ΕΙΣΑΓΩΓΗ ΣΤΟ ΔΟΜΗΜΕΝΟ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟ

Μερικές χρήσιμες οδηγίες

Ιωάννινα-Κοζάνη, Ιανουάριος 2008.

Α. Στάδια Ανάπτυξης

Γενικά, διακρίνουμε τη φάση της ανάλυσης, του σχεδιασμού και της υλοποίησης. Για το κάθε ένα από αυτά υπάρχει πολλή θεωρία και "σχολές" για το πώς πρέπει να γίνονται όταν πρόκειται για κάποιο μεγάλο project. Εδώ θα περιοριστούμε βέβαια, σε απλές καιπρακτικές συμβουλές.

Ανάλυση

Πριν προχωρήσουμε στην υλοποίηση ενός προγράμματος για την επίλυση συγκεκριμένου προβλήματος, αναλύουμε αυτό το πρόβλημα. Όσο καλύτερη η ανάλυση τόσο περισσότερο κόπο γλιτώνουμε όταν αναπτύσσουμε και αργότερα αναβαθμίζουμε το πρόγραμμά μας.

Σχεδιασμός

Όταν σχεδιάζουμε ένα πρόγραμμα είναι καλό να αναρωτηθούμε: αν χρειαστεί να κάνω κάτι ελαφρά διαφορετικό από αυτό που θέλω τώρα, πόσο εύκολα θα μπορέσω να τροποποιήσω το πρόγραμμά μου για να το πετύχω αυτό; Θα μπορώ να το προσαρμόσω με όσο το δυνατό πιο λίγες αλλαγές σε όσο το δυνατό πιο λίγα σημεία; Ή μήπως θα χρειαστεί να το ξαναγράψω από την αρχή; Πρακτικά, αυτό επιτυγχάνεται με το να επιδιώκουμε να βρούμε τουλάχιστον δύο τρόπους για την υλοποίηση μιας ενέργειας και να διαλέγουμε αυτόν που έχει λιγότερα στοιχεία τα οποία ενδέχεται να αλλάξουν στο μέλλον.

Υλοποίηση

Αυτό είναι πολύ απλά ο καθαυτό προγραμματισμός. Σε αυτό το σημείο θα πρέπει να πούμε ότι είναι χρήσιμο να "σπάζουμε" το πρόγραμμά μας σε επιμέρους αρχεία με συναρτήσεις που μπορεί να χρησιμοποιήσουμε και σε άλλα έργα. Όσο μεγαλώνει ένα project και γίνεται πιο πολύπλοκο, με περισσότερα ιεραρχικά επίπεδα, αυξάνεται η ανάγκη να χρησιμοποιούμε αρχεία επικεφαλίδας για να περιλαμβάνουμε συναρτήσεις και άλλα στοιχεία που έχουμε ορίσει, όπου χρειάζεται.

Σε ένα αρχείο επικεφαλίδας μπορώ να περιλάβω τα εξής:

- ορισμούς συμβολικών σταθερών
- άλλες οδηγίες #include
- δηλώσεις δομών και ενώσεων δεδομένων που θα μου χρειαστούν στις συναρτήσεις μου
- δηλώσεις τύπων απαρίθμησης
- δηλώσεις νέων τύπων (απλών και παραγώγων) με struct και typedef
- δηλώσεις πρωτότυπων συναρτήσεων
- εξωτερικές μεταβλητές

  κλπ.

Τα αρχεία επικεφαλίδας είναι καλό να έχουν και το κατάλληλο "περιτύλιγμα" που τα προφυλάσσει από πολλαπλή συμπερίληψη:

```
#ifndef MYHEADER_H
#define MYHEADER_H
... (περιεχόμενα του αρχείου) ...
#endif
```

Όσο πιο μεγάλο το πρόγραμμα τόσο πιο χρήσιμη είναι αυτή η πρακτική.

Μερικά πρακτικά θέματα

Κρατάμε ένα ημερολόγιο της ανάπτυξης του λογισμικού. Επίσης, καθώς αναπτύσσουμε τις συναρτήσεις μας, καλό είναι να γράφουμε κάπου πώς λειτουργούν και τι κάνουν αυτές, είτε για προσωπική μας χρήση είτε για να βοηθήσουμε όποιον άλλον βασιστεί στο πρόγραμμά μας.

Όταν θέλουμε να κάνουμε κάτι πολυσύνθετο, γράφουτε στο χαρτί ή στον ΗΥ με λόγια ένα-ένα τα βήματα που θα κάνουμε 1) πρώτα αυτό 2) μετά εκείνο και τα ακολουθούμε πιστά.

Κρατάμε αντίγραφα ασφαλείας του κώδικα που αναπτύσσετε! Δεν είναι αδύνατο να σβήσουμε

κατά λάθος το αποτέλεσμα δουλειάς πολλών ημερών!
Η C μας δίνει μεγάλη ελευθερία στο στυλ γραφής. Δεν πρέπει να κάνουμε κακή χρήση αυτής της ελευθερίας. Να υιοθετήσουμε ένα ευανάγνωστο ύφος και με αυτό να γράφουμε πάντα τα δικά μας προγράμματα. Επίσης, προσθέτουμε διευκρινιστικά σχόλια σε διάφορα σημεία.
Σαν ένας πρακτικός ορισμός μπορεί να ειπωθεί ότι το ιδανικό δομημένο πρόγραμμα είναι αυτό που μπορούμε να το διαβάσουμε από την αρχή μέχρι το τέλος χωρίς το μάτι να πηδήσει σε προηγούμενες ή επόμενες σειρές ούτε μία φορά **και** στο τέλος να έχουμε καταλάβει 100% τι κάνει. Το να επιτύχουμε πλήρως αυτό το στόχο είναι πρακτικά ανέφικτο, επιδιώκουμε όμως να τείνουμε όσο το δυνατό πιο κοντά στο ιδανικό.
Μια συνοπτική παρουσίαση των σταδίων ανάπτυξης μπορεί να είναι και η εξής:

- ξεκινάμε από απαιτήσεις εισόδου και εξόδου
- κάνουμε διάγραμμα ροής δεδομένων και καθορίζουμε τις συναρτήσεις
- Σπάμε το πρόγραμμα σε υποπρογράμματα και αυτά σε μικρότερα, αν χρειαστεί. Αυτά καλό είναι να κυμαίνονται από 20 ως 50 γραμμές κώδικα (περίπου μία οθόνη).
- Ανάλυση από πάνω προς τα κάτω, υλοποίηση (γράψιμο κώδικα) από κάτω προς τα πάνω. Δηλαδή, αφού έχουμε σχεδιάσει το πρόγραμμα, αρχίζουμε να γράφουμε τις θεμελιωδέστερες συναρτήσεις και μετά αυτές που τις καλούν κ.ο.κ. μέχρι να φτάσουμε στην "κορυφή" που είναι η main.

Μετά, ακολουθεί το στάδιο των διορθώσεων (εδώ μπορεί και να διαπιστώσουμε ότι πρέπει να αλλάξουν διάφορα στοιχεία του σχεδιασμού επειδή δεν είχαμε προβλέψει όσα μπορεί να πάνε στραβά).

Β. Δοκιμές και διορθώσεις
Αυτό είναι ίσως και το κομμάτι που φοβίζει περισσότερο και φαίνεται σαν η πιο μεγάλη ταλαιπωρία (ίσως όχι άδικα). Γι' αυτό και δίνουμε μεγαλύτερη έκταση σε αυτό το μέρος.
Συνηθισμένα λάθη και "ύποπτα" σημεία
Μερικά λάθη είναι τόσο συνηθισμένα (και όχι μόνο μεταξύ των αρχάριων!) ώστε αξίζει να τα καταγράψουμε και να τα ελέγχουμε ως "συνήθεις ύποπτους" κάθε φορά που κάτι πάει στραβά – ακόμη κι όταν δεν πάει κάτι φανερά στραβά! Τα περισσότερα από αυτά θα εκδηλωθούν είτε κατά τη μεταγλώττιση είτε κατά την εκτέλεση, αλλά υπάρχουν και μερικά που θα περάσουν και τα δύο αυτά στάδια και θέλουν ιδιαίτερη προσοχή. Αν κάθε φορά που έχουμε σφάλμα ψάχνουμε πρώτα ανάμεσα σε αυτά, τα συνήθη λάθη, σύντομα θα συνηθίσουμε και θα τα βρίσκουμε κατευθείαν. Ας κάνουμε λοιπόν στον εαυτό μας τις παρακάτω ερωτήσεις:

1. βάλαμε ελληνικό ερωτηματικά (;) στο τέλος κάθε *απλής* εντολής; μήπως λείπει από καμία;
2. βάλαμε ελληνικό ερωτηματικό στο τέλος της do...while; μήπως λείπει από καμία τέτοια;
3. μήπως κατά λάθος βάλαμε και κανένα παραπανήσιο ελληνικό ερωτηματικό σε καμία σύνθετη εντολή if, for, while; (π.χ. έπρεπε να γράψουμε while(x>0){...διάφορες εντολές....} και γράψαμε while(x>0) ; {...διάφορες εντολές....} που είναι συντακτικά σωστό αλλά τελείως διαφορετικό από εκείνο που θέλαμε!
4. κλείσαμε όλα τα άγκιστρα { και } ; Τα άγκιστρα πάνε πάντα σε ζευγάρια` όποιος ξέρει να βάζει σωστά τα άγκιστρα ξέρει το 50% του δομημένου προγραμματισμού!
5. κλείσαμε όλες τις παρενθέσεις (και) ;
6. κλείσαμε όλα τα διπλά εισαγωγικά " και ";
7. βάλαμε τη σωστή προδιαγραφή μορφοποίησης στις εντολές printf και scanf (%d για ακέραιους, %f για πραγματικούς, %c για χαρακτήρες, %s για αλφαριθμητικά) ;
8. γράψαμε με τον ίδιο συνδυασμό μικρών και κεφαλαίων ένα όνομα μεταβλητής ή συνάρτησης σε όλη του την εμβέλεια (κύριο πρόγραμμα ή συνάρτηση όπου εμφανίζεται); Όπως δηλώσαμε ένα όνομα, έτσι πρέπει και να εμφανίζεται παντού. Άλλο το abc, άλλο το Abc και άλλο το ABC.
9. μήπως μπερδέψαμε κάπου το 0 (μηδέν) με το O (κεφαλαίο όμικρον);
10. έχουμε μία και μόνο μία συνάρτηση main στο πρόγραμμά μας ή κάνοντας copy-paste από εδώ κι από εκεί προσθέσαμε και... διεκδικητή για την αρχηγία του προγράμματος;
11. έχουμε κάνει include το αρχείο επικεφαλίδας που χρειάζεται για τη δουλειά μας;
12. μήπως έχουμε μπερδέψει τα σημεία στίξης (τελεία αντί για κόμμα, κόμμα αντί για τελεία,

κόμμα ή τελεία αντί για ελληνικό ερωτηματικό μέσα σε ένα for κλπ) ;

13. μήπως μπερδέψαμε κάποιο άγκιστρο με παρένθεση ή το ανάποδο;
14. μήπως αντί για αγκύλες [ ] στα στοιχεία ενός πίνακα βάλαμε παρενθέσεις (και); (συνηθισμένο σφάλμα του διδάσκοντα που επί χρόνια προγραμμάτιζε σε Fortran όπου δεν υπάρχουν αγκύλες!) Σε αυτή την περίπτωση ο μεταγλωττιστής θα νομίσει ότι πρόκειται για κάποια συνάρτηση και πιθανότατα δε θα πει τίποτε!
15. Μήπως έχουμε ξεχάσει να δηλώσουμε κάποια συνάρτηση ή την ορίζουμε μετά τη main χωρίς να προηγείται έστω το πρωτότυπο ή έχουμε ξεχάσει κάποιο δικό μας header file όπου δηλώνουμε τις συναρτήσεις μας;
16. Μήπως έχουμε βάλει τελεστή ανάθεσης (απλό "ίσον", =) αντί για τελεστή ελέγχου ισότητας (διπλό "ίσον", = =); Π.χ. το if(x == y) και το if(x = y) είναι και τα δύο συντακτικά σωστά και ο μεταγλωττιστής δε θα παραπονεθεί καθόλου! Αλλά το πρώτο σημαίνει: "αν το x είναι ίσο με το y, τότε...". Ενώ το δεύτερο σημαίνει: "θέσε x ίσο με το y· αν τώρα, το x είναι διαφορετικό από μηδέν, τότε...". Καμία σχέση το ένα με το άλλο!
17. Μήπως περνάμε σε μια συνάρτηση ορίσματα κατά τιμή ενώ θέλαμε να τα περάσουμε κατά διεύθυνση; ή το αντίστροφο;
18. Μήπως το όνομα μιας εξωτερικής μεταβλητής είναι ακριβώς ίδιο με αυτό μιας τοπικής; Θυμηθείτε ότι σε μια τέτοια περίπτωση, η τοπική υπερισχύει (overrides) πάνω στην εξωτερική. Οσο πιο "τοπική" τόσο περισσότερο υπερισχύει και είναι σα να μην υπάρχει η πιο εξωτερική.
19. Μήπως οι αριθμοδείκτες ενός πίνακα έγιναν μεγαλύτεροι από το μέγεθός του ή μικρότεροι από το μηδέν;
20. Μήπως ζητάμε το περιεχόμενο (κάνουμε dereference) ενός δείκτη που δε δείχνει κάπου;

Ένα συνηθισμένο σφάλμα εκτέλεσης είναι το λεγόμενο segmentation fault: αυτό οφείλεται σε αναφορά σε περιοχή της μνήμης που δεν έχει δεσμευτεί, π.χ. όταν ένας αριθμοδείκτης ξεπερνά τα όρια ενός πίνακα, για παράδειγμα, έχουμε έναν πίνακα με 5 στοιχεία, example[5] και ζητάμε το έκτο: i = 5; ...; x = example[i];
Αλλη συχνή περίπτωση που οδηγεί σε τέτοιο σφάλμα είναι κακή χρήση των δεικτών. Οι δείκτες πρέπει να δείχνουν κάπου (εκτός από NULL) αν θέλουμε να πάρουμε το περιεχόμενό τους. Αν ένας δείκτης δεν έχει οριστεί να δείχνει σε κάποια μεταβλητή θα πάρουμε τέτοιο σφάλμα.
Ιδιαίτερα "πονηρό" σφάλμα είναι εκείνο κατά το οποίο ένας δείκτης έχει τεθεί ίσος με τη διεύθυνση κάποιου άλλου ο οποίος σε κάποια στιγμή γίνεται NULL. Παράδειγμα:

```
int x, **p, *q;
...
q = &x; p = &q;
...
q = NULL;
...
printf("έχουμε και λέμε: %d \n", **p); /* segmentation fault! */
```

Ενέργειες και πρωτοβουλίες για διόρθωση σφαλμάτων

Αφού τελειώσουμε με τα συντακτικά σφάλματα απομένουν τα σφάλματα εκτέλεσης και τα λογικά σφάλματα. Για να τα βρούμε, αν αποτύχει ο έλεγχος με βάση τις ερωτήσεις που δώσαμε πιο πάνω, θα πρέπει να δοκιμάσουμε το πρόγραμμα βάζοντάς του δεδομένα για τα οποία ξέρουμε το αποτέλεσμα (π.χ. από υπολογισμό με το χέρι, από τη θεωρία τη σχετική με το πρόβλημα, από άλλα προγράμματα που ξέρουμε ότι δουλεύουν σωστά για το συγκεκριμένο πρόβλημα και τα συγκεκριμένα δεδομένα κλπ). Αν υποθέσουμε ότι βγαίνει ένα λάθος αποτέλεσμα ή ακόμη και ένα σφάλμα εκτέλεσης και δεν ξέρουμε πού ακριβώς προκύπτει τότε, για να βρούμε την πηγή του, εφαρμόζουμε τη "μέθοδο της διχοτόμησης". Αυτή συνίσταται στα εξής βήματα:
α) βεβαιωνόμαστε ότι τα δεδομένα εισόδου είναι εν τάξει (π.χ. δεν έχουμε βάλει χαρακτήρα αντί για ακέραιο, τιμή μιας μεταβλητής στη θέση μιας άλλη κλπ).
β) τότε το λάθος προφανώς βρίσκεται κάπου μεταξύ της ανάγνωσης δεδομένων και του τέλους του προγράμματος. Βάζουμε κάπου στη μέση του προγράμματος μια εντολή printf για την εκτύπωση μιας ή περισσότερων μεταβλητών που έχουν σχέση με το λάθος που αντιμετωπίζουμε. Έτσι το

πρόγραμμα χωρίστηκε σε δύο μέρη. Αν παίρνουμε λογικές και αναμενόμενες τιμές τότε το λάθος βρίσκεται στο δεύτερο μέρος, μετά από την printf, αλλιώς το πρόβλημα βρίσκεται στο πρώτο μέρος, πριν από την printf.
γ) Στο μέρος του προγράμματος που εντοπίσαμε εφαρμόζουμε και πάλι την ίδια διαδικασία, δηλαδή βάζουμε μία printf κάπου στη μέση και με τη βοήθειά της διαπιστώνουμε σε ποια υποπεριοχή του προγράμματος εντοπίζεται το λάθος.
δ) Συνεχίζοντας έτσι και "μπαίνοντας" σε καλούμενα υποπρογράμματα αν χρειαστεί, εντοπίζουμε ακριβώς το σημείο όπου παρουσιάζεται το λάθος. Ειδικά στην περίπτωση που εντοπίζουμε λάθος στην κλήση μιας συνάρτησης, πριν "μπούμε" στο εσωτερικό της, πρέπει να ελέγξουμε αν τα ορίσματα περνάνε μέσω τιμής ή μέσω διεύθυνσης και να βεβαιωθούμε αν είναι αυτό που θέλουμε.

Τώρα, έστω ότι εντοπίσαμε πού ακριβώς υπάρχει σφάλμα αλλά δεν καταλαβαίνουμε τι φταίει. Μία λύση είναι να τροποποιήσουμε το πρόγραμμα και να δοκιμάσουμε να κάνουμε αυτό που γίνεται εκεί με κάποιον άλλο ισοδύναμο τρόπο. Αν το λάθος διορθώθηκε τότε είμαστε εν τάξει και επίσης, μπορεί να γίνει φανερό τι έφταιγε. Αν δε διορθώθηκε μπορεί τουλάχιστον να γίνει πιο σαφής η προέλευσή του.
Μία άλλη ενέργεια που πρέπει να κάνουμε είναι να ψάξουμε όχι μόνο αυτό που υπάρχει αλλά και αυτό που δεν υπάρχει. Συγκεκριμένα, ας δώσουμε ένα παράδειγμα από πραγματική εμπειρία με τις σειρές ασκήσεων του μαθήματος. Στην έκτη σειρά ασκήσεων υπήρχε η παρακάτω εντολή:

```
vector vec = newVector( );
```

Η συνάρτηση newVector αρχικοποιούσε μια μεταβλητή τύπου vector όπου vector ήταν μια δομή δεδομένων για την αναπαράσταση διανυσμάτων στο επίπεδο. Ενώ όλο το υποπρόγραμμα ήταν ακριβώς ίδιο με αυτό που δινόταν στο φυλλάδιο (το οποίο ήταν δοκιμασμένο), σε κάποια δοκιμή το πρόγραμμα δεν μεταγλωττιζόταν αλλά σταματούσε με το μήνυμα "invalid initializer" με αριθμό γραμμής αυτόν της παραπάνω εντολής. Αυτό που είχε συμβεί ήταν ότι στο αντίστοιχο header file έλειπε η δήλωση της συνάρτησης newVector, δηλαδή η παρακάτω γραμμή

```
vector newVector( void );
```

η οποία είχε σβηστεί κατά λάθος σε κάποιο "copy-paste".
Εξ αιτίας αυτού του τυχαίου σβησίματος δεν πάει αμέσως το μυαλό στο να ελέγξουμε το αντίστοιχο αρχείο, ωστόσο τέτοια λάθη συμβαίνουν αρκετά συχνά, με πιο συνηθισμένη περίπτωση το να ξεχάσουμε μια προσωρινή, δοκιμαστική αλλαγή που κάναμε προηγουμένως. Τα παραπάνω μας οδηγούν να διατυπώσουμε τους εξής κανόνες:
α) όταν εμφανίζεται ένα ακατανόητο πρόβλημα σε μια εντολή, ελέγχουμε αν όλα είναι εν τάξει με τις μεταβλητές, συναρτήσεις κλπ που εμφανίζονται σε αυτή την εντολή (π.χ. οι συναρτήσεις είναι δηλωμένες στα header files, οι μεταβλητές είναι του τύπου που θέλουμε κλπ).
β) προσοχή στα "copy-paste"!
γ) ο,τι αλλάζουμε, αφαιρούμε ή προσθέτουμε προσωρινά, θυμόμαστε να το επαναφέρουμε ή διαγράψουμε όταν τελειώσουμε με τις δοκιμές!

Άλλες πρακτικές οδηγίες

Όταν το πρόγραμμα τυπώνει λάθος αποτελέσματα, πρώτα κυττάζουμε μήπως έχουμε λάθος κωδικό μορφοποίησης σε κάποια εντολή printf ή fprintf, π.χ. θέλουμε να τυπώσουμε ένα float και αντί για %d βάζουμε %f ή θέλουμε να το αλλάξουμε και να τυπώσουμε ένα double αλλά ξεχνάμε να αλλάξουμε το %f σε %lf.
Η πρόσθεση δεικτών δεν είναι αποδεκτή πράξη. Αλλά η αφαίρεση είναι αποδεκτή κι έτσι π.χ. μπορούμε να βρούμε τη "μεσαία" θέση μεταξύ δύο δεικτών ως εξής: mid = low+(high-low)/2; (ενώ το mid = (low+high)/2 θα ήταν λάθος).
Έστω ένας πίνακας A[n]. Τότε τα στοιχεία A[-1] και A[n] είναι έξω από τα όρια του πίνακα. Ωστόσο, οι αριθμητικές πράξεις δεικτών που περιλαμβάνουν το πρώτο στοιχείο μετά το τέλος του πίνακα, δηλαδή το A[n], επιτρέπονται και θα λειτουργήσουν σωστά.

Γ. Οι μεταγλωττιστές στην πράξη
Windows, πρόγραμμα Dev-C++

Οι οδηγίες αυτές βασίστηκαν στην έκδοση 4.9.9.2, αλλά λίγο-πολύ ισχύουν και για την παλιότερη έκδοση που χρησιμοποιήθηκε στο μάθημα.

Έστω ότι θέλουμε να αναπτύξουμε ένα πρόγραμμα που αποτελείται από τρία αρχεία: το **pmain.c**, το **plib.c** και το **plib.h**. Το πρώτο περιέχει την κύρια συνάρτηση (main), το δεύτερο ορίζει κάποιες συναρτήσεις που θα αναπτύξουμε και το τρίτο έχει τις δηλώσεις αυτών των συναρτήσεων και άλλα στοιχεία (εξωτερικές μεταβλητές, ορισμούς συμβολικών σταθερών κλπ).
Ακολουθούμε τα εξής βήματα:

`Αρχείο ---> Νέο... ---> Έργο [Enter]:`

μας βγάζει σε ένα παράθυρο που έχει διάφορους τύπους εφαρμογών.

Διαλέγουμε: `Console Application`

και από κάτω επιλέγουμε C project (ή έργο C). Στο κουτάκι κάτω αριστερά γράφουμε ένα όνομα για το έργο, π.χ. ptest και πατάμε OK.
Μας βγάζει το σχετικό πλαίσιο όπου ζητείται να διαλέξουμε πού θα σώσουμε το project. Αφού διαλέξουμε φάκελλο, μας ανοίγει το project με ένα αρχείο που έχει τα παρακάτω περιεχόμενα:

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

int main(int argc, char *argv[])
{

  system("PAUSE");
  return 0;
}
```

(στην έκδοση που είχαμε στο μάθημα λείπουν το stdlib.h και το system("PAUSE")). Αριστερά, φαίνεται η δομή του project με τα αρχεία που περιέχει (προς το παρόν ένα με default όνομα main.c).
Αυτό το αρχείο που έχει ήδη τη main (εμείς απλώς θα συμπληρώσουμε ο,τι μας χρειάζεται) θα το σώσουμε ως pmain.c Πάμε:

`Αρχείο ---> Αποθήκευση ως... [Enter]`

και στο σχετικό πλαίσιο γράφουμε όνομα pmain. Από κάτω από εκεί που βάζουμε το όνομα υπάρχει ο τύπος του αρχείου. Εκεί λέει "`C source files (*.c)`". Δεν το αλλάζουμε. Σώζουμε το αρχείο.

Τώρα, θα προσθέσουμε το αρχείο plib.c που θα περιέχει τις συναρτήσεις μας. Πάμε:

`Αρχείο ---> Νέο... ---> Πηγαίο Αρχείο [Enter]`

Μας βγάζει ερώτηση "`Να προστεθεί το νέο αρχείο στο έργο;`" Πατάμε Yes. Στο αριστερό πλαίσιο θα δούμε το νέο αρχείο στη δομή του project ως "Ανώνυμο". Το αρχείο αυτό είναι στο προσκήνιο και είναι κενό. Θα το σώσουμε ως plib.c. Πάμε:

`Αρχείο ---> Αποθήκευση ως... [Enter]`

και στο σχετικό πλαίσιο γράφουμε όνομα plib. Από κάτω από εκεί που βάζουμε το όνομα υπάρχει ο τύπος του αρχείου. Εκεί λέει "`C source files (*.c)`". Δεν το αλλάζουμε. Σώζουμε το αρχείο.

Τώρα, θα προσθέσουμε το αρχείο plib.h που θα περιέχει τις δηλώσεις των συναρτήσεων μας. Πάμε:

`Αρχείο ---> Νέο... ---> Πηγαίο Αρχείο [Enter]`

Μας βγάζει ερώτηση "`Να προστεθεί το νέο αρχείο στο έργο;`" Πατάμε Yes. Στο αριστερό πλαίσιο θα δούμε το νέο αρχείο στη δομή του project ως "Ανώνυμο". Το αρχείο αυτό είναι στο προσκήνιο και είναι κενό. Θα το σώσουμε ως plib.h. Πάμε:

`Αρχείο ---> Αποθήκευση ως... [Enter]`

και στο σχετικό πλαίσιο γράφουμε όνομα plib. Από κάτω από εκεί που βάζουμε το όνομα υπάρχει ο τύπος του αρχείου. Εκεί λέει "`C source files (*.c)`". Πατάμε το μικρό βελάκι για να ανοίξει το πτυσσόμενο μενού και διαλέγουμε "`Header files (*.h, *.hpp, *.rh, *.hh)`". Σώζουμε το αρχείο.

Προσθέτουμε στο αρχείο plib.h το παρακάτω περιεχόμενο:

```
#ifndef PLIB_H
#define PLIB_H
#include <stdio.h>
void myfunction( void );
#endif
```

Παρατηρείστε ότι περιλάβαμε και το αρχείο stdio.h.
Σώζουμε το αρχείο από

`Αρχείο ---> Αποθήκευση`

ή από το σχετικό εικονίδιο.

Προσθέτουμε στο αρχείο plib.c το παρακάτω περιεχόμενο:

```
#include "plib.h"
void myfunction ( void ) {
     printf("\nThis is a function\n");
}
```

Τώρα καταλαβαίνουμε ότι έπρεπε να μπει κάπου και το stdio.h αφού υπάρχει μία printf. Εμείς επιλέξαμε να το βάλουμε στο plib.h το οποίο μετά κάνουμε include εδώ.
Σώζουμε το αρχείο από

`Αρχείο ---> Αποθήκευση`

ή από το σχετικό εικονίδιο.

Προσθέτουμε στο αρχείο pmain.c το παρακάτω περιεχόμενο (με έντονα γράμματα τα καινούρια στοιχεία που προσθέτουμε):

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

#include "plib.h"

int main(int argc, char *argv[])
{
  myfunction();
  system("PAUSE");
  return 0;
}
```

Σώζουμε το αρχείο από

`Αρχείο ---> Αποθήκευση`

ή από το σχετικό εικονίδιο.

`Μεταγλώττιση: Εκτέλεση ---> Δημιουργία.`

Διαβάζουμε τα μηνύματα στο κάτω μέρος. Αν υπάρχει κάποιο μήνυμα λάθους μπορούμε να μεταβούμε στο σημείο στο οποίο αναφέρεται κάνοντας διπλό κλικ στο μήνυμα. Τότε, η αντίστοιχη γραμμή στο πρόγραμμα θα τονιστεί.

Εκτέλεση:

`Εκτέλεση ---> Εκτέλεση`

ή

`Εκτέλεση ---> Δημιουργία και Εκτέλεση.`

UNIX/Linux

Για αρκετά μικρές εφαρμογές τα πράγματα είναι πολύ απλά αφού μπορούμε να βάλουμε όλα τα

αρχεία μας (πηγαία και επικεφαλίδας) σε ένα κατάλογο, να μεταβούμε εκεί από τη γραμμή εντολών και να τυπώσουμε

```
cc pmain.c plib.c
```

και να πατήσουμε Enter (δε χρειάζεται να περιλάβουμε το plib.h). Το εκτελέσιμο αρχείο θα έχει το όνομα a.out οπότε για να το τρέξουμε αρκεί να τυπώσουμε

```
./a.out
```

και να πατήσουμε Enter. Αν θέλουμε να δώσουμε άλλο όνομα στο αρχείο, π.χ. myprog, μπορούμε να τυπώσουμε:

```
cc pmain.c plib.c -o myprog
```

και μετά, για να το τρέξουμε:

```
./myprog
```

Για πιο μεγάλα projects με πολλά αρχεία πηγαίου κώδικα και επικεφαλίδας μπορούμε να χρησιμοποιήσουμε το make, ένα utility του UNIX/Linux που επιτρέπει να μεταγλωττίζονται μόνο τα αρχεία που τροποποιήσαμε και όχι όλο το πρόγραμμα από την αρχή. Για να τρέξει το make πρέπει να φτιάξουμε ένα αρχείο κειμένου το οποίο λέγεται makefile και περιέχει οδηγίες για το make ώστε να κάνει τη μεταγλώττιση. Το makefile πρέπει να βρίσκεται στον ίδιο κατάλογο με τα άλλα αρχεία μας. Ένα αρχείο makefile για το παράδειγμά μας θα μπορούσε να είναι το εξής:

```
.c.o:
     cc -c -g $*.c

OBJECTS = pmain.o plib.o

CMD = test.exe

$(CMD): $(OBJECTS) makefile
     cc -o $(CMD) $(OBJECTS)

clean:
     /bin/rm *.o

## Dependencies
pmain.o: plib.h
plib.o: plib.h

plib.h:
```

Οι γραμμές που αρχίζουν με # είναι σχόλια (και όχι οδηγίες στον προεπεξεργαστή! - αυτό δεν είναι γραμμένο σε γλώσσα C!) Στην αρχή, δίνεται ο “κανόνας” που λέει οτι ο μεταγλωττιστής (cc) θα κάνει μεταγλώττιση κάθε πηγαίου αρχείου με κατάληξη c. Από αυτά θα προκύψουν τα αντικειμενικά αρχεία με κατάληξη o. Στα αρχεία αυτά δίνουμε συλλογικά το όνομα OBJECTS, ενώ το εκτελέσιμο αρχείο (CMD) θέλουμε να λέγεται test.exe. Οι επόμενες γραμμές λένε ότι ο μεταγλωττιστής (βασικά ο linker) θα πάρει τα εκτελέσιμα και θα τα ενώσει για να δώσει το εκτελέσιμο με το όνομα που ορίσαμε. Μπορούμε να ορίσουμε επίσης και πρόσθετες λειτουργίες όπως με το προσδιοριστικό clean που μπορούμε να το θέσουμε σαν επιλογή στο make και τότε αυτό θα έχει σαν αποτέλεσμα να σβηστούν τα αντικειμενικά αρχεία και να μείνουν μόνο τα πηγαία και το εκτελέσιμο. Τέλος, καθορίζονται και οι εξαρτήσεις, δηλαδή ποιο αρχείο σχετίζεται με ποια, ώστε αν αλλάξουν αυτά να μεταγλωττιστούν εκ νέου και όσα εξαρτώνται από αυτά.

<u>Παραδείγματα</u>:
Με έναν editor (όπως ο vi, emacs ή gedit) μπορούμε να γράψουμε τα τρία αρχεία pmain.c, plib.c και plib.h που περιγράψαμε πιο πάνω, καθώς και το makefile με τα παραπάνω περιεχόμενα. Αν γράψουμε απλά

```
make
```
και πατήσουμε Enter θα δούμε τα εξής:
```
cc -c -g pmain.c
cc -c -g plib.c
cc -o test.exe pmain.o plib.o
```
δηλαδή πρώτα μεταγλωττίστηκαν ένα-ένα τα αρχεία c και μετά ενώθηκαν. Η επιλογή -g επιτρέπει να ενσωματωθούν χρήσιμες πληροφορίες για τη χρήση προγραμμάτων αποσφαλμάτωσης (debugging). Η επιλογή -c εμποδίζει το linker να δράσει, δηλαδή να ενώσει το αντικειμενικό αρχείο με τις βιβλιοθήκες πριν μεταγλωττιστούν και τα υπόλοιπα, ώστε να μπορέσει να τα ενώσει όλα μαζί στο τέλος.
Αν πάμε να αλλάξουμε ένα αρχείο, π.χ. το plib.c (π.χ. να προσθέσουμε ένα σχόλιο) και μετά ξαναγράψουμε make, θα δούμε το εξής:
```
cc -c -g plib.c
cc -o test.exe pmain.o plib.o
```
δηλαδή μόνο το αλλαγμένο αρχείο μεταγλωττίστηκε ξανά.
Αν ξανατρέξουμε το make, θα δούμε το εξής:
```
make: `test.exe' is up to date.
```
με προφανές νόημα.
Μπορούμε να δοκιμάσουμε και την επιλογή clean:
```
> make clean
/bin/rm *.o
```
και αν ξανατρέξουμε το make θα πάρουμε:
```
cc -c -g pmain.c
cc -c -g plib.c
cc -o test.exe pmain.o plib.o
```
αφού τώρα δεν υπάρχουν μεταγλωττισμένα αρχεία πηγαίου κώδικα (τα σβήσαμε όλα με το clean).
Τέλος, αν αλλάξουμε το plib.h από το οποίο εξαρτώνται τα άλλα δύο, όπως περιγράφεται και στο makefile, και τρέξουμε το make θα πάρουμε το εξής:
```
cc -c -g pmain.c
cc -c -g plib.c
cc -o test.exe pmain.o plib.o
```
αφού λόγω της εξάρτησης πρέπει να ενημερωθούν και τα άλλα δύο αρχεία που έχουν κάνει include το plib.h