Περιεχόμενα

Πρόλογος

Αυτό είναι ένα ΒΙΒΛΙΟ ΑΠΑΝΤΗΣΕΩΝ. Παρέχει τις λύσεις για όλες τις ασκήσεις του βιβλίου *Η Γλώσσα Προγραμματισμού C*, δεύτερη έκδοση, των Brian W. Kernighan και Dennis M. Ritchie (Κλειδάριθμος, 1990)*.

Το Αμερικανικό Ινστιτούτο Εθνικών Προτύπων (American National Standards Institute, ANSI) δημιούργησε το πρότυπο ANSI για τη γλώσσα C, και οι K&R τροποποίησαν την αρχική έκδοση του βιβλίου *Η Γλώσσα Προγραμματισμού C*. Ξαναγράψαμε λοιπόν τις λύσεις ώστε να συμβαδίζουν τόσο με το πρότυπο ANSI όσο και με τη δεύτερη έκδοση του βιβλίου των K&R.

Η προσεκτική μελέτη του βιβλίου *Απαντήσεις στα Προβλήματα της C*, δεύτερη έκδοση, σε συνδυασμό με τη μελέτη του βιβλίου των K&R, θα σας βοηθήσει να κατανοήσετε τη γλώσσα C και θα σας διδάξει καλές προγραμματιστικές τεχνικές σε C. Χρησιμοποιήστε το βιβλίο των K&R για να μάθετε C, εργαστείτε με τις ασκήσεις, και κατόπιν μελετήστε τις λύσεις που παρουσιάζονται εδώ. Οι λύσεις μας χρησιμοποιούν τις δομές της γλώσσας οι οποίες είναι γνωστές στο σημείο του βιβλίου K&R όπου εμφανίζονται οι ασκήσεις. Ο σκοπός είναι να μπορείτε να παρακολουθήσετε τα βήματα του βιβλίου K&R. Στη συνέχεια, καθώς μαθαίνετε περισσότερα σχετικά με τη γλώσσα C, πιθανώς να μπορείτε να δώσετε καλύτερες λύσεις. Για παράδειγμα, μέχρι να εξηγηθεί η εντολή

```
if (παράσταση)
   εντολή1
else
   εντολή2
```

στη σελίδα 41 του K&R, δεν τη χρησιμοποιούμε. Παρόλα αυτά, μπορείτε να βελτιώσετε τις λύσεις των Ασκήσεων 1-8, 1-9, και 1-10 (σελίδα 39 K&R) αν τη χρησιμοποιήσετε. Μερικές φορές, παρέχουμε επίσης λύσεις χωρίς τέτοιους περιορισμούς.

Οι λύσεις εξηγούνται. Υποθέτουμε ότι έχετε μελετήσει το βιβλίο K&R μέχρι το σημείο της άσκησης. Προσπαθούμε να μην επαναλαμβάνουμε το βιβλίο, αλλά να περιγράφουμε απλώς τα σημαντικά στοιχεία σε κάθε λύση.

---

* Από εδώ και πέρα θα αναφέρεται ως K&R.

Δεν είναι δυνατόν να μάθει κανείς μια γλώσσα προγραμματισμού διαβάζοντας απλώς τις εντολές τής γλώσσας αυτής. Πρέπει επίσης να προγραμματίσει — να γράψει το δικό του κώδικα και να μελετήσει τον κώδικα άλλων. Στο βιβλίο αυτό χρησιμοποιήσαμε ισχυρές λειτουργίες της γλώσσας, χωρίσαμε τον κώδικα σε λειτουργικές μονάδες, κάναμε εκτεταμένη χρήση των ρουτινών βιβλιοθήκης, και μορφοποιήσαμε τα προγράμματά μας έτσι ώστε να μπορείτε να παρακολουθήσετε τη λογική ροή τους. Ελπίζουμε ότι το βιβλίο αυτό θα σας βοηθήσει να γίνετε ειδικοί στη γλώσσα C.

Ευχαριστούμε τους φίλους που βοήθησαν στη δημιουργία αυτής της δεύτερης έκδοσης: τους Brian Kernighan, Don Kostuch, Bruce Leung, Steve Mackey, Joan Magrabi, Julia Mistrello, Rosemary Morrissey, Andrew Nathanson, Σοφία Παπανικολάου, Dave Perlin, Carlos Tondo, John Wait, και Eden Yount.

Clovis L. Tondo

ΚΕΦΑΛΑΙΟ 3 **Η ροή του ελέγχου**

Άσκηση 3-1: (σελίδα 89 K&R)

Η δυαδική μας αναζήτηση κάνει δύο ελέγχους μέσα στο βρόχο, ενώ ένας θα ήταν αρκετός (με το τίμημα περισσότερων ελέγχων έξω από το βρόχο). Γράψτε μια εκδοχή με ένα μόνο έλεγχο μέσα στο βρόχο και μετρήστε τη διαφορά στο χρόνο εκτέλεσης.

```
/* binsearch: εύρεση του x στο v[0] <= v[1] <= ... <= v[n-1]      */
int binsearch(int x, int v[], int n)
{
   int low, high, mid;
   low = 0;
   high = n - 1;
   mid = (low + high) / 2;
   while (low <= high && x != v[mid]) {
      if (x < v[mid])
         high = mid - 1;
      else
         low = mid + 1;
      mid = (low + high) / 2;
   }
   if (x == v[mid])
      return mid;         /* βρέθηκε ταύτιση                      */
   else
      return -1;          /* δεν βρέθηκε ταύτιση                  */
}
```

Αλλάξαμε την παράσταση στο βρόχο `while` από

```
low <= high
```

σε

```
low <= high && x != v[mid]
```

ώστε να μπορούμε να χρησιμοποιήσουμε μία μόνο εντολή `if` μέσα στο βρόχο. Αυτό σημαίνει ότι θα πρέπει να υπολογίζουμε τη `mid` πριν ξεκινήσει ο βρόχος καθώς και κάθε φορά που εκτελείται ο βρόχος.

Θα πρέπει να έχουμε μια συνθήκη ελέγχου έξω από το βρόχο `while` προκειμένου να διαπιστώνουμε αν ο βρόχος τερματίστηκε επειδή βρέθηκε το `x` μέσα στον πίνακα `v`. Αν το `x` υπάρχει στον πίνακα η συνάρτηση `binsearch` επιστρέφει την τιμή `mid`, ενώ διαφορετικά επιστρέφει `-1`.

Η διαφορά στο χρόνο εκτέλεσης είναι ελάχιστη. Δεν κερδίσαμε πολλά σε επιδόσεις και είχαμε κάποιες απώλειες ως προς την αναγνωσιμότητα του κώδικα. Ο αρχικός κώδικας στη σελίδα 88 του βιβλίου K&R είναι συνολικά πιο ευανάγνωστος.

Άσκηση 3-2: (σελίδα 91 K&R)

Γράψτε μια συνάρτηση `escape(s, t)` η οποία, καθώς αντιγράφει το αλφαριθμητικό `t` στο `s`, μετατρέπει χαρακτήρες όπως οι αλλαγές γραμμών και οι στηλοθέτες σε ορατές ακολουθίες διαφυγής, όπως `\n` και `\t`. Χρησιμοποιήστε μια εντολή `switch`. Γράψτε και μια συνάρτηση για το αντίθετο, δηλαδή τη μετατροπή ακολουθιών διαφυγής σε πραγματικούς χαρακτήρες.

```
/* escape: επέκταση των χαρακτήρων αλλαγής γραμμής και στηλοθέτη  */
/*         σε ορατές ακολουθίες διαφυγής κατά την αντιγραφή       */
/*         του αλφαριθμητικού t στο s                              */
void escape(char s[], char t[])
{
   int i, j;

   for (i = j = 0; t[i] != '\0'; i++)
      switch (t[i]) {
         case '\n':                          /* αλλαγή γραμμής      */
            s[j++] = '\\';
            s[j++] = 'n';
            break;
         case '\t':                          /* στηλοθέτης          */
            s[j++] = '\\';
            s[j++] = 't';
            break;
         default:                           /* οι άλλοι χαρακτήρες  */
            s[j++] = t[i];
            break;
         }
   s[j] = '\0';
}
```

Η εντολή

```
for (i = j = 0; t[i] != '\0'; i++)
```

ελέγχει το βρόχο. Η μεταβλητή `i` είναι ο δείκτης ευρετηρίου για το αρχικό αλφαριθμητικό `t`, ενώ το `j` είναι ο δείκτης ευρετηρίου για το τροποποιημένο αλφαριθμητικό `s`.

Υπάρχουν τρεις περιπτώσεις στην εντολή `switch`: το `'\ n'` για το χαρακτήρα αλλαγής γραμμής, το `'\t'` για το χαρακτήρα στηλοθέτη, και η προεπιλογή. Αν ο χαρακτήρας `t[i]` δεν ταιριάζει με καμία από τις δύο πρώτες περιπτώσεις, η συνάρτηση `escape` εκτελεί την περίπτωση με την ετικέτα `default`: την αντιγραφή του `t[i]` στο αλφαριθμητικό `s`.

Η συνάρτηση `unescape` είναι παρόμοια:

```
/* unescape: μετατροπή ακολουθίας διαφυγής σε πραγματικούς        */
/* χαρακτήρες κατά την αντιγραφή του αλφαριθμητικού t στο s       */
void unescape(char s[], char t[])
{
   int i, j;

   for (i = j = 0; t[i] != '\0'; i++)
      if (t[i] != '\\')
         s[j++] = t[i];
      else                          /* βρέθηκε ανάποδη κάθετος      */
         switch(t[++i]) {
         case 'n':                  /* αλλαγή γραμμής               */
            s[j++] = '\n';
            break;
         case 't':                  /* στηλοθέτης                   */
            s[j++] = '\t';
            break;
         default:                   /* όλοι οι άλλοι χαρακτήρες     */
            s[j++] = '\\';
            s[j++] = t[i];
            break;
         }
   s[j] = '\0';
}
```

Αν ο χαρακτήρας στο `t[i]` είναι μια ανάποδη κάθετος, χρησιμοποιούμε την εντολή `switch` για να μετατρέψουμε το αλφαριθμητικό `\n` σε αλλαγή γραμμής και το αλφαριθμητικό `\t` σε χαρακτήρα στηλοθέτη. Η περίπτωση `default` χειρίζεται το ενδεχόμενο μιας ανάποδης καθέτου που ακολουθείται από οτιδήποτε άλλο — αντιγράφει απλώς την ανάποδη κάθετο και το `t[i]` στο αλφαριθμητικό `s`.

Οι εντολές `switch` μπορούν να είναι και ένθετες. Η ακόλουθη είναι μια ακόμα λύση για το ίδιο πρόβλημα.

```
/* unescape: μετατροπή ακολουθίας διαφυγής σε πραγματικούς        */
/* χαρακτήρες κατά την αντιγραφή του αλφαριθμητικού t στο s       */
void unescape(char s[], char t[])
{
   int i, j;

   for (i = j = 0; t[i] != '\0'; i++)
      switch (t[i]) {
         case '\\':                    /* ανάποδη κάθετος         */
            switch(t[++i]) {
            case 'n':                   /* αλλαγή γραμμής         */
               s[j++] = '\n';
               break;
            case 't':                   /* στηλοθέτης             */
               s[j++] = '\t';
               break;
            default:                   /* όλοι οι άλλοι χαρακτήρες */
               s[j++] = '\\';
               s[j++] = t[i];
               break;
            }
            break;
         default:                      /* όχι ανάποδη κάθετος     */
            s[j++] = t[i];
            break;
         }
   s[j] = '\0';
}
```

Η εξωτερική εντολή `switch` χειρίζεται το χαρακτήρα ανάποδης καθέτου και όλες τις υπόλοιπες περιπτώσεις (`default`). Η περίπτωση της ανάποδης καθέτου χρησιμοποιεί μια ακόμα εντολή `switch`, όπως και στην προηγούμενη λύση.

Άσκηση 3-3: (σελίδα 95 K&R)

Γράψτε μια συνάρτηση `expand(s1, s2)` που αναπτύσσει συντομογραφικούς συμβολισμούς όπως ο `a-z` από το αλφαριθμητικό `s1` στην ισοδύναμη πλήρη λίστα `abc...xyz` στο αλφαριθμητικό `s2`. Επιτρέψτε τη χρήση πεζών και κεφαλαίων γραμμάτων καθώς και αριθμητικών ψηφίων, και προετοιμαστείτε για να αντιμετωπίσετε περιπτώσεις όπως οι `a-b-c` και `a-z0-9` ή `-a-z`. Φροντίστε ώστε μια παύλα (-) στην αρχή ή στο τέλος να αντιμετωπίζεται "κυριολεκτικά", δηλαδή να μην γίνεται ειδική ερμηνεία του συμβόλου.

```
/* expand: ανάπτυξη συντομογραφικού συμβολισμού από το s1 στο      */
/* αλφαριθμητικό s2                                                 */
void expand(char s1[], char s2[])
{
   char c;
   int i, j;
   i = j = 0;
   while ((c = s1[i++]) != '\0') /* λήψη χαρακτήρα από το s1[]     */
      if (s1[i] == '-' && s1[i+1] >= c) {
         i++;
         while (c < s1[i])        /* επέκταση συντομογραφίας       */
            s2[j++] = c++;
      } else
         s2[j++] = c;             /* αντιγραφή του χαρακτήρα       */
   s2[j] = '\0';
}
```

Η συνάρτηση παίρνει ένα χαρακτήρα από το `s1`, τον αποθηκεύει στο `c`, και κατόπιν ελέγχει τον επόμενο χαρακτήρα. Αν ο επόμενος χαρακτήρας είναι η παύλα και ο μεθεπόμενος χαρακτήρας είναι μεγαλύτερος ή ίσος από το χαρακτήρα του `c`, η συνάρτηση `expand` προχωρά στην ανάπτυξη του συντομογραφικού συμβολισμού. Διαφορετικά, η συνάρτηση `expand` αντιγράφει το χαρακτήρα στο `s2`.

Η συνάρτηση `expand` λειτουργεί για χαρακτήρες ASCII. Η συντομογραφία `a-z` αναπτύσσεται στο ισοδύναμο αλφαριθμητικό `abc... xyz`. Η συντομογραφία `!-~` αναπτύσσεται σε `!"#..ABC..XYZ..abc..xyz..|}~`.

Η λύση αυτή μας δόθηκε από τον Axel Schreiner του Πανεπιστημίου του Osnabruck, στη Δυτική Γερμανία.

Άσκηση 3-4: (σελίδα 97 K&R)

Σε μια αριθμητική αναπαράσταση συμπληρώματος ως προς δύο, η έκδοσή μας της `itoa` δεν χειρίζεται το μεγαλύτερο αρνητικό αριθμό, δηλαδή μια τιμή του `n` ίση με $-(2^{\text{μέγεθος\_λέξης-1}})$. Εξηγήστε γιατί συμβαίνει αυτό. Τροποποιήστε τη συνάρτηση ώστε να εμφανίζει σωστά την τιμή, ανεξάρτητα από το μηχάνημα στο οποίο εκτελείται.

```
#define  abs(x)   ((x) < 0 ? -(x) : (x))
/* itoa: μετατροπή του n σε χαρακτήρες του s - τροποποιημένη      */
void itoa(int n, char s[])
{
   int i, sign;
   void reverse(char s[]);

   sign = n;          /* καταγραφή του προσήμου                   */
   i = 0;
   do {               /* παραγωγή ψηφίων σε αντίστροφη σειρά      */
      s[i++] = abs(n % 10) + '0';   /* λήψη επόμενου ψηφίου        */
   } while ((n /= 10) != 0);        /* διαγραφή του               */
   if (sign < 0)
      s[i++] = '-';
   s[i] = '\0';
   reverse(s);
}
```

Το

$-(2^{\text{μέγεθος\_λέξης-1}})$

δεν μπορεί να μετατραπεί όπως είναι σε θετικό αριθμό με μια απλή πράξη όπως η

```
n = -n;
```

επειδή ο μεγαλύτερος θετικός αριθμός σε μια αναπαράσταση συμπληρώματος ως προς δύο είναι ο:

$(2^{\text{μέγεθος\_λέξης-1}}) - 1$

Η μεταβλητή `sign` αποθηκεύει την αρχική τιμή του `n`. Η μακροεντολή `abs` βρίσκει την απόλυτη τιμή του `n % 10`. Αυτό αποφεύγει το πρόβλημα με το

$-(2^{\text{μέγεθος\_λέξης-1}})$

επειδή γίνεται θετικό μόνο το αποτέλεσμα του υπολοίπου.

Η παράσταση υπό συνθήκη στην εντολή `do-while` άλλαξε από

```
(n /= 10) > 0
```

σε

```
(n /= 10) != 0
```

επειδή, αν το `n` παρέμενε αρνητικό καθόλη τη διάρκεια του βρόχου, ο βρόχος δεν θα τερματιζόταν ποτέ.

Άσκηση 3-5: (σελίδα 97 K&R)

Γράψτε τη συνάρτηση `itob(n, s, b)` που μετατρέπει τον ακέραιο `n` σε αναπαράσταση χαρακτήρων, με αριθμητική βάση `b`, η οποία αποθηκεύεται στο αλφαριθμητικό `s`. Πιο συγκεκριμένα, η `itob(n, s, 16)` μορφοποιεί τον `n` σαν 16δικό ακέραιο στο `s`.

```
/* itob: μετατρέπει τον n σε χαρακτήρες στο s - με βάση b          */
void itob(int n, char s[], int b)
{
   int i, j, sign;
   void reverse(char s[]);

   if ((sign = n) < 0)       /* καταγραφή προσήμου                 */
      n = -n;                /* κάνε τον n θετικό                  */
   i = 0;
   do {                    /* δημιουργία ψηφίων σε αντίστροφη σειρά  */
      j = n % b;           /* πάρε το επόμενο ψηφίο                 */
      s[i++] = (j <= 9) ? j+'0' : j+'a'-10;
   } while ((n /= b) > 0);          /* διαγραφή του                 */
   if (sign < 0)
      s[i++] = '-';
   s[i] = '\0';
   reverse(s);
}
```

Τα περιεχόμενα του `n` μετατρέπονται σε αριθμητική βάση `b` έτσι ώστε η παράσταση

```
n % b
```

να επιστρέφει μια τιμή μεταξύ `0` και `b-1`, και η παράσταση

```
n /= b
```

να διαγράφει αυτόν τον αριθμό από το `n`. Ο βρόχος συνεχίζεται όσο το `n/b` είναι μεγαλύτερο του μηδενός.

Άσκηση 3-6: (σελίδα 97 K&R)

Γράψτε μια έκδοση της `itoa` που να δέχεται τρία ορίσματα, αντί για δύο. Το τρίτο όρισμα είναι το ελάχιστο πλάτος πεδίου — ο αριθμός που προκύπτει από τη μετατροπή θα πρέπει, εφόσον χρειάζεται, να συμπληρωθεί από τα αριστερά με κενά ώστε να έχει αρκετό πλάτος.

```
#define   abs(x)   ((x) < 0 ?  -(x) : (x))

/* itoa: μετατροπή του n σε χαρακτήρες στο s                    */
/*       με πλάτος w χαρακτήρες                                  */
void itoa(int n, char s[], int w)
{
   int i, sign;
   void reverse(char s[]);

   sign = n;               /* καταγραφή προσήμου                */
   i = 0;
   do {                    /* παραγωγή ψηφίων σε αντίστροφη σειρά */
      s[i++] = abs(n % 10) + '0';   /* λήψη επόμενου ψηφίου       */
   } while ((n /= 10) != 0);        /* διαγραφή του               */
   if (sign < 0)
      s[i++] = '-';
   while (i < w)                    /* συμπλήρωμα με κενά         */
      s[i++] = ' ';
   s[i] = '\0';
   reverse(s);
}
```

Η συνάρτηση είναι παρόμοια με την `itoa` της Άσκησης 3-4. Η απαραίτητη τροποποίηση είναι η

```
while (i < w)
   s[i++] = ' ';
```

Ο βρόχος `while` συμπληρώνει, εφόσον χρειάζεται, το αλφαριθμητικό `s` με κενά διαστήματα.