Αρχικά κάναμε define τα πεδία που παραμετροποιούνται για την άσκηση όπως δίνει η εκφώνηση:
#define M 5
#define K1 3
#define K2 2
#define K3 2
καθώς και μερικά άλλα πεδία βοηθητικά για την υλοποίηση:
#define RED 0
#define GREEN 1
#define YELLOW 2
#define TRUE 1
#define FALSE 0

Φτιάξαμε δύο struct τα οποία θα χρησημοποιήσουμε για την υλοποίηση του παιχνιδιού.

```
typedef struct {
	int number, color;
} Move;
```

Το struct Move αναπαριστά μία κίνηση που κάνει ένας παίκτης στο παιχνίδι, δηλαδή κρατάει ένα χρώμα και έναν αριθμό κύβος που θα αφαιρεθούν από το τραπέζι.

```
typedef struct {
	int red, green, yellow;
} Table;
```

Το struct Table αναπαριστά μία κατάσταση τραπεζιού, δηλαδή ένα στιγμιότυπο κατάστασης παιχνιδιού. Κρατάει τρία πεδία red, green, yellow που το κάθε ένα παριστάνει το πόσα κυβάκια εκάστοτε χρώματος υπάρχουν στο τραπέζι.

Με αυτό το struct θα δουλέψουμε σε όλο το παιχνίδι. Δηλαδή, θα φτιάξουμε ένα τραπέζι τύπου Table, θα το αρχικοποιήσουμε ώστε να έχει M κυβάκια για κάθε χρώμα και από αυτό θα αφαιρούμε κύβους για κάθε κίνηση του παίκτη ή του υπολογιστή.

Φτιάξαμε επίσης ένα πίνακα με όλες τις επιτρεπτές κινήσεις από κάποιον παίκτη. Η εκφώνηση λέει ότι έχουμε τις εξής επιτρεπτές κινήσεις:

1 κόκκινο κύβο
1 πράσινο κύβο
1 κίτρινο κύβο
(K1=3) κόκκινους κύβους
(K2=2) πράσινους κύβους
(K3=2) κίτρινους κύβους

Συνολικά δηλαδή 6 πιθανές κινήσεις είτε από τον παίκτη είτε από τον υπολογιστή κάθε φορά.

Αυτός ο πίνακας είναι ο possibleMoves. Είναι τύπου Move (το struct που ορίσαμε παραπάνω) και έχει μέγεθος 6, μία θέση για κάθε επιτρεπτή κίνηση.

**Η συνάρτηση main:**
Στην main υλοποιούμε ολόκληρο το παιχνίδι χρησημοποιώντας τις υπόλοιπες βοηθητικές συναρτήσεις.

Συγκεκριμένα, οι μεταβλητές gameOver και aiWins είναι ψευδο-boolean μεταβλητές για το αν έχει τελειώσει το παιχνίδι και για το αν έχει κερδίσει ο υπολογιστής ή ο χρήστης. Τις δηλώνουμε στη main και η γλώσσα C τις αρχικοποιεί αυτόματα σε 0 (το 0 είναι το false και το 1 το true).

Μετά καλούμε την initializePossibleMoves η οποία φτιάχνει τον πίνακα που αναφέραμε παραπάνω με τις επιτρεπτές κινήσεις.

Φτιάχνουμε το κύριο τραπέζι μας (tbl) με το οποίο θα δουλέψουμε για την εκτέλεση όλου του παιχνιδιού.

Καλούμε την initializeHead η οποία παίρνει με αναφορά το τραπέζι και το αρχικοποιεί δηλαδή του βάζει τον αριθμό των κύβων που πρέπει να έχει το τραπέζι κατά την έναρξη του παιχνιδιού.

Καλούμε την printTable(tbl) για να τυπώσει το τραπέζι. Αυτή την έχουμε φτιάξει εμείς παρπαάνω. Εδώ έχουμε βάλει απλό πέρασμα του tbl δηλαδή με τιμή και όχι με αναφορά (δε χρησημοποιούμε pointer) γιατί δεν θέλουμε να πειράξουμε κάτι στο τραπέζι και να μείνει η αλλαγή θέλουμε απλά να διαβάσουμε τις τρείς μεταβλητές του με τα τρία χρώματα.

Βάζουμε ένα while(1) δηλαδή επανάληψη επ άπειρο και μέσα υλοποιούμε το παιχνίδι δηλαδή δίνουμε τη σειρά για να παίξει μία στον παίκτη και μία στον υπολογιστή. Τσεκάρουμε κάθε φορά μετά από κάθε κίνηση αν έχουμε gameover και αν έχουμε αλλάζουμε τις μεταβλγητές gameOver και aiWins καταλλήλως και μετά κάνουμε break από τον ατελείωτο βρόγχο.

Εκτός βρόγχου, τυπώνουμε απλά ποιος νίκησε.

**Άλλες βοηθητικές συναρτήσεις:**

void placeMove(Table *t, Move m): παίρνει ως όρισμα το τραπέζι που θέλουμε να γίνει η κίνηση τύπου Table και μια κίνηση τύπου Move και απλά αφαιρεί από τον κατάλληλο από τους τρείς μετρητές για τα τρεία χρώματα του τραπεζιού, τον αριθμό που λέει η κίνηση που περάσαμε. Εδώ όπως βλέπουμε περνάμε το Table με αναφορά και το δουλεύουμε με pointer γιατί θέλουμε οι αλλαγές που θα κάνουμε να μείνουν και μετά την κλήση της συνάρτησης.

int checkGameOver(Table t): Τσεκάρει για το τραπέζι που περάσαμε ως όρισμα αν έχουμε gameOver δηλαδή αν έχουμε μηδενιστεί όλοι οι κύβου του τραπεζιού.

void printTable(Table t): τυπώνει το τραπέζι

int isValidMove(Table t, Move m): επιστρέφει 0 αν η κίνηση που πήρε ως όρισμα στο τραπέζι που πήρε ως όρισμα δεν είναι επιτρεπτή και 1 αν είναι επιτρεπτή. Εδώ τσεκάρουμε 2 πράγματα: αν υπάρχουν διαθέσιμοι κύβοι για να σηκώσει και αν η κίνηση που δώθηκε ανήκει στις επιτρεπτές κινήσεις (τις 6 που είπαμε παραπάνω).

Table newTableAfterMove(Table t, Move m): Επιστρέφει ένα τραπέζι. Το τραπέζι αυτό έχει όσα κυβάκια θα είχε το τραπέζι που περάσαμε ως όρισμα αν κάναμε την κίνηση move που επίσης

περάσαμε ως όρισμά. Δηλαδή επιστρέφει ένα αντίγραφο του τραπεζιού που δέχτηκε, μόνο που σε αυτό το αντίγραφο κάνει και την κίνηση που δέχτηκε.

void userPlay(Table *t): Την καλούμε όταν θέλουμε να κάνει κίνηση ο παίκτης. Διαβάζει μία κίνηση από τον παίκτη και εφόσον αυτή είναι επιτρεπτεί την εκτελεί στο τραπέζι που πήρε ως όρισμα.

void computerPlay(Table *t): Την καλούμε όταν θέλουμε να κάνει κίνηση ο υπολογιστής. Με την βοήθεια της nextMoveAi (θα τη δούμε σε λίγο) κάνει την κίνηση που προβλέπει ο Minimax.

int nextMoveAi(Table t): Επιστρέφει ένα νούμερο από το 0 ως το 5 δηλαδή ένα από τα 6 πιθανά σενάρια κίνησης. Κάθε ένας από αυτούς τους αριθμούς αναπαριστούν μία πιθανή κίνηση δηλαδή μία θέση στον πίνακα possibleMoves. Συγκεκριμένα, για κάθε μία από τις επιτρεπτές κινήσεις καλούμε τον minimax για να μας επιστρέψει το score. Στο τέλος επιλέγουμε την κίνηση με το υψηλότερο score όπως προβλέπει ο minimax. Και επιστρέφουμε το index αυτής της κίνησης ώστε να την κάνει η συνάρτηση computerPlay δηλαδή να κάνει την κίνηση possibleMoves[τιμή που επιστρέψαμε].
Εδώ όπως βλέπουμε χρησημοποιώ την newTableAfterMove για να μην μπλέξω τον αλγόριθμό του Minimax με το τραπέζι που ήδη έχω και γίνει αχταρμάς, φτιάχνω για κάθε κίνηση καινούρια εικονικά τραπεζάκια με εκτελεσμένη την πιθανή κίνηση για την οποία θέλω το score κάθε φορά και τα περνάω στον minimax με τιμή και όχι με αναφορά ώστε να μην τα μπλέξω.

int minimax(Table t, int aiTurn, int depth): Η υλοποίηση του Minimax. Μας επιστρέψει το score της τρέχουσας κατάστασης του τραέζιού. Όπως είπα παραπάνω στον Minimax περνάω κατάσταση τραπεζιού με ήδη παιγμένη κίνηση για να μου επιστρέψει το score που έχω. Αν σας ρωτήσει γιατί το κάνατε έτσι, γιατί έτσι μας ήρθε. Τα ορίσματα είναι το τραπέζι με την ήδη παιγμένη κίνηση όπως είπαμε, η boolean aiTurn που είναι 1 όταν παίζει το pc και 0 όταν παίζει ο χρήστης (αυτό το χρησημοποιώ κατά την αναδρομή στον Minimax και κάθε φορά το αλλάζω για να πετύχω αυτό που είπαμε με τα φύλλα και την εναλλαγή από min σε max κάθε φορά, και το Depth για να ξέρω πόσο βαθιά είμαι στο δέντρο κάθε φορά. Η διαδικασία είναι η εξής: για κάθε φύλλο του κόμβου στον οποίο είμαι (δηλαδή για όλες τις πιθανές επιτρεπτές κινήσεις), αν είμαι στην κατάσταση max επιλέγω αυτή με το υψηλότερο score και αν είμαι στην min επιλέγω αυτή με το μικρότερο. Ο αλγόριθμος δουλεύει αναδρομικά και όταν βρεί gameOver γυρίζει προς τα πάνω και αναθέτει τις τιμές σε κάθε κόμβο που διατρέξαμε σύμφωνα με τους κανόνες που ξέρουμε (αυτό με το min και το max για τα αρτια-περιττα, αυτό που είπα και παραπάνω).

Ο υπολογισμός του score γίνεται με την getScore. Της περνάμε το βάθος και την τρέχουσα κατάσταση min ή max και μας επιστρέφει το score. Με αυτό πετυχένουμε χοντρικά ότι όταν ο υπολογιστής βρεθεί σε δίλλημα, επιλέγει την κίνηση που θα του αποφέρει συντομότερη νίκη δηλαδή ότι υπάρχει σε μικρότερο βάθος φύλλο του δέντρου (όπου φύλλο σημαίνει gameover).

Αυτά για τον κώδικα.
Τώρα γιατί δεν φτιάξαμε κανονικό δέντρο και να κρατάμε σε κάθε node το score ώστε να μην χρειαστεί να τρέχουμε αναδρομικά τον minimax κάθε φορά: γιατί είχε μεγάλη πολυπλοκότητα χώρου. Το δοκιμάσαμε στην αρχή και έπιανε πάρα πολύ ram και επίσης, έκανε πολλή ώρα η αρχικοποίηση στην αρχή όπου καλούσαμε τον minimax για να βρεί τα score από όλα τα nodes. Οπότε το αλλάξαμε και το κάναμε έτσι.