δεξιά του = δεν έχει ισοδύναμο τύπο με το όνομα/έκφραση στα αριστερά
  - (3) Κατά το πέρασμα των input παραμέτρων σε κλήσεις συναρτήσεων, όταν οι πραγματικές (actual) παράμετροι δεν έχουν ισοδύναμο τύπο με τις τυπικές (typical) παραμέτρους της συνάρτησης
  - (4) Σε return expressions κατά την επιστροφή τιμών από συναρτήσεις, όταν δεν είναι ισοδύναμος ο τύπος με τον τύπο της τιμής επιστροφής
- Ποιοι τύποι επιτρέπεται να μετατραπούν έμμεσα; (a) αριθμητικοί τύποι και (b) pointer τύποι

Expressions: Usual arithmetic conversions

- Στις εκφράσεις γίνονται έμμεσες μετατροπές, από τον compiler χωρίς να ορίσει κάτι το πρόγραμμα
- Γιατί θέλουμε εκφράσεις της μορφής 3 + 5.2 + 3 / 7;
  - Διότι είναι βολικό για τον προγραμματιστή – δεν έχει νόημα να απαγορέψουμε κάτι που είναι τόσο συνηθισμένο σε όσους προσπαθούν να λύσουν προβλήματα
- Γιατί δεν μπορούμε να έχουμε τελεστές/operators, π.χ. +, που να παίρνουν operands διαφορετικών τύπων, π.χ. το πρώτο operand να είναι int και το δεύτερο float;
  - Διότι οι operators τελικά αναπαριστούν εντολές τους επεξεργαστή και δεν είναι αποδοτικό να έχουμε τέτοιους operators
- Οπότε καταλήγουμε ότι οι operators έχουν operands του ίδιου τύπου, π.χ. πρόσθεση από δύο ints ή δύο floats, και ο compiler κάνει τις έμμεσες μετατροπές σε συγκεκριμένα σημεία και με καλά ορισμένο τρόπο
- Ο γενικός κανόνας είναι: Όλα τα operands σε εκφράσεις μετατρέπονται στον ”στενότερο” (narrowest) τύπο που μπορεί να αναπαραστήσει τις τιμές των operands.
- Αυτό οδηγεί στο να γίνεται μετατροπή από ”μικρούς” σε “μεγαλύτερους” τύπους, η μετατροπές αυτές ονομάζονται type promotions

Type promotions

- (1) Όταν τουλάχιστον ένα από τα δύο operands έχει κάποιο floating τύπο

```
float → double → long double
```

  - Π.χ. Αν ένα operand είναι float και το άλλο double, ο float θα γίνει double
- (2) Όταν κανένα operands δεν έχει floating τύπο
- short or char → int → unsigned int → long int → unsigned long int
  - Π.χ. Αν ένα operand είναι int και το άλλο long int, ο int θα γίνει long int
  - Αν ένα operand είναι short και το άλλο char, θα γίνουν και τα δύο int
- Θέλει ιδιαίτερη προσοχή η (αυτόματη) μετατροπή των int σε unsigned, π.χ.

```
int i=-10;
unsigned int j=2;
if ((i+j) < 0) {...}
```

  - Θα περιμέναμε το αποτέλεσμα της σύγκρισης (i+j)<0 να είναι true
  - Λόγω των έμμεσων μετατροπών το i θα γίνει unsigned int στην έκφραση i+j
  - Οπότε το i+j θα είναι ένας μεγάλος θετικός αριθμός και το αποτέλεσμα θα είναι να μην εκτελεστεί το if clause
  - Αν αλλάζαμε τους κανόνες να γίνονται οι unsigned int → int, θα είχαμε παρόμοια θέματα σε άλλες περιπτώσεις
  - Το πρόβλημα ξεκινά από το ότι οι τύποι αυτοί έχουν το ίδιο μέγεθος (σε bit) αλλά αναπαριστούν διαφορετικό εύρος τιμών ο καθένας

Assignments / Function call, return

- Διαφορετικός κανόνας: Ο compiler μετατρέπει ότι υπάρχει στα δεξιά στον τύπο που υπάρχει στα αριστερά
- Αν ο τύπος στα αριστερά είναι ”μικρότερος” πάλι γίνεται η μετατροπή
  - Σε μερικές περιπτώσεις warning, αλλά όχι πάντα
  - Για ιστορικούς λόγους
  - Εμείς αυτό πρέπει να το θεωρούμε λάθος και κακή τακτική
    Οφείλουμε να γνωρίζουμε τους τύπους μας και πως γίνονται τα assignments και οι κλήσεις των συναρτήσεων στο πρόγραμμά μας
  - Αν πρέπει να το κάνουμε, πρέπει να ελέγχουμε πριν το assignment σε “μικρότερο” η τιμή που μετατρέπουμε να είναι μέσα σε όρια που αναπαρίστανται από τον “μικρότερο” τύπο

Άμεσες (explicit) μετατροπές αριθμητικών τύπων (type casting)

- Ας σκεφτούμε την εξής ειδική περίπτωση: float x = 3/2;
  - Ποια θα είναι η τιμή του x; Το 1.0 και όχι το 1.5 όπως ίσως θα ελπίζαμε…
  - Γιατί; Το / είναι ένας τελεστής που μπορεί να χρησιμοποιηθεί και για int και για float διαίρεση. Ωστόσο, επειδή και τα 2 operands είναι int, σε αυτή την περίπτωση θα κάνει int διαίρεση, άρα το αποτέλεσμα θα είναι 1. Στη συνέχεια ο int 1 θα μετατραπεί σε float 1.0 για να γίνει το assignment στο float x.
- Είναι σωστό αυτό; Σωστό είναι με βάση τον ορισμό την γλώσσας, αλλά δεν είναι βολικό…
- Πως θα μπορούσαμε να το διορθώσουμε; Π.χ. μπορούμε να γράψουμε: float x = 2; x = 3/x;
  - Αυτο θα μετατρέψει τον int 2 σε float κατά την ανάθεση στο x και μετά στο x/3, το 3 θα γίνει float και το / θα κάνει float διαίρεση με float αποτέλεσμα το 1.5
- Επειδή αυτό φαίνεται κάπως δυσνόητο όταν διαβάζουμε τον κώδικα, η C μας επιτρέπει να πραγματοποιούμε άμεσες (explicit) μετατροπές τύπων με έναν νέο τελεστή, τις παρανθέσεις (type), ως εξής: float x = x = 2 / (float)3;
  - Στην περίπτωση αυτή, το 3 θα μετατραπεί σε float και μετά τα υπόλοιπα βήματα θα γίνουν κατά τον επιθυμητό/βολικό τρόπο
- Οι άμεσες μετατροπές τύπων είναι το γνωστό μας type casting
- ΑΛΛΑ:
  - Γενικά δεν χρησιμοποιούμε explicit type conversions γιατί δεν θα έπρεπε να τις χρειάζεται το πρόγραμμά μας
  - Θα πρέπει να ορίζουμε τους τύπους μας και να γράφουμε τον κώδικά μας ώστε να μην χρειάζεται να μετατρέπουμε τύπους με τρόπο που ενδεχομένως είναι απρόβλεπτος. Κάθε φορά που χρησιμοποιείτε μια άμεση μετατροπή τύπου, θα πρέπει να αναρωτιέστε γιατί χρειάζεται και ποιος είναι ο λόγος που οι τύποι σας δεν ταιριάζουν με βάση τους ορισμούς της γλώσσας για τις πράξεις;
  - Ειδικές περιπτώσεις, όπως η παραπάνω, είναι λιγοστές, και σε αυτές τις περιπτώσεις η χρήση του type casting κάνει το πρόγραμμά μας πιο σαφές και ευανάγνωστο

Εξάσκηση

```
#include <stdio.h>
#include <limits.h>
int main(void){
          float f = 1000.0;
          int si = 0;
          unsigned ui = 0;
          printf("size of int, unsigned, float = %lu %lu %lu bytes\n",
                    sizeof(int), sizeof(unsigned), sizeof(float));
          printf("UINT_MAX=%u\n", UINT_MAX);
          printf("INT_MAX=%d\n", INT_MAX);
          printf("INT_MIN=%d\n", INT_MIN);
          /* be careful with implicit conversions between types in assignments */
          f = f * f * f * f; /* result is 10^12 which is > 32 bits */
          si = f;
          ui = f;
          printf("f=%f\n", f);
          printf("si=%d\n", si);
          printf("ui=%u\n", ui);
          if (si == ui) printf("this would be nice!\n");
          if (ui == UINT_MAX) printf("this would be nice too!\n");
          if (si == INT_MAX) printf(”and this one would be nice too!\n");
          /* arithmetic conversions occur in all expressions */
          si = -1;
          ui = 0;
          if (si > ui) printf("implicit conversion of signed to unsigned in > as well\n");
          else printf("no implicit conversion of signed to unsigned in >\n");
          return 0;
}
```

Πολυμορφισμός (Polymorphism)

- Ας σκεφτούμε τη συνάρτηση malloc
- Μπορούμε να την χρησιμοποιήσουμε με πολλούς τύπους, π.χ.

```
int *A;
float *F;
A = malloc(10*sizeof(int));
F = malloc(10*sizeof(float));
```

  - Τι είδους συνάρτηση είναι η malloc που μια φορά μας επιστρέφει τιμή int * και άλλη φορά float *
- Ουσιαστικά η malloc μπορεί να επιστρέψει μια τιμή που δεν είναι ενός μόνο τύπου, αλλά ανήκει σε ένα σύνολο τύπων
- Μερικές φορές λοιπόν χρειάζεται να αναφερθούμε όχι σε έναν μόνο τύπο, αλλά σε σύνολα από τύπους
- Η C δεν έχει κάποιο sophisticated σύστημα τύπων που να επιτρέπει arbitrary σύνολα τύπων όπως άλλες γλώσσες (πχ ML)
- Παρόλα αυτά μας επιτρέπει να χρησιμοποιούμε ένα (και μόνο ένα) σύνολο τύπων που περιλαμβάνει όλους τους τύπους “pointer σε …”
  - Επομένως ο τύπος void * ορίζει το μέγεθος (όσο το μεγεθος των pointers, για μας 4 bytes) και τις επιτρεπόμενες πράξεις (μόνο σύγκριση)
- Αυτό το σύνολο του δίνουμε το όνομα void *, π.χ.
  - void *p; # Το p είναι μια μεταβλητή τύπου “οποιοσδήποτε τύπος pointer σε …”
  - void *f(void *); # Η f είναι μια συνάρτηση που παίρνει μια παράμετρο και επιστρέφει μια τιμή, όπου και οι δύο έχουν τύπο “οποιοσδήποτε τύπος pointer σε …”
- Σημείωση: Το void δεν έχει καμιά σχέση με το void *
  - void = ένα keyword που σημαίνει ότι μια συνάρτηση δεν παίρνει παραμέτρους ή δεν επιστρέφει τιμή, ώστε να είναι σαφής η προθεσή μας όταν γράφουμε τέτοιες συναρτήσεις
  - void * = ένας τύπος (σύνολο τύπων) που ελέγχεται από το type system της γλώσσας
- Συναρτήσεις που οι τιμές επιστροφής ή οι παράμετροί του μπορούν να είναι διαφόρων τύπων ονομάζονται πολυμορφικές
  - Στη C αυτές οι συναρτήσεις αναγκαστικά δηλώνονται με void * στον αντίστοιχο τύπο
- Οπότε η malloc είναι μια πολυμορφική συνάρτηση που δηλώνεται ως
  - `void *malloc(int);`

Έμμεσες μετατροπές τύπων ”pointer σε …”

- Ο τύπος void * είναι ένας νέος τύπος. Μπορούμε να γράψουμε;

```
int *A;
A = malloc(10 *sizeof(int));
```

- Αν οι τύποι int * και void * είναι ισοδύναμοι θα μπορούσαμε, αλλά δεν είναι
- Αλλά: Η C ορίζει ότι γίνονται έμμεσες μετατροπές τύπων και σε μεταβλητές τύπου pointer
- Οποιοσδήποτε τύπος “pointer σε …“ μπορεί να μετατραπεί σε void * και το ανάποδο, έμμεσα, χωρίς να ορίσει κάτι το πρόγραμμα
- Άρα τελικά, λόγων των έμμεσων μετατροπών τύπων, μπορούμε να γράψουμε: int *A = malloc(10 *sizeof(int));
  - Σημείωση: Θα δείτε ότι μερικές φορές, κάποιοι προτιμούν να γράφουν:

```
A = (int *)malloc(10 *sizeof(int));
```

  - Για να είναι σαφής η μετατροπή που γίνεται σε int * και να βλέπουμε με ποιον τρόπο θα χρησιμοποιηθεί η μνήμη που μας δίνει η malloc
- Οι έμμεσες μετατροπές τύπων pointer γίνονται στα ίδια σημεία όπως και για τους αριθμητικούς τύπους (εκφράσεις, αναθέσεις, συναρτήσεις)
- Αντίστοιχα λοιπόν με τις αριθμητικές μετατροπές, θα πρέπει να γράφουμε τα προγράμματά μας με τρόπο που οι τύποι pointer να είναι ορισμένοι σωστά και να μην χρειάζονται explicit μετατροπές (type casting) παρά μόνο σε ειδικές περιπτώσεις που θα πρέπει να τις κατανοούμε προσεκτικά

Εξάσκηση

- Κοιτάξτε πάλι την 2η άσκησή σας και τις μετατροπές τύπων που γίνονται στα σημεία που χρησιμοποιείται δηλώσεις “char *s” και “char const *s”
  - Εξετάστε τι σημαίνουν τα warnings που βλέπετε
  - Σκεφτείτε αν είναι εφικτό να μην υπάρχει κανένα warning

Παράδειγμα πολυμορφικής συνάρτησης: qsort

- Στην standard βιβλιοθήκη της C υπάρχουν πολλές πολυμορφικές συναρτήσεις σαν την malloc, π.χ. η qsort
- qsort: διατάσσει ένα array από στοιχεία οποιουδήποτε τύπου
- Πως ορίζεται και πως καλείται η qsort;

```
void qsort(  void *base, size_t nmemb, size_t size,
             int (*compar)(const void *, const void *)
          );
```

base: Η αρχή του array, τα στοιχεία του οποίου θέλουμε να διατάξουμε

nmemb: ο αριθμός των στοιχείων του array

size: το μέγεθος του κάθε στοιχείου

  - Παρατηρείστε ότι με αυτές τις τρεις παραμέτρους μπορεί κανείς να ορίσει ένα array με στοιχεία οποιουδήποτε τύπου, δίνοντας τα στοιχεία της μνήμης: που ξεκινά το array, κάθε πότε αρχίζει το επόμενο στοιχείο, και μετά από πόσα στοιχεία τελειώνει
  - Με αυτές τις παραμέτρους επομένως μπορεί κανείς να διατρέξει ένα οποιοδήποτε array

compar: Μια συνάρτηση που συγκρίνει δύο στοιχεία του array και επιστρέφει 1,0, -1 αν το πρώτο στοιχείο είναι μεγαλύτερο, ίσο, μικρότερ από το δεύτερο

  - Παρατηρείστε ότι η συνάρτηση compar γράφεται από όποιον καλεί την qsort και αναφέρεται στους τύπους των στοιχείων του array που γνωρίζει μόνο αυτός που καλεί την qsort (και φυσικά δεν ήταν γνωστοί όταν γράφτηκε η qsort)

Παράδειγμα κλήσης qsort

- Ας διατάξουμε ένα array από
  - struct node {
    - char *s;
    - int value;
  - }
  - Όπου θέλουμε το sorting να γίνει με βάση το value
- Κλήση της qsort:

```
# συνάρτηση που συγκρίνει δύο structs με τον επιθυμητό τρόπο
int icompar(const void *p, const void *q){
        struct node *p1 = p;
        struct node *q1 = q;
        if (p1->value < q1->value) return -1;
        else if (p1->value == q1->value) return 0;
        else   if (p1->value > q1->value) return 1;
}

int main(void){
        struct node A[100] = {...}; /* Δήλωση και αρχικοποίηση του array A */
        qsort(A, 100, sizeof(struct node), icopmar);
        # σε αυτό το σημείο το array A έχει διαταχθεί
        return;
}
```

Reading

- King Ch. 7.4, 19.{3,4,5}
- (online) The C Book, Mike Banahan, Declan Brady and Mark Doran, Second Edition, 2003. [html] [pdf (updated 2020)]
  - Section 2.8.1
  - Section 5.3.5, 5.7.1