Σε αυτήν την σειρά άρθρων θα μιλήσω για συναρτήσεις στην assembly και πώς αυτές υλοποιούνται. Επειδή είναι αρκετά εξειδικευμένο θέμα, απαιτείται από τον αναγνώστη να έχει γνώση του πώς λειτουργούν οι συναρτήσεις/μέθοδοι σε τουλάχιστον μία γλώσσα προγραμματισμού (δηλαδή γνώση εννοιών όπως παράμετροι συναρτήσεων, επιστροφή από συναρτήσεις, σκοπιές, κλήση συναρτήσεων, τοπικές μεταβλητές κτλ). Οποιεσδήποτε γνώσεις σε assembly (εισαγωγικές και μη) δεν κάνουν κακό ωστόσο, διότι αυτά τα άρθρα δεν αποτελούν εισαγωγή σε assembly. Απλά ακουμπάνε το θέμα της υλοποίησης ενός είδους συναρτήσεων σε αυτήν.

Επίσης, σε αυτά τα άρθρα θα χρησιμοποιήσω σύνταξη AT&T (unix style) και μπορεί να κάνω χρήση κλήσεων συστήματος linux (για να αποφύγω την στάνταρ βιβλιοθήκη της C), άρα ο κώδικας είναι σίγουρο πως θα δουλέψει σε διανομές linux. Χρησιμοποιώ την οικογένεια από assemblers, GNU AS, κατά την οποία αποθηκεύουμε ένα αρχείο κώδικα assembly με κατάληξη .s ή .S, κάνουμε assemble με την εντολή as και linking με την εντολή ld. Έτσι αν αποθηκεύσουμε ένα αρχείο κώδικα assembly με το όνομα test.s, τότε με τις παρακάτω εντολές μπορούμε να το μετατρέψουμε σε εκτελέσιμο:

```
as test.s -o test.o
ld test.o -o test
```

Η σύνταξη AT&T χρησιμοποιείται από τον GCC όταν κάνει assemble κώδικες C/C++. Ο GCC σας επιτρέπει να μεταφράσετε κώδικες C/C++ σε κώδικα assembly, χωρίς να χρειαστεί να κάνετε εκτελέσιμο, απλά με το όρισμα -S:

```
gcc -S main.c
```

ή

```
g++ -S main.cpp
```

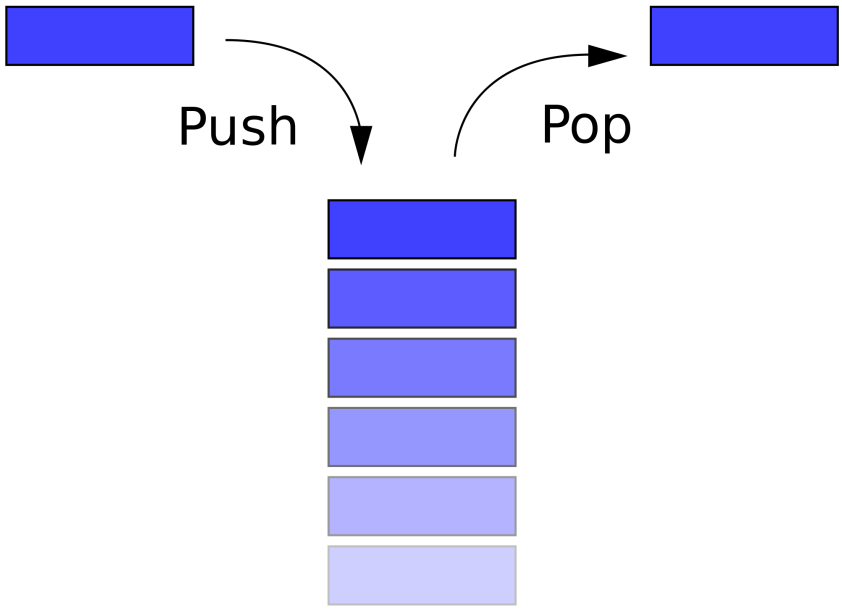
Πολύ χρήσιμο αν θέλετε να δείτε πώς μετατρέπονται διάφορα προγράμματά σας υψηλού επιπέδου, σε assembly.

Στην assembly δεν υπάρχει συγκεκριμένος τρόπος υλοποίησης συναρτήσεων. Δεν υπάρχει κάποιο στάνταρ δηλαδή. Έτσι κάθε γλώσσα μπορεί να ορίζει την δικιά της σύμβαση με την οποία υλοποιεί συναρτήσεις σε επίπεδο γλώσσας μηχανής. Οι συμβάσεις αυτές ονομάζονται calling conventions. Ένα calling convention ορίζει το πώς μεταφέρονται οι παράμετροι και οι τοπικές μεταβλητές στην συνάρτηση, καθώς και το πώς επιστρέφουν τιμές από αυτήν. Αφού δεν υπάρχει στάνταρ calling convention θα μπορούσαμε να φτιάξουμε και εμείς ένα, αν νιώθουμε αρκετά γενναίοι. Παρ’ ολ’ αυτά, σε αυτήν την σειρά άρθρων θα χρησιμοποιήσουμε ένα ήδη υπάρχον calling convention που χρησιμοποιείται κατά κύριο λόγο στην γλώσσα προγραμματισμού C.

Το calling convention στο οποίο αναφέρομαι, ονομάζεται cdecl. Στην C βέβαια μπορούμε να χρησιμοποιήσουμε διάφορα άλλα calling conventions, γι' αυτό το cdecl δεν είναι το μοναδικό calling convention που απαντάται στην γλώσσα αυτή. Το cdecl ορίζεται κυρίως για συναρτήσεις της στανταρ βιβλιοθήκης της C, καθώς και για συναρτήσεις που δημιουργούμε εμείς μέσα στην γλώσσα, χωρίς να ορίζουμε άμεσα αν θέλουμε η συνάρτηση να ακολουθεί διαφορετικό calling convention. Το προεπιλεγμένο calling convention για συναρτήσεις στην C, με άλλα λόγια, είναι το cdecl, το οποίο και θα αναλύσουμε παρακάτω.

Οι συναρτήσεις με βάση το cdecl, διαμορφώνονται γύρω από μία πολύ βασική έννοια που μπορεί ο αναγνώστης να έχει ξανακούσει: την στοίβα. Η στοίβα είναι το κύριο υλικό από το οποίο είναι «χτισμένες» οι συναρτήσεις, καθώς και τα προγράμματα τα ίδια, στο χαμηλότερο επίπεδο προγραμματισμού ενός υπολογιστή. Μα τι είναι η στοίβα και για πιο σκοπό χρησιμοποιείται;

Με αντιγραφή από την σειρά μου με τα memory leaks:
"Η στόιβα (stack) είναι μία δομή δεδομένων, που εκτελεί δύο λειτουργίες. Push και pop:



Τα δεδομένα είναι τα μπλε κουτιά σε αυτήν την εικόνα. Όταν κάνουμε push στην στοίβα, ένα δεδομένο μεγέθους x bytes προστίθεται πάνω πάνω. Όταν κάνουμε pop, αφαιρούμε το τελευταίο δεδομένο που προσθέσαμε στην στοίβα. Αυτή είναι μία στοίβα τύπου LIFO (last-in, first-out)."

Στην assembly, η στοίβα είναι μία περιοχή της μνήμης που θυμίζει την παραπάνω δομή δεδομένων και κάνει ακριβώς αυτές τις δύο λειτουργίες που προανέφερα. Στην σύνταξη AT&T έχουμε τα instructions, `pushl` και `popl` τα οποία εκτελούν αυτές τις λειτουργίες. Το l στο τέλος των instructions συμβολίζει τον τύπο δεδομένων (δηλαδή το μέγεθος της μνήμης) που θέλουμε να κάνουμε push ή pop, και βγαίνει από τη λέξη

long. Στην assembly x86, ο τύπος δεδομένων long έχει μήκος 32 bit (4 byte). Έτσι όταν λέμε `popl` εννοούμε πως θέλουμε να κάνουμε pop τα πρώτα 4 byte από την κορυφή της στοίβας.

Σε αρκετά instructions που χρησιμοποιούμε μέσω της σύνταξης AT&T μπορούμε να διαφοροποιούμε τον τύπο δεδομένων που χρησιμοποιούμε. Οι εντολές `pushl`/`popl` δεν υποστηρίζουν κάτι διαφορετικό πέρα από long, αλλά για να κάνουμε mov 1 byte θα χρησιμοποιούμε την εντολή `movb` για παράδειγμα. Αντίστοιχα για να κάνουμε mov δεδομένα μήκους 4 byte θα χρησιμοποιούσαμε την εντολή `movl`.

Τα `pushl`/`popl` δέχονται ένα και μοναδικό operand, το καθένα. Το `pushl` δέχεται την τιμή που θέλουμε να «σπρώξουμε» στην κορυφή της στοίβας, και το `popl` δέχεται έναν register στον οποίο θα αποθηκευτεί η τιμή που αφαιρέθηκε από την κορυφή της στοίβας. Σκεφτείτε το σαν μια στοίβα από πιάτα στην οποία μπορούμε είτε να βάζουμε πιάτα στην κορυφή είτε να τα βγάζουμε.

Αξίζει να σημειωθεί, πως η στοίβα στις αρχιτεκτονικές x86, δεν υλοποιείται από κάτω προς τα πάνω αλλά από πάνω προς τα κάτω. Με άλλα λόγια, η στοίβα μεγαλώνει από τις υψηλότερες θέσεις μνήμης προς τις χαμηλότερες. Όταν βάζουμε μία τιμή πάνω στην στοίβα με την εντολή `pushl`, τότε η κορυφή της στοίβας μετακινείται 4 byte προς τα κάτω. Αντίστοιχα όταν κάνουμε `popl`, η κορυφή της στοίβας μετακινείται 4 byte προς τα πάνω. Για να δούμε σε ποια θέση μνήμης βρίσκεται η στοίβα κάθε φορά, από το πρόγραμμά μας, μπορούμε να χρησιμοποιήσουμε τον ειδικό register esp (από το stack pointer). Ο esp αποθηκεύει την θέση μνήμης της κορυφής της στοίβας, αυτόματα, μετά από λειτουργίες του τύπου `pushl`/`popl`.

Για παράδειγμα ας πούμε πως σε αυτό το μοντέλο μνήμης κάθε θέση μνήμης είναι 1 byte (δηλαδή η απόσταση από την θέση μνήμης 0xcdddfef1 μέχρι την 0xcdddfef2 είναι 8 bit ακριβώς). Τότε αν η κορυφή της στοίβας βρίσκεται στην τυχαία θέση μνήμης 0xcdddfef8, μία αναπαράσταση της μνήμης θα ήταν κάπως έτσι:

```
0x00000000
.
.
0xcdddfef1
0xcdddfef2
0xcdddfef3
0xcdddfef4
0xcdddfef5
0xcdddfef6
0xcdddfef7
0xcdddfef8 <- %esp
.
.
0xffffffff
```

Με μία λειτουργία `pushl` θα αφαιρεθούν 4 bytes από την θέση μνήμης της κορυφής στοίβας (δηλαδή από τον esp register) και τα προηγούμενα bytes θα πάρουν την τιμή που κάναμε `pushl`. Τρέχουμε το instruction `pushl $FFFFF18C` για να τοποθετήσουμε τον δεκαεξαδικό αριθμό FFFFF18C στην κορυφή της στοίβας και παίρνουμε:

```
0x00000000
.
.
0xcdddfef1
0xcdddfef2
0xcdddfef3
0xcdddfef4 = 0010 1001 <- %esp
0xcdddfef5 = 1100 0000
0xcdddfef6 = 0001 1000
0xcdddfef7 = 0000 1111
0xcdddfef8
.
.
0xffffffff
```

Ο δεκαεξαδικός αριθμός FFFFF18C σε δυαδική αναπαράσταση των 4 byte (32 bit) είναι 00001111 00011000 11000000 00101001. Οι επεξεργαστές Intel όμως, ακολουθούν αρχιτεκτονική little endian που σημαίνει πως το λιγότερο σημαντικό byte αποθηκεύεται πρώτο. Έτσι πάνω βλέπουμε πως το byte 00101001 είναι στην χαμηλότερη θέση μνήμης και στην συνέχεια ακολουθούν όλα τα άλλα bytes ανάποδα. Αρχιτεκτονικές στις οποίες τα δεδομένα αποθηκεύονται στην μνήμη, με το πιο σημαντικό byte στην χαμηλότερη θέση μνήμης, ονομάζονται big endian. Σε αρχιτεκτονικές big endian για παράδειγμα, η παραπάνω αναπαράσταση μνήμης θα έμοιαζε κάπως έτσι:

```
0x00000000
.
.
0xcdddfef1
0xcdddfef2
0xcdddfef3
0xcdddfef4 = 0000 1111 <- %esp
0xcdddfef5 = 0001 1000
0xcdddfef6 = 1100 0000
0xcdddfef7 = 0010 1001
0xcdddfef8
.
.
0xffffffff
```

Πέρα από αυτήν την μικρή παρένθεση, ας δούμε τι θα έκανε το instruction `popl` σε αυτήν την περίπτωση. Αν τρέξουμε την εντολή `popl`, ο stack pointer (δηλαδή η κορυφή της στοίβας) θα ανέβει 4 byte και θα αφαιρεθούν τα δεδομένα τα οποία προσθέσαμε με την `pushl`. Αυτό σημαίνει πως ο υπολογιστής είναι ελεύθερος να χρησιμοποιήσει τις παραπάνω θέσεις μνήμης που πλέον δεν ανήκουν στην στοίβα, όπως αυτός θελήσει, άρα δεν θα πρέπει να βασιζόμαστε πλέον σε αυτές. Ωστόσο η εντολή `popl` παίρνει σαν operand έναν register στον οποίο μπορούμε να αποθηκεύσουμε προσωρινά τα δεδομένα που μόλις κάναμε pop.

Εννοείται πως αφού είμαστε σε x86 assembly μπορούμε κάθε φορά να κάνουμε push και pop δεδομένα μήκους μόνο 4 byte (32 bit). Άρα και τα δεδομένα που θα αφαιρέσουμε από την κορυφή της στοίβας μας, δεν μπορούν να είναι περισσότερα από 4 bytes κάθε φορά. Ας τρέξουμε το `popl %eax` να δούμε τι θα γίνει:

```
0x00000000
.
.
0xcdddfef1
0xcdddfef2
0xcdddfef3
0xcdddfef4 = 0000 1111
0xcdddfef5 = 0001 1000
0xcdddfef6 = 1100 0000
0xcdddfef7 = 0010 1001
0xcdddfef8 <- %esp
.
.
0xffffffff
```

Βλέπουμε πως η κορυφή της στοίβας μας, αυξήθηκε κατά 4 bytes και πλέον δείχνει στην θέση μνήμης 0xcdddfef8. Η

```
popl %eax
```

είναι η αντίστοιχη των δύο instructions

```
movl (%esp), %eax
addl $4, %esp
```

Δεν έχουμε καμία εγγύηση για το τι συνέβη στις θέσεις μνήμης που βρίσκονταν στην προηγούμενη κορυφή της στοίβας, καθώς αφαιρέθηκαν από αυτή. Η στοίβα δρα κατά κάποιον τρόπο σαν προστάτης των δεδομένων μας στην μνήμη. Δεν επιτρέπει στον υπολογιστή να πειράξει τα δεδομένα που έχουμε κάνει push σε αυτήν, καθώς αυτά τα δεδομένα ανήκουν πλέον στο πρόγραμμά μας.

Πολλές φορές, δεν μας χρειάζονται οι τιμές που αφαιρούμε από την κορυφή της στοίβας, αλλά αφού το instruction `popl` θέλει αναγκαστικά ένα operand, εμείς πολύ απλά μπορούμε να του δώσουμε έναν register που δεν χρησιμοποιούμε. Εδώ στο παράδειγμα χρησιμοποιήσαμε τον register eax. Μετά την εντολή `popl %eax`, μέσα στον eax θα βρίσκεται η τιμή FFFFF18C.

Κλείνοντας, η στοίβα στην assembly μας δίνει την δυνατότητα να δημιουργούμε stack frames. Με τα stack frames μπορούμε να δημιουργούμε σκοπιές, μέσα στις οποίες υπάρχουν τοπικές μεταβλητές οι οποίες καταστρέφονται αφού το πρόγραμμα βγει από την σκοπιά. Επίσης, στην υλοποίηση συναρτήσεων, βάζουμε και έξτρα πληροφορίες στην στοίβα, όπως είναι η θέση μνήμης που θα επιστρέψει η συνάρτηση για να συνεχιστεί η εκτέλεση του προγράμματος, από το σημείο που έγινε η κλήση. Στα επόμενα άρθρα θα μιλήσω περισσότερο για το cdecl και θα δείξω παράδειγμα με συνάρτηση σε assembly.