Learning C

THE ULTIMATE GUIDE

Theodoros Tsiftelidis
ΔΙΠΑΕ | ΣΙΝΔΟΣ

Table of Contents

1. Εισαγωγή

Ο παρών οδηγός απευθύνεται άμεσα στον αναγνώστη. Δηλαδή δεν τηρείται η επαγγελματική μορφή λόγου που συναντάται στις επιστημονικές εργασίες. Ακόμα έχει σχεδιαστεί ώστε να διαβαστεί γραμμικά, από την αρχή ως το τέλος. Επομένως κάθε ενότητα και κεφάλαιο προϋποθέτουν ότι έχετε κατανοήσει τα προηγούμενα.

Ο σκοπός αυτού του οδηγού είναι μια σύντομη αλλά λεπτομερείς περιγραφή της γλώσσας προγραμματισμού C. **Συντάχθηκε σε κάτω από 2 βδομάδες από έναν προπτυχιακό φοιτητή, αυτό σημαίνει σχεδόν σίγουρα θα περιέχει κάποιο λάθος η αστοχία και δεν αποτελεί αυθεντία**.
Σημαντική σημείωση, πολλά από τα παραδείγματα αποτελούν μια υπεραπλούστευση της πραγματικότητας και γίνονται για καθαρά ακαδημαϊκούς λόγους.
Τέλος, προφανές είναι ότι δεν καλύπτονται σε 100 σελίδες όλες οι έννοιες που μπορούν να καλυφθούν ή καλύπτονται από συγγράμματα των 600 σελίδων, αλλά καλύπτονται τα απολύτως βασικά που θα θέσουν στην σωστή πορεία έναν νέο προγραμματιστή και θα κάνουν μια καλή επανάληψη σε έναν πιο έμπειρο. Προς το τέλος καλύπτονται και κάποιες πιο προχωρημένες έννοιες όπως κανονικές εκφράσεις και διαχείριση αρχείων.

Για να ακολουθήσει κανείς τον οδηγό δεν απαιτείται κάποια πρότερη γνώση αλλά προτείνεται η γνώση στα εξής

- Τι είναι το τερματικό και κάποιες βασικές λειτουργίες του
- Δυαδικό σύστημα

1.1 Ιστορία και χρησιμότητα:

Η γλώσσα προγραμματισμού C είναι μια γλώσσα γενικού σκοπού που δημιουργήθηκε το 1972 από τον Dennis Ritchie στα Bell Labs [1]. Σαν

γλώσσα στοχεύει στην βέλτιστη αξιοποίηση των πόρων του επεξεργαστή γιαυτό έχει έντονη παρουσία στον προγραμματισμό λειτουργικών συστημάτων, οδηγούς συσκευών και πολλά άλλα [1]. Ακόμα αξιοποιείται από υπερυπολογιστές μέχρι και μικροελεγκτές / μικροεπεξεργαστές [1]. Είναι αρκετά κοντά στην μνήμη και θεωρείται από τις πιο low level γλώσσες προγραμματισμού και κατά συνέπεια από τις γρηγορότερες. Επιπροσθέτως είναι μια compile type γλώσσα για δομημένο προγραμματισμό. Το 1985 κυκλοφόρησε από τον Bjarne Stroustrup η C++ γλώσσα προγραμματισμού που είναι η C τροποποιημένη για αντικειμενοστραφή προγραμματισμό [2]. Κάποιες φημισμένες εφαρμογές που χτίστηκαν κυρίως σε C είναι:

- Linux kernel
- OS X kernel
- Bash
- OpenSSH
- UNIX kernel
- Doom
- Μέρη του Git
- Σχεδόν όλοι οι hardware drivers (εκτός από αυτούς που είναι σε Assembly.)
- Βασική υλοποίηση Python και Ruby
- Τα περισσότερα ενσωματωμένα συστήματα των αμαξιών (ECU, ABS…)

1.2 Εγκατάσταση C

Με τον όρο «Εγκατάσταση C» εννοούμε την εγκατάσταση εργαλειών που επιτρέπουν τον χρήστη να αναπτύξει και εκτελέσει προγράμματα γραμμένα σε C.

Για την συγγραφή προγράμματος χρησιμοποιούνται Editors ή IDE.

- Ως text editor χαρακτηρίζεται οποιαδήποτε εφαρμογή δέχεται ως είσοδο κείμενο, χαρακτηριστικό παράδειγμα το notepad των windows [3].
- Επόμενο σκαλοπάτι είναι κάποιο editor που προσθέτει κάποιες επιπλέον δυνατότητες από σκέτο γράψιμο. Αυτό μπορεί να είναι αυτόματη συμπλήρωση, χρωματισμός λέξεων και πολλά άλλα. Συνήθως αναφέρονται ως Source-code editors [4]. Τέτοια είναι τα notepad++, Visual Studio Code, Sublime, Atom, κ.α.
- Στην κορυφή της πυραμίδας βρίσκονται τα IDEs (Integrated Development Environment) που περιλαμβάνουν μια πληθώρα εργαλειών με σκοπό την διευκόλυνση του χρήστη όπως source-code editor, compiler, debugger, version control system και πολλά άλλα [5]. Κάποια χαρακτηριστικά παραδείγματα είναι το Code::Blocks, Eclipse, Dev-C++, Visual Studio.

Για την εκτέλεση ενός προγράμματος απαιτείται ένας compiler. Για το IDE εγκαθίστανται ένας αυτόματα με την εγκατάσταση του IDE. Για text editors και source-code editors χρειάζεται χειροκίνητη εγκατάσταση. Οι πιο γνωστοί compilers για C και C++ είναι:

- GCC, the GNU Compiler Collection
- Orwell Dev-C++
- Clang / LLVM
- tcc

Με τον πιο γνωστό και διαδεδομένο να είναι ο GCC και αυτό που θα χρησιμοποιήσουμε για το υπόλοιπο του οδηγού.

Για την εγκατάσταση, προτείνεται η εγκατάσταση της σουίτας MSYS2 που περιλαμβάνει το MinGW-w64 το πιο διαδεδομένο port του GCC μαζί με πολλά άλλα χρήσιμα εργαλεία για την ανάπτυξη εφαρμογών windows. Προτείνεται μέσω της MSYS2 σουίτας καθώς προσφέρει εύκολη

εγκατάσταση και την τελευταία έκδοση όλων των εργαλειών. Φυσικά είναι δυνατή και η απευθείας εγκατάσταση του MinGW-w64 ή ακόμα και GCC compiler αλλά δεν προτείνεται εκτός αν υπάρχει κάποιος πολύ συγκεκριμένος λόγος που αφορά έμπειρους χρήστες.

Tutorial για MSYS2: https://www.freecodecamp.org/news/how-to-install-c-and-cpp-compiler-on-windows/.

Για περιβάλλον ανάπτυξης προτείνεται ο source-code editor Visual Studio Code, διότι είναι αρκετά ελαφρύς και παρέχει τεράστια γκάμα δυνατοτήτων μέσω της εγκατάστασης επεκτάσεων. Συγκεκριμένα για το παρών έγγραφο είναι απαραίτητη η εγκατάσταση των εξής επεκτάσεων:

- C/C++ επέκταση της Microsoft που προσθέτει αυτόματη συμπλήρωση, χρωματισμός λέξεων και πολλά άλλα «καρυκεύματα».
- Code Runner του Jun Han που επιτρέπει το πρόγραμμα να εκτελείται τοπικά από την σελίδα του editor.
- Υπάρχει και η C/C++ extension pack που προσθέτει ακόμα περισσότερα εργαλεία. Προαιρετικά μπορεί να εγκατασταθεί και αυτή.

Στην συνέχεια δημιουργούμε έναν φάκελο που θα στεγάζει τα αρχεία μας και τον ανοίγουμε μέσα στο VScode. Δημιουργούμε ένα αρχείο και το ονομάζουμε “HelloWorld.c”. Πατώντας terminal και configure default build task μας εμφανίζει τον gcc compiler που εγκαταστήσαμε προηγουμένως, τον επιλέγουμε και παρατηρούμε δημιουργείται ένα αρχείο json μέσα στον φάκελο μας. Είναι ένα αρχείο που περιέχει πληροφορίες για τον compiler πως να χειρίζεται τα αρχεία που δημιουργούμε.

Για την εκτέλεση του προγράμματος:

- Αν έχουμε source-code editor ή IDE γίνεται απευθείας μέσα από το περιβάλλον πατώντας το κουμπί Run.
- Για text editors πρέπει χειροκίνητα να δημιουργηθεί το εκτελέσιμο αρχείο. Ανοίγουμε cmd και κάνουμε cd στον φάκελο με το αρχείο που θέλουμε να εκτελέσουμε. Γράφουμε gcc HelloWorld.c και πατώντας enter δημιουργείτε ένα εκτελέσιμο εν ονόματι a.exe. Γράφοντας απλά το όνομα του αρχείου δηλαδή a.exe εκτελείται.

Μια ακόμη σημαντική ρύθμιση στο VScode είναι File -> Preferences -> Settings -> Αναζήτηση code runner run in terminal και επιλογή το κουτάκι. Αυτή η ρύθμιση κάνει το πρόγραμμα να τρέχει σε terminal και έτσι μπορούμε σε επόμενα κεφάλαια να στείλουμε δεδομένα στο πρόγραμμα μας και πολλά άλλα.

1.3 Το πρώτο πρόγραμμα

Το πλέον καθιερωμένο πρώτο πρόγραμμα είναι το Hello World πρόγραμμα δηλαδή ένα πρόγραμμα που όταν το τρέχουμε εμφανίζει το μήνυμα “Hello World”. Αυτό στην C είναι:

```c
#include <stdio.h>
int main()
{
   // This program prints Hello World
   printf("Hello, World!");
   return 0;
}
```

Πάμε να δούμε γραμμή προς γραμμή τι κάνει.

1) Η πρώτη γραμμή (#include <stdio.h>) περιλαμβάνει (αυτό σημαίνει το include) την βιβλιοθήκη “stdio.h” που περιλαμβάνει την υλοποίηση εντολών όπως η printf() που χρησιμοποιείται παρακάτω. Μπορούμε να παραλείψουμε την δήλωση της αλλά αυτό σημαίνει

ότι δεν μπορούμε να χρησιμοποιήσουμε την printf() ή οποιαδήποτε άλλη εντολή χρησιμοποιούσαμε και προερχόταν από αυτή την βιβλιοθήκη. Επειδή σχεδόν πάντα χρησιμοποιούμε την printf() σε αυτόν τον οδηγό θα την συμπεριλαμβάνουμε πάντα.

2) Στην δεύτερη γραμμή δηλώνουμε την βασική συνάρτηση από την οποία ξεκινάει η εκτέλεση οποιουδήποτε προγράμματος.
3) Στην τρίτη γραμμή το { συμβολίζει την αρχή της main συνάρτησης.
4) Η τέταρτη γραμμή είναι ένα σχόλιο του προγραμματιστεί που δεν ερμηνεύεται από το πρόγραμμα και αποτελεί μήνυμα / σχόλιο για όποιον άλλον προγραμματιστή δει τον κώδικα. Το παρόν μήνυμα εξηγεί την λειτουργία του προγράμματος που είναι η εκτύπωση του μηνύματος Hello World.
5) Στην πέμπτη γραμμή βρίσκεται η εντολή που εκτυπώνει το μήνυμα μας. Πιο συγκεκριμένα επρόκειτο για την printf() συνάρτηση που πήραμε από το stdio.h αρχείο. Η δουλειά της είναι να εκτυπώνει στην κονσόλα ό,τι της δώσουμε. Στην παρούσα περίπτωση επειδή θέλουμε να εκτυπώσουμε ένα μήνυμα βάζουμε «αυτάκια» (quotation marks) γύρω από το μήνυμα ώστε να καταλάβει ο compiler ότι είναι κείμενο.
6) Στην έκτη και προτελευταία γραμμή η εντολή return 0 «επιστρέφει» στην τιμή μηδέν στην συνάρτηση main και τερματίζεται. Μια χρησιμότητα είναι αν εξετάζουμε την main συνάρτηση και δούμε ότι τερματίστηκε με τον κωδικό 0 τότε καταλαβαίνουμε ότι το πρόγραμμα έφτασε μέχρι το τέλος άρα εκτελέστηκε σωστά. Μπορεί να παραληφθεί η συγκεκριμένη εντολή καθώς πλέον προστίθεται αυτόματα από την γλώσσα αλλά είναι καλή προγραμματιστική συνήθεια να την συμπεριλαμβάνουμε.
7) Στην έβδομη και τελευταία γραμμή το } δείχνει ότι εδώ τελειώνει η main συνάρτηση.

Αξίζει να σημειωθεί πως στο τέλος κάθε γραμμής πρέπει να μπαίνει ένα ελληνικό ερωτηματικό ; καθώς συμβολίζει το τέλος γραμμής και το χρησιμοποιεί ο compiler για να καταλάβει που τελειώνει κάθε γραμμή. Εφόσον το ; συμβολίζει τέλος γραμμής μπορούμε να έχουμε ολόκληρο το πρόγραμμα μας σε μία γραμμή και απλά κάθε φορά που τελειώνει μια εντολή να βάζουμε ; αυτό θα δουλέψει μια χαρά αλλά δεν είναι πρακτικό διότι δεν φαίνεται καλά ο κώδικας και είναι πιο δύσκολο για κάποιον να τον διαβάσει και διορθώσει.

1.4 Πως εκτελείται το πρώτο μας πρόγραμμα:

Το αρχείο που γράψαμε (σε οποιοδήποτε περιβάλλον αναφέραμε πριν) ονομάζεται source code αρχείο. Αυτό διότι περιλαμβάνει τον source code – πηγαίο κώδικα δηλαδή τον κώδικα που γράφει και καταλαβαίνει ο άνθρωπος. Ωστόσο δεν μπορεί να τον καταλάβει ο υπολογιστής. Εδώ έρχεται ο compiler – μεταγλωττιστής που η δουλειά του είναι να μεταφράσει το αρχείο σε κάτι που καταλαβαίνει ο υπολογιστής. Αυτό το αρχείο στην συνέχεια εκτελεί ο υπολογιστής.

2. Βασικά στοιχεία:

2.1 Σχόλια (Comments):

Όπως προαναφέρθηκε τα σχόλια αγνοούνται από τον compiler και είναι καθαρά μηνύματα προς όποιον διαβάζει τον πηγαίο κώδικα. Καλή συνήθεια είναι να αναγράφεται, δίπλα ή πάνω από κάθε γραμμή ή τμήμα κώδικα, μια σύντομη περιγραφή του τι κάνει. Μπορεί τώρα να φαίνεται γελοίο που έχουμε 7 γραμμές κώδικα αλλά σε πιο σύνθετα και μεγάλα προγράμματα που δουλεύουν ταυτόχρονα πολλοί προγραμματιστές είναι απαραίτητο για συνοχή και αποφυγή συγχύσεων. Ακόμη πολλές φορές όταν ένας προγραμματιστής επιστρέφει σε ένα πρόγραμμα που είχε γράψει πριν από καιρό δεν θυμάται πως και τι έκανε ο κώδικας του, οπότε

τα μηνύματα δεν είναι απλά για τους άλλους προγραμματιστές αλλά και για τους ίδιους μας του εαυτούς. Υπάρχουν δύο κατηγορίες σχολίων.

- Σχόλια μονής γραμμής (single-line comments)
- Σχόλια πολλαπλών γραμμών (multi-line comments)

Τα σχόλια μονής γραμμής εισάγονται με τους χαρακτήρες

```
//
```

και ότι ακολουθεί μετά από αυτούς εκλαμβάνεται ως σχόλιο. Συνήθως χρησιμοποιούνται για μικρές σημειώσεις πάνω, δίπλα ή κάτω από μια γραμμή ή τμήμα κώδικα. Για παράδειγμα:

```
// Αυτό είναι ένα σχόλιο μονής γραμμής.
```

Τα σχόλια πολλαπλών γραμμών εισάγονται με το

```
/*
```

και τελειώνουν με το

```
*/
```

ό,τι τοποθετηθεί ανάμεσα τους εκλαμβάνεται ως σχόλιο.

<u>Παράδειγμα:</u>

```
/*
Αυτό
Είναι
Ένα
Σχόλιο πολλαπλών γραμμών
*/
```

Συνήθως συναντάμε σχόλια στην αρχή ενός προγράμματος και εξηγούν με λεπτομέρειες τι κάνει το πρόγραμμα.

Φυσικά επειδή είναι σχόλια ο καθένας τα χρησιμοποιεί όπως θέλει χωρίς περιορισμούς, εδώ απλά αναφερόμαστε στις πιο συνηθισμένες χρήσεις τους.

2.2 Escape Sequences (\x):

Έτσι ορίζεται ένας συνδυασμός χαρακτήρων που αποτελείται από:

```
\ (backslash)
```

και ένα γράμμα ή συνδυασμό γραμμάτων. Τα πιο γνωστά είναι:

- \n: Δημιουργεί νέα γραμμή στο τερματικό και ότι ακολουθεί μετά τον χαρακτήρα, μπαίνει στην νέα γραμμή
- \t: Δημιουργεί το κενό που δημιουργεί αν πατήσουμε το tab.

Παραδείγματα:

```
printf("Hello,\nWorld!");
```

Το αποτέλεσμα αντί για Hello, World! Θα είναι:

```
Hello,
World!
```

Αντίστοιχα αν είχαμε

```
printf("Hello,\tWorld!");
```

Το αποτέλεσμα θα ήταν:

```
Hello,  World!
```

Μια ακόμη σημαντική λειτουργία για το backslash είναι ότι ακυρώνει την ειδική σημασία των συμβόλων. Έτσι αν θέλαμε να εκτυπώσουμε τα

```
" "
```

μέσα στο κείμενο μας με το Hello World! θα τα βρίσκαμε σκούρα, διότι τα " " δηλώνουν αρχή και τέλος μηνύματος. Εδώ έρχεται η χρήση του backslash όπου αν το βάλουμε πριν έναν τέτοιο ειδικό χαρακτήρα τότε ακυρώνει την σημασία του και θα μας επιτρέψει αν το εκτυπώσουμε. Έτσι αν είχαμε

```
printf("Hello, \"Beautiful\" ,World!");
```

Το αποτέλεσμα είναι:

```
Hello, "Beautiful" ,World!
```

Αντίστοιχα αυτό ισχύει αν θέλαμε να εκτυπώσουμε οποιοδήποτε από τα

```
' ', ?
```

Ή ακόμα και το ίδιο το

```
\
```

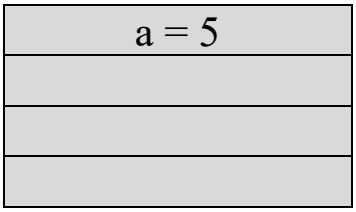
αντίστοιχα βάζουμε ένα \ πριν από αυτά.

Υπάρχουν πολλά ακόμη escape sequences εκτός απο το \n και \t όπου με μία απλή αναζήτηση στο ίντερνετ βρίσκεται μια πληθώρα επιλογών. Μια ακόμη ίσως χρήσιμη αναφορά είναι το \v που προσθέτει κάθετο tab.

2.3 Μεταβλητές (Variables):

Έστω ότι έχουμε ένα πιο σύνθετο πρόγραμμα που θέλουμε να κάνει κάποιες πράξεις όπως για παράδειγμα να προσθέτει δύο αριθμούς, να εκτυπώνει το αποτέλεσμα και να κλείνει. Έστω ότι θέλουμε να προσθέσουμε τους αριθμούς 5 και 3 και να εκτυπώσουμε το αποτέλεσμα της πρόσθεσης τους δηλαδή το 8. Πρώτα θα χρειαστούμε ένα προσωρινό χώρο για να αποθηκεύσουμε αυτούς τους αριθμούς και στην συνέχεια το αποτέλεσμα της πράξης τους. Αυτό το ρόλο εξυπηρετεί η κύρια μνήμη (RAM) του υπολογιστή μας. Βολεύει να απεικονίζουμε την μνήμη ως μια κάθετη συστοιχία με κουτάκια:

Κάθε κουτάκι ονομάζεται θέση μνήμης ή κελί μνήμης και μπορούμε να του αναθέσουμε μια τιμή. Αυτή η διεργασία χωρίζεται σε δύο βήματα, αρχικά έχουμε την δέσμευση μιας θέσης και σε δεύτερο χρόνο την ανάθεση μιας τιμής όπως το 5.



Στην C και σε άλλες γλώσσες χαμηλού επιπέδου (δηλαδή που είναι κοντά στη μνήμη) πρέπει κατά την δέσμευση μιας θέσης να δηλώσουμε και τι τύπου δεδομένα θα την κατοικήσουν, π.χ. ακέραιος αριθμός, αριθμός με υποδιαστολή, χαρακτήρες (α, β…), κ.α.

Παράδειγμα:

```
int x;
```

Στην παραπάνω γραμμή γίνεται η δέσμευση μιας θέσης (κουτάκι) στην μνήμη που ακούει στο όνομα x και είναι τύπου int δηλαδή ακεραίου (int = integer = ακέραιος).

Επόμενο βήμα είναι η ανάθεση κάποιας τιμής σε αυτή τη x μεταβλητή.

```
x = 5;
```

Με την παραπάνω γραμμή βάζουμε στην μεταβλητή / κουτάκι με το όνομα x την τιμή 5.

Τα δυο παραπάνω βήματα μπορούν να γίνουν σε μια γραμμή με την εξής σύνταξη:

```
int x = 5;
```

2.4 Σταθερές (Constants):

Έστω ότι έχουμε στο πρόγραμμα μας μια μεταβλητή με πολύ σημαντική τιμή. Για παράδειγμα,

```
float pi = 3.14;
```

Υπάρχει σοβαρή περίπτωση κάποια άλλη στιγμή μέσα στο πρόγραμμα είτε από αμέλεια είτε καταλάθος να αλλάξουμε την τιμή της μεταβλητής (ειδικά όσο θα μεγαλώνουν τα προγράμματα). Λόγω της σημαντικότητας της μεταβλητής αυτό δεν είναι κάτι που θέλουμε να ρισκάρουμε. Για να σιγουρέψουμε λοιπόν ότι δεν μπορεί να αλλάξει τιμή, κατά την δήλωση της θα βάλουμε την λέξη const πριν το float, κάπως έτσι:

```
const float pi = 3.14;
```

και αυτό κάνει σίγουρο ότι η τιμή δεν μπορεί να αλλάξει μετά την πρώτη ανάθεση.

Μια προγραμματιστική συνήθεια είναι όλες τις μεταβλητές που είναι σταθερές (const = constant) το όνομα τους να είναι στα κεφαλαία, δηλαδή:

```
const float PI = 3.14;
```

Μπορούμε επίσης να ορίσουμε μια σταθερά στο C χρησιμοποιώντας τον προεπεξεργαστή #define. Οι σταθερές που ορίζονται χρησιμοποιώντας το #define είναι μακροεντολές που συμπεριφέρονται σαν σταθερές. Αυτές οι σταθερές δεν αντιμετωπίζονται από τον μεταγλωττιστή, τις χειρίζεται ο προεπεξεργαστής και αντικαθίστανται η τιμή τους πριν από τη μεταγλώττιση.

```
#include <stdio.h>
#define pi 3.14

int main()
{
```

```
   printf("The value of pi: %f", pi); // Καλύπτεται σε
επόμενη ενότητα η εκτύπωση μεταβλητών - σταθερών.
   return 0;
}
```

2.5 Κυριολεκτικές Σταθερές (Literals):

Πολλές φορές χρειάζεται να χρησιμοποιήσουμε συγκεκριμένους αριθμούς, χαρακτήρες ή ονόματα σε εντολές εκχώρησης όπως:

```
int x = 5;
```

Αυτή η τιμή 5 που ανατίθεται στο x ονομάζεται κυριολεκτική σταθερά. Αντίστοιχα κυριολεκτική σταθερά θα είναι οποιοσδήποτε χαρακτήρας ή όνομα τοποθετηθεί σε κάποια μεταβλητή η σε δομή ελέγχου.

2.6 Κανόνες Ονομασίας:

Ως όνομα μπορεί να χρησιμοποιηθεί μια ακολουθία χαρακτήρων που ξεκινά από αλφαβητικό χαρακτήρα και περιέχει μόνο γράμματα, ψηφία ή χαρακτήρες κάτω παύλας (_). Για παράδειγμα:

```
int x;
int number_of_elements;
int Fourier_transform;
int z2;
int Polygon;
```

Τα παρακάτω δεν είναι ονόματα:

```
int 2x;          // Ένα όνομα πρέπει να ξεκινά με
αλφαβητικό χαρακτήρα
int Start menu; // Το κενό (space) δεν είναι γράμμα,
ψηφίο ή κάτω παύλα
```

Μια πιο λεπτομερείς λίστα με κανόνες ονομασίας είναι:

- Τα ονόματα μπορεί να περιέχουν γράμματα, ψηφία και κάτω παύλες
- Τα ονόματα πρέπει να ξεκινούν με γράμμα ή κάτω παύλα (_)

- Τα ονόματα έχουν διάκριση πεζών-κεφαλαίων (myVar και myvar είναι διαφορετικές μεταβλητές)
- Τα ονόματα δεν μπορούν να περιέχουν κενά διαστήματα ή ειδικούς χαρακτήρες όπως !, #, %, κ.λπ.
- Οι δεσμευμένες λέξεις (όπως int) δεν μπορούν να χρησιμοποιηθούν ως ονόματα

Τα παρακάτω δεν είναι κανόνες αλλά συστάσεις:

- Συνιστάται η χρήση περιγραφικών ονομάτων για τη δημιουργία κατανοητού και διατηρήσιμου κώδικα:

```
// Αντί για
int m = 60;
// Προτιμάται
int minutesPerHour = 60;
```

- Αν και τα myVar και myvar αντιμετωπίζονται από το σύστημα ως διαφορετικές μεταβλητές, καλό είναι να μην επιλέγεται η χρήση τους από τον χρήστη.
- Η γλώσσα επιτρέπει ένα όνομα μεταβλητής να ξεκινάει από κάτω παύλα _ για παράδειγμα _foo. Ωστόσο καλό είναι να μην χρησιμοποιείται αυτή η αναγραφή σε ονόματα μεταβλητών επειδή τέτοια ονόματα δεσμεύονται για οντότητες σχετιζόμενες με την υλοποίηση της γλώσσας και οντότητες συστήματος. Εάν αποφεύγετε τη χρήση κάτω παύλας ως αρχικού χαρακτήρα, δεν θα αντιμετωπίσετε ποτέ προβλήματα σύγκρουσης των ονομάτων σας με τέτοια «ονόματα συστήματος».

3. Τύποι δεδομένων (Data Types):

Όπως είδαμε στο προηγούμενο κεφάλαιο, με την εξής γραμμή κώδικα:

```
int x;
```

Δεσμεύεται έναν χώρο στην μνήμη που ακούει στο όνομα x και μπορεί να αποθηκεύσει ακέραιες τιμές. Αυτή η «λέξη» που μπαίνει πριν το όνομα της μεταβλητή, το int στην προκειμένη περίπτωση, ανήκει σε μια κατηγορία ειδικών λέξεων που λέγονται τύποι δεδομένων (data types). Γενικά οι τύποι δεδομένων χωρίζονται σε 3 κατηγορίες.

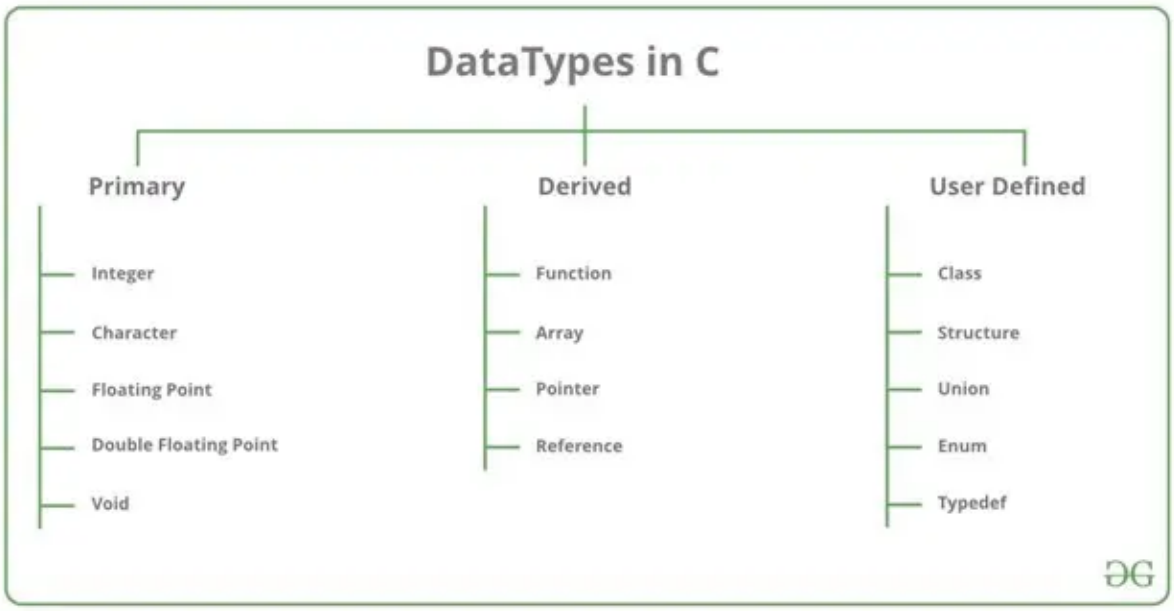


Figure 1: Data Types in C - GeeksforGeeks

- Η primary ή πρωτεύων κατηγορία περιλαμβάνει τους απλούς τύπους δεδομένων και είναι ενσωματωμένοι στην γλώσσα οπότε υποστηρίζονται αυτόματα από όλους τους compilers.
- Η derived κατηγορία αναφέρεται στους τύπους δεδομένων που εκπίπτουν ευθέως από τους πρωτεύων.
- Η user defined κατηγορία περιγράφει τους τύπους δεδομένων που δηλώνει ο χρήστης / προγραμματιστής.

Για αρχή θα περιγραφούν μόνο οι πρωτεύων τύποι και σε επόμενα κεφάλαια θα αναλυθούν και οι υπόλοιποι.

Οι πρωτεύων τύποι δεδομένων είναι:

- int: Για αναπαράσταση ακεραίων.

- float: Για αναπαράσταση δεκαδικών έως 6-7 ψηφία. (γνωστό ως single point precision ή single precision). Αν ένας αριθμός δεν έχει τόσα ψηφία τότε αυτόματα θα γεμίσει μέχρι τα 6-7 ψηφία με μηδενικά, δηλαδή ο αριθμός 2.2 θα εμφανιστεί 2.200000.
- double: Για αναπαράσταση δεκαδικών έως 14-15 ψηφία. (γνωστό ως double point precision ή double precision), ίδια παρατήρηση με πάνω.
- char: Για αναπαράσταση μεμονωμένων ψηφίων (α,1,-).
- void: Για αναπαράσταση του κενού, δεν χρησιμοποιείται σε μεταβλητές και διίστανται οι απόψεις για το αν θεωρείται τύπος δεδομένων, σταματάνε οι αναφορές σε αυτό μέχρι να μελετηθεί αναλυτικότερα σε παρακάτω κεφάλαιο.

Οι float και double τύποι δεδομένων είναι γνωστοί και ως πραγματικοί τύποι δεδομένων.

Επιπροσθέτως υπάρχει μια λίστα προθεμάτων που όταν μπουν μπροστά από κάποιους συγκεκριμένους τύπους δεδομένων αλλάζουν κάποιες ιδιότητες τους. Λέμε μπροστά από κάποιους συγκεκριμένους καθώς δεν επιδέχονται όλοι όλα τα προθέματα.

Αυτά τα προθέματα είναι:

- short: Μικραίνει το πόσο μνήμη καταλαμβάνει μια μεταβλητή
- long: Αυξάνει το πόσο μνήμη καταλαμβάνει μια μεταβλητή
- signed: Δηλώνει ότι η μεταβλητή υποστηρίζει και θετικές και αρνητικές τιμές.
- unsigned: Δηλώνει ότι η μεταβλητή υποστηρίζει μόνο θετικές τιμές.

Αν και ορίζεται signed πρόθεμα στην πράξη δεν θα δούμε ποτέ να αναγράφεται πριν μια μεταβλητή καθώς signed είναι μια μεταβλητή εκ προεπιλογής, για παράδειγμα signed int = int. Άρα ότι μεταβλητή δημιουργήσουμε είναι εκ προεπιλογής signed και αλλάζει μόνο αν

μπροστά βάλουμε κάποιο από τα άλλα προθέματα (short, long, unsigned). Κάθε μνήμη ram έχει έναν συγκεκριμένο διαθέσιμο χώρο ο οποίος διαμορφώνεται κατά της κατασκευή της και μετριέται σε bits. Για παράδειγμα μια ram των 8gb μπορεί να αποθηκεύσει 64000000000 bits πληροφορίας. Επομένως αν ένα πρόγραμμα καταναλώνει 2gb ram, η διαθέσιμη ram γίνεται 8 – 2 = 6gb δηλαδή 48000000000 bits πληροφορίας.

Κάθε μεταβλητή αναλόγως και με τον τύπο της και το πρόθεμα που μπορεί να έχει καταλαμβάνει διαφορετικό πλήθος bits. Εδώ φαίνεται και η χρησιμότητα των άλλον προθεμάτων, καθώς αν είναι σίγουρο ότι μια μεταβλητή δέχεται αυστηρά μικρές τιμές (πχ κρατάει την μέρα γενεθλίων) τότε αντί για int μπορεί να δηλωθεί short int που όπως θα δούμε και παρακάτω είναι τα μισά bits.

Για να μην χρησιμοποιούμε μεγάλους αριθμούς συχνά μετράμε τον χώρο στην μνήμη σε bytes αντί για bits όπου 1 byte = 8 bits.

Το πόσα bits δεσμεύει από την μνήμη μια μεταβλητή καθορίζει και την ελάχιστη και μέγιστη τιμή. Στην παρακάτω εικόνα έχουμε 8bits δηλαδή 1byte και στην τιμή που αναλογεί σε καθένα. Έστω ότι χρησιμοποιούνται για την απεικόνιση μια μεταβλητής αν πρόκειται για signed μεταβλητή τότε ένα bit από τα 8 πρέπει να χρησιμοποιηθεί για απεικόνιση του προσήμου. Άρα μόνο τα 7bits χρησιμοποιούνται για απεικόνιση της αξίας της μεταβλητής. Έτσι η μέγιστη αξία που μπορεί να λάβει είναι 1111111 δηλαδή 64+32+16+8+4+2+1 = 127, και η ελάχιστη τιμή το -128. Δεν θα μπούμε σε λεπτομέρειες γιατί δεν είναι στο -127 η ελάχιστη τιμή και είναι στο -128 γιατί ξεφεύγει από τους στόχους του βιβλίου, έχει να κάνει με την τεχνική που χρησιμοποιείται για την απεικόνιση δεκαδικών αριθμών στο δυαδικό σύστημα (συμπλήρωμα ως προς 2 - Two's complement).

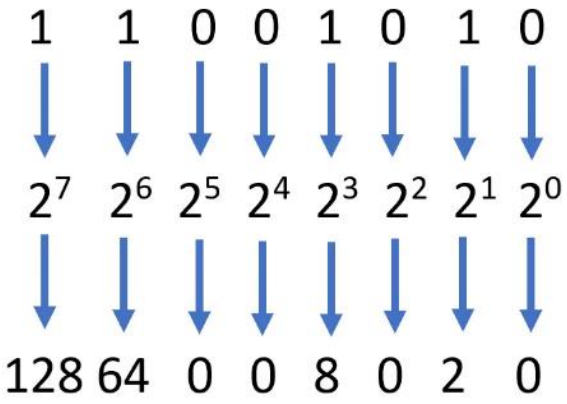


Figure 2: Value of 8 bits

Σημαντική σημείωση, ανάλογα με:

- Έκδοση γλώσσας
- Compiler
- Επεξεργαστή (αρχιτεκτονική / μικροαρχιτεκτονική – οργάνωση), κ.α.
- Είδος, γενιά, ποσότητα κύριας μνήμης.

Μπορεί να υπάρξει διαφορά σε bits / bytes άρα και σε ελάχιστη και μέγιστη τιμή, τα παρακάτω πινακάκια είναι καθαρά θεωρητικά για ακαδημαϊκούς λόγους.

3.1 Ακέραιοι (Integers – int):

Ο παρακάτω πίνακας περιλαμβάνει όλους τους πιθανούς συνδυασμούς προθεμάτων με τους ακέραιους τύπους δεδομένων, τον χώρο στην μνήμη που καταλαμβάνουν (bits / bytes) και την ελάχιστη με μέγιστη τιμή.

Τύπος	Bits	Bytes	Ελάχιστη Τιμή	Μέγιστη Τιμή
short int	16	2	-32768	32767

unsigned short int	16	2	0	65535
int	(16 -) 32	2 - 4	-2147483648	2147483647
unsigned int	(16 -) 32	2 - 4	0	4294967295
long int	(32 -) 64	4 - 8	int – long long int	int – long long int
unsigned long int	(32 -) 64	4 - 8	0	int – long long int * 2
long long int	64	8	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
unsigned long long int	64	8	0	18,446,744,073,709,551,615

[6]

Σημειώσεις:

- short int = short
- Παρατηρούμε ότι στους unsigned τύπους η μέγιστη τιμή είναι διπλάσια, αυτό γιατί χρησιμοποιούνται όλα τα bits για την αξία του αριθμού και κανένα για πρόσημο.
- Παρατηρείται μια διακύμανση στα bits για τους λόγους που αναφέρθηκαν προηγουμένως, ανάλογα την έκδοση κ.α.
- Όταν ορίζουμε μεταβλητή τύπου long συνηθίζεται μετά την τιμή να μπαίνει το γράμμα L. Στους περισσότερους compilers δεν υπάρχει θέμα, αλλά σε κάποιους άλλους ή σε άλλες εκδόσεις για μεγάλους αριθμούς μπορεί να υπάρξει πρόβλημα.

3.2 Πραγματικοί αριθμοί:

Οι πραγματικές μεταβλητές (float, double) αφιερώνουν ένα bit στο πρόσημο, κάποια για το ακέραιο μέρος και κάποια για το δεκαδικό μέρος.

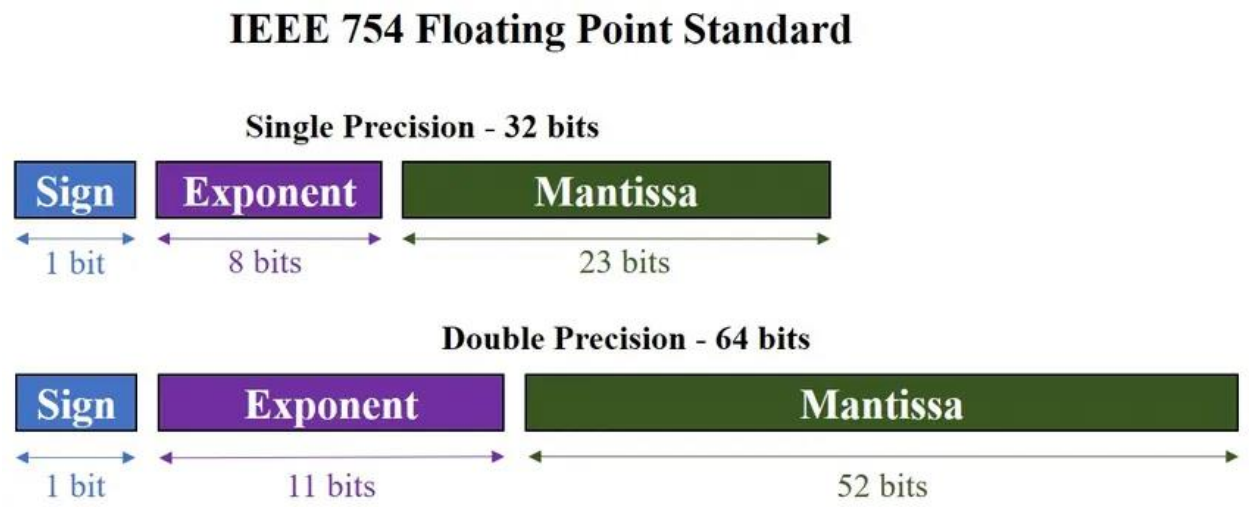


Figure 3: Float and double bits distribution

Όπως έχει αναφερθεί οι float μεταβλητές αναφέρονται και ως single precision και φαίνεται ότι αφιερώνουν 8 bit για τον ακέραιο και 23 για το δεκαδικό μέρος, με τις double αντίστοιχα στα 11 bit και 52 bit.

Έτσι το αντίστοιχο πινακάκι θα είναι:

Τύπος	Bits	Bytes	Ελάχιστη Τιμή	Μέγιστη Τιμή
float	32	4	-3.4*10^38	3.4*10^38
double	64	8	-1.7*10^308	1.7*10^308
long double	80	10	-1.1*10^4932	1.1*10^4932

[6]

Αντίστοιχα και εδώ μετά την τιμή σε μια float μεταβλητή τοποθετείται f ή F.

3.3 Χαρακτήρες (Characters – char):

Το σύνολο χαρακτήρων της C είναι το ASCII. Σε μεταβλητές τύπου char μπορεί να αποθηκευτεί ένας χαρακτήρας ή το αριθμητικό του ισοδύναμο.

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Παράδειγμα:

```
char ch;
ch = 'X'; ή
ch = 88;
```

Αντίστοιχο πινακάκι για χαρακτήρες είναι το:

Τύπος	Bits	Bytes	Ελάχιστη Τιμή	Μέγιστη Τιμή
char	8	1	-128	127
signed char	8	1	-128	127

unsigned char	8	1	0	255

[6]

Εδώ έχουμε και char και signed char καθώς αρνητική τιμή σε απλό char μπορεί να δημιουργήσει πρόβλημα σε κάποιους compilers, για την ώρα δεν είναι κάτι που μας απασχολεί.

Πάντα όταν έχουμε χαρακτήρα βάζουμε ‘ ‘ πριν και μετά.

3.4 Boolean - bool:

Πολύ συχνά, στον προγραμματισμό, θα χρειαστεί ένας τύπος δεδομένων που μπορεί να έχει μόνο μία από τις δύο τιμές, όπως:

- ΝΑΙ - ΟΧΙ
- ON - OFF
- ΑΛΗΘΗΣ - ΨΕΥΔΗΣ

Για αυτόν τον σκοπό, η C έχει έναν ειδικό τύπο δεδομένων που είναι γνωστός ως Boolean ή bool. Ο τύπος αυτός δεν είναι ενσωματωμένος στην γλώσσα όπως οι int, char, κ.α. Αλλά εισήχθη στην έκδοση C99 και πρέπει να συμπεριληφθεί η ακόλουθη βιβλιοθήκη για να το χρησιμοποιηθεί:

```
#include <stdbool.h>
```

Μια μεταβλητή bool δηλώνεται με τη λέξη-κλειδί bool και μπορεί να λάβει μόνο τις τιμές true ή false:

Οι Booleans αντιπροσωπεύουν τιμές που είναι είτε αληθείς είτε ψευδείς.

```
   bool isProgrammingFun = true;
```

Προτού επιχειρήσουμε να εκτυπώσουμε μια bool μεταβλητή πρέπει να ξέρουμε ότι επιστρέφει την τιμή 0 αν είναι ψευδής και οποιοδήποτε άλλον

αριθμό, συνήθως 1 αν είναι αληθής, άρα εκτυπώνεται με το %d format specifier.

```
bool isProgrammingFun = true;
bool isFishTasty = false;

printf("%d\n", isProgrammingFun); // Returns 1 (true)
printf("%d\n", isFishTasty);      // Returns 0 (false)
```

Το αποτέλεσμα θα είναι:

```
1
0
```

3.5 Εκτύπωση μεταβλητών:

Σε προηγούμενο κεφάλαιο είδαμε πως μπορούμε να εκτυπώσουμε ένα μήνυμα απλά γράφοντας

```
printf("message");
```

Πως μπορούμε όμως να εκτυπώσουμε μια μεταβλητή?

Έστω η μεταβλητή

```
int a = 5;
```

Η εκτύπωση της μεταβλητής είναι εφικτή με την εξής γραμμή.

```
printf("%d", a);
```

Το %d πληροφορεί τον compiler ότι σε αυτή τη θέση πρέπει να βάλει μια ακέραια μεταβλητή.

Πως ξέρει ότι είναι ακέραια ? Από το γράμμα “d”, αν υπήρχε άλλο γράμμα όπως το “f” θα περίμενε για float μεταβλητή.

Ποια μεταβλητή βάζει σε αυτή τη θέση ? Την μεταβλητή που ακολουθεί μετά το κόμμα. Στην προκειμένη περίπτωση την μεταβλητή “a”.

Αυτά τα γράμματα d,f μαζί με το σύμβολο % λέγονται format specifiers και δείχνουν στον compiler τι είδους μεταβλητής μπαίνει σε εκείνη την θέση. Παρακάτω ακολουθεί ένας πίνακας με όλα τα format specifiers:

Data Type	Format Specifier
int	%d ή %i
unsigned int	%u
short int	%hd ή %hi
unsigned short int	%hu
long int	%ld ή %li
unsigned long int	%lu
long long int	%lld ή %lli
unsigned long long int	%llu
float	%f ή %F
double	%lf ή %lF
long double	%Lf ή %LF
char	%c
unsigned char	%c

Προφανώς είναι εφικτή η εκτύπωση δύο αριθμών με τον εξής τρόπο:

```
printf("%d, %d", a, b);
```

και η ενσωμάτωση τους μαζί με μήνυμα:

```
printf("a = %d ", a);
```

Αν έχουμε μια μεταβλητή char f = 120; Τότε μπορούμε να χρησιμοποιήσουμε το %d και να εκτυπώσουμε τον ακέραιο ενώ η ίδια μεταβλητή με %c θα εμφανίσει ότι αντιστοιχεί στο 120 που είναι το x.

Αν προσπαθήσουμε να εκτυπώσουμε τιμή εκτός των ορίων, πχ char f = 128 τότε θα γίνει overflow – υπερχείλιση και θα κάνει κύκλο και θα πάει στο -127, ωστόσο θα εμφανίσει και μήνυμα error πρόγραμμα.

3.6 sizeof():

Το sizeof() είναι ένας πολυχρησιμοποιούμενος τελεστής που υπολογίζει πόσο χώρο πιάνει μια μεταβλητή στην μνήμη (πλήθος bytes). Αυτές τις ενσωματωμένες στην γλώσσα λέξεις που έχουν μια συγκεκριμένη λειτουργία τις λέμε keywords. Η δομή του είναι:

```
sizeof(Expression);
```

Για παράδειγμα:

```
int b = 100;
printf("int b = %d and size = %d\n", b, sizeof(b));
```

Οι παραπάνω γραμμές δηλώνουν μια ακέραια μεταβλητή με όνομα b που έχει την τιμή 100, εκτυπώνεται η τιμή της μεταβλητής με το κατάλληλο μήνυμα και στην συνέχεια πόσα bytes καταναλώνει από την μνήμη.

Η παραπάνω δομή μπορεί να χρησιμοποιηθεί για την εξακρίβωση τον προηγουμένων πινάκων.

4. Τελεστές:

4.1 Αριθμητικοί Τελεστές και Casting:

Οι κυριότεροι αριθμητικοί τελεστές είναι:

1. Πρόσθεση: + (int z = x + y;)
2. Αφαίρεση: - (int z = x - y;)
3. Πολλαπλασιασμός: * (int z = x * y;)

4. Διαίρεση: / (int z = x / y;)
5. Modulus: % (int z = x % y;)
6. Αύξηση: ++
7. Μείωση: --

Διαίρεση: Έστω int x = 5 και int y = 2.

Τότε για την διαίρεση το z θα είναι 5 / 2. Εμείς γνωρίζουμε από τα μαθηματικά ότι το αποτέλεσμα αυτής της πράξης είναι 2.5. Ωστόσο επειδή όλες μας οι μεταβλητές είναι ακέραιες το z θα πάρει μόνο την ακέραια τιμή της διαίρεσης δηλαδή το 2. Αυτό γιατί αυτό που μας νοιάζει στην διαίρεση δεν είναι το σε τι είδος μεταβλητής αποθηκεύεται το αποτέλεσμα, αλλά τι είδους μεταβλητής είναι οι δύο συντελεστές, έτσι εδώ επειδή έχουμε διαίρεση ακεραίων αριθμών, πάντα θα «πετιέται» το δεκαδικό μέρος.

Πως μπορεί να γίνει σωστά η διαίρεση ?

Αρχικά, πρέπει να δηλώσουμε το z ως float ή double (έστω ότι θέλουμε float) ώστε να μπορεί να αποθηκεύσει δεκαδικό αριθμό. Στη συνέχεια αν δοκιμάσουμε να το τρέξουμε θα πάρουμε 2.000000, δηλαδή το πρόβλημα εξακολουθεί να υπάρχει. Ο

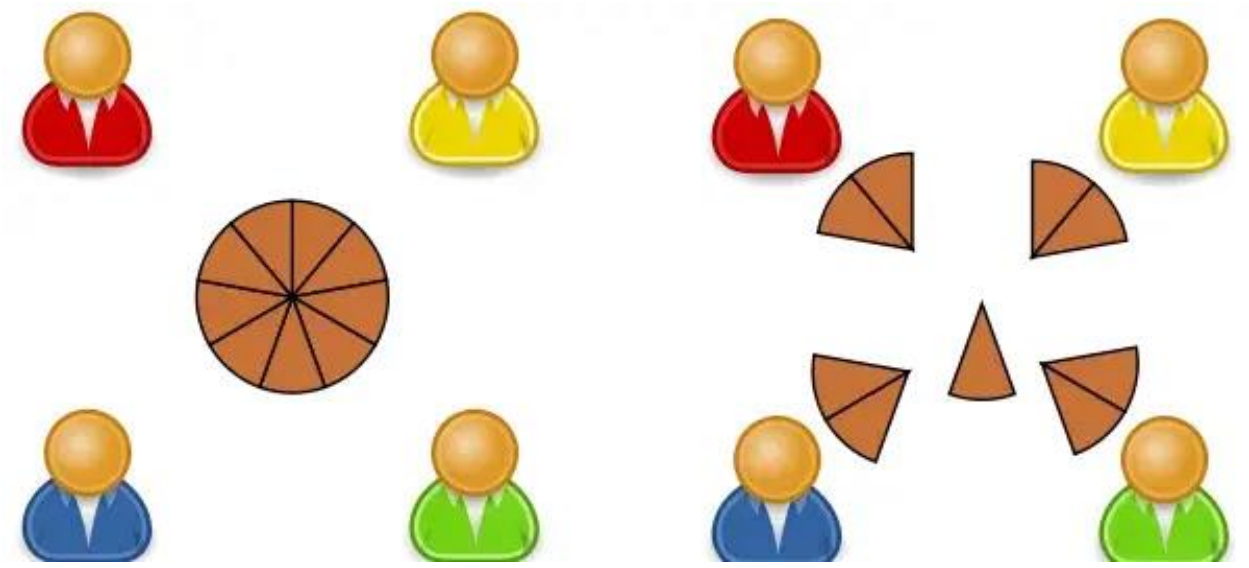
Για να λύσουμε αυτό το πρόβλημα υπάρχουν δύο επιλογές:

- Πρώτον μπορεί μια από τις δύο x, y μεταβλητές να δηλωθεί επίσης ως float πχ float y = 2; Αυτό μπορεί να λύνει το παραπάνω πρόβλημα αλλά μπορεί να επίσης να ανοίγει την πόρτα και σε ένα μεγαλύτερο πρόβλημα. Αν είχαμε ένα μεγαλύτερο πρόγραμμα και αυτή η μεταβλητή χρησιμοποιούταν και αλλού, δεν είναι συνετό να την αλλάζαμε είδος επειδή μας βολεύει για μια πράξη καθώς μπορεί να χαλάμε κάποια άλλη πράξη σε άλλο σημείο του προγράμματος.
- Δεύτερη επιλογή είναι μια προσωρινή μετατροπή της μεταβλητής σε float, έτσι δεν αλλάζουμε τον τύπο της σε όλο το πρόγραμμα

αλλά μόνο για την πράξη που μας νοιάζει, αυτή η μέθοδος λέγεται casting και υλοποιείται έτσι:

```
int z = x / (float)y;
```

Modulus: Πρόκειται για έναν τελεστή που μας δίνει για αποτέλεσμα το υπόλοιπο της ακέραιας διαίρεσης, δηλαδή:



Αύξηση: Γενικά μπορούμε να αυξήσουμε την τιμή μιας μεταβλητής γράφοντας x = x + 1, αλλά ένας πιο γρήγορος τρόπος είναι x++.

Μείωση: Αντίστοιχα αντί για x = x – 1, έχουμε x--. Σημείωση ισχύει μόνο για αύξηση / μείωση με το 1.

Έστω μια διαίρεση το αποτέλεσμα της οποίας είναι 2.2 και το κρατάει η float μεταβλητή z. Αν εκτυπώσουμε την μεταβλητή z τότε θα γεμίσει τις κενές θέσεις μέχρι 6-7 ψηφία με μηδενικά, ένας τρόπος να το αποφύγουμε αυτό είναι να περιορίσουμε το αποτέλεσμα να εμφανίζει μόνο 1 δεκαδικό μετά το κόμμα, αυτό γίνεται από τον ειδικό χαρακτήρα %.1 πριν το f, δηλαδή:

```
printf("%.1f", z);
```

4.2 Τελεστές Ανάθεσης:

Έστω μια μεταβλητή x = 10 και έστω ότι θέλουμε να την αυξήσουμε κατά 5. Από όσα ξέρουμε μπορούμε να γράψουμε πέντε φορές x++, ή μπορούμε να γράψουμε x = x + 5. Υπάρχει και ένας τρίτος τρόπος όμως και αυτός είναι να γράψουμε x += 5. Αυτή η μορφή επεκτείνεται και στις άλλες πράξεις.

Τελεστής	**Αντίστοιχο**
x += 3	x = x + 3
x -= 3	x = x - 3
x *= 3	x = x * 3
x /= 3	x = x / 3
x %= 3	x = x % 3

4.3 Τελεστές Σύγκρισης:

Οι τελεστές σύγκρισης χρησιμοποιούνται για τη σύγκριση δύο τιμών (ή μεταβλητών). Αυτό είναι σημαντικό στον προγραμματισμό, γιατί μας βοηθά να βρίσκουμε απαντήσεις και να παίρνουμε αποφάσεις.

Η επιστρεφόμενη τιμή μιας σύγκρισης είναι είτε 1 είτε 0, που σημαίνει true (1) ή false (0).

Στο παρακάτω παράδειγμα, χρησιμοποιούμε τον τελεστή μεγαλύτερο από (>) για να μάθουμε εάν το 5 είναι μεγαλύτερο από το 3:

```
int x = 5;
int y = 3;
printf("%d", x > y); // returns 1 (true) because 5 is
greater than 3
```

Μια λίστα με τους τελεστές σύγκρισης.

Τελεστής	**Σημασία**
==	Ίσο με

!=	Όχι ίσο
>	Μεγαλύτερο από
<	Μικρότερο από
>=	Μεγαλύτερο ίσο
<=	Μικρότερο ίσο

Σημείωση: ένα = είναι για ανάθεση τιμής, ενώ διπλό == είναι για έλεγχο ισότητας.

4.4 Λογικοί Τελεστές:

Μπορείτε επίσης να ελέγξετε για αληθείς ή ψευδείς τιμές με λογικούς τελεστές.

Οι λογικοί τελεστές χρησιμοποιούνται για τον προσδιορισμό της λογικής μεταξύ μεταβλητών ή τιμών, συνδυάζοντας πολλαπλές συνθήκες:

Τελεστής	**Σημασία – Όνομα**	**Παράδειγμα**
&&	AND	x < 5 && x < 10
||	OR	x < 5 || x < 4
!	NOT	!(x < 5 && x < 10)

5. Δομές Ελέγχου:

5.1 if statement:

Συχνά στα προγράμματα απαιτείται να γίνεται κάποιος έλεγχος στις μεταβλητές, για παράδειγμα σε ένα πρόγραμμα καζίνου πρέπει να ελέγχεται ότι η ηλικία είναι μεγαλύτερη των 21. Σε προηγούμενο κεφάλαιο είδαμε ποιον τελεστή χρησιμοποιούμε για σύγκριση τιμών και εδώ θα μάθουμε πως τον χρησιμοποιούμε.

```
int age = 20;
if (age >= 21)
{
   printf("You can enter");
```

```
    }
    return 0;
```

Το παραπάνω πρόγραμμα ελέγχει αυτό ακριβώς, αν δηλαδή η ηλικία είναι μεγαλύτερη των 21 τότε μπορεί να εισέλθει στο καζίνο αλλιώς όχι. Αυτό γίνεται με την δομή ελέγχου if.

Η δομή ξεκινάει με την λέξη if ακολουθούμενη από μια παρένθεση μέσα στην οποία γίνεται ο έλεγχος μας. Τα δύο άγκιστρα παρακάτω περιέχουν τον κώδικα που θα εκτελεστεί αν η παραπάνω σύγκριση είναι αληθής. Αν δεν είναι αληθής τότε δεν εκτελούνται και το πρόγραμμα συνεχίζει με τις από κάτω γραμμές. Στο συγκεκριμένο πρόγραμμα η παρακάτω γραμμή είναι η return 0; οπότε θα τερματίσει. Μπορούμε να εμπλουτίσουμε το πρόγραμμα εμφανίζοντας κάποιο μήνυμα και στην περίπτωση που ο έλεγχος είναι ψευδής, αυτό γίνεται έτσι:

```
    int age = 20;
    if (age >= 21)
    {
        printf("You can enter");
    }
    else
    {
        printf("You shall not pass");
    }
```

Το else μαζί με τον κώδικα μέσα στις αγκύλες είναι για την περίπτωση που ο έλεγχος στο if βγάλει ψευδή. Εδώ αν είναι αληθής ο πρώτος έλεγχος τότε κανονικά εμφανίζεται το μήνυμα You can enter και το πρόγραμμα προσπερνάει τον κώδικα του else, αν όμως ο έλεγχος βγει ψευδής τότε εκτελεί τον κώδικα που βρίσκεται μέσα στο else. Βλέπουμε ότι στο else δεν υπάρχει παρένθεση με έλεγχο, αυτό γιατί το else εκτελείται σε οποιαδήποτε περίπτωση δεν ισχύει το if.

Γενικά βλέπουμε ότι μια δομή ελέγχου if είναι αυτοτελής, δηλαδή δεν απαιτείται να μπει και else, όταν όμως προστίθεται και το else είναι αλυσιδωτά δηλαδή αν εκτελεστεί το if δεν εκτελείται το else και αντίστοιχα όταν εκτελείται το else είναι επειδή δεν εκτελέστηκε το if.

Υπάρχει και μια τρίτη επιλογή και αυτή είναι του else if. Μπαίνει ανάμεσα από το if και το else και αν δεν υπάρχει else τότε είναι απλά κάτω από το if. Μια τέτοια δομή είναι:

```
int age = 20;
if (age >= 21)
{
   printf("You can enter");
}
else if (age < 18)
{
   printf("You are not even an adult...");
}
else
{
   printf("You shall not pass");
}
```

Αν ο πρώτος έλεγχος είναι αληθής τότε εκτελείται το πρώτο κομμάτι κώδικα και μετά βγαίνει από το πρόγραμμα από την δομή. Αν είναι ψευδής τότε ελέγχεται το else if και για αληθής εκτελείται και για ψευδής πάει στο else.

Δεν γίνεται να χρησιμοποιηθούν δύο δομές if ? Γίνεται προφανώς, απλά σε πολλές περιπτώσεις όπως θα δούμε παρακάτω είναι περισσότερο μπελάς από το να χρησιμοποιηθεί απλά ένα else if.

Κάτι ακόμα πολύ χρήσιμο είναι ότι μπορούμε να βάλουμε όσα else if θέλουμε σε μια δομή.

5.2 switch statement:

Το switch statement είναι ένας πιο αποδοτικός τρόπος να κάνουμε πολλαπλούς ελέγχους, αντί δηλαδή να έχουμε πολλά else if, χρησιμοποιούμε την δομή switch. Προσοχή μπορεί να ελέγξει μόνο για ισότητα και όχι για μικρότερο η μεγαλύτερο.

```
switch (age)
{
case 18:
   printf("You are an adult");
   break;
case 21:
   printf("You are eligible for casino");
   break;
default:
   printf("age value is not 18 or 21");
   break;
}
```

Η δομή ξεκινάει με την λέξη switch, στην παρένθεση τοποθετείται η μεταβλητή που εξετάζεται και κάθε case έχει μια τιμή η οποία συγκρίνεται για ισότητα με την μεταβλητή, αν είναι αληθής τότε εκτελείται το κομμάτι κώδικα μέσα σε αυτό το case. Με το keyword break το πρόγραμμα σταματάει να ελέγχει τα άλλα cases, δηλαδή βγαίνει από την δομή. Αν δεν υπήρχε το break τότε θα έλεγχε για κάθε case. Η default περίπτωση είναι κάτι σαν το else στην δομή if, εκτελείται για οποιαδήποτε άλλη τιμή από τα cases.

6. Δομές Επανάληψης:

Έστω ένα πρόγραμμα στο οποίο απαιτείται η εμφάνιση αριθμών από το 1 έως το 10. Σύμφωνα με όσα έχουν διδαχτεί η λύση σε αυτό το πρόβλημα είναι να δηλωθούν δέκα μεταβλητές η κάθε μία με μία τιμή από το 1 έως

το 10 και εκτυπώνεται κάθε μία, ή αρκετά καλύτερα μία μεταβλητή που μετά από κάθε εκτύπωση αυξάνεται κατά ένα. Δηλαδή να μοιάζει κάπως έτσι:

```
int a = 1;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
a++;
printf("a = %d\n", a);
```

Τώρα μπορεί να είναι μόνο για δέκα τιμές αλλά τι θα γινόταν αν είχαμε 100 ή ακόμα και 1000. Γιαυτό το λόγο υπάρχουν οι δομές επανάληψης όπου ένα κομμάτι κώδικα που είναι να εκτελεστεί πολλές φορές το γράφουμε μια φορά και ορίζουμε πόσες φόρες θα εκτελεστεί.

Εκτός από δομές επανάληψης συχνά συναντάται και ο όρος βρόχος επανάληψης.

6.1 Δομή while:

Η πρώτη τέτοια δομή που θα εξεταστεί είναι η while με την εξής σύνταξη:

```
int a = 1;
while (a <= 10)
{
   printf("a = %d\n", a);
   a++;
}
```

Στην αρχή το πρόγραμμα δημιουργεί μια μεταβλητή a με τιμή το 1. Όταν φτάνει στην while ελέγχετε αν είναι μικρότερη η τιμή της a από το 10 και αν είναι τότε μπορεί να μπει μέσα στην λούπα όπως λέγεται. Εκεί εκτυπώνεται η τιμή της a και στην συνέχεια αυξάνεται κατά ένα. Στην συνέχεια ξανά περνάει από τον έλεγχο αν το a είναι μικρότερο του 10, αν δεν είναι τότε ξαναμπαίνει στην λούπα άρα ξαναεκτελείται ο κώδικας. Έτσι μόνο τέσσερις γραμμές κώδικα εκτελούν ότι χρειάστηκε πριν είκοσι γραμμές.

Αν ξεχνούσαμε να γράψουμε a++ η τιμή του a δεν θα έφτανε ποτέ να γίνει 10 άρα η εκτύπωση θα εκτελούνταν για πάντα (ή πιο ρεαλιστικά μέχρι να τελειώσει η μνήμη), αυτό λέγεται ατέρμων βρόχος.

6.2 Δομή do/while:

Μοιάζει πολύ στην while με μόνη διαφορά ο έλεγχος γίνεται στο τέλος της επανάληψης και όχι στην αρχή, άρα πάντα θα εκτελεστεί τουλάχιστον μια φόρα, εν αντιθέσει με την while που μπορεί να εκτελεστεί και καμία φορά (για παράδειγμα a = 20). Επομένως χρησιμοποιείται σε περιπτώσεις που σίγουρα θέλουμε μια επανάληψη.

```
int b = 1;
do
{
```

```
        printf("b = %d\n", b);
        b++;
    } while (b <= 10);
```

6.3 Δομή for:

Η for είναι η πιο χρησιμοποιούμενη δομή επανάληψης και ορίζεται ως εξής:

```
    for (int i = 0; i <= 10; i++)
    {
        printf("i = %d\n", i);
    }
```

Γενικά στις δομές επανάληψης για μεταβλητή που μετρά τις επαναλήψεις χρησιμοποιείται το i. Βλέπουμε πως σε αυτή τη δομή δεν χρειάζεται να έχουμε ορίσει από πριν την μεταβλητή αλλά μπορούμε μέσα στην δομή, αν και προφανώς γίνεται να έχει οριστεί προηγουμένως. Στην παρένθεση σημαντική σημείωση ότι χωρίζονται από ερωτηματικό οι εντολές και όχι κόμμα.

Είναι επίσης δυνατό να τοποθετηθεί ένας βρόχος for μέσα σε έναν άλλο. Αυτό ονομάζεται εμφωλευμεύνος βρόχο (nested loop).

Ο "εσωτερικός βρόχος" θα εκτελείται μία φορά για κάθε επανάληψη του "εξωτερικού βρόχου". Συνηθίζεται η μεταβλητή του εξωτερικού βρόχου να είναι το i και του εσωτερικού το j.

```
    // Outer loop
    for (int i = 1; i <= 2; ++i)
    {
        printf("Outer: %d\n", i); // Executes 2 times
        // Inner loop
        for (int j = 1; j <= 3; ++j)
        {
```

```
            printf(" Inner: %d\n", j); // Executes 6 times
(2 * 3)
        }
    }
```

Με αποτελέσματα:

```
Outer: 1
 Inner: 1
 Inner: 2
 Inner: 3
Outer: 2
 Inner: 1
 Inner: 2
 Inner: 3
```

6.4 Break / Continue:

Όπως είδαμε σε προηγούμενο κεφάλαιο με το keyword break σταματούσε ο έλεγχος στα cases του switch, έτσι και στις δομές επανάληψης μπορεί να σταματήσει η επανάληψη, για παράδειγμα:

```
    for (int i = 0; i < 10; i++)
    {
        if (i == 4)
        {
            break;
        }
        printf("%d\n", i);
    }
```

Στο παραπάνω πρόγραμμα μόλις εμφανιστεί το 4 σταματάει η επανάληψη και το πρόγραμμα βγαίνει από την for.

Έστω όμως ότι δεν θέλουμε να σταματάει στο 4 αλλά να το προσπερνάει, αυτό γίνεται με το keyword continue αντί για break:

```c
for (int i = 0; i < 10; i++)
{
    if (i == 4)
    {
        continue;
    }
    printf("%d\n", i);
}
```

Έτσι όταν εμφανιστεί το 4 θα το προσπεράσει το πρόγραμμα και θα πάει απευθείας στο 5.

Προφανώς η χρήση του break και continue δεν περιορίζεται μόνο για τις δομές switch και for αλλά όπου μπορεί να υπάρξει διακοπή της κανονικής πορείας.

7. Πίνακες:

Οι πίνακες χρησιμοποιούνται για την αποθήκευση πολλαπλών τιμών σε μια μεμονωμένη μεταβλητή, αντί να δηλώνονται ξεχωριστά μεταβλητές για κάθε τιμή. Αν στις μεταβλητές ένα όνομα αντιστοιχούσε σε ένα κουτάκι μνήμης, εδώ ένα όνομα αντιστοιχεί σε πολλά κουτάκια.

Δηλαδή αντί για:

a = 5

Έχουμε:

a [0] = 5
a [1] = 6
a [2] = 7
a [3] = 8

Όπου το σύνολο των κελιών ονομάζονται a και κάθε κελί όπως θα δούμε παρακάτω αναγράφεται με το όνομα του πίνακα και έναν αριθμό.
Για να δημιουργηθεί ένας πίνακας, αρχικά ορίζεται ο τύπος δεδομένων (όπως int) και στην συνέχεια καθορίζεται το όνομα του πίνακα ακολουθούμενο από αγκύλες []. Προσοχή ένας πίνακας δεν μπορεί να περιέχει πολλαπλούς τύπους δεδομένων, δηλαδή δεν μπορεί να έχει και ακέραιους και χαρακτήρες. Αν δηλωθεί ακέραιος (int) τότε μπορεί να περιέχει μόνο ακέραιους.
Οι τιμές εκχωρούνται με τον εξής τρόπο:

```
int myNumbers[] = {25, 50, 75, 100};
```

Όπως βλέπουμε τον πίνακα μπορούμε να καταλάβουμε ότι έχει 4 στοιχεία ή κελιά όπως αλλιώς λέγονται. Για να αναφερθούμε στο πρώτο στοιχείο γράφουμε

```
myNumbers[0];
```

Αντίστοιχα με 1,2,3 και για τα επόμενα.

Για αλλαγή της τιμής ενός κελιού γράφουμε:

```
myNumbers[0] = 33;
```

Για παράδειγμα:

```
int myNumbers[] = {25, 50, 75, 100};
myNumbers[0] = 33;

printf("%d", myNumbers[0]);
// Now outputs 33 instead of 25
```

Για να εκτυπώσουμε όλα τα στοιχεία / κελιά του πίνακα χρησιμοποιείται η for δομή:

```
for (int i = 0; i < 4; i++)
{
   printf("%d\n", myNumbers[i]);
```

```
}
```

Γράφοντας i < 4 σημαίνει ότι η επανάληψη γίνεται μέχρι και την τελευταία ακέραια τιμή (int i) που είναι μικρότερη του 4 δηλαδή το 3. Άρα είναι σαν να γράφει i <= 3.

Καμιά φορά όμως δεν είναι γνωστό εξαρχής ποια στοιχεία θα κρατάει ο πίνακας οπότε δηλώνεται απλά το πόσα θα κρατήσει με τον εξής τρόπο:

```
int myNumbers[4];
```

Στην συνέχεια μπορούν να του ανατεθούν τιμές με τον εξής τρόπο:

```
myNumbers[0] = 25;
myNumbers[1] = 50;
myNumbers[2] = 75;
myNumbers[3] = 100;
```

Αφού δημιουργηθεί ένας πίνακας δεν είναι εφικτή η αλλαγή μεγέθους του οπότε πρέπει να είμαστε σίγουροι κατά την δημιουργία του.

7.1 Μέγεθος Πίνακα:

Για να βρεθεί το μέγεθος ενός πίνακα μπορεί να χρησιμοποιηθεί ο τελεστής sizeof.

```
int myNumbers[] = {10, 25, 50, 75, 100};
printf("%d", sizeof(myNumbers)); // Prints 20
```

Με την εκτέλεση του παραπάνω κώδικα παίρνουμε όμως κάτι αναπάντεχο, εμείς ξέρουμε ότι ο πίνακας έχει 5 στοιχεία αλλά το αποτέλεσμα είναι 20. Αυτό συμβαίνει διότι ο τελεστής sizeof μετράει το μέγεθος ενός στοιχείου σε bytes οπότε μιας και είναι πίνακας ακεραίων και κάθε ακέραιος καταλαμβάνει 4 bytes το μέγεθος του πίνακα σε bytes είναι 4*5 = 20bytes.

Αν θέλουμε να πάρουμε τον αριθμό στοιχείων του πίνακα δηλαδή το 5 πρέπει να γράψουμε:

```
int length = sizeof(myNumbers) / sizeof(myNumbers[0]);
```

Όπου διαιρεί τα συνολικά bytes του πίνακα με αυτά του πρώτου κελιού και έτσι παίρνουμε πόσα κελιά υπάρχουν.

Έτσι πλέον αντί να γράφει ο προγραμματιστής μέχρι ποιο νούμερο να κάνει επαναλήψεις η for, γίνεται αυτόματα με την μεταβλητή length. Δηλαδή αντί για:

```
for (int i = 0; i < 5; i++)
{
   printf("%d\n", myNumbers[i]);
}
```

Πλέον έχουμε:

```
for (int i = 0; i < length; i++)
{
   printf("%d\n", myNumbers[i]);
}
```

7.2 Πολυδιάστατοι Πίνακες:

Με τον όρο πολυδιάστατος πίνακας εννοούμε έναν πίνακα που έχει πολλές διαστάσεις. Ο πίνακας που εξετάστηκε στην προηγούμενη ενότητα ονομάζεται μονοδιάστατος πίνακας μιας και είναι μια απλή συστοιχία κελιών. Ο πιο γνωστός πολυδιάστατος είναι ο δισδιάστατος πίνακας όπου έχει δύο διαστάσεις. Μπορούμε να τον φανταστούμε ως έναν μονοδιάστατο όπου κάθε κελί είναι ένας άλλος μονοδιάστατος.

i/j	Στήλη-1	Στήλη-2	Στήλη-3	Στήλη-4
Σειρά-1	[0][0]	[0][1]	[0][2]	[0][3]
Σειρά-2	[1][0]	[1][1]	[1][2]	[1][3]
Σειρά-3	[2][0]	[2][1]	[2][2]	[2][3]
Σειρά-4	[3][0]	[3][1]	[3][2]	[3][3]
Σειρά-5	[4][0]	[4][1]	[4][2]	[4][3]

Το σύνολο των κάθετων κελιών λέγεται στήλες (columns) και των οριζόντιων σειρές (rows).
Για να δημιουργήσουμε έναν δισδιάστατο πίνακα γράφουμε:

```
int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} };
```

Ο πρώτος αριθμός είναι ο αριθμός των γραμμών και ο δεύτερος ο αριθμός των στηλών. Ισχύει ότι ίσχυε και στον μονοδιάστατο απλά με έξτρα την δεύτερη αγκύλη.

Για την εκτύπωση του πίνακα πλέον απαιτούνται δύο επαναλήψεις for η μία μέσα στην άλλην.

```
for (int i = 0; i < 2; i++)
{
   for (int j = 0; j < 3; j++)
   {
      printf("%d\n", matrix[i][j]);
   }
}
```

8. String:

Μέχρι στιγμής είδαμε τον τύπο δεδομένων char όπου μπορεί να κρατήσει οποιονδήποτε χαρακτήρα ASCII. Για παράδειγμα:

```
char a = 'a';
```

Ωστόσο υπάρχει ένας προφανής περιορισμός, σε κάθε char μεταβλητή μπορούμε να αναθέσουμε μόνο έναν χαρακτήρα, οπότε τι κάνουμε όταν θέλουμε μια μεταβλητή να περιέχει μια ολόκληρη λέξη ?

Αν δοκιμάσουμε να γράψουμε:

```
char a = 'test';
printf("%c", a);
```

Θα εμφανιστεί error και δεν θα τρέξει το πρόγραμμα. Αυτό είναι απολύτως λογικό μιας και όπως αναφέρθηκε προηγουμένως μια μεταβλητή char διαθέτει μόνο 1 byte μνήμης ram και μόνο 1 byte είναι κάθε χαρακτήρας, άρα προσπαθούμε να αποθηκεύσουμε 4 bytes πληροφορίας σε θέση που χωράει 1 byte.

Μια λύση είναι να χρησιμοποιήσουμε 4 μεταβλητές τύπου char όπου η κάθε μία αποθηκεύει έναν χαρακτήρα. Όσο όμως μεγαλώνει το πρόγραμμα μας η συντήρηση αλλά και η ανάγνωση ενός τέτοιου κώδικα γίνεται δυσκολότερη.

Η λύση σε αυτό το πρόβλημα είναι αντί για μια μεταβλητή char να χρησιμοποιήσουμε έναν πίνακα char.

```
char a[] = "test";
printf("%s", a);
```

Αυτός ο πίνακας από χαρακτήρες λέγεται string «συμβολοσειρά» και για την εκτύπωση του απαιτείται %s αντί για %c. Ακόμη για την ανάθεση τιμής πρέπει να χρησιμοποιηθούν διπλά εισαγωγικά

```
" "
```

Και όχι μονά όπως στους μεμονωμένους χαρακτήρες.

Αν επιχειρήσουμε να δούμε τον αριθμό χαρακτήρων του πίνακα με τον τρόπο που περιγράφηκε σε προηγούμενη ενότητα θα πάρουμε κάτι το αναπάντεχο:

```
char a[] = "test";
int length = sizeof(a) / sizeof(a[0]);
printf("%d", length);
```

Αποτέλεσμα:

```
5
```

Όμως φαίνεται πολύ καλά ότι έχει 4 (t, e, s, t) χαρακτήρες γιατί το αποτέλεσμα είναι 5?

Η απάντηση είναι επειδή υπάρχει ένας ακόμα κρυφός χαρακτήρας και αυτός είναι το:

```
\0
```

Είναι γνωστός ως null terminating character και μπαίνει αυτόματα. Η χρησιμότητα του είναι ώστε να ξέρει ο compiler που τελειώνει ο πίνακας. Αν δημιουργούσαμε εμείς τον πίνακα γράφοντας αναλυτικά την τιμή για κάθε κελί τότε θα έπρεπε να τον συμπεριλάβουμε, για παράδειγμα:

```
char a[] = {'t', 'e', 's', 't', '\0'};
printf("%s", a);
```

Πρόκειται για μια τόσο χρήσιμη και σημαντική χρήση πινάκων με χαρακτήρες που άλλες γλώσσες έχουν εισάγει τον πίνακα χαρακτήρων δηλαδή το string ως τύπο δεδομένων (σε πολλές είναι και ενσωματωμένος). Στην C δεν ισχύει κάτι τέτοιο. Ωστόσο έχουν δημιουργηθεί ειδικές βιβλιοθήκες που προσθέτουν λειτουργίες. Μια από αυτές είναι η:

```
#include <string.h>
```

9. Συναρτήσεις (functions):

Συχνά απαιτείται η επανάληψη κάποιου κομματιού κώδικα, σε απλές περιπτώσεις που απλά απαιτείται η επανεμφάνιση ενός μηνύματος μπορεί είτε απλά να επαναληφθεί η printf εντολή

```
printf("message");
printf("message");
```

ή να χρησιμοποιηθεί κάποια δομή επανάληψης.

```
for (int i = 0; i <= 2; i++)
{
   printf("message");
}
```

Τι γίνεται όμως στην περίπτωση που η επανάληψη κώδικα που πρέπει να γίνει περιλαμβάνει πολλές και σύνθετες εντολές ?

Τότε μπορεί να χρησιμοποιηθεί κάτι που λέγεται συνάρτηση (function).

```
dataType myFunction(parameters)
{
   // code to be executed
}
```

Αλλά ας μην προτρέχουμε

Ήδη γνωρίζουμε και έχουμε χρησιμοποιήσει μια συνάρτηση. Η γνωστή σε όλους μας int main() είναι μια συνάρτηση.

```
#include <stdio.h>
int main()
{
   printf("message");
   return 0;
}
```

Λέγεται main γιατί είναι η βασική συνάρτηση που ψάχνει ο υπολογιστής όταν είναι να εκτελέσει το πρόγραμμα. Αφού βρει την main ξεκινάει και εκτελεί τις εντολές της. Βλέποντας την συνάρτηση, παρατηρούμε ότι δεν διαφέρει και τόσο από την δήλωση μιας μεταβλητής. Υπάρχει το όνομα της συνάρτησης (main) και τι τύπου είναι (int). Κάθε συνάρτηση πρέπει να έχει έναν τύπο, λέγονται τύποι επιστροφής.

Η τελευταία γραμμή στο παραπάνω πρόγραμμα είναι return 0; Αυτό που κάνει είναι επιστρέφει την τιμή 0. Την επιστρέφει στην συνάρτηση στην οποία ανήκει δηλαδή στην main. Ακολουθεί, μια σύντομη περιγραφή του τι γίνεται όταν τρέχουμε ένα πρόγραμμα:

- Το λειτουργικό σύστημα τρέχει το πρόγραμμα, βρίσκει και φορτώνει την main συνάρτηση στην μνήμη και ξεκινάει να εκτελεί τις εντολές της.
- Το συγκεκριμένο πρόγραμμα έχει μόνο μια εντολή και αυτή είναι να εκτυπώσει την λέξη message στο τερματικό
- Η επόμενη εντολή είναι η return 0 όπου επιστρέφει την τιμή 0 στην main και τερματίζεται το πρόγραμμα. Αν υπήρχαν εντολές κάτω από αυτή την γραμμή τότε δεν θα εκτελούνταν ποτέ καθώς το πρόγραμμα τελειώνει μόλις του επιστραφεί κάποιος αριθμός. Αυτό σημαίνει ότι αντί για 0 μπορούμε να βάλουμε να επιστρέφει το 1 ή οποιονδήποτε άλλο αριθμό. Απλά έχει γίνει μια σύμβαση όταν ένα πρόγραμμα εκτελείται σωστά να επιστρέφει το 0 και όταν συναντά κάποια απρόσμενη έξοδο που οφείλεται σε σφάλμα τότε 1 ή άλλον αριθμό (ανάλογα το σφάλμα).
- Γιατί αριθμός ? Γιατί η main είναι τύπου int οπότε δέχεται μόνο ακέραιες τιμές.
- Πως ξέρει ότι εκτελέστηκε σωστά ? Αφού έφτασε στο τέλος του προγράμματος δηλαδή στην τελευταία γραμμή τότε μπορούμε να πούμε ότι εκτελέστηκε σωστά.
- Κάθε συνάρτηση περιμένει να της επιστραφεί μια τιμή ανάλογα με τον τύπο της. Αυτό ονομάζεται τύπος επιστροφής. Η main για παράδειγμα έχει τύπο επιστροφής ακέραιο (φαίνεται από την λέξη int στην δήλωση).
- Η C99 έκδοση όρισε πως όταν μια συνάρτηση φτάνει στο τέλος της χωρίς να έχει επιστρέψει κάποιον κωδικό λάθους τότε αντιμετωπίζεται σαν να επέστρεψε το 0. Γιαυτό σε μοντέρνα προγράμματα καμιά φορά δεν βάζουμε return 0.

Πάμε να φτιάξουμε μια συνάρτηση:

Αρχικά πρέπει να την ορίσουμε, όπως είπαμε πριν μια συνάρτηση ορίζεται με τον εξής τρόπο:

```
dataType myFunction(parameters)
{
   // code to be executed
}
```

Δηλαδή έχουμε τον τύπο της συνάρτησης και μετά το όνομα της με παρενθέσεις. Μέσα στις παρενθέσεις μπαίνουν οι παράμετροι / ορίσματα, αυτό συνήθως είναι μεταβλητές που θέλουμε να μεταφέρουμε στην συνάρτηση και είναι καθαρά προαιρετικές, περισσότερα παρακάτω. Τέλος μέσα στις αγκύλες όπως και στην main μπαίνει ο κώδικας που θα εκτελεί η συνάρτηση.

Που τοποθετείτε όμως μέσα στον κώδικα μας:

Πρώτη επιλογή πάνω από την main:

```
#include <stdio.h>

dataType myFunction(parameters)
{
   // code to be executed
}

int main()
{
   printf("message");
   return 0;
   printf("message2");
}
```

Δεύτερη επιλογή κάτω από την main:

```
#include <stdio.h>
```

```c
int main()
{
   printf("message");
   return 0;
   printf("message2");
}

dataType myFunction(parameters)
{
   // code to be executed
}
```

Δεν μπορεί μια συνάρτηση να τοποθετηθεί μέσα στην main και ούτε και σε κάποια άλλη συνάρτηση. Κάθε συνάρτηση είναι ξεχωριστό μέρος στο πρόγραμμα.

Όπως είπαμε πριν όταν τρέχει το πρόγραμμα ξεκινάει να εκτελεί τις εντολές της main, άρα πως γίνεται να εκτελεστεί κώδικας που είναι μέσα σε άλλη συνάρτηση. Η απάντηση είναι πως πρέπει να «καλέσουμε», όπως ονομάζεται, αυτή την συνάρτηση μέσα στην main.

Το παραπάνω γίνεται κατανοητό με ένα παράδειγμα:

```c
#include <stdio.h>

int myFunction()
{
   return 2;
}

int main()
{
   int a = myFunction();
```

```
    printf("a = %d\n", a);
    return 0;
}
```

Στο παραπάνω πρόγραμμα δηλώνεται μια ακέραιου τύπου συνάρτηση που απλά επιστρέφει την τιμή 2. Στην main δηλώνεται μια ακέραια μεταβλητή και βλέπουμε της ανατίθεται αυτό: myFunction()

Πως γίνεται σε μια ακέραια μεταβλητή να αντιστοιχεί λέξη ?

Το παραπάνω δεν είναι λέξη αλλά κλήση της συνάρτησης που επιστρέφει την τιμή 2, άρα δεν ανατίθεται η λέξη myFunction() στην μεταβλητή a αλλά αυτό που επιστρέφει δηλαδή το 2.

Αν και δεν περνάμε κάποια παράμετρο στην myFunction πάλι πρέπει να βάλουμε τις παρενθέσεις απλά κενές.

Τι γίνεται όμως όταν θέλουμε να κάνουμε μια συνάρτηση που δεν επιστρέφει κάποιον αριθμό αλλά να εκτυπώνει ένα μήνυμα ?

Επειδή είναι συνάρτηση τότε πρέπει πάλι να επιστρέφει κάτι αλλά μπορούμε να την κάνουμε να επιστρέφει το τίποτα το οποίο θεωρείται κάτι. Αυτό γίνεται με το να την θέσουμε τύπου void, είναι ο τύπος που αντιπροσωπεύει το τίποτα / κενό.

```
#include <stdio.h>

void myFunction()
{
    printf("Hello World!");
    return;
}

int main()
{
```

```
    myFunction();
    return 0;
}
```

Με τις παραπάνω αλλαγές δηλώνουμε μια void τύπου συνάρτηση που απλά επιστρέφει μήνυμα Hello World!. Τώρα επειδή δεν είναι κάποιος τύπος μεταβλητής δεν μπορεί να ανατεθεί σε μεταβλητή και καλείται απλά με το όνομα της. Γιαυτό δεν διαθέτει και τίποτα μετά το return και είναι κενό. Σε αυτή την περίπτωση μπορεί να παραληφθεί το return. Στους άλλους τύπος όμως όπως int, float… δεν μπορεί να παραληφθεί.

Άρα είδαμε ότι υπάρχουν int και void συναρτήσεις, τι άλλοι τύποι όμως μπορούν να υπάρξουν ? Η λίστα αποτελείται από όλους τους πρωτεύων τύπους δεδομένων και τους derived τύπους.

Ακόμα πιθανή είναι μια συνάρτηση να καλεί μια άλλη συνάρτηση:

```
void myFunction()
{
    printf("Some text in myFunction\n");
    myOtherFunction();
}

void myOtherFunction()
{
    printf("Hey! Some text in myOtherFunction\n");
}

int main()
{
    myFunction();
    return 0;
}
```

9.1 Παράμετροι:

Οι μεταβλητές που δηλώνονται σε μια συνάρτηση μπορούν να χρησιμοποιηθούν μόνο μέσα σε αυτή, για παράδειγμα αν η main ορίσει μια μεταβλητή a, τότε αυτή η μεταβλητή υπάρχει μόνο μέσα στην main και μπορεί να χρησιμοποιηθεί μόνο μέσα σε αυτήν. Για να πάρει πρόσβαση σε αυτή την μεταβλητή μια συνάρτηση πρέπει στο κάλεσμα της να περαστεί παραμετρικά, δηλαδή ως παράμετρος.
Οι παράμετροι μπορούν να χρησιμοποιηθούν για να περαστεί πληροφορία μέσα στην συνάρτηση:

```c
#include <stdio.h>

int myFunction(int a)
{
   return a+2;
}

int main()
{
   int a = 1;
   printf("a = %d\n", a);
   printf("a + 2 = %d\n", myFunction(a));
   return 0;
}
```

Το παραπάνω πρόγραμμα αποτελεί ένα παράδειγμα αυτού. Περνάει ως παράμετρος η μεταβλητή a στην συνάρτηση και επιστρέφεται η τιμή της a + 2. Στην δήλωση της συνάρτησης πρέπει να προσδιοριστεί ο τύπος και το πλήθος των παραμέτρων που δέχεται. Στην προκειμένη περίπτωση ορίζεται ότι θα δέχεται μια ακέραια μεταβλητή. Όταν καλείται η συνάρτηση πρέπει να δοθεί το όνομα μιας ακέραιας μεταβλητής αλλιώς θα υπάρξει σφάλμα. Στην συνάρτηση δεν περνάει η πραγματική

μεταβλητή αλλά ένα αντίγραφο αυτής (call / pass by value), οπότε είναι αδύνατη η γενική αλλαγή της τιμής της a και ότι επεξεργασία γίνεται πάνω της αφορά την συνάρτηση myFunction, πρακτικά δηλαδή πρόκειται για μια νέα μεταβλητή. Ακόμη είναι δυνατό να χρησιμοποιηθεί άλλο όνομα στην συνάρτηση αντί για a:

```c
#include <stdio.h>

int myFunction(int b)
{
   return b + 2;
}

int main()
{
   int a = 1;
   printf("a = %d\n", a);
   printf("a + 2 = %d\n", myFunction(a));
   return 0;
}
```

Εφικτό είναι να περαστεί και πίνακας παραμετρικά:

```c
#include <stdio.h>

void myFunction(int myNumbers[5])
{
   for (int i = 0; i < 5; i++)
   {
      printf("%d\n", myNumbers[i]);
   }
}
```

```c
int main()
{
   int myNumbers[5] = {10, 20, 30, 40, 50};
   myFunction(myNumbers);
   return 0;
}
```

Ένα θετικό των συναρτήσεων είναι ότι μπορούμε να χρησιμοποιήσουμε την συνάρτηση για διαφορετική μεταβλητή κάθε φορά και να γλυτώσουμε γραμμές κώδικα καθώς διαφορετικά έπρεπε να ξαναγράφουμε μια εντολή για κάθε μεταβλητή. Ακόμα γίνεται πιο ευανάγνωστος ο κώδικας μας και ευκολότερος στην συντήρηση. Το τελευταίο τώρα δεν είναι εμφανές αλλά θα γίνει όσο αυξάνεται το μέγεθος και η πολυπλοκότητα ενός προγράμματος.

Ακόμη καλή προγραμματιστική συνήθεια είναι κάθε συνάρτηση να εξυπηρετεί έναν συγκεκριμένο σκοπό και να δέχεται παραμέτρους μόνο για την εκπλήρωση του σκοπού αυτού.

9.2 Global / Local μεταβλητές:

Όπως αναφέρθηκε οι μεταβλητές που δηλώνονται μέσα σε μία συνάρτηση (myFunction, main…) είναι προσβάσιμες μόνο μέσα από την συνάρτηση. Γιαυτό είναι γνωστές και ως local δηλαδή τοπικές μεταβλητές.

Υπάρχει ακόμα ένας τύπος μεταβλητών γνωστός ως global δηλαδή παγκόσμιες, αυτές δηλώνονται εκτός συνάρτησης (συνήθως κάτω από τα #include) και είναι προσβάσιμες από όλες τις συναρτήσεις:

```c
#include <stdio.h>

int x = 5;

void myFunction()
```

```
{
   printf("%d\n", x);
   int x = 1;
   printf("%d\n", x);
}

int main()
{
   myFunction();
   printf("%d\n", x);
   int x = 2;
   printf("%d\n", x);
   return 0;
}
```

Ωστόσο όπως φαίνεται αν δηλωθεί μέσα σε μια συνάρτηση μια μεταβλητή με το ίδιο όνομα τότε παραγράφεται η παγκόσμια και επιλέγεται η τοπική. Γιαυτό πρέπει να αποφεύγεται η χρήση ίδιου ονόματος στις μεταβλητές. Τέλος οι παγκόσμιες μεταβλητές καλό είναι να χρησιμοποιούνται μόνο όταν είναι απαραίτητο, η πρώτη επιλογή δηλαδή πάντα είναι η χρήση τοπικών μεταβλητών.

9.3 Δήλωση και υλοποίηση συναρτήσεων:

Όπως και στις μεταβλητές μια συνάρτηση μπορεί να δηλωθεί και να υλοποιηθεί σε διαφορετικές γραμμές κώδικα:

Μεταβλητές:

```
int x;
x = 5;
```

Αντίστοιχα για συνάρτηση:

```
#include <stdio.h>
```

```c
void myFunction();

int main()
{
   myFunction();
   return 0;
}

void myFunction()
{
   printf("Hello World!");
}
```

Συνηθίζεται η δήλωση να γίνεται πάνω από την main και η υλοποίηση κάτω από αυτήν.

Αν υπάρχουν παράμετροι τότε αναγράφονται και αυτές:

```c
#include <stdio.h>

void myFunction(int x, int y);

int main()
{
   myFunction(2, 3);
   return 0;
}

void myFunction(int x, int y)
{
   printf("x = %d, y = %d\n", x, y);
}
```

10. Βιβλιοθήκη <string.h>:

Τώρα που μάθαμε τι είναι η συνάρτηση πάμε να δούμε κάποιες χρήσιμες συναρτήσεις που παρέχει η string.h βιβλιοθήκες για τον χειρισμό strings δηλαδή πινάκων χαρακτήρων. Για να χρησιμοποιηθούν οι παρακάτω συναρτήσεις απαιτείται η προσθήκη της βιβλιοθήκης

```
#include <string.h>
```

10.1 Μήκος συμβολοσειράς (String Length – strlen()):

Έστω η παρακάτω συμβολοσειρά:

```
char alphabet[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

Η συνάρτηση strlen() θα μας δώσει το μήκος της συμβολοσειράς:

```
printf("%d", strlen(alphabet));
```

Με έξοδο το:

```
26
```

Διαφέρει από την μέθοδο με το sizeof() που είδαμε σε προηγούμενη ενότητα στο ότι η strlen() αν και τον μετρά κανονικά δεν συμπεριλαμβάνει τον χαρακτήρα \0 στην εμφάνιση δηλαδή:

```
printf("%d", strlen(alphabet));          // 26
printf("%d", sizeof(alphabet) / sizeof(alphabet [0]));
// 27
```

10.2 Ένωση συμβολοσειρών (Concatenate Strings - strcat()):

```
#include <stdio.h>
#include <string.h>

int main()
{
   char str1[20] = "Hello ";
   char str2[] = "World!";
```

```
    strcat(str1, str2);

    printf("%s", str1);
}
```

Στην πρώτη γραμμή δηλώνεται μια string (πίνακας χαρακτήρων) μεταβλητή str1 όπου μπορεί να αποθηκεύσει μέχρι 20 bytes – χαρακτήρες. Του ανατίθεται η λέξη Hello και ένα κενό άρα καταναλώνονται 6 bytes και μένουν 20 – 6 = 14 bytes. Στην δεύτερη γραμμή η string μεταβλητή str2 που τις ανατίθεται η λέξη World! Επίσης 6 bytes, δεν ορίζεται κάποιο όριο στην δήλωση άρα η μεταβλητή είναι ακριβώς όσο η λέξη που τις δόθηκε δηλαδή 7 bytes (+1 λόγω του \0).

Στην συνέχεια χρησιμοποιείται η συνάρτηση strcat όπου ενώνει τις δύο μεταβλητές. Το δεύτερο όρισμα δηλαδή το str2 είναι αυτό που προστίθεται στο πρώτο όρισμα str1. Για να είναι επιτυχής αυτή η προσθήκη απαιτείται η str1 να χωράει και την δεύτερη. Τα ελεύθερα bytes είναι 14 οπότε χωράει η str2 άρα και είναι εφικτή η ένωση των δύο συμβολοσειρών.

10.3 Αντιγραφή συμβολοσειρών (Copy Strings – strcpy()):

```
#include <stdio.h>
#include <string.h>

int main()
{
    char str1[20] = "Hello World!";
    char str2[20];

    strcpy(str2, str1);

    printf("%s", str2);
```

```
    return 0;
}
```

Αντίστοιχα και εδώ ο προορισμός πρέπει να είναι αρκετά μεγάλος ώστε να χωράει άρα >20 bytes.

10.4 Σύγκριση τιμών (Compare Strings - strcmp()):

```
#include <stdio.h>
#include <string.h>

int main()
{
    char str1[] = "Hello";
    char str2[] = "Hello";
    char str3[] = "Hi";

    // Compare str1 and str2, and print the result
    printf("%d\n", strcmp(str1, str2)); // Returns 0 (the
strings are equal)

    // Compare str1 and str3, and print the result
    printf("%d\n", strcmp(str1, str3)); // Returns -4
(the strings are not equal)
    return 0;
}
```

Επιστρέφει 0 αν οι δύο συμβολοσειρές είναι ίσες αλλιώς κάποιον τυχαίο άλλο αριθμό.

11. Είσοδος Χρήστη (User Input):

Σε αυτό το κεφάλαιο μαθαίνουμε πως μπορούμε να κάνουμε το πρόγραμμα μας πιο διαδραστικό επιτρέποντας τον χρήστη να δώσει αυτός τιμές στο πρόγραμμα μας.

11.1 scanf():

Αποτελεί την πιο γνωστή συνάρτηση για εισαγωγή από τον χρήστη.

Γίνεται κατανοητό με ένα παράδειγμα:

```c
#include <stdio.h>

int main()
{
   int age;
   char name[20];

   printf("Dose thn hlikia sou: ");
   scanf("%d", &age);
   printf("Dose to onoma sou: ");
   scanf("%s", &name);

   printf("\nEisai %d xronwn kai legesai %s\n", age,
name);

   return 0;
}
```

Στο παραπάνω πρόγραμμα ζητάμε από τον χρήστη να δώσει την ηλικία του και χρησιμοποιώντας την συνάρτηση scanf() που βρίσκεται στην βιβλιοθήκη stdio.h αποθηκεύουμε ότι γράψει ο χρήστης στην μεταβλητή age.

Έξοδος προγράμματος:

```
Dose thn hlikia sou: 20
Dose to onoma sou: Theodoros


Eisai 20 xronwn kai legesai Theodoros
```

Βλέπουμε ότι η συνάρτηση scanf() παίρνει δύο παραμέτρους, τον τύπο της μεταβλητής και την διεύθυνση στην μνήμη. Το &age δεν δίνει αντίγραφο της μεταβλητής αλλά την διεύθυνση της μνήμης στην οποία κατοικεί η μεταβλητή, περισσότερα γιαυτό σε επόμενο κεφάλαιο.

Αντίστοιχα επαναλαμβάνεται ίδια φιλοσοφία και για είσοδο ονόματος.

Για την δήλωση του ονόματος επειδή δεν ξέρουμε εξαρχής το όνομα του χρήστη δεν μπορούμε να δηλώσουμε έναν πίνακά ακριβώς όσα bytes απαιτείται οπότε για σιγουριά δηλώνουμε ένα μεγάλο αριθμό bytes όπως τα 20 bytes στην προκειμένη περίπτωση που αντιστοιχούν σε 20 πιθανούς χαρακτήρες. Αν το όνομα είναι μικρό όπως “Πέτρος” με 6 χαρακτήρες τότε 20 – 6 = 14 bytes έχουν δεσμευτεί από την μνήμη αλλά δεν χρησιμοποιούνται ποτέ, αυτό όμως είναι μια παραχώρηση που πρέπει να γίνει.

Τι θα γίνει αν στο προηγούμενο παράδειγμα δώσουμε ονοματεπώνυμο που περιέχει και ένα κενό, πχ theo tsif:

```
Dose thn hlikia sou: 20
Dose to onoma sou: Theo Tsif


Eisai 20 xronwn kai legesai Theo
```

Παρατηρούμε ότι το επίθετο χάθηκε. Αυτό συνέβη διότι η scanf() διαβάζει από το τερματικό μέχρι να τις δοθεί ο ένας whitespace χαρακτήρας όπως το κενό (space) οπότε καθώς γράφαμε το ονοματεπώνυμο σταμάτησε να διαβάζει στο κενό οπότε το επίθετο χάθηκε. Άλλα γνωστά whitespaces είναι το enter το tab και πολλά άλλα.

11.2 fgets():

Η fgets() είναι μια συνάρτηση που δεν σταματάει να διαβάζει όταν της δοθεί το κενό, ιδανική δηλαδή για το παράδειγμα μας που θέλουμε να εισάγουμε ονοματεπώνυμο.

Δομείται με τον εξής τρόπο:

```
fgets(name, 20, stdin);
```

Απαιτεί δηλαδή το όνομα της μεταβλητής, το μέγεθος της και την λέξη stdin που σημαίνει standard output, είναι μια διεργασία του τερματικού που αναλαμβάνει το λειτουργικό και είναι ώστε αυτά που γράφει ο χρήστης στο πληκτρολόγιο να τα μεταφέρει στο πρόγραμμα, είναι μια «αόρατη» διεργασία που εκτελεί αυτόματα το λειτουργικό και χρησιμοποιείται από όποια άλλη διεργασία θέλει είσοδο από τον χρήστη, απλά εδώ πρέπει να δηλωθεί χειροκίνητα.

Τρέχοντας το παρακάτω κομμάτι κώδικα όμως συναντάμε κάτι αναπάντεχο:

```
#include <stdio.h>

int main()
{
   char name[20];

   printf("Dose to onoma sou: ");
   fgets(name, 20, stdin);

   printf("legesai %s, pws eisai ? ", name);
   return 0;
}
```

Η έξοδος του προγράμματος θα είναι:

```
Dose to onoma sou: theo tsif
legesai theo tsif
, pws eisai ?
```

Αν και εμείς δεν βάλαμε \n πριν το μήνυμα pws eisai βλέπουμε ότι υπάρχει μια νέα γραμμή στην έξοδο. Αυτό συμβαίνει διότι η fgets() διαβάζει και το enter που πατάει ο χρήστης στην εισαγωγή του ονόματος, και το enter είναι ο χαρακτήρας νέας γραμμής δηλαδή το \n.

Για να μην συμβαίνει αυτό μπορούμε να χρησιμοποιήσουμε την strlen() συνάρτηση ώστε να αφαιρέσουμε τον χαρακτήρα \n, με τον εξής τρόπο:

```
name[strlen(name) - 1] = '\0';
```

Ο πίνακας μας μοιάζει κάπως έτσι:

```
"John\n\0"
```

Και η παραπάνω γραμμή τον κάνει:

```
"John\0\0"
```

Ένας άλλος τρόπος θα ήταν με την χρήση της strcspn() συνάρτησης:

```
name[strcspn(name, "\n")] = '\0';
```

Όπου βρίσκει την πρώτη εμφάνιση του χαρακτήρα \n και τον αλλάζει σε \0.

12. Επιπλέον χρήσιμα στοιχεία:

12.1 Βιβλιοθήκη <math.h>:

```
#include <stdio.h>
#include <math.h>

int main()
{
   double A = sqrt(9);    // Ρίζα του 9 = 3.
   double B = pow(2, 4);  // 2^4 = 16.
```

```
   int C = round(3.14);   // Στρογγυλοποίηση = 3.
   int D = ceil(3.14);    // Άνω στρογγυλοποίηση = 4.
   int E = floor(3.14);   // Κάτω στρογγυλοποίηση = 3.
   double F = fabs(-100); // Απόλυτο = 100.
   double G = log(3);     // Λογάριθμος του 3 =
1.098612.
   double H = sin(45);    // ημχ
   double I = cos(45);    // συνχ
   double J = tan(45);    // εφχ
}
```

Κάποιες χρήσιμες συναρτήσεις της βιβλιοθήκης math.h με το τι κάνουν.

12.2 Ternary - Conditional Operator:

Αποτελεί μια συντόμευση (shortcut) για if/else δομές όταν αναθέτουν ή επιστρέφουν μια τιμή. Για παράδειγμα:

```
#include <stdio.h>

int findMax(int x, int y)
{
   if (x > y)
   {
      return x;
   }
   else
   {
      return y;
   }
}

int main()
{
```

```
    int max = findMax(3, 4);
    printf("max = %d", max);
    return 0;
}
```

Στο παραπάνω πρόγραμμα η συνάρτηση findMax δέχεται ως όρισμα δύο αριθμούς και βλέπει ποιος είναι μεγαλύτερος και τον επιστρέφει. Αυτό μπορεί να γίνει πολύ πιο γρήγορα με την εξής δομή:

```
//(condition) ? value if true : value if false
```

Παράδειγμα:

```
#include <stdio.h>

int findMax(int x, int y)
{
    return (x > y) ? x : y;
}

int main()
{
    int max = findMax(3, 4);
    printf("max = %d", max);
    return 0;
}
```

12.3 Ανταλλαγή μεταβλητών:

Για ακέραιους, χαρακτήρες και πραγματικές μεταβλητές:

```
int x = 1;
int y = 2;
int temp;

temp = x;
```

```
x = y;
y = temp;
```

Για strings:

```
char x[] = "water";
char y[] = "lemonade";
char temp[15];

strcpy(temp, x);
strcpy(x, y);
strcpy(y, temp);
```

Ωστόσο αν το μήκος της y είναι μικρότερο της x θα συναντήσουμε σφάλμα. Η πιο απλή λύση είναι να ορίσουμε ίδιο μήκος x και y:

```
char x[15] = "water";
char y[15] = "lemonade";
char temp[15];

strcpy(temp, x);
strcpy(x, y);
strcpy(y, temp);
```

12.4 typedef:

τεστ

Είναι ένα keyword που δίνει ψευδώνυμο σε μεταβλητές.

Αντί να έχουμε:

```
char user1[25] = "Test";
```

Μπορούμε έξω από την main να ορίσουμε ένα typedef με τον εξής τρόπο:

```
typedef char user[25];
```

Και έτσι όταν θέλουμε να φτιάξουμε έναν χρήστη αρκεί μόνο η γραμμή:

```
user user1 = "Test";
```

Όλο μοιάζει κάπως έτσι:

```c
#include <stdio.h>

typedef char user[25];

int main()
{
   user user1 = "Test";
   return 0;
}
```

Το typedef είναι κάτι που χρησιμοποιείται πολύ στα structs που θα δούμε σε επόμενο κεφάλαιο.

12.5 Random numbers:

Το να παράγουμε έναν πραγματικά τυχαίο αριθμό είναι κάτι το πολύ δύσκολο και απαιτεί εκτός από έναν πολυσύνθετο αλγόριθμο και έναν πολύ ισχυρό υπολογιστή. Επομένως στην πραγματικότητα τις περισσότερες φορές που έχουμε έναν τυχαίο αριθμό σε ένα πρόγραμμα είναι στατιστικά τυχαίος και όχι πραγματικά τυχαίος και γιαυτό λέγεται ψευδό-τυχαίος αριθμός (pseudo random numbers).

Για να δημιουργήσουμε έναν τέτοιο αριθμό αρχικά πρέπει να συμπεριλάβουμε δύο βιβλιοθήκες:

```c
#include <stdlib.h>
#include <time.h>
```

Η stdlib.h περιλαμβάνει την συνάρτηση srand() και rand() που θα χρειαστούμε και η time.h την time().

Η srand() δέχεται ως όρισμα έναν αριθμό και ανάλογα με αυτόν διαμορφώνει και τις ψευδό-τυχαίες τιμές που παράγει η rand(). Αυτόν τον αριθμό τον λέμε συνήθως σπόρο (seed). Συνηθίζεται αυτός ο σπόρος να

είναι ένας αριθμός που εκφράζει πόσα δευτερόλεπτα πέρασαν από την 1 Ιανουαρίου 1970, επομένως σε κάθε διαφορετική εκτέλεση του προγράμματος επειδή αυξάνεται αυτός ο χρόνος, αυξάνεται και ο σπόρος κάτι που δημιουργεί διαφορετικά αποτελέσματα κάθε φορά. Αυτή την διαδικασία την πραγματοποιεί η συνάρτηση time όπου για να πάρουμε τα τωρινά δευτερόλεπτα γράφουμε:

```
time(0)
```

Και όλο μαζί:

```
srand(time(0));
```

Επομένως αφού έχουμε σπείρει όπως ονομάζεται την συνάρτηση rand() πάμε να δημιουργήσουμε έναν τυχαίο αριθμό:

```
int number1 = rand();
```

Με την παρακάτω γραμμή η μεταβλητή number1 θα λάβει τυχαία μια τιμή από το 0 έως το 32.767. Αν δεν χρειαζόμαστε μια τόσο μεγάλη τιμή και θέλουμε κάτι μικρότερο μπορούμε να το περιορίσουμε με τον εξής τρόπο:

```
int number1 = rand() % 6;
```

Όπου 6 ο μέγιστος αριθμός που επιθυμούμε, η παραπάνω μεταβλητή θα έχει έναν τυχαίο αριθμό από το 0 έως το 5. Μπορεί να ενισχυθεί περαιτέρω με την γραμμή:

```
int number1 = (rand() % 6) + 1;
```

Όπου τώρα αποδίδει έναν τυχαίο αριθμό μεταξύ του 0 και 6.

12.6 Bitwise operators:

Επρόκειτο για ειδικούς χαρακτήρες που χρησιμοποιούνται στο bit επίπεδο προγραμματισμού. Προϋποθέτει γνώσεις σχετικά με το δυαδικό σύστημα.

Έστω τρείς μεταβλητές:

```
int x = 6;
```

```
int y = 12;
int z = 0;
```

Το δυαδικό ισοδύναμο τους είναι:

```
int x = 6;  // 00000110
int y = 12; // 00001100
int z = 0;  // 00000000
```

Η παρακάτω γραμμή κώδικα θα εκτελέσει την δυαδική πρόσθεση – AND για τους δύο αριθμούς:

```
z = x & y;
```

Δηλαδή:

```
// 00000110
//        +
// 00001100
//        =
// 00000100 δηλαδή το 4
```

Για επαλήθευση:

```
printf("z = %d", z);
```

Αποτέλεσμα:

```
z = 4
```

Αντίστοιχα η παρακάτω γραμμή εκτελεί δυαδικό Ή – OR:

```
z = x | y;
```

Με αποτέλεσμα:

```
z = 14
```

Επίσης για αποκλειστικό Ή – Exclusive OR – XOR:

```
z = x ^ y;
```

Με αποτέλεσμα:

```
z = 10
```

Θυμάμαι: Ολίσθηση κατά 1 προς τα αριστερά είναι διπλασιασμός και προς τα δεξιά διαίρεση.

Με την παρακάτω γραμμή το z παίρνει τα bits του x αφού έχουν δεχθεί ολίσθηση προς τα αριστερά κατά 1, δηλαδή διπλασιασμός άρα 12.

```
z = x << 1;
```

Και για προς τα δεξιά:

```
z = x >> 1;
```

Αντίστοιχα μπορεί να γίνει και ολίσθηση κατά 2,3…

Τέλος με την εξής γραμμή:

```
z = ~x;
```

Το z παίρνει την τιμή του x σε συμπλήρωμα ως προς 1 (Ones complement) δηλαδή για x = 00000110 => z = 11111001.

13. Τεχνικές γραφής κώδικα:

13.1 Never Nesting:

Στο παρών κεφάλαιο θα κάνουμε μια παύση για την C και θα μιλήσουμε για μια τεχνική γραφής κώδικα από την μεριά του χρήστη που αφορά όλες τις γλώσσες. Τονίζεται πως η παρακάτω τεχνική είναι καθαρή προτίμηση του συγγραφέα και δεν είναι απαραίτητο να την ακολουθήσει κανείς.

Ως nesting ορίζεται ο εμφωλευμεύνος κώδικας δηλαδή η χρήση δομών η μία μέσα στην άλλην.

Ως 1D δηλαδή μία διάσταση ορίζεται μια δομή με την εξής μορφή:

```
int calculate(int bottom, int top) // 1
{
   return bottom + top;
}
```

Για 2D γίνεται έτσι:

```
int calculate(int bottom, int top) // 1
{
   if (top > bottom) // 2
   {
      return bottom + top;
   }
   else
   {
      return 0;
   }
}
```

Και για 3D:

```
int calculate(int bottom, int top) // 1
{
   if (top > bottom) // 2
   {
      int sum = 0;
      for (int number = bottom; number <= top; number++)
// 3
      {
         sum += number;
      }
      return sum;
   }
   else
   {
      return 0;
   }
}
```

Αυτό είναι και το άνω όριο για κάποιον που ακολουθεί αυτή την τεχνική η οποία είναι ποτέ εμφωλευμεύνο κώδικα πάνω από τρείς διαστάσεις.

```
int calculate(int bottom, int top) // 1
{
   if (top > bottom) // 2
   {
      int sum = 0;
      for (int number = bottom; number <= top; number++)
// 3
      {
         if (number % 2 == 0) // 4
         {
            sum += number;
         }
      }
      return sum;
   }
   else
   {
      return 0;
   }
}
```

Για την αποφυγή παραπάνω διαστάσεων υπάρχουν δύο μέθοδοι:

- Εξαγωγή (extraction)
- Αντιστροφή (inversion)

Εξαγωγή:

Στην μέθοδο της εξαγωγής μια δομή βγαίνει έξω από την συνάρτηση για να γίνει μια δική της συνάρτηση:

```
int filterNumber(int number)
{
   if (number % 2 == 0)
```

```
    {
        return number;
    }
    return 0;
}

int calculate(int bottom, int top)
{
    if (top > bottom)
    {
        int sum = 0;
        for (int number = bottom; number <= top; number++)
        {
            if (number % 2 == 0)
            {
                sum += filterNumber(number);
            }
        }
        return sum;
    }
    else
    {
        return 0;
    }
}
```

Αντιστροφή:

Η τεχνική γραφής κατά την οποία το αρνητικό κομμάτι κώδικα μπαίνει στην αρχή:

```
int calculate(int bottom, int top)
{
```

```
    if (top < bottom)
    {
        return 0;
    }
    else
    {
        int sum = 0;
        for (int number = bottom; number <= top; number++)
        {
            if (number % 2 == 0)
            {
                sum += filterNumber(number);
            }
        }
        return sum;
    }
}
```

Έτσι αφού αν δεν πληροί τις προϋποθέσεις θα μπει στην if και θα τερματίσει αφού θα επιστραφεί 0, μπορούμε να αφαιρέσουμε την δομή else. Δηλαδή:

```
int calculate(int bottom, int top)
{
    if (top < bottom)
    {
        return 0;
    }
    int sum = 0;
    for (int number = bottom; number <= top; number++)
    {
        if (number % 2 == 0)
        {
```

```
            sum += filterNumber(number);
        }
    }
    return sum;
}
```

Αν είχαμε μια μεγαλύτερη συνάρτηση τότε επαναλαμβάνεται μέχρι το θεμιτό αποτέλεσμα. Με αυτόν τον τρόπο έχουμε στην αρχή της συνάρτησης τους λόγους για τους οποίους μπορεί να τερματίσει με άλλα λόγια έχουμε τις προϋποθέσεις τις συναρτήσεις και όλος ο υπόλοιπος κώδικας είναι η πραγματική χρησιμότητα της συνάρτησης. Έτσι πετυχαίνουμε να έχουμε έναν πιο ευανάγνωστο κώδικα που είναι ευκολότερο για όλους να τον καταλάβουν.

13.2 Θέση αγκύλων:

Όταν μια δομή εκτελεί μόνο μια γραμμή κώδικα μπορούν να παραληφθούν οι αγκύλες, δηλαδή:

```
if (/* condition */)
    return 1;
```

Και είναι ισοδύναμο με το:

```
if (/* condition */)
{
    return 1;
}
```

Μια άλλη δυνατότητα είναι η πρώτη αγκύλη να είναι στην ίδια γραμμή με τον έλεγχο ή και η δεύτερη με την τελευταία γραμμή, δηλαδή:

```
if (/* condition */){
    return 1;}
```

Προσωπικά, προτείνω πάντα μα πάντα να μπαίνουν αγκύλες ακόμα και αν πρόκειται μόνο για μια γραμμή κώδικα και κάθε αγκύλη σε ξεχωριστή γραμμή.

13.3 < ή <=:

Έστω ότι θέλουμε μια for να εκτυπώνει αριθμούς μέχρι και το 10, υπάρχουν δύο τρόποι να γίνει αυτό:

```
for (int i = 0; i < 11; i++)
{
   printf("%d\n", i);
}
```

Στον πρώτο τρόπο λέμε i < 11 και επειδή είναι ακέραιος ο τελευταίος ακέραιος μικρότερος αριθμός είναι το 10 άρα θα δουλέψει.

Και ο δεύτερος τρόπος και αυτός που προτιμάω προσωπικά είναι:

```
for (int i = 0; i <= 10; i++)
{
   printf("%d\n", i);
}
```

Προτιμώ αυτό τον τρόπο δηλαδή γιατί σου δείχνει ευθέως μέχρι ποια τιμή θα πάει το i και δεν χρειάζεται ο παραπάνω υπολογισμός.

Όταν δουλεύουμε με άγνωστες τιμές πχ αριθμό επαναλήψεων n τότε είναι:

- i < n ή
- i <= n – 1

13.4 Σχόλια ΟΧΙ – Documentation ΝΑΙ:

τεστ

14. Αναδρομή:

Αναδρομή ονομάζεται η τεχνική μια συνάρτηση να καλεί τον εαυτό της. Αυτή η τεχνική παρέχει ένα μεγάλο πλεονέκτημα και αυτό είναι ότι μπορούμε να σπάσουμε τα περίπλοκα προβλήματα σε πιο απλά κάνοντας

τα έτσι πιο εύκολο να επιλυθούν. Ωστόσο αυτό συνοδεύεται από ένα επίσης μεγάλο μειονέκτημα και αυτό είναι ότι έχουμε εκτεταμένη δέσμευση μνήμης αφού σε κάθε κλήση δημιουργούνται νέες μεταβλητές.

Είδη αναδρομής:

- Άμεση αναδρομή έχουμε όταν μια συνάρτηση καλεί τον εαυτό της.
- Έμμεση αναδρομή έχουμε όταν για παράδειγμα μια συνάρτηση Α καλεί μια συνάρτηση Β και αυτή ξανά την Α.
- Ατέρμονη αναδρομή έχουμε όταν μια αναδρομική συνάρτηση δεν διαθέτει τρόπο να σταματήσει τις αναδρομικές κλήσεις.

Από το τελευταίο συμπεραίνουμε ότι μια αναδρομική συνάρτηση πρέπει να διαθέτει ένα σημείο τερματισμού (break case).

14.1 Παραγοντικό (factorial):

Το πιο συνηθισμένο παράδειγμα αναδρομής είναι ο υπολογισμός του παραγοντικού ενός αριθμού:

```
#include <stdio.h>
long int multiplyNumbers(int n);
int main()
{
   int n;
   printf("Enter a positive integer: ");
   scanf("%d", &n);
   printf("Factorial of %d = %ld", n,
multiplyNumbers(n));
   return 0;
}

long int multiplyNumbers(int n)
{
```

```
    if (n <= 1) // break case
    {
        return 1;
    }
    return n * multiplyNumbers(n - 1);
}
```

Το σημείο τερματισμού είναι αν το n είναι 0 ή 1 όπου και των δύο αριθμών το παραγοντικό είναι το 1 οπότε και γιαυτό επιστρέφουμε την τιμή 1. Επομένως η συνάρτηση δέχεται έναν αριθμό και τον πολλαπλασιάζει με την τιμή που θα επιστρέψει η κλήση της ίδιας συνάρτησης και τον αριθμό αυτόν μείον 1. Αυτό θα επαναλαμβάνεται μέχρι να φτάσουμε μια τιμή <= 1 όπου θα επιστραφεί το 1. Τότε αναδρομικά από πίσω προς τα μπροστά θα γίνουν όλοι οι πολλαπλασιασμοί και θα επιστραφεί το τελικό αποτέλεσμα.

```
Παράδειγμα Εκτέλεσης:
Αν ο χρήστης εισάγει τον αριθμό 5:
Η multiplyNumbers(5) καλεί τη multiplyNumbers(4).
Η multiplyNumbers(4) καλεί τη multiplyNumbers(3).
Η multiplyNumbers(3) καλεί τη multiplyNumbers(2).
Η multiplyNumbers(2) καλεί τη multiplyNumbers(1).
Η multiplyNumbers(1) επιστρέφει 1.
Η multiplyNumbers(2) επιστρέφει 2 * 1 = 2.
Η multiplyNumbers(3) επιστρέφει 3 * 2 = 6.
Η multiplyNumbers(4) επιστρέφει 4 * 6 = 24.
Η multiplyNumbers(5) επιστρέφει 5 * 24 = 120.
```

14.2 Ύψωση σε δύναμη:

```
double powerRec(double base, double exp)
{
    if (exp == 0) // break case
```

```
    {
        return 1;
    }
    if (exp == 1) // optimization
    {
        return base;
    }
    else
    {
        return base * powerRec(base, exp - 1);
    }
}
```

Η παρακάτω συνάρτηση υπολογίζει την ύψωση σε κάποια δύναμη για έναν αριθμό. Εκτός από το σημείο τερματισμού έχει και μια γραμμή που είναι για βελτιστοποίηση ώστε να γλυτώσουμε τουλάχιστον μία επιπλέον κλήση της συνάρτησης.

```
Ας δούμε ένα παράδειγμα με την συνάρτηση `powerRec` για
τον υπολογισμό της δύναμης \( 2^3 \):

1. Κλήση: `powerRec(2, 3)`
   - Ο εκθέτης δεν είναι 0 ούτε 1, έτσι επιστρέφει \( 2
\times powerRec(2, 2) \).

2. Κλήση: `powerRec(2, 2)`
   - Ο εκθέτης δεν είναι 0 ούτε 1, έτσι επιστρέφει \( 2
\times powerRec(2, 1) \).

3. Κλήση: `powerRec(2, 1)`
   - Ο εκθέτης είναι 1, έτσι επιστρέφει την βάση, δηλαδή
2.
```

```
Συνδυάζοντας τα αποτελέσματα από τις αναδρομικές
κλήσεις:
- `powerRec(2, 1)` επιστρέφει 2.
- `powerRec(2, 2)` επιστρέφει \( 2 \times 2 = 4 \).
- `powerRec(2, 3)` επιστρέφει \( 2 \times 4 = 8 \).

Έτσι, το αποτέλεσμα του \( 2^3 \) είναι 8.
```

14.3 Σειρά Fibonacci:

```c
int fibo(int num)
{
   if (num == 1 || num == 2)
   {
      return 1;
   }
   else
   {
      return fibo(num - 1) + fibo(num - 2);
   }
}
```

Τι κάνει:

```
Η συνάρτηση `fibo` υπολογίζει τον \(n\)-οστό αριθμό της
ακολουθίας Fibonacci χρησιμοποιώντας αναδρομή. Η
ακολουθία Fibonacci ορίζεται ως εξής: οι δύο πρώτοι
αριθμοί είναι 1, και κάθε επόμενος αριθμός είναι το
άθροισμα των δύο προηγούμενων αριθμών. Εάν η είσοδος
`num` είναι 1 ή 2, η συνάρτηση επιστρέφει 1, καθώς αυτοί
είναι οι δύο πρώτοι αριθμοί της ακολουθίας. Για
μεγαλύτερες τιμές του `num`, η συνάρτηση επιστρέφει το
άθροισμα των δύο προηγούμενων αριθμών της ακολουθίας,
δηλαδή `fibo(num - 1) + fibo(num - 2)`.
```

```
### Παράδειγμα:
Ας δούμε τον υπολογισμό του 5ου αριθμού της ακολουθίας Fibonacci, δηλαδή `fibo(5)`:

1. Κλήση: `fibo(5)`
   - Δεν είναι 1 ή 2, επιστρέφει `fibo(4) + fibo(3)`.

2. Κλήση: `fibo(4)`
   - Δεν είναι 1 ή 2, επιστρέφει `fibo(3) + fibo(2)`.

3. Κλήση: `fibo(3)`
   - Δεν είναι 1 ή 2, επιστρέφει `fibo(2) + fibo(1)`.

4. Κλήση: `fibo(2)`
   - Είναι 2, επιστρέφει 1.

5. Κλήση: `fibo(1)`
   - Είναι 1, επιστρέφει 1.

Συνδυάζοντας τα αποτελέσματα:
- `fibo(3)` επιστρέφει \(1 + 1 = 2\).
- `fibo(4)` επιστρέφει \(2 + 1 = 3\) (το αποτέλεσμα του `fibo(3)` και `fibo(2)`).
- `fibo(3)` (ξανά από `fibo(4)`) επιστρέφει \(1 + 1 = 2\).
- `fibo(5)` επιστρέφει \(3 + 2 = 5\) (το αποτέλεσμα του `fibo(4)` και `fibo(3)`).

Έτσι, ο 5ος αριθμός της ακολουθίας Fibonacci είναι 5.
```

15. Pointers:

15.1 Διευθύνσεις μνήμης (Memory Address):

Όπως μάθαμε στην ενότητα με τις μεταβλητές η RAM είναι μια συστοιχία από κουτάκια και μπορούμε να χρησιμοποιήσουμε ένα κουτάκι ώστε να αποθηκεύσουμε μια μεταβλητή. Κάθε κουτάκι για να ξεχωρίζει από τα άλλα έχει έναν μοναδικό αριθμό που χρησιμοποιείται για να προσδιοριστεί. Αυτός ο μοναδικός αριθμός λέγεται διεύθυνση μνήμης και μπορούμε να τον μάθουμε για κάθε μεταβλητή με την παρακάτω εντολή:

```
int age = 43;
printf("%p", &age);
```

Το αποτέλεσμα είναι:

```
0000003df41ffe7c
```

Και αυτό αντιστοιχεί σε κάποιο κουτάκι μέσα στην μνήμη, προφανώς δεν είναι σταθερός αριθμός και σε κάθε εκτέλεση του προγράμματος παίρνουμε διαφορετικό αποτέλεσμα. Η διεύθυνση όπως φαίνεται είναι εκφρασμένη σε δεκαεξαδικό (hexadecimal) σύστημα.

Βλέπουμε δεν διαφέρει και πολύ από την μορφή που έχουμε συνηθίσει για εκτύπωση της τιμής της μεταβλητής:

```
int age = 43;
printf("%d", age);  // Εκτύπωση τιμής μεταβλητής age
printf("%p", &age); // Εκτύπωση διεύθυνσης μνήμης
μεταβλητής age
```

Οι διαφορές είναι ότι αντί για %d που εκφράζει ακέραιο χρησιμοποιούμε %p που εκφράζει διεύθυνση μνήμης και αντί για age βάζουμε &age με το έξτρα σύμβολο & να εκφράζει ότι θέλουμε την διεύθυνση μνήμης που κατοικεί η μεταβλητή age και όχι την τιμή της.

Μια καλή αναλογία είναι να σκεφτόμαστε την συνολική μνήμη RAM σαν μια οδό όπου κάθε κελί της μνήμης είναι ένα σπίτι και κάθε σπίτι έχει μια διεύθυνση.

15.2 Pointer μεταβλητές:

Μια pointer μεταβλητή (μεταβλητή δείκτη) είναι μια μεταβλητή που αντί να περιέχει κάποια τιμή, περιέχει μια διεύθυνση μνήμης στην οποία υπάρχει κάποια άλλη μεταβλητή.

Έστω η παρακάτω κανονική μεταβλητή:

```
int age = 21;
```

Μπορούμε να δημιουργήσουμε μια pointer μεταβλητή που έχει για τιμή την διεύθυνση μνήμης που βρίσκεται η age με τον εξής τρόπο:

```
int *pAge = &age;
```

- Αρχικά ο τύπος του pointer πρέπει να είναι ίδιος με τον τύπο της μεταβλητής.
- Το * σύμβολο είναι απαραίτητο για να δείξουμε στον compiler ότι είναι pointer και όχι απλή μεταβλητή άρα μπορεί να αποθηκεύσει διεύθυνση μνήμης.
- Συνηθίζεται το όνομα μιας pointer μεταβλητής να ξεκινά με μικρό p και να ακολουθεί το όνομα της μεταβλητής που δείχνει με κεφαλαίο το πρώτο γράμμα.
- Τέλος όπως αναφέραμε και σε προηγούμενες ενότητες με το σύμβολο & και μετά το όνομα μιας μεταβλητής παίρνουμε την διεύθυνση μνήμης της μεταβλητής και όχι την τιμή της.

Εκτελώντας τις δύο παρακάτω εντολές βλέπουμε ότι η ***διεύθυνση*** της μεταβλητής age είναι ίδια με την ***τιμή*** της μεταβλητής pAge:

```
printf("address of age: %p\n", &age);
printf(" value of pAge: %p\n", pAge);
```

Με αποτέλεσμα:

```
address of age: 000000c6849ffa4c
 value of pAge: 000000c6849ffa4c
```

Με την παρακάτω εντολή μπορούμε να δούμε που είναι αποθηκευμένη η pointer μεταβλητή:

```
printf("address of pAge: %p\n", &pAge);
```

Και για αποτέλεσμα παίρνουμε:

```
address of pAge: 000000c6849ffa40
```

Άρα στο κελί της μνήμης με αριθμό 000000c6849ffa40 και όνομα pAge υπάρχει αποθηκευμένη η τιμή 000000c6849ffa4c που είναι η διεύθυνση της μεταβλητής age με τιμή 21.

Από τα παραπάνω μπορούμε να συμπεράνουμε ότι μέσω της pointer μεταβλητής έχουμε πρόσβαση στην τιμή της μεταβλητής που δείχνει δηλαδή της age δηλαδή στο 21. Για να εκτυπώσουμε την τιμή 21 χρειαζόμαστε την παρακάτω γραμμή κώδικα:

```
printf("value at stored address: %d\n", *pAge);
```

Το *pAge είναι γνωστό ως dereferencing / ξε-αναφορά. Πάλι χρησιμοποιείται το * για να δείξει σε αυτό που δείχνει η μεταβλητή και όχι στην τιμή της.

Αξίζει να σημειωθεί ότι στην δήλωση ενός pointer υπάρχει και ένας δεύτερος τρόπος να γίνει και αυτός είναι:

```
int* pAge = &age;
```

Δηλαδή το * σύμβολο να είναι μαζί με τον τύπο και όχι με το όνομα, συνήθως χρησιμοποιείται ο πρώτος τρόπος.

Ωστόσο μια άλλη καλή συνήθεια στους pointers είναι κατά την δήλωση τους να δίνεται η τιμή NULL (εκφράζει το τίποτα) και σε δεύτερο χρόνο να τους ανατεθεί κάποια τιμή, δηλαδή:

```
int *pAge = NULL;
pAge = &age;
```

15.3 Call / pass by reference:

Σε προηγούμενο κεφάλαιο μάθαμε πως όταν δίνουμε μια μεταβλητή παράμετρο σε μια συνάρτηση, στην πραγματικότητα δίνεται στην συνάρτηση ένα αντίγραφο της μεταβλητής οπότε και δεν μπορούμε να αλλάξουμε οριστικά την αξία μιας μεταβλητής μέσω συνάρτησης. Αυτό που περιγράφηκε είναι γνωστό ως call / pass by value. Τώρα που μάθαμε pointers ξεκλειδώθηκε άλλη μια δυνατότητα και αυτή είναι η call / pass by reference όπου αντί να στέλνουμε ένα αντίγραφο της μεταβλητής στέλνουμε μια pointer μεταβλητή και μέσω αυτής μπορούμε να αλλάξουμε οριστικά μια μεταβλητή.

```
#include <stdio.h>
void printAge(int *pAge);
int main()
{
    int age = 21;
    int *pAge = NULL;
    pAge = &age;

    printAge(pAge);
}

void printAge(int *pAge)
{
    printf("You are %d years old\n", *pAge);
    *pAge = 30;
    printf("You are now %d years old\n", *pAge);
}
```

Με αποτέλεσμα:

```
You are 21 years old
You are now 30 years old
```

Η παραπάνω ιδιότητα χρησιμοποιείται ιδιαιτέρως στην επεξεργασία πινάκων.

15.4 Pointers με πίνακες:

Έστω η παρακάτω δήλωση πίνακα χαρακτήρων:

```
int numbers[4] = {1, 2, 3, 4};
```

Μάθαμε ότι αυτό που συμβαίνει είναι ότι δεσμεύονται από την μνήμη τέσσερις διαδοχικές θέσεις / κελιά και κάθε ένα από αυτά παίρνει μια τιμή, σαν τέσσερις μεταβλητές.

Μπορούμε να μάθουμε σε ποιο κελί είναι αποθηκευμένη κάθε τιμή με τον εξής τρόπο:

```
printf("first cell = %p\n", &numbers[0]);
printf("second cell = %p\n", &numbers[2]);
printf("third cell = %p\n", &numbers[3]);
printf("fourth cell = %p\n", &numbers[4]);
```

Και έχουμε ως αποτέλεσμα:

```
first cell = 0000002b947ff820
second cell = 0000002b947ff828
third cell = 0000002b947ff82c
fourth cell = 0000002b947ff830
```

Όμως εκτός από τα κελιά με τις τιμές τους υπάρχει και το όνομα μεταβλητής, αυτό που είναι άραγε:

```
printf("array name = %p\n", &numbers);
```

Η απάντηση δεν είναι κάτι που περιμέναμε καθώς βγαίνει αυτό:

```
array name = 0000002b947ff820
```

Δείχνει δηλαδή στην ίδια διεύθυνση με το πρώτο κελί.

Αυτό επειδή η μνήμη μοιάζει κάπως έτσι:

```
         +---+
numbers: | 1 | numbers[0]
         +---+
         | 2 | numbers[1]
         +---+
         | 3 | numbers[2]
         +---+
         | 4 | numbers[3]
         +---+
```

Αυτό γενεί το εξής ερώτημα:

Είναι αποθηκευμένη σε μία άλλη θέση και απλά δείχνει στο numbers[0] κελί ? Η μήπως αυτό το κελί έχει δύο ονόματα και αν είναι αυτό τότε πως μπορεί και το κάνει:

Συνοπτικά:

An array name is not itself a pointer, but decays into a pointer to the first element of the array in most contexts. It's that way because the language defines it that way.

Αναλυτικότερα:

- Όταν δημιουργείται ένας πίνακας ο μόνος χώρος που δεσμεύεται είναι ο αυτός που απαιτείται για τις τιμές, δηλαδή δεν δεσμεύεται χώρος για κάποιον έξτρα δείκτη (pointer) ή μεταδεδομένα.
- Ο κανόνας στην C είναι ότι κάθε φορά που ο μεταγλωττιστής βλέπει μια έκφραση τύπου πίνακα (όπως numbers, που έχει τύπο int [4]) και αυτή η έκφραση δεν είναι ο τελεστής του sizeof και άλλα παρεμφερή ο τύπος αυτής της έκφρασης μετατρέπεται ("decays") σε τύπο δείκτη (int *) και η τιμή της έκφρασης είναι η διεύθυνση του πρώτου στοιχείου του πίνακα. Επομένως, η παράσταση numbers

έχει τον ίδιο τύπο και τιμή με την έκφραση numbers [0] (και κατ' επέκταση, η έκφραση * numbers έχει τον ίδιο τύπο και τιμή με την παράσταση numbers [0]).

- ***Δηλαδή ο compiler αντικαθιστά το numbers με το numbers[0].***
- Η C προήλθε από μια παλαιότερη γλώσσα που ονομαζόταν B, και στο B το numbers ήταν ένας ξεχωριστός δείκτης και δεν είχε σχέση με τα στοιχεία του πίνακα numbers[0], numbers[1], κ.λπ. Ο Ritchie ήθελε να διατηρήσει τη σημασιολογία του πίνακα του B, αλλά δεν ήθελε να μπλέξει με την αποθήκευση του ξεχωριστού δείκτη. Έτσι το ξεφορτώθηκε. Αντίθετα, ο μεταγλωττιστής θα μετατρέψει εκφράσεις πίνακα σε εκφράσεις δείκτη κατά τη μετάφραση, όπως είναι απαραίτητο.

Αυτό οδηγεί στα εξής:

Όταν περνάμε έναν πίνακα σε μια συνάρτηση, το μόνο που λαμβάνει η συνάρτηση είναι ένας δείκτης στο πρώτο στοιχείο - δεν έχει ιδέα πόσο μεγάλος είναι ο πίνακας . Για να γνωρίζει η συνάρτηση πόσα στοιχεία έχει ο πίνακας, πρέπει είτε να χρησιμοποιήσετε μια τιμή φρουρού (sentinel) (όπως ο τερματιστής 0 σε συμβολοσειρές C) είτε να περάσουμε τον αριθμό των στοιχείων ως ξεχωριστή παράμετρο.

Ένα ακόμα που μας προσφέρει η παραπάνω ιδιότητα είναι να κάνουμε πράξεις με μόνο το όνομα του πίνακα:

```
// Get the value of the second element in myNumbers
printf("%d\n", *(numbers + 1));
```

Έτσι το numbers [i] είναι ισοδύναμο με το *(numbers + i). Γιαυτό μπορεί να έχετε δει a[5] και 5[a] είναι επειδή ο compiler αυτό που βλέπει είναι *(a + 5) οπότε και ανάποδα να μπουν δεν κάνει διαφορά: *(5 + a).

Αν περάσουμε παραμετρικά έναν πίνακα σε μια συνάρτηση αυτή θα έχει πρόσβαση μόνο στο πρώτο στοιχείο του πίνακα και στα υπόλοιπα έχει

πρόσβαση μέσω αριθμητικής δεικτών *(array + i). Γιαυτό χρειάζεται και το μέγεθος του πίνακα για να ξέρει μέχρι ποιον αριθμό μπορεί να προσθέσει χωρίς να βγει από τον πίνακα. Αυτό επειδή όταν δηλώνεται ένας πίνακας όλες οι τιμές είναι δίπλα δίπλα οπότε ξεκινώντας από την πρώτη και προσθέτοντας 1 κάθε φορά μπορούμε να τον περάσουμε όλο, αν δεν ξέρουμε που τελειώνει και συνεχίζουμε να προσθέτουμε 1 κάποια στιγμή θα καταλήξουμε σε κελί μνήμης που έχει τιμή άσχετη με τις τιμές του πίνακα, γιαυτό είναι απαραίτητο να ξέρει μια συνάρτηση το μέγεθος του πίνακα.

15.5 Μέγεθος μεταβλητής δείκτη:

Αν και μια μεταβλητή δείκτη δηλώνεται πάντα ίδιο τύπο με την μεταβλητή στην οποία δείχνει, δεν δεσμεύει και τα ίδια bytes με τον τύπο.

```
int a = 5;
printf("sizeof a = %d\n", sizeof(a));
int *ptr = &a;
printf("sizeof ptr = %d\n", sizeof(ptr));
```

Με τον παραπάνω κώδικα παίρνουμε ως αποτέλεσμα:

```
sizeof a = 4
sizeof ptr = 8
```

Αν αλλάξουμε τον τύπο σε char, long, short και γενικά σε οποιοδήποτε πρωτεύων τύπο, το αποτέλεσμα θα παραμείνει ίδιο, πάντα μια μεταβλητή δείκτη δεσμεύει 8bytes από την μνήμη RAM.

16. Πολυπλοκότητα:

Το παρακάτω κεφάλαιο αποτελεί αντιγραφή των διαφανειών της καθηγήτριας Ασδρέ Κατερίνα από το μάθημα «αντικειμενοστραφής προγραμματισμός».

Ως αλγόριθμος (algorithm) ορίζεται μια πεπερασμένη, αιτιοκρατική και αποτελεσματική μέθοδος επίλυσης ενός προβλήματος. Κλασικό

παράδειγμα ο αλγόριθμος του Ευκλείδη για υπολογισμό του μέγιστου κοινού διαιρέτη δύο ακεραίων αριθμών.

Για κάθε (επιλύσιμο) πρόβλημα Π υπάρχει ένα σύνολο αλγορίθμων {$A1$, $A2$ … , $Aκ$} που το επιλύουν, k ≥ 1. Εμείς θέλουμε να επιλέξουμε τον «καλύτερο» δηλαδή αποδοτικότερο αλγόριθμο. Τα κριτήρια είναι:

- Χρόνος εκτέλεσης
- Απαιτούμενοι πόροι (μνήμη)
- Βαθμός δυσκολίας υλοποίησης

Εστιάζοντας στον χρόνο αναδιατυπώνουμε το προηγούμενο ερώτημα:

Ποιος από όλους τους k αλγορίθμους που επιλύουν το πρόβλημα είναι ο πιο αποτελεσματικός, δηλαδή ποιος αλγόριθμος έχει τη μικρότερη πολυπλοκότητα χρόνου (=μικρό χρόνο εκτέλεσης) ?

Εμπειρικά:

Κωδικοποιούμε τον αλγόριθμο σε μια γλώσσα προγραμματισμού και τον εκτελούμε σε ένα υπολογιστή με πολλά και διάφορα μεγέθη εισόδων μετρώντας τον χρόνο εκτέλεσης. Εξαρτάται από:

- Τον υπολογιστή
- Την γλώσσα προγραμματισμού
- Τις ικανότητες του προγραμματιστή

Θεωρητικά:

Έστω n είναι το μέγεθος της εισόδου. Εάν $T1(n)$ και $T2(n)$ είναι οι χρόνοι εκτέλεσης δύο διαφορετικών εφαρμογών του αλγόριθμου Α τότε υπάρχει πάντα σταθερά c, τέτοια ώστε $T1(n) = c \cdot T2(n)$, με n πολύ μεγάλο (από την αρχή της σταθερότητας). Η παραπάνω αρχή ΔΕΝ εξαρτάται από τον Η/Υ, την γλώσσα και τις ικανότητες του προγραμματιστή!

Θα χρειαστούμε τον αριθμό των βασικών πράξεων που εκτελούνται από τον αλγόριθμο και όχι τον ακριβή χρόνο που απαιτούν. Το πλήθος των βασικών πράξεων ενός αλγόριθμου εξαρτάται από το μέγεθος της εισόδου του.

Βασικές πράξεις:

- Ανάθεση μιας τιμής σε κάποια μεταβλητή.
- Σύγκριση δύο τιμών.
- Βασικές αριθμητικές πράξεις (π.χ. πρόσθεση, πολλαπλασιασμός, κλπ.).
- Εύρεση της τιμής ενός συγκεκριμένου στοιχείου σε έναν πίνακα.

Εκφράζουμε την πολυπλοκότητα χρόνου $T(n)$ ενός αλγόριθμου ως συνάρτηση του μεγέθους της εισόδου του, δηλαδή:

$T(n) = f(|I|)$, όπου $|I|$ το μέγεθος της εισόδου.

Αναλύοντας την πολυπλοκότητα χρόνου ενός αλγόριθμου, διακρίνουμε τις παρακάτω δύο περιπτώσεις:

- **Ανάλυση Χειρότερης Περίπτωσης** (worst-case analysis)
- **Ανάλυση Αναμενόμενης Περίπτωσης** (average-case analysis)

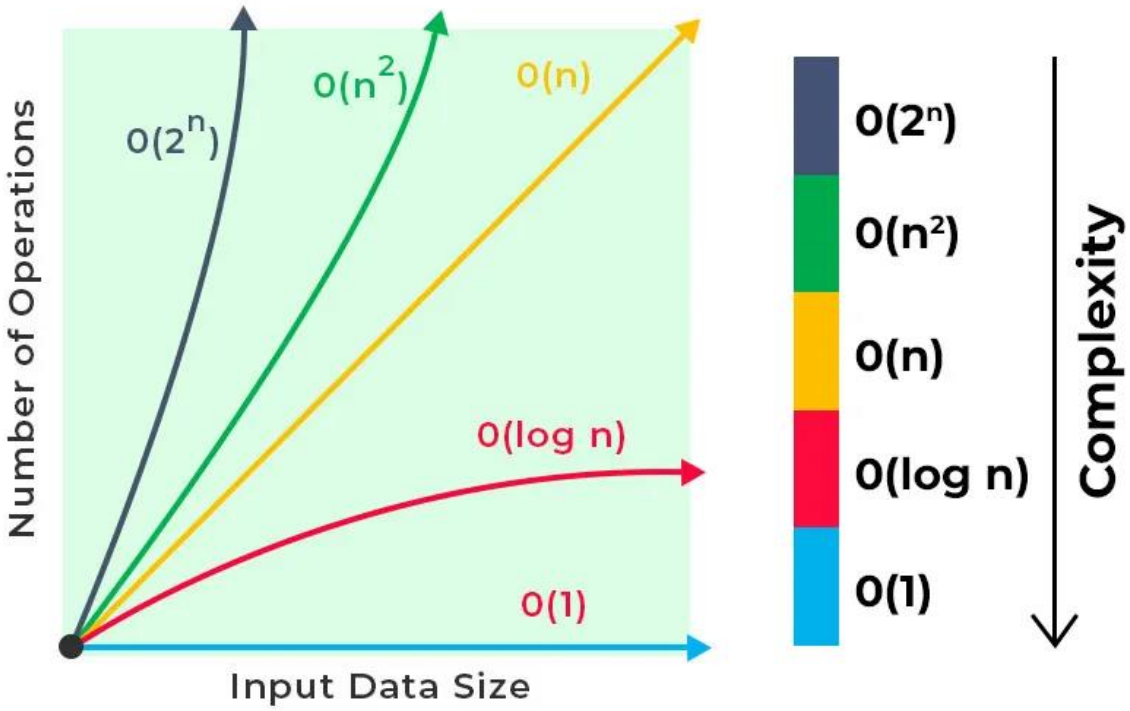
Έστω ότι έχουμε T(n)=8n + 6. Τότε T(n)=8n ή T(n) = n Άρα: η ασυμπτωτική συμπεριφορά της T(n) = 8n + 6 περιγράφεται από τη συνάρτηση T(n) = n (όσο το n μεγαλώνει). Τότε λέμε ότι ο αλγόριθμός μας έχει χρονική πολυπλοκότητα Θ(T(n)) ή Θ(n).

Θέτουμε άνω όριο (χειρότερη περίπτωση) και εισάγεται ο λεγόμενος συμβολισμός O (O – notation ή big-O), από την αγγλική λέξη order.

Κλάσεις Πολυπλοκότητας:

- **Λογαριθμικοί Αλγόριθμοι:** Έχουν χρόνο εκτέλεσης $T(n) = O(\log\text{^}k * n)$, όπου k είναι μια θετική σταθερά.

- **Γραμμικοί Αλγόριθμο:** Έχουν χρόνο εκτέλεσης $T(n) = O(n)$.
- **Πολυωνυμικοί Αλγόριθμο:** Έχουν χρόνο εκτέλεσης $T(n) = O(nk)$, όπου k είναι μια θετική σταθερά.
- **Εκθετικοί Αλγόριθμοι:** Έχουν χρόνο εκτέλεσης $T(n) = O(cn)$, όπου $c > 1$.



17. Αλγόριθμοι αναζήτησης:

Ως αλγόριθμος αναζήτησης μπορεί να χαρακτηριστεί μια συνάρτηση που αναζητεί ένα στοιχείο σε μια δομή. Για παράδειγμα έναν αριθμό σε έναν πίνακα. Για τα παρακάτω παραδείγματα εμείς θα αναζητούμε την τιμή της μεταβλητής target σε έναν πίνακα ακεραίων με όνομα array. Άρα η συνάρτηση μας πρέπει ως όρισμα να δέχεται τον πίνακα, το στοιχείο που αναζητούμε και το μέγεθος του πίνακα για τον λόγο που είδαμε στο προηγούμενο κεφάλαιο. Αξίζει να σημειωθεί πως δεν μπορούμε να υπολογίσουμε το μέγεθος του πίνακα μέσα στην συνάρτηση, αυτό πρέπει

να το υπολογίζουμε εκτός συνάρτησης και να δίνεται στην συνάρτηση ως παράμετρος.

17.1 Σειριακή αναζήτηση (Sequential Search):

Η πιο γνωστή και απλή μεθοδολογία αναζήτησης ενός στοιχείου όπου περνάμε όλο τον πίνακα στοιχείο την φορά και συγκρίνουμε την τιμή που ψάχνουμε με κάθε στοιχείο, όταν το βρούμε σπάει η επανάληψη:

```
int sequential_search(int array[], int array_length, int target)
{
    for (int i = 0; i < array_length; i++)
    {
        if (array[i] == target)
        {
            return i;
        }
    }

    return -1;
}
```

Όπως προαναφέρθηκε η συνάρτηση δέχεται για ορίσματα τον πίνακα, το μέγεθος του και την τιμή που ψάχνει. Περνάει ένα ένα τα στοιχεία του πίνακα συγκρίνοντας καθένα με την τιμή που ψάχνουμε, μόλις το βρει επιστρέφει τον αριθμό του κελιού και τερματίζει η συνάρτηση. Αν δεν βρεθεί τότε επιστρέφετε το -1.

17.2 Δυαδική αναζήτηση (Binary Search):

Η δυαδική αναζήτηση είναι ένας κατά πολύ πιο αποδοτικός αλγόριθμος από την σειριακή αναζήτηση ιδίως στους μεγάλους πίνακες, αλλά προϋποθέτει ο πίνακας να είναι ταξινομημένος. Έστω ότι ο πίνακας είναι

ταξινομημένος με αύξουσα σειρά. Τα βήματα που ακολουθεί είναι τα εξής:

- Υπολογίζει το μέσον του πίνακα και συγκρίνει τον αριθμό που ψάχνουμε με τον αριθμό στο μέσον.
- Αν ο αριθμός που ψάχνουμε είναι μεγαλύτερος από το μέσον τότε σίγουρα βρίσκεται στα δεξιά από το μέσον οπότε «πετιέται το αριστερό κομμάτι του πίνακα. Στην ουσία θέτουμε την αρχή του πίνακα να είναι στο μέσον + 1.
- Στην συνέχεια υπολογίζεται το καινούργιο μέσον στον νέο πίνακα και επαναλαμβάνεται ο έλεγχος και η διαίρεση μέχρι είτε να βρεθεί το στοιχείο είτε να τελειώσει ο πίνακας.

```
int binarySearch(int array[], int length, int target)
{
    int lowerBound = 0, upperBound = length - 1;

    while (lowerBound <= upperBound)
    {
        int mid = lowerBound + (upperBound - lowerBound)
/ 2;

        if (array[mid] == target)
        {
            return mid;
        }
        else if (array[mid] < target)
        {
            lowerBound = mid + 1;
        }
        else
        {
```

```
            upperBound = mid - 1;
        }
    }
    return -1;
}
```

17.3 Αναδρομική δυαδική αναζήτηση (Recursive Binary Search):

Ίδια ακριβώς λογική αλλά αντί να έχουμε μια δομή while κάνουμε τις επαναλήψεις με αναδρομικές κλήσεις της ίδιας συνάρτησης:

```
#include <stdio.h>

int binarySearch(int array[], int x, int low, int high)
{
    if (high >= low)
    {
        int mid = low + (high - low) / 2;

        // If found at mid, then return it
        if (array[mid] == x)
            return mid;
        // Search the left half
        if (array[mid] > x)
            return binarySearch(array, x, low, mid - 1);
        // Search the right half
        return binarySearch(array, x, mid + 1, high);
    }
    return -1;
}
```

18. Αλγόριθμοι ταξινόμησης:

Αφορά αντίστοιχα συναρτήσεις που ταξινομούν έναν πίνακα δηλαδή θέτουν τα στοιχεία του κατά έναν τρόπο, αύξων ή φθίνων. Για όλα τα παραδείγματα θεωρούμε ότι θέλουμε να ταξινομήσουμε τον πίνακα κατά αύξουσα σειρά. Για φθίνουσα αντικαθίστανται το > από <.

18.1 Bubble Sort (Φυσαλίδα):

Ο πιο διαδεδομένος τρόπος ταξινόμησης. Λέγεται bubble (φυσαλίδα) επειδή κάθε στοιχείο μετακινείται προς το τέλος του πίνακα σε κάθε επανάληψη, όπως και οι φυσαλίδες που μετακινούνται προς την επιφάνεια του νερού.

Βήματα:

- Ξεκινώντας από το πρώτο στοιχείο, συγκρίνεται το πρώτο και το δεύτερο στοιχείο.
- Εάν το πρώτο στοιχείο είναι μεγαλύτερο από το δεύτερο στοιχείο, ανταλλάσσονται.
- Τώρα, συγκρίνετε το δεύτερο και το τρίτο στοιχείο. Ανταλλάσσονται αντίστοιχα αν δεν είναι όπως θέλουμε.
- Η παραπάνω διαδικασία συνεχίζεται μέχρι το τελευταίο στοιχείο.

Αλγόριθμος:

```
void bubbleSort(int array[], int size)
{
    for (int i = 0; i < size - 1; i++)
    {
        int swapped = 0;
        for (int j = 0; j < size - i - 1; j++)
        {
            if (array[j] > array[j + 1])
            {
```

```
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
                swapped = 1;
            }
        }

        if (swapped == 0)
        {
            break;
        }
    }
}
```

Ο ρόλος της swapped μεταβλητής είναι να ελέγχει αν ο πίνακας είναι ήδη ταξινομημένος καθώς γίνεται 0 μόνο για επανάληψη που δεν ίσχυε η συνθήκη ελέγχου. Αυτή αποτελεί μια βελτιωμένη έκδοση της bubble sort καθώς η απλή δεν έχει τέτοιον έλεγχο.

Οπτικά:

7	3	8	6	4

1ο Πέρασμα: 3-7-6-4-8

2ο Πέρασμα: 3-6-4-7-8

3ο Πέρασμα: 3-4-6-7-8

Ο πίνακας είναι ταξινομημένος, ωστόσο αναλόγως και την υλοποίηση, η συνάρτηση μπορεί να κάνει ακόμα ένα πέρασμα και τότε θα σταματήσει.

Πολυπλοκότητα:

Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$

Στην πρώτη επανάληψη είναι (n-1) πλήθος ελέγχων, στην δεύτερη (n-2) φτάνοντας στην τελευταία με 1. Άρα έχουμε:

```
(n-1) + (n-2) + (n-3) +.....+ 1 = n(n-1)/2
```

Που ισούται με n^2 άρα και $O(n^2)$.

Ακόμα:

- **Worst Case Complexity:** $O(n^2)$
  Αν θέλουμε να ταξινομήσουμε σε αύξουσα σειρά και ο πίνακας είναι ήδη σε φθίνουσα σειρά.
- **Best Case Complexity:** O(n)
  Όταν ο πίνακας είναι ήδη ταξινομημένος οπότε η εξωτερική επανάληψη τρέχει για έναν αριθμό n ενώ η εσωτερική καθόλου. Έτσι έχουμε μόνο n πλήθος ελέγχων οπότε γραμμική πολυπλοκότητα.
- **Average Case Complexity:** $O(n^2)$
  Τυχαιότητα στην θέση των στοιχείων.

Ο αλγόριθμος αυτός χρησιμοποιείται στις εξής περιπτώσεις:

- Όταν δεν μας νοιάζει η πολυπλοκότητα.
- Θέλουμε έναν απλό και λιτό κώδικα στο πρόγραμμα μας.

18.2 Insertion Sort (Εισαγωγής):

Σε αυτή τη μέθοδο σε κάθε επανάληψη επιλέγεται ένα στοιχείο του πίνακα και τοποθετείται απευθείας στην τελική σωστή του θέση. Λέγεται insertion (εισαγωγής) επειδή σε κάθε επανάληψη επιλέγεται ένα στοιχείο, βολεύει να το σκεφτόμαστε σαν τα χαρτιά που φτιάχνουμε στο χέρι μας σε ένα παιχνίδι χαρτιών.

Βήματα:

- Το πρώτο στοιχείο του πίνακα θεωρείται ταξινομημένο οπότε παίρνουμε το δεύτερο στοιχείο και το αποθηκεύουμε.
- Αν το δεύτερο στοιχείο είναι μικρότερο από το πρώτο τότε το πρώτο αντιγράφεται στο δεύτερο και το αποθηκευμένο μπαίνει στο πρώτο.
- Στην επόμενη επανάληψη διαλέγεται το τρίτο στοιχείο και ελέγχεται με τα δύο αριστερά του και γίνεται ανταλλαγή αν πρέπει.
- Επανάληψη των βημάτων μέχρι τέλους.

Αλγόριθμος:

```
void insertionSort(int array[], int size)
{
    for (int i = 1; i < size; i++)
    {
        int current = array[i], pos = i;
        while (pos > 0 && array[pos - 1] > current)
        {
            array[pos] = array[pos - 1];
            pos--;
        }
        array[pos] = current;
    }
}
```

Οπτικά:

7	3	8	6	4

1ο Πέρασμα: 7-7-8-6-4

[3]

3-7-8-6-4

2ο Πέρασμα: 3-6-7-8-4

3ο Πέρασμα: 3-4-6-7-8

Πολυπλοκότητα:

Best	O(n)
Worst	$O(n^2)$
Average	$O(n^2)$

- **Worst Case Complexity:** $O(n^2)$
  Αν θέλουμε να ταξινομήσουμε σε αύξουσα σειρά και ο πίνακας είναι ήδη σε φθίνουσα σειρά.
- **Best Case Complexity:** O(n)
  Όταν ο πίνακας είναι ήδη ταξινομημένος οπότε η εξωτερική επανάληψη τρέχει για έναν αριθμό n ενώ η εσωτερική καθόλου. Έτσι έχουμε μόνο n πλήθος ελέγχων οπότε γραμμική πολυπλοκότητα.
- **Average Case Complexity:** $O(n^2)$
  Τυχαιότητα στην θέση των στοιχείων.

Ο αλγόριθμος αυτός χρησιμοποιείται στις εξής περιπτώσεις:

- Ο πίνακας είναι μικρός.
- Μόνο λίγα στοιχεία θέλουν ταξινόμηση.

18.3 Selection Sort (Επιλογής):

Επιλέγει τον μικρότερο αριθμό του πίνακα κάθε φορά και τον τοποθετεί στην αρχή.

Αλγόριθμος:

```
void selectionSort(int array[], int size)
{
    int min, temp;
    for (int i = 0; i < size - 1; i++)
    {
        min = i;
```

```
        for (int k = i + 1; k < size; k++)
        {
            if (array[k] < array[min])
            {
                min = k;
            }
            temp = array[min];
            array[min] = array[i];
            array[i] = temp;
        }
    }
}
```

Οπτικά:

7	3	8	6	4

1ο Πέρασμα: 3-7-8-6-4

2ο Πέρασμα: 3-4-8-6-7

3ο Πέρασμα: 3-4-6-8-7

4ο Πέρασμα: 3-4-6-7-8

Πολυπλοκότητα:

Best	$O(n^2)$
Worst	$O(n^2)$
Average	$O(n^2)$

Ο αλγόριθμος αυτός χρησιμοποιείται στις εξής περιπτώσεις:

- Έχουμε μια μικρή λίστα για ταξινόμηση
- το κόστος της ανταλλαγής δεν έχει σημασία
- ο έλεγχος όλων των στοιχείων είναι υποχρεωτικός

- Το κόστος εγγραφής σε μια μνήμη έχει σημασία όπως στη μνήμη flash (ο αριθμός εγγραφών/ανταλλαγών είναι O(n) σε σύγκριση με O(n2) τύπου bubbleSort).

18.4 Merge Sort:

Ο merge sort είναι ένας αναδρομικός αλγόριθμος που βασίζεται στην αρχή του Divide and Conquer, Διαίρει και βασίλευε. Εδώ, ένα πρόβλημα χωρίζεται σε πολλά υποπροβλήματα. Κάθε υποπρόβλημα επιλύεται ξεχωριστά. Τέλος, τα υποπροβλήματα ενώνονται για να σχηματίσουν την τελική λύση.

Αλγόριθμος:

```
void merge(int arr[], int p, int q, int r)
{
    // Create L ← A[p..q] and M ← A[q+1..r]
    int n1 = q - p + 1;
    int n2 = r - q;

    int L[n1], M[n2];

    for (int i = 0; i < n1; i++)
        L[i] = arr[p + i];
    for (int j = 0; j < n2; j++)
        M[j] = arr[q + 1 + j];
    // Maintain current index of sub-arrays and main
array
    int i, j, k;
    i = 0;
    j = 0;
    k = p;
```

```
    // Until we reach either end of either L or M, pick
larger among
    // elements L and M and place them in the correct
position at A[p..r]
    while (i < n1 && j < n2)
    {
        if (L[i] <= M[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = M[j];
            j++;
        }
        k++;
    }
    // When we run out of elements in either L or M,
    // pick up the remaining elements and put in A[p..r]
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2)
    {
        arr[k] = M[j];
        j++;
        k++;
    }
```

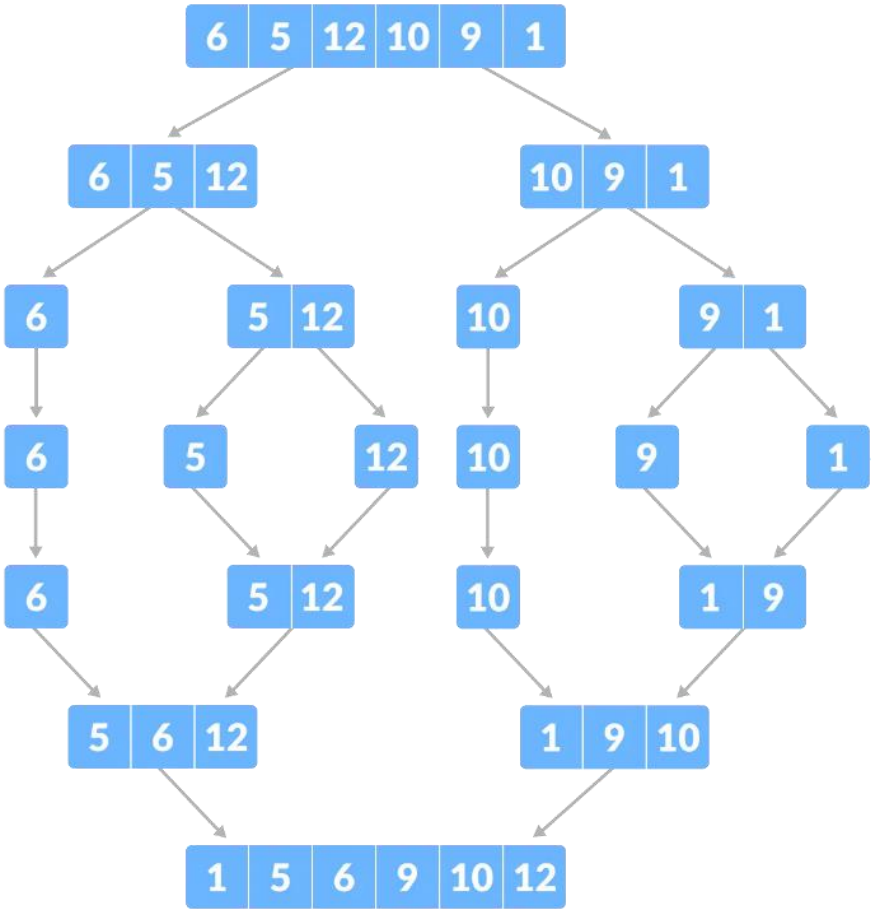
```
}

// Divide the array into two subarrays, sort them and
merge them
void mergeSort(int arr[], int l, int r)
{
    if (l < r)
    {
        // m is the point where the array is divided
into two subarrays
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        // Merge the sorted subarrays
        merge(arr, l, m, r);
    }
}
```

Πολυπλοκότητα:

Best	O(n*log n)
Worst	O(n*log n)
Average	O(n*log n)

Οπτικά:



18.5 Quick Sort:

Η quick sort είναι επίσης μια αναδρομική μέθοδος και μοιράζεται την ίδια λογική του divide and conquer. Εδώ σε κάθε πέρασμα ορίζεται ένα pivot και στοιχεία μικρότερα από αυτό πάνε αριστερά του και μεγαλύτερα δεξιά. Συνήθως ως pivot ορίζεται το τελευταίο στοιχείο αλλά δεν είναι απαραίτητη αυτή η προσέγγιση.

Αλγόριθμος:

```
// function to swap elements
void swap(int *a, int *b)
{
    int t = *a;
```

```
    *a = *b;
    *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high)
{
    // select the rightmost element as pivot
    int pivot = array[high];
    // pointer for greater element
    int i = (low - 1);
    // traverse each element of the array
    // compare them with the pivot
    for (int j = low; j < high; j++)
    {
        if (array[j] <= pivot)
        {
            // if element smaller than pivot is found
            // swap it with the greater element pointed
by i
            i++;
            // swap element at i with element at j
            swap(&array[i], &array[j]);
        }
    }
    // swap the pivot element with the greater element
at i
    swap(&array[i + 1], &array[high]);
    // return the partition point
    return (i + 1);
}
```

```c
void quickSort(int array[], int low, int high)
{
    if (low < high)
    {
        // find the pivot element such that
        // elements smaller than pivot are on left of
pivot
        // elements greater than pivot are on right of
pivot
        int pi = partition(array, low, high);
        // recursive call on the left of pivot
        quickSort(array, low, pi - 1);
        // recursive call on the right of pivot
        quickSort(array, pi + 1, high);
    }
}
```

Οπτικά:

7	3	8	6	4

1ο Πέρασμα: pivot = 4 άρα τα στοιχεία μικρότερα αυτού πάνε αριστερά του: 3-4-8-6-7

2ο Πέρασμα: Έχουν δημιουργηθεί δύο διαμερίσματα: [3,4] και [8,6,7], το αριστερό είναι έτοιμο οπότε pivot = 7 για το δεύτερο => [6,7,8].

Ενώνοντας τα: 3-4-6-7-8.

Πολυπλοκότητα:

Best	O(n*log n)
Worst	$O(n^2)$
Average	O(n*log n)

19. Structs:

Ένα struct είναι μια δομή που απαρτίζεται από ένα σύνολο μεταβλητών διαφορετικών ή και ίδιων τύπου δεδομένων. Είναι δηλαδή μια συλλογή μεταβλητών, που μπορούν να είναι και διαφορετικοί τύποι μεταξύ τους, που ακούν συνολικά στο όνομα struct. Είναι δηλαδή αποθηκευμένα στην μνήμη με τον εξής τρόπο:

char name[] = "Bob"
int age = 20
float gpa = 7.5

Όπου όλη η παραπάνω συστοιχία ακούει στο όνομα struct, για την ακρίβεια στο όνομα που θα δώσουμε στο struct μας. Για να ορίσουμε ένα struct γράφουμε:

```
struct Student
{
    char name[20];
    int age;
    float gpa;
};
```

Μεγάλη προσοχή, δεν ορίζουμε τιμές στις μεταβλητές, απλά τις δηλώνουμε. Κάτι άλλο που παρατηρούμε είναι ότι μετά την εξωτερική αγκύλη χρειάζεται ένα ερωτηματικό.

Το struct που φτιάξαμε υπακούει στο όνομα Student και έχει τέσσερα πεδία. Μπορούμε με αυτό να φτιάξουμε πολλούς students ο καθένας με διαφορετικά πεδία, γιαυτό δεν ορίζονται και πεδία κατά την δήλωση. Μέχρι στιγμής ότι έχουμε γράψει είναι για την αρχική δήλωση ενός struct . Για να φτιάξουμε ένα struct και να βάλουμε τιμές στα πεδία του πρέπει αρχικά να το δηλώσουμε στην main και να δώσουμε ένα όνομα:

```
struct Student Student1;
struct Student Student2;
```

Θυμίζω ότι Student είναι το όνομα του Struct στην γενική μορφή του που απλά δηλώνει μεταβλητές και δεν έχει τιμές, για να πάρει τιμές πρέπει να δηλωθεί ένα άλλο struct με άλλο όνομα που είναι τύπου Student. Πρακτικά είναι σαν να δημιουργούμε τον δικό μας τύπο δεδομένων.

Επόμενο βήμα είναι να γεμίσουμε τα πεδία από κάθε student. Ένα θα το κάνουμε δυναμικά δηλαδή θα δώσουμε εμείς οι προγραμματιστές τιμές και στο άλλο θα διαβάζουμε από τον χρήστη και θα βάζουμε ότι τιμές δώσει.

Επειδή κάθε student έχει τέσσερα πεδία όταν πάμε να βάλουμε τιμή σε ένα πρέπει να δηλώσουμε σε ποιο βάζουμε, γιαυτό γράφουμε το όνομα του struct και το ακολουθεί μια τελεία (.) μετά την οποία μπαίνει το πεδίο που μας ενδιαφέρει:

```
Student1.age = 21;
```

Με την ίδια λογική μπορούμε να δώσουμε τιμές σε όλα τα πεδία εκτός του ονόματος:

```
Student1.gpa = 3.5;
```

Για το όνομα δεν μπορούμε απλά να γράψουμε name = κάτι. Αυτό διότι είναι ένας πίνακας χαρακτήρων, άρα το όνομα που θέλουμε να δώσουμε πρέπει να σπάσει σε χαρακτήρες και κάθε χαρακτήρας να μπει σε ένα κελί. Αυτή την δουλειά μπορούμε να την κάνουμε εύκολα με την συνάρτηση strcpy() που είδαμε σε προηγούμενο κεφάλαιο:

```
strcpy(Student1.name, "Bob");
```

Μπορούμε να ελέγξουμε ότι δόθηκαν σωστά τα στοιχεία εκτυπώνοντας τα:

```
printf("Name: %s\n", Student1.name);
printf("Age: %d\n", Student1.age);
```

```
printf("GPA: %f\n", Student1.gpa);
```

Και βλέπουμε ότι περάστηκαν σωστά:

```
Name: Bob
Age: 21
GPA: 3.500000
```

Για τον δεύτερο student θα παίρνουμε τα στοιχεία από τον χρήστη και θα γεμίσουμε τα πεδία του με αυτά.

```
printf("Give the name of the second student: ");
scanf("%s", Student2.name);
printf("Give the age of the second student: ");
scanf("%d", &Student2.age);
printf("Give the gpa of the second student: ");
scanf("%f", &Student2.gpa);
```

Και αντίστοιχα για να τα δούμε:

```
printf("Name: %s\n", Student2.name);
printf("Age: %d\n", Student2.age);
printf("GPA: %f\n", Student2.gpa);
```

Πολύ συχνά τα struct συνδυάζονται με το keyword typedef με τον εξής τρόπο:

```
typedef struct
{
    char name[20];
    int age;
    float gpa;
} Student;
```

Και έτσι πλέον δεν χρειάζεται κάθε φορά που έχουμε struct να το γράφουμε αλλά μπορούμε απευθείας έτσι:

```
Student student1;
```

Ένας άλλος τρόπος να δώσουμε τιμές στα πεδία ενός struct είναι ο εξής:

```
Student student1 = {"Bob", 20, 3.5};
```

19.1 Πίνακας από structs:

Αντί να φτιάχνουμε πολλά ξεχωριστά structs μπορούμε να κάνουμε έναν πίνακα από αυτά που δηλώνεται έτσι:

```
Student students[] = {student1, student2, student3};
```

Προφανώς προϋποθέτει την ύπαρξη των student1,2,3.

Μπορούμε για την εκτύπωση των πεδίων να χρησιμοποιήσουμε μια δομή επανάληψης for όπως και στους απλούς πίνακες. Αντίστοιχα πρέπει πρώτα να υπολογίσουμε το μέγεθος του struct με τον γνωστό τρόπο:

```
int size = sizeof(students) / sizeof(students[0]);
```

και στην συνέχεια μπορούμε να εκτυπώσουμε τα στοιχεία έτσι:

```
for (int i = 0; i < size; i++)
    {
        printf("Name: %s, Age: %d, GPA: %f\n",
students[i].name, students[i].age, students[i].gpa);
    }
```

20. Enums:

Enumeration (ή enum), πρόκειται για έναν τύπο δεδομένων που ορίζει ο χρήστης και χρησιμοποιείται κυρίως για την εκχώρηση ονομάτων σε ακέραιες σταθερές, κάνοντας έτσι ένα πρόγραμμα εύκολο στην ανάγνωση και τη συντήρηση. Ορίζονται εξωτερικά της main με τον εξής τρόπο:

```
enum Day
{
    Sunday,
```

```
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};
```

Κάθε σταθερά αντιπροσωπεύεται από έναν ακέραιο αριθμό. Η πρώτη σταθερά δηλαδή η Sunday έχει τον αριθμό 0, η δεύτερη τον 1 και ούτω καθεξής. Αυτές οι τιμές είναι οι προκαθορισμένες τιμές, μπορούμε να θέσουμε δικές μας με τον εξής τρόπο:

```
enum Day
{
    Sunday = 1,
    Monday = 2,
    Tuesday = 3,
    Wednesday = 4,
    Thursday = 5,
    Friday = 6,
    Saturday = 7
};
```

Για να χρησιμοποιήσουμε το enum αρχικά δημιουργούμε μια μεταβλητή τύπου enum και της αναθέτουμε κάποια τιμή:

```
enum Day today = Sunday;
```

Σημαντική σημείωση οι σταθερές μεταχειρίζονται από τον compiler ως ακέραιοι και όχι strings. Οπότε πρέπει να τις συμπεριφερόμαστε σαν ακέραιους αριθμούς, άρα για να εκτυπώσουμε ένα enum γράφουμε:

```
printf("Today is %d\n", today);
```

Μπορούμε να ελέγξουμε αν σήμερα είναι σαββατοκύριακο με τον παρακάτω κώδικα:

```c
if (today == 1 || today == 7)
{
    printf("Today is weekend\n");
}
```

Ή για να είναι ακόμα πιο ευκρινής ο κώδικας μπορούμε και:

```c
if (today == Sunday || today == Saturday)
{
    printf("Today is weekend\n");
}
```

21. Κανονικές Εκφράσεις (Regular Expressions):

Το παρών κεφάλαιο είναι μια αντιγραφή των διαφανειών του μαθήματος «Εισαγωγή στα λειτουργικά συστήματα» του αναπληρωτή καθηγητή Σιδηρόπουλου Αντώνη.

Οι κανονικές εκφράσεις (regular expressions – regex – regexp) είναι ένας σύντομος και σαφής τρόπος έκφρασης ενός μοτίβου αναζήτησης χαρακτήρων. Συνήθως τέτοια μοτίβα χρησιμοποιούνται από αλγόριθμους αναζήτησης συμβολοσειρών για λειτουργίες "εύρεσης" ή "εύρεσης και αντικατάστασης" σε συμβολοσειρές ή για επικύρωση εισόδου.

Όλες οι γλώσσες προγραμματισμού έχουν υποστήριξη για κανονικές εκφράσεις:

- Είτε μέσω βιβλιοθηκών, πχ: C, C++
- Είτε είναι ενσωματωμένες στην γλώσσα, πχ: perl, php, Javascript

Προτάθηκαν για πρώτη φορά το 1951 (S. Kleene). Ο Ken Thomson (1968) τις χρησιμοποίησε στην εντολή grep (global regular expression print). Η εντολή grep αναζητά μια κανονική έκφραση μέσα σε ένα αρχείο. Διαβάζει το αρχείο γραμμή-γραμμή και ελέγχει αν στην τρέχουσα γραμμή

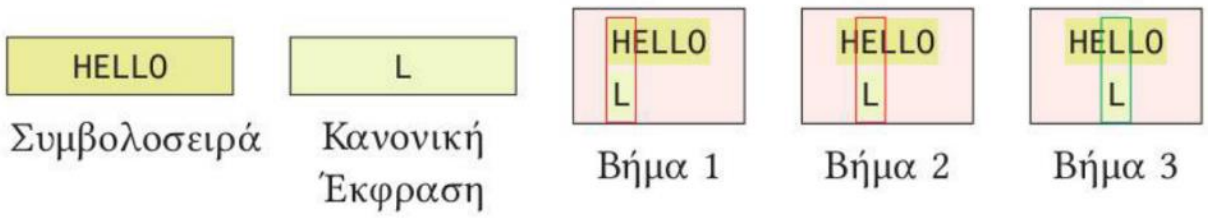
ταιριάζει η κανονική έκφραση. Αν ναι, τότε τυπώνει την γραμμή, αν όχι τότε προχωρά στην επόμενη.

Μια κανονική έκφραση είναι παρόμοια με μια μαθηματική έκφραση. Μια μαθηματική έκφραση αποτελείται από τελεστέους (operands) και τελεστές (operators).

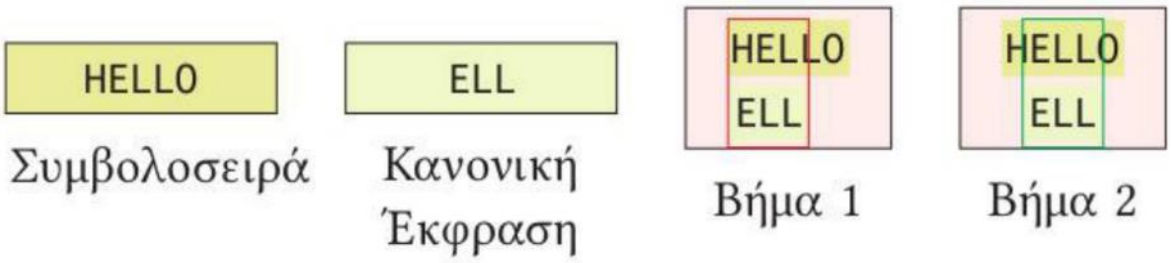
Μια κανονική έκφραση αποτελείται από:

- Άτομα (atoms): Προσδιορίζουν τι αναζητούμε.
- Τελεστές (operators): Προσδιορίζει τις πράξεις.

Ένα απλό παράδειγμα είναι η αναζήτηση του γράμματος L στην λέξη HELLO:



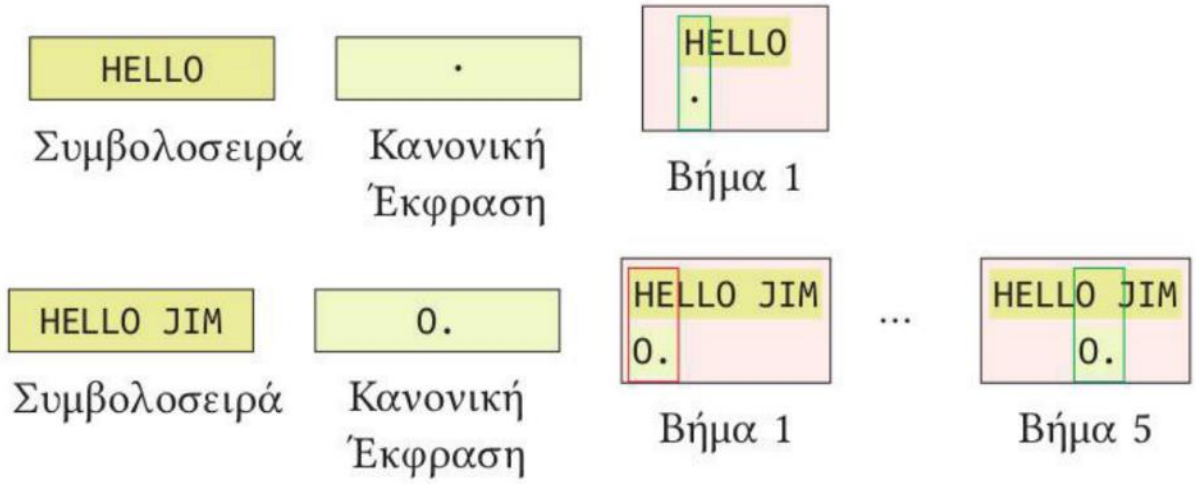
Εκτός μεμονωμένων γραμμάτων εφικτή είναι και η αναζήτηση συστοιχιών γραμμάτων:



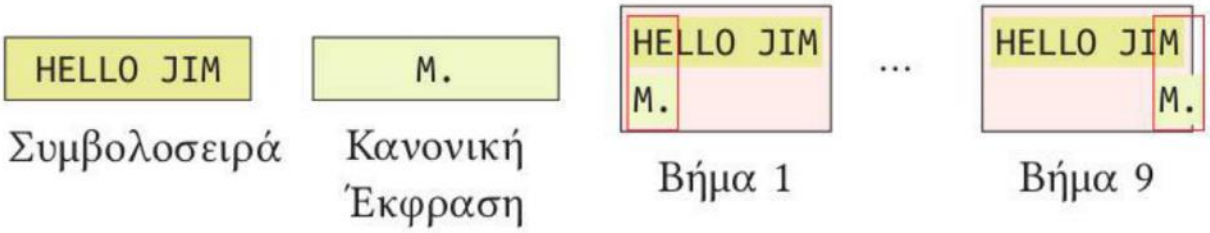
Εδώ πρέπει να βρεθούν και τα τρία γράμματα με αυτή τη σειρά στην λέξη.

21.1 Άτομα:

Αν θέλουμε να αναζητήσουμε την ύπαρξη οποιουδήποτε χαρακτήρα, όχι κάποιου συγκεκριμένου δηλαδή, τοποθετούμε τον τελεστή τελεία (.):



Το οποιοδήποτε χαρακτήρας δεν περιλαμβάνει και το κενό.



Αν θέλουμε να αναζητήσουμε πολλαπλούς χαρακτήρες μπορούμε να γράψουμε τα εξής:

- [αβγδ]: ύπαρξη α ή β ή γ ή δ.
- [α-δ]: ύπαρξη οποιουδήποτε αριθμού μέσα στο σύνολο α έως δ.
- [^αβγδ] ή [^α-δ]: οποιοσδήποτε χαρακτήρας εκτός από τους αβγδ.

Υπάρχουν και έτοιμα σύνολα χαρακτήρων:

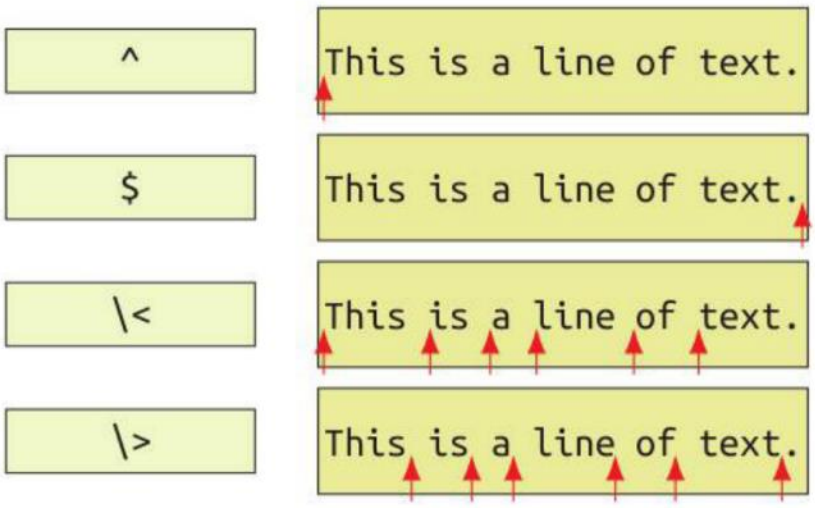
- [:alnum:] ➔ 0-9a-zA-Z, eg: [[:alnum:],+] □ [0-9a-zA-Z,+]
- [:alpha:] ➔ a-zA-Z (Περιλαμβάνονται και Α-Ωα-ω και όλα τα γράμματα όλων των εθνικών αλφάβητων όταν το σύνολο χαρακτήρων είναι UTF. Όταν το σύνολο χαρακτήρων είναι ASCII τότε ισχύει [:alpha:] == a-zA-Z)

- [:cntrl:], [:print:] ➔ Κοντρόλ χαρακτήρας (<31 στον πίνακα ASCII), εκτυπώσιμος χαρακτήρας (visible and spaces).
- [:digit:], [:xdigit:] ➔ 0-9 , 0-9A-F
- [:graph:] ➔ εκτυπώσιμος χαρακτήρας εκτός spaces/tabs.
- [:lower:], [:upper:], ➔ πεζός, κεφαλαίος
- [:punct:], [:space:] ➔ στίξη (\]\[!"#$%&'()*+,./:;<=>?@\^_`{|}~-)

21.2 Άγκυρες:

Είναι άτομα που χρησιμοποιούνται για να αντιστοιχήσουν το πρότυπο σε ένα συγκεκριμένο τμήμα, δεν αντιστοιχούν σε χαρακτήρες αλλά καθορίζουν ένα σημείο.

- ^: Σημαίνει αρχή συμβολοσειράς.
- $: Σημαίνει τέλος συμβολοσειράς.
- \<: Σημαίνει αρχή λέξης. Οι λέξεις οριοθετούνται από τα κενά και τα σημεία στίξης.
- \>: Σημαίνει τέλος λέξης.



21.3 Τελεστές:

Ο τελεστής **ακολουθίας** (sequence operator) δεν εκφράζεται με σύμβολο. Αυτό σημαίνει ότι αν μια σειρά από atoms φαίνονται σε μια κανονική έκφραση, υποδηλώνεται η παρουσία ενός αόρατου sequence operator ανάμεσά τους.

test	Αναζήτηση του t, να ακολουθεί το e, να ακολουθεί το s, να ακολουθεί το t (ουσιαστικά η λέξη test)
x[0-9A-F][0-9A-F]	Αναζήτηση του x, να ακολουθεί ένα ψηφίο από τα (0, 1, 2, … , 9 , A , B , … , F) και να ακολουθεί ακόμη ένα ψηφίο από το ίδιο σύνολο. Ας σημειωθεί ότι δεν υπάρχει με κανέναν τρόπο συσχέτιση των δύο τελευταίων ψηφίων, δηλαδή ταιριάζει η συμβολοσειρά x66 αλλά και η x6E . Οι δύο κλάσεις χαρακτήρων που ορίσαμε, αν και ίδιες, μπορούν να ταιριάζουν σε διαφορετικά ψηφία.
^[0-9]	Αναζήτηση της αρχής συμβολοσειράς και έπειτα ένα αριθμητικό ψηφίο. Ουσιαστικά θα ταιριάξει σε συμβολοσειρές που ξεκινούν με αριθμό.
^[0-9]$	Αναζήτηση της αρχής συμβολοσειράς, έπειτα ένα αριθμητικό ψηφίο και να ακολουθεί το τέλος συμβολοσειράς. Ουσιαστικά, θα ταιριάξει σε συμβολοσειρές που αποτελούνται από έναν μόνο αριθμητικό χαρακτήρα.

Ο τελεστής **εναλλαγής** (alternation operator) χρησιμοποιείται για να ορίσει μια ή περισσότερες εναλλακτικές περιπτώσεις:

job|hobby	Θα ταιριάξει, είτε την κανονική έκφραση «job» είτε την «hobby»
here|there|away	Θα ταιριάξει μία από τις τρεις κανονικές εκφράσεις. Δεν υπάρχει περιορισμός στο πλήθος των εναλλακτικών που μπορούν να μπουν σε συνεχόμενες OR.

^Test|^This|\<Mr\>	Θα ταιριάξει μία από τις τρεις κανονικές εκφράσεις. Οι κανονικές εκφράσεις μπορούν να περιέχουν οτιδήποτε, ακόμη και άγκυρες, και είναι ανεξάρτητες μεταξύ τους.

Ο τελεστής **επανάληψης** (repetition operator) καθορίζει για το atom που υπάρχει ακριβώς πριν από τον τελεστή πόσες φορές πρέπει να επαναληφθεί:

{n}	Το προηγούμενο atom ακριβώς n φορές
{n,m}	Το προηγούμενο atom από n έως m φορές
{n,}	Το προηγούμενο atom n ή περισσότερες φορές
* ή {0,}	Το προηγούμενο atom 0 ή περισσότερες φορές
+ ή {1,}	Το προηγούμενο atom 1 ή περισσότερες φορές
? ή {0,1}	Το προηγούμενο atom 0 ή 1 φορές

Η χρήση των τελεστών επανάληψης αυξάνει ιδιαίτερα τον χρόνο ελέγχου. Για αυτό, θα πρέπει να γίνεται προσεκτική «κατασκευή» μιας κανονικής έκφρασης, ώστε να μην δημιουργεί άχρηστους συνδυασμούς, ιδίως εάν πρόκειται να ελεγχθεί σε μεγάλο όγκο δεδομένων (πχ. Αρχεία καταγραφής).

ΚΕ	εξήγηση	Εφαρμογή
A{3}	AAA	**AAA**, B**AAA**BBB, XYZ**AAA**AAATT
B{2,4}	BB ή BBB ή BBBB ⇔ BB|BBB|BBBB	**BB**, XY**BB**BBZX, QWE**BBBB**BBBB**BB**XZY
AB{1,3}	AB|ABB|ABBB	**ABB**, **ABBB**BABB
A{1,3}B	AB|AAB|AAAB	**AB**BB, B**AAB**BBB**AB**BB
A{1,3}B{1,3}	AB|ABB|ABBB| AAB|AABB|AABBB| AAAB|AAABB|AAABBB	**ABBB**, CXZ**AABBB**BB**AB**ZX
AB{4,}C	ABBBBC|ABBBBBC|ABBBBBBC| ..	**ABBBBC**, CXZA**BBBBBBC**CBB, CB**BBBBBBC**ABBBBBZX
AB*C	AC|ABC|ABBC|ABBBC|..	

Τελεστής **ομαδοποίησης** (Group Operator), είναι ένα ζεύγος παρενθέσεων που ανοίγουν και κλείνουν. Όταν μια ομάδα χαρακτήρων περικλείεται σε παρενθέσεις ο επόμενος τελεστής εφαρμόζεται σε όλη την ομάδα:

ΚΕ	Εφαρμογή
^([+][0-9]{1,3})?[0-9]{10}$	0123456789 +300123456789 +10123456789
^[[:alnum:]_.-]+@[[:alnum:]-]{2,}[.][a-z]{2,5}$	test@test.com test_d@test.com test_@t-.com __@--.com

Μόνο αν υπάρχει το «+» θέλουμε του επιπλέον αριθμητικούς χαρακτήρες

ΚΕ	Εφαρμογή
^[[:alnum:]_.-]+@([[:alnum:]-]{2,}[.]){1,}[a-z]{2,5}$	test@test.com test_d@test.com test_@t-.com __@--.com test@test2.test.com test_d@first.second.third.test.com

Τελεστής **αποθήκευσης** (Save) & **Back Reference.** Με την ομαδοποίηση δίνεται συγχρόνως και «εντολή αποθήκευσης σε buffer» του string που ταίριαξε στο τμήμα της RE που είναι στις παρενθέσεις. Υπάρχουν 9 buffers που μπορούν να χρησιμοποιηθούν για αποθήκευση. Οι buffers συμβολίζονται με \1, \2, \3, …, \9.

σύμβ.	εξήγηση
(.)\1	AA,BB,ΓΓ,…,00,11,22 … οποιουσδήποτε 2 χαρακτήρες που επαναλαμβάνονται. YGDF//GHSFD, hgaasf
([0-9])[0-9]*\1	55, 58765, 1675566551, 435231, 4543, 122334, ~~34567~~, asdd22tf, ~~sdf45drr5~~, a2as5235
([0-9]{3})\1	346346, 987987, 123123, 51231237 …
([0-9]{3})\1[A-Z]*\1	346346346, 987987ADS987, 123123TPEE123, A123123A4B123,
([0-9])\1{3} ⇔ ([0-9])\1\1\1	5555, 9999, 2222, 2332222243, 342222222456, …
([0-9]{2})\1{2} ⇔ ([0-9]{2})\1\1	555555 , 828282, 161616, …
(.)(.).{2}\2\1	ABXYBA, EP98PE, 6DYRERE, A6DYREYD6R
(.).{3}\1	AXCDA, XCXADSX, ZUAXCDAOP,

Ο χαρακτήρας \ (backslash) χρησιμοποιείται είτε για να αποδώσει ειδική σημασία στον χαρακτήρα που ακολουθεί, είτε για να αναιρέσει την ειδική σημασία του (αν είναι ειδικός χαρακτήρας):

*	Τελεστής επανάληψης
*	Ο χαρακτήρας «*»
\[	Ο χαρακτήρας «[»

21.4 Ειδικοί χαρακτήρες:

*	τελεστής
+	τελεστής
?	Τελεστής

[	Κλάση
{	Τελεστής
(,)	Ομαδοποίηση
|	Or
]	μόνο μετά από [
}	μόνο μετά από {
^	στην αρχή της RE (αρχή string), όταν είναι πρώτος χαρακτήρας μέσα σε κλάση (άρνηση), σε άλλα σημεία δεν έχει ειδική σημασία.
-	μέσα σε κλάση (εύρος). Αλλιώς ο χαρ. «-».
$	στο τέλος της RE (τέλος string). Αλλιώς ο χαρ. «$».
.	οποιοσδήποτε χαρακτήρας. Μέσα σε [] ο χαρακτήρας «.».

Ο χαρακτήρας \ (backslash) χρησιμοποιείται είτε για να αποδώσει ειδική σημασία στον χαρακτήρα που ακολουθεί, είτε για να αναιρέσει την ειδική σημασία του (αν είναι ειδικός χαρακτήρας).

\1..\9	back reference
\<, \>	anchors
\w	word character □ [a-zA-Z0-9_]
\d	digit □ [0-9]
\s	spaces□ spaces ad tabs and lines breaks
\W, \D, \S	άρνηση των παραπάνω

μέσα σε μια κλάση χαρακτήρων, (σχεδόν) όλοι οι ειδικοί χαρακτήρες χάνουν την ειδική σημασία τους. πχ: [{}*()/+*?|.] σημαίνει ένας χαρακτήρας από τους {}*()/+*?|.

21.5 Σύνολα Κανονικών Εκφράσεων:

Υπάρχουν διάφορα σύνολα κανονικών εκφράσεων (όχι σε όλα συμβατοί μεταξύ τους):

IEEE POSIX:

- Basic Regular Expressions (BRE)
- Extended Regular Expressions (ERE)

Perl (PCRE, PCRE2)

Και άλλοι νεότεροι (ανάλογα με την γλώσσα προγραμματισμού)

- πχ: η php είχε αναπτύξει δικό της μηχανισμό κανονικών εκφράσεων, αλλά πλέον καταργήθηκε και χρησιμοποιείται της perl (PCRE2) μέσω της βιβλιοθήκης preg

Στο BRE οι ειδικοί χαρακτήρες ?, +, {, |, (, and) χάνουν την ειδική τους σημασία. Για να αποκτήσουν ειδική σημασία χρειάζονται το «\»:

\?,\+, \{, \|, \(, and \).

ERE	BRE
?	\?
+	\+
{ }	\{ \}
( )	\(\)
|	|

21.6 Παράδειγμα χρήσης:

Σε μια φόρμα (ή σε οποιοδήποτε πρόγραμμα) ζητούμε από τον χρήστη να εισάγει αριθμό τηλεφώνου. Δεν θέλουμε όμως να πληκτρολογήσει spaces, παύλες, τελείες ή άλλους χαρακτήρες εκτός από αριθμούς. Επίσης, το μήκος τηλεφώνου πρέπει να είναι 10.

Για να γίνει αυτός ο έλεγχος μέσα από μια γλώσσα προγραμματισμού χωρίς την χρήση κανονικών εκφράσεων απαιτείται να γράψουμε πάνω από 10 γραμμές κώδικα.

Με χρήση κανονικών εκφράσεων αρκεί να ελέγξουμε ότι το string που έδωσε ο χρήστης ταιριάζει στην κανονική έκφραση:

- ^[0-9]{10}$
- ^[26][0-9]{9}$

Σε περίπτωση που θέλουμε να δεχόμαστε και την διεθνή μορφή τηλεφώνου, πχ: +302310999888, θα πρέπει απλά να επεκτείνουμε την κανονική έκφραση:

^[0-9]{10}$|^(\+[0-9]){1,3}{10}$

22. Διαχείριση αρχείων:

Σε αυτή την ενότητα θα δούμε πως μπορούμε να δημιουργήσουμε ένα αρχείο μέσα από ένα πρόγραμμα C και πως μπορούμε να γράψουμε σε αυτό ή να διαβάσουμε από ένα υπάρχων.

22.1 Writing files:

Αν τρέξουμε τον παρακάτω κώδικα:

```c
#include <stdio.h>

int main()
{
    // file opened in write mode
    FILE *pf = fopen("hello.txt", "w");
    // code that gets executed inside the file
    fprintf(pf, "Hello World");
    // file closed
    fclose(pf);
    return 0;
}
```

Παρατηρούμε ότι δημιουργεί ένα αρχείο hello.txt στον φάκελο που έχουμε ανοιχτό στο VScode και μέσα στο αρχείο υπάρχει το κείμενο Hello World. Πως το κάνει όμως αυτό ?

Η πρώτη γραμμή κώδικα είναι η εξής:

```c
FILE *pf = fopen("hello.txt", "w");
```

Βλέπουμε ότι δηλώνει μια μεταβλητή δείκτη με όνομα pf που είναι τύπου FILE.

Αυτός ο τύπος FILE είναι ένα struct της βιβλιοθήκης stdio.h και έχει για πεδία του μεταβλητές που περιγράφουν ένα αρχείο. Επομένως με την εντολή FILE *pf δημιουργούμε μια μεταβλητή δείκτη που μπορεί να

αποθηκεύσει το που βρίσκεται ένα αντικείμενο FILE στην μνήμη. Με την παρακάτω εντολή:

```
fopen("hello.txt", "w");
```

Δημιουργείται ένα αρχείο με όνομα "hello.txt" σε λειτουργία εγγραφής (writing mode), δηλαδή μπορούμε να γράψουμε σε αυτό. Το αρχείο βρίσκεται στον δίσκο αλλά η fopen το «ανοίγει» δηλαδή δημιουργεί ένα αντικείμενο τύπου FILE στην μνήμη που συμπληρώνει τα πεδία του FILE για το συγκεκριμένο αρχείο, στην συνέχεια επιστρέφεται η θέση αυτού του αντικειμένου στην μεταβλητή δείκτη pf.

Η επόμενη εντολή:

```
fprintf(pf, "Hello World");
```

«γράφει» την συμβολοσειρά Hello World μέσα στο αρχείο.

Και για τέλος πρέπει να κλείσουμε το αρχείο και αυτό γίνεται με την εντολή:

```
fclose(pf);
```

Αν ξανατρέξουμε το πρόγραμμα όπως είναι και απλά αλλάξουμε την εξής γραμμή:

```
fprintf(pf, "Hello Big World");
```

Τότε θα αντικατασταθεί το αρχείο που δημιουργήσαμε στην πρώτη εκτέλεση και έτσι πλέον αντί για Hello World θα γράφει Hello Big World. Αν θέλουμε να προσθέσουμε μια γραμμή κειμένου και να μην αντικαταστήσουμε τότε πρέπει να οριστεί το αρχείο σε λειτουργία append (append mode). Τότε ότι γράφουμε θα προστίθεται στο αρχείο:

```
FILE *pf = fopen("hello.txt", "a");
```

22.2 Διαγραφή αρχείου:

Για να διαγράψουμε ένα αρχείο αρκεί να γράψουμε την γραμμή:

```
remove("hello.txt");
```

Μπορούμε να εμπλουτίσουμε το πρόγραμμα μας με τον παρακάτω κώδικα:

```
if (remove("hello.txt") == 0)
{
    printf("File deleted successfully.\n");
}
else
{
    printf("Error deleting file.\n");
}
```

Έτσι πληροφορούμε και τον χρήστη αν διαγράφηκε το αρχείο. Προφανώς αν τρέξουμε το πρόγραμμα πάνω από μία φορά για ένα αρχείο ή το τρέξουμε για ένα αρχείο που δεν υπάρχει θα πάρουμε το μήνυμα λάθους ότι δεν διαγράφηκε.

22.3 Δημιουργία αρχείου με απόλυτη διαδρομή:

Στο πρώτο παράδειγμα δημιουργήσαμε ένα αρχείο στον φάκελο που ήδη βρίσκεται το πρότζεκτ μας απλά γράφοντας το όνομα του. Μπορούμε ωστόσο να δημιουργήσουμε ένα αρχείο οπουδήποτε στο σύστημα γράφοντας την απόλυτη διαδρομή του:

```
FILE *pf = fopen("C:/Users/NINTZ/Desktop/hello.txt",
"w");
```

Ή

```
FILE *pf = fopen("C:\\Users\\NINTZ\\Desktop\\hello.txt",
"w");
```

Αν θέλουμε να χρησιμοποιήσουμε backslash “\” τότε πρέπει να μπει διπλό ώστε να ακυρωθεί η ειδική σημασία του και να συμπεριφερθεί ως σύμβολο ή εναλλακτικά είναι εφικτή η χρήση απλού slash.

22.4 Ανάγνωση αρχείων (Reading Files):

Για να διαβάσουμε τα περιεχόμενα ενός αρχείου αλλάζουμε την πρώτη εντολή σε:

```
FILE *pf = fopen("C:\\Users\\NINTZ\\Desktop\\hello.txt",
"r");
```

Τώρα για να διαβάσουμε το περιεχόμενο πρέπει να δηλώσουμε έναν πίνακα χαρακτήρων με όσο μέγεθος θέλουμε και αυτό εκφράζει πόσους χαρακτήρες θα διαβάσει από την πρώτη γραμμή:

```
char buffer[255];
```

Για να διαβάσουμε τώρα την πρώτη γραμμή γράφουμε:

```
fgets(buffer, 255, pf);
```

Και αυτό θα διαβάσει την πρώτη γραμμή του αρχείου μέχρι 255 χαρακτήρες.

Μπορούμε να δούμε αυτή την γραμμή με την παρακάτω εντολή:

```
printf("%s", buffer);
```

Για να διαβάσουμε όλες τις γραμμές αρκεί να το βάλουμε σε μια δομή επανάληψης. Πόσες φορές θέλουμε όμως να επαναλαμβάνεται αυτή η ανάγνωση ? Η απάντηση είναι όσο υπάρχει κείμενο δηλαδή για όσο δεν υπάρχει κενό:

```
while (fgets(buffer, 255, pf) != NULL)
{
    printf("%s", buffer);
}
```

Ένα ακόμα κομμάτι κώδικα που μπορούμε να προσθέσουμε είναι να ελέγχουμε την ύπαρξη του αρχείου πριν επιχειρήσουμε να το διαβάσουμε:

```
if (pf == NULL)
{
```

```
    printf("File not found");
    return 0;
}
```

23. Σφάλματα και είδη μνήμης:

Συντακτικά, λογικά λάθη… και stack, heap… uint32_t, inline ….

For completeness, it is worth mentioning stack memory. Stack memory is a type of dynamic memory which is reserved for variables that are declared inside functions. Variables declared inside a function use stack memory rather than static memory.

When a function is called, stack memory is allocated for the variables in the function. When the function returns the stack memory is freed.

It is good to be aware of stack memory to be able to handle the memory usage of nested function calls and recursion. Recursion that repeats itself too many times may take up too much stack memory. When that happens it is called a **stack overflow**.

Πολύ απίθανο να γραφεί αυτό το κεφάλαιο.

24. Διαχείριση μνήμης:

Στην C, πρέπει να διαχειριστούμε τη μνήμη μόνοι μας. Είναι μια περίπλοκη εργασία, αλλά είναι επίσης αρκετά ισχυρή όταν χρησιμοποιείται σωστά. Η σωστή διαχείριση της μνήμης του υπολογιστή βελτιστοποιεί την απόδοση του προγράμματος, επομένως είναι χρήσιμο να γνωρίζετε πώς να απελευθερώνετε τη μνήμη όταν δεν απαιτείται πλέον και να χρησιμοποιείτε μόνο όσο χρειάζεται.

24.1 Δέσμευση μνήμης (memory allocation):

Δέσμευση / εκχώρηση μνήμης ονομάζεται η διαδικασία που καταλαμβάνουμε θέσεις από την μνήμη ram για αποθήκευση στοιχείων σχετικά με το πρόγραμμα μας. Αυτή η διαδικασία στην C μπορεί να επιτευχθεί με δύο τρόπους:

- Στατική δέσμευση (static allocation)
- Δυναμική δέσμευση (dynamic allocation)

24.1.1 Στατική δέσμευση (static allocation):

Με την στατική δέσμευση είμαστε ήδη γνώριμοι καθώς ότι μεταβλητή έχουμε δηλώσει μέχρι στιγμής ήταν χρησιμοποιώντας αυτή τη τεχνική. Λέγεται στατική διότι η μνήμη δεσμεύεται πριν το πρόγραμμα ξεκινήσει να εκτελεί τις εντολές. Είναι γνωστή και ως compile-time memory allocation. Με αυτή τη τεχνική δεσμεύονται θέσεις για όλες τις μεταβλητές του προγράμματος.

```
int students[20];
```

Για παράδειγμα ο παραπάνω πίνακας που είναι 20 θέσεων και αυτό δηλώνεται στην αρχή του προγράμματος οπότε δεσμεύονται και 80 bytes από την μνήμη. Έστω ότι ο πίνακας εκφράζει πόσοι μαθητές κατά μέσο όρο εκτιμάται ότι θα παρακολουθούν ένα μάθημα, αν στο τέλος του εξαμήνου γίνει απολογισμός και βρεθεί ότι το παρακολούθησαν 12 άτομα αυτό σημαίνει ότι 8 θέσεις δηλαδή 32 bytes δεν χρησιμοποιήθηκαν ποτέ, άρα τσάμπα εμείς τα «φάγαμε».

24.1.2 Δυναμική δέσμευση (dynamic allocation):

Για την λύση σε αυτό το πρόβλημα υπάρχει ο δυναμικός τρόπος δέσμευσης μνήμης. Με αυτή τη τεχνική μπορούμε να δεσμεύουμε και απελευθερώνουμε μνήμη και κατά την εκτέλεση του προγράμματος. Σημαντική σημείωση δεν είναι ο ένας ή ο άλλος τρόπος, μπορούν και οι δύο να εφαρμοστούν ταυτόχρονα στο πρόγραμμα απλά όχι για την ίδια

μεταβλητή. Αυτή είναι γνωστή και ως runtime memory allocation και έχουμε πλήρη έλεγχο από το πόση μνήμη δεσμεύουμε κάθε φορά. Για την χρήση της δυναμικής μνήμης απαιτείται η χρήση δεικτών (pointers) και των συναρτήσεων malloc() και calloc() που βρίσκονται στην βιβλιοθήκη stdlib.h την οποία θα πρέπει και να κάνουμε include.

malloc():

Πρόκειται για μια συνάρτηση που βρίσκεται στην βιβλιοθήκη <stdlib.h> και μας επιτρέπει να δεσμεύουμε μνήμη από την σωρό (heap) της RAM. Συντάσσεται με τον εξής τρόπο:

```
malloc(size);
```

Όπου για size τοποθετούμε έναν ακέραιο πχ 16, και αυτό εκφράζει τον αριθμό bytes που θα δεσμεύσει.

Συνήθως ορίζουμε μια μεταβλητή δείκτη να ισούται με την παραπάνω έκφραση:

```
int *ptr1 = malloc(16);
```

Η malloc δεσμεύει στην σωρό 16 bytes και επιστρέφει την διεύθυνση του πρώτου byte. Απλά επειδή η διεύθυνση που επιστρέφει είναι υπό της μορφής void* πρέπει να την κάνουμε cast στον τύπο που θέλουμε, στην προκειμένη περίπτωση int. Ο λόγος που επιστρέφει την διεύθυνση με τύπο void* είναι για να μπορούμε να την κάνουμε cast σε ότι θέλουμε, να μην υπάρχει δηλαδή κάποιος περιορισμός. Άρα μια πλήρως σωστή δήλωση είναι:

```
int *ptr1 = (int *)malloc(16);
```

Τι μπορούμε να κάνουμε αυτή τη μνήμη που δεσμεύσαμε ? Μπορούμε να χρησιμοποιήσουμε τα 16 bytes για να δηλώσουμε 4 ακέραιους (16 bytes συνολικά / 4 bytes ανά ακέραιο = 4 ακέραιες μεταβλητές), αλλά αυτό είναι κάτι που θα δούμε σε λίγο.

Παράδειγμα:

Το πιο απλό παράδειγμα είναι να δεσμεύσουμε δυναμικά μια ακέραια μεταβλητή, να της δώσουμε μια τιμή και να την εκτυπώσουμε:

```c
#include <stdio.h>
#include <stdlib.h> // Required for malloc and free

int main()
{
    // Step 1: Allocate memory for an integer
    int *ptr = (int *)malloc(sizeof(int));
    // Check if memory allocation was successful
    if (ptr == NULL)
    {
        printf("Memory allocation failed\n");
         return 1; // Return non-zero value to indicate
failure
    }
    // Step 2: Assign a value to the allocated memory
    *ptr = 10;
    // Step 3: Print the value
    printf("%d\n", *ptr);
    // Step 4: Free the allocated memory
    free(ptr);
    return 0;
}
```

Το παραπάνω πρόγραμμα κάνει ακριβώς αυτό.

Δηλώνουμε στην σωρό όσα bytes καταλαμβάνει ένα int (4 bytes) και αναθέτουμε αυτή τη μνήμη σε έναν δείκτη, στην συνέχεια έχουμε έναν έλεγχο στον δείκτη να σιγουρευτούμε ότι η παραπάνω διεργασία έγινε σωστά. Μετά κάνοντας dereference τον δείκτη αναθέτουμε στην θέση

που δείχνει την τιμή 10. Υπάρχει δηλαδή ένα κουτάκι μνήμης στην σωρό που έχει μέσα την τιμή 10. Εκτυπώνουμε την τιμή αυτή και επειδή τελειώσαμε με ότι την χρειαζόμασταν γράφουμε free(ptr) ώστε να αποδεσμευτεί η μνήμη.

Αν θέλαμε 4 ακέραιες μεταβλητές τότε θα μπορούσαμε είτε όπως πριν να δώσουμε για όρισμα 16 είτε θα μπορούσαμε να γράψουμε:

```
int *ptr = (int *)malloc(4 * sizeof(int));
```

και είναι ακριβώς το ίδιο.

Άρα έχουμε 4 θέσεις στην μνήμη heap που κάθε μια δέχεται έναν ακέραιο, πως βάζουμε τιμές ?

Συμπεριφερόμαστε στον δείκτη σαν να είναι πίνακας, δηλαδή:

```
ptr[1] = 10;
ptr[2] = 20;
ptr[3] = 30;
```

Και μπορούμε να τις εκτυπώσουμε κανονικά:

```
printf("Value at index 1: %d\n", ptr[1]);
printf("Value at index 2: %d\n", ptr[2]);
printf("Value at index 3: %d\n", ptr[3]);
```

Επίτηδες δεν έχω βάλει τιμή και στο πρώτο κελί, τι θα γίνει αν προσπαθήσουμε να το εκτυπώσουμε ?

```
printf("Value at index 0: %d\n", ptr[0]);
```

Το αποτέλεσμα που παίρνουμε είναι το εξής:

```
Value at index 0: -1872403040
```

Αυτό είναι μια τιμή «ξεχασμένη» που υπήρχε στην σωρό, εμείς δεσμεύσαμε την θέση της αλλά δεν της βάλαμε κάποια τιμή οπότε πήραμε ότι υπήρχε αρχικά στην θέση. Αυτό είναι το βασικό χαρακτηριστικό της

malloc(), ότι δηλαδή απλά δεσμεύει θέσεις στην μνήμη, δεν τις αρχικοποιεί κιόλας.

calloc():

Η άλλη συνάρτηση που αναφέραμε η calloc() κάνει ακριβώς αυτό, δεσμεύει και αρχικοποιεί τις θέσεις που πήρε στο 0. Η σύνταξη της είναι:

```
calloc(amount, size);
```

Παίρνει δύο ορίσματα, το πρώτο είναι ο αριθμός των στοιχείων που θέλουμε και το δεύτερο το μέγεθος. Έτσι εμείς που θέλαμε 4 ακεραίους θα γράφαμε:

```
int *ptr = (int *)calloc(4, sizeof(int));
```

Και όταν πάμε να εκτυπώσουμε ένα κελί στο οποίο δεν αναθέσαμε τιμή παίρνουμε:

```
Value at index 0: 0
```

24.2 Ανακατανομή μνήμης (Reallocate Memory):

Σε περίπτωση που δεν μας φτάνει η μνήμη που έχουμε δεσμεύσει μπορούμε να την ανακατανέμουμε για να την κάνουμε μεγαλύτερη. Σε αυτή την διαδικασία δεσμεύεται ένα νέο, μεγαλύτερο, κομμάτι της σωρού χωρίς να χαθούν ότι δεδομένα είχαμε σε αυτό που δεν μας έκανε. Χρησιμοποιείται η realloc() συνάρτηση, που δομείται έτσι:

```
int *ptr2 = realloc(ptr1, size);
```

Όπου:

- Το πρώτο όρισμα είναι ο δείκτης της μνήμης την οποία επιθυμούμε να αυξήσουμε.
- Το δεύτερο όρισμα είναι το νέο μέγεθος μνήμης που επιθυμούμε να δεσμεύσουμε.

Η συνάρτηση realloc() προσπαθεί να αλλάξει το μέγεθος της μνήμης που έχει δεσμευτεί από τον δείκτη ptr1 και να επιστρέψει την ίδια διεύθυνση μνήμης. Εάν δεν μπορεί να αλλάξει το μέγεθος της μνήμης στην τρέχουσα διεύθυνση, τότε θα εκχωρήσει νέα μνήμη σε διαφορετική διεύθυνση και θα επιστρέψει αυτή τη νέα διεύθυνση.

Όταν η συνάρτηση realloc() επιστρέφει διαφορετική διεύθυνση μνήμης, η μνήμη στην αρχική διεύθυνση δεν είναι πλέον δεσμευμένη και δεν είναι ασφαλής για χρήση. Αυτό σημαίνει ότι η παλιά διεύθυνση μνήμης που δείχνει ο αρχικός δείκτης δεν μπορεί να χρησιμοποιηθεί ξανά, καθώς η μνήμη αυτή έχει απελευθερωθεί ή επαναχρησιμοποιηθεί από το σύστημα.

Για να αποφευχθούν πιθανά σφάλματα και ανεπιθύμητη πρόσβαση σε μη έγκυρη μνήμη, συνιστάται να αντιστοιχίσετε τον νέο δείκτη που επιστρέφει η realloc() στην αρχική μεταβλητή δείκτη. Με αυτόν τον τρόπο, ο παλιός δείκτης δεν θα παραμείνει διαθέσιμος και δεν θα υπάρχει κίνδυνος να χρησιμοποιηθεί κατά λάθος. Η ενέργεια αυτή εξασφαλίζει ότι ο δείκτης θα δείχνει πάντα σε έγκυρη και σωστά δεσμευμένη μνήμη.

```
ptr1 = realloc(ptr1, size);
```

Με αυτήν την ανάθεση, η ptr1 δείχνει πλέον στη νέα διεύθυνση μνήμης που επιστρέφει η realloc(), και η παλιά διεύθυνση μνήμης δεν είναι πλέον προσβάσιμη.

24.3 Αποδέσμευση μνήμης (Deallocate Memory):

Όπως προαναφέρθηκε καλό είναι αφού τελειώσουμε να χρησιμοποιούμε μια μεταβλητή που δηλώθηκε δυναμικά, να αποδεσμεύσουμε αυτή τη μνήμη ώστε να μπορεί να τη χρησιμοποιήσει ο υπολογιστής ή κάποιο άλλο πρόγραμμα, μια ακόμη καλή συνήθεια είναι αφού το κάνουμε αυτό για ασφάλεια να θέσουμε και τον δείκτη σε null για να μη τυχόν τον χρησιμοποιήσουμε.

```
free(ptr);
```

```
ptr = NULL;
```

24.4 Διαρροή μνήμης (Memory Leaks):

Διαρροή μνήμης λέμε ότι έχουμε όταν δεσμεύουμε δυναμικά μνήμη και δεν την αποδεσμεύουμε. Αν αυτό συμβεί σε μια δομή επανάληψης ή συνάρτησης που καλείται συχνά τότε το πρόγραμμα θα καταλήξει να χρησιμοποιεί αχρείαστα πολύ μνήμη.

Ένας άλλος κίνδυνος είναι να «χαθεί» ο δείκτης πριν μπορέσουμε να αποδεσμεύσουμε την μνήμη που δείχνει. Για παράδειγμα αν είχαμε κάτι τέτοιο:

```
int x = 5;
int *ptr;
ptr = calloc(2, sizeof(*ptr));
ptr = &x;
```

Αλλάξαμε καταλάθος το που δείχνει ο δείκτης πριν αποδεσμεύσουμε την μνήμη που έδειχνε άρα τώρα «κλείδωσε» και δεν μπορούμε να την αποδεσμεύσουμε.

Ένα άλλο παράδειγμα είναι αυτό:

```
int *ptr;
ptr = malloc(sizeof(*ptr));
ptr = realloc(ptr, 2 * sizeof(*ptr));
```

Αν αποτύχει η ανακατανομή μνήμης τότε επιστρέφει NULL στον δείκτη και έτσι χάνεται πάλι η μνήμη. Γιαυτό είναι αναγκαίο πάντα να κάνουμε τον έλεγχο που δείξαμε στο πρώτο κανονικό πρόγραμμα του κεφαλαίου, να μην ξεχνάμε να αποδεσμεύουμε την μνήμη και να βάζουμε τον δείκτη σε NULL αφού την αποδεσμεύσουμε.

25. Χρήση κανονικών εκφράσεων στην C:

25.1 Πως εκτελείται ένα πρόγραμμα σε βάθος:

Linker

25.2 Πρόβλημα σύνδεσης (linking error):

Για αρχή απλά μας νοιάζει να μπορεί να εκτελεστεί ο παρακάτω κώδικας χωρίς να αναλύσουμε τι κάνει:

```
#include <regex.h>
#include <stdio.h>

int main()
{
    // Variable to create regex
    regex_t reegex;
    // Variable to store the return value after creation
of regex
    int value;
    // Function call to create regex
    value = regcomp(&reegex, "[:word:]", 0);
    // If compilation is successful
    if (value == 0)
    {
        printf("RegEx compiled successfully.");
    }
    // Else for Compilation error
    else
    {
        printf("Compilation error.");
    }
    return 0;
```

```
}
```

Αν επιχειρήσουμε να τρέξουμε τον παραπάνω κώδικα με τον τυπικό τρόπο, δηλαδή να πατήσουμε απλά το κουμπί run θα παρατηρήσουμε ότι δεν τρέχει (αν σε κάποιον τρέχει τότε αγνοήστε τα παρακάτω).

Προσωπικά το μήνυμα που μου βγάζει είναι:

```
undefined reference to `regcomp'
collect2.exe: error: ld returned 1 exit status
```

Αυτό συμβαίνει διότι αν και η βιβλιοθήκη regex.h γίνεται compile δεν γίνεται link με τον κώδικα μας και έτσι δεν μπορεί να αναγνωριστεί η συνάρτηση regcomp. Η λύση σε αυτό το πρόβλημα είναι να τοποθετήσουμε μια ειδική εντολή για τον linker και να του πούμε να κάνει link την βιβλιοθήκη. Αυτή η ειδική εντολή στον linker είναι γνωστή και ως linker flag. Για compile που γίνεται σε τερματικό αρκεί μετά το «gcc HelloWorld.c» να βάλουμε την εντολή «-lregex» (εικάζω το l συμβολίζει το link και μετά ακολουθεί το όνομα της βιβλιοθήκης). Κάνοντας αυτό το πρόβλημα λύνεται και θα πρέπει κανονικά όταν τρέχουμε το πρόγραμμα να εμφανίζεται το μήνυμα `RegEx compiled successfully.`

Στο visual studio code όμως που προτάθηκε για χρήση στην αρχή του οδηγού τα πράγματα δεν είναι τόσο απλά. Αφού δοκίμασα με πάρα πολλούς διαφορετικούς τρόπους δεν κατάφερα να έχω σταθερά αποτελέσματα, δηλαδή μια έτρεχε μια όχι επομένως κατέληξα στην εξής μπακάλικη μεθοδολογία. Αν ακολουθείτε τον οδηγό από την αρχή τότε έχετε βάλει την ρύθμιση στο code runner να εκτελεί το πρόγραμμα σε terminal, στην πραγματικότητα το vscode από ότι καταλαβαίνω ανοίγει ένα powershell και τρέχει εκεί μια εντολή που εντοπίζει το προτζεκτ, το κάνει compile και το εκτελεί. Αυτή η εντολή φαίνεται πριν την εκτέλεση του προγράμματος και για εμένα είναι η εξής:

```
cd "c:\Users\NINTZ\Desktop\Learning C\" ; if ($?) { gcc
hellowolrd.c -o hellowolrd } ; if ($?) { .\hellowolrd }
```

Η λύση στην οποία κατέληξα είναι απλά να τροποποιήσουμε το μεσαίο κομμάτι τις εντολής και να προσθέσουμε την εντολή -lregex:

```
cd "c:\Users\NINTZ\Desktop\Learning C\" ; if ($?) { gcc
hellowolrd.c -o hellowolrd -lregex} ; if ($?) { .\hellowolrd }
```

Εκτελώντας τώρα το πρόγραμμα πρέπει να παίρνουμε το εξής μήνυμα:

```
RegEx compiled successfully.
```

25.3 Το πρώτο μας λειτουργικό πρόγραμμα με regex:

τεστ

26. map:

Εξίσου απίθανο να γραφεί, ίσως στη δεύτερη έκδοση…

to-do:

- κλασική δήλωση main με ορίσματα
- κεφάλαιο 23, 25.3+, 26.

27. References

[1] [Online]. Available: https://en.wikipedia.org/wiki/C_(programming_language).

[2] [Online]. Available: https://en.wikipedia.org/wiki/C%2B%2B.

[3] [Online]. Available: https://en.wikipedia.org/wiki/Text_editor.

[4] [Online]. Available: https://en.wikipedia.org/wiki/Source-code_editor.

[5] [Online]. Available: https://en.wikipedia.org/wiki/Integrated_development_environment.

[6] [Online]. Available: https://en.cppreference.com/w/c/language/arithmetic_types.