Adatszerkezetek és forrásfájlok választása

Az includeolt forrásfájlok

```
#include <stdio.h>     //Kiíráshoz
#include <stdlib.h>    //Memóriakezelés
#include <stdbool.h> //Igaz-hamis érték
#include <string.h>    //strcpy és társai, stringkezelés
#include <time.h>      //randomizálás, illetve a delay beállítása
#include "ranklist.h"  //dicsőséglista kezelése
#include "debugmalloc.h"    //memóriakezelés figyelése
#include "debugmalloc-impl.h"       //memóriakezelés figyelése
#include "econio.h"   //szebb konzolhasználat, színesítés
#include "tombkezelo.h"      //dinamikus tomb kezelése (kiírása, átírása…)
#include "jatekvezerles.h"    //A gép taktikájának vezérlése
#include "keret.h"            //A játék elejét és végét irányítja, keretet ad a programnak
#include "sablon.h"           //A váza a programnak, felhasználóbarátibb kezelés
#include "structures.h"       //Az adatszerkezetek headerfileja
```

A program működésének fő szereplői:

```
typedef struct{
   char nicknev[31];
   int pontszam;
}Gyoztes;
```

Ez fogja tárolni a felhasználónak a legfeljebb 30 karakterből álló nevét, valamint ahány lépés történik a játék ideje alatt, annyi lesz a pontszáma. Ha a gép győz, akkor ez a nicknev átíródik „RobotAI'-ra, viszont a pont marad. Majd ezt a struktúrát fogjuk átadni a dicsőséglistát kezelő függvénynek, és íratjuk ki azt.

```
typedef struct{
   char karakter;
   bool vege;
}Jatek;
```

Ez fogja a játék állását ellenőrizni. Minden körben megnézzük, hogy kigyűlt-e már valakinek öt egymás melletti (azonos) jel, és ha igen, akkor a karakter eltárolja a kigyűlt jelet (O vagy X), a bool pedig truera értékelődik, (minden már körben false volt, ez a ciklusunk bennmaradási feltétele is), és innen tudjuk majd megnézni, hogy vége van-e a játéknak. (A karakterből pedig következtetünk, hogy a gép nyert, vagy a játékos.)

```
typedef struct{
   int veszely;
   int lehetoseg;
   int sor;
   int oszlop;
   int mennyi;
   bool helyzet;
}Mezo;
```

Ez fogja a gépnek a lépését vezényelni. Az egymás mellett lévő azonos jeleket fogjuk számolni a függvényben, és ennek a számát fogjuk a ’mennyiben’ eltárolni. A sor és oszlop kapja annak a mezőnek a koordinátáit, amelyiket éppen vizsgálunk. (Ez az átmenetileg más jelre beállított mező koordinátája lesz, de erről részletesebben lent a függvényeknél írok. A pálya összes vizsgált mezői közül megnézzük melyiknek a legnagyobb a ’mennyi’ értéke, és annak a mezőnek az x koordinátája lesz a sor, az y pedig az oszlop. A helyzettel tudjuk majd összehasonlítani, hogy a gépnek támadni vagy védekezni lenne hatékonyabb. Két Mezo típust hozunk létre, az egyik a védekezésre a másik a támadásra.

```
typedef struct Ranklist{
   char nev[31];
   int pontszam;
   struct Ranklist *next;
}Ranklist;
```

Ez a dicsőséglistához szükséges struktúra. A név itt is egy fix 30 karakteres, csakúgy mint feljebb a Gyoztes.nicknev, hisz ezeket a nickneveket fogjuk belemásolni és a pontszam is hasonlóképp a Gyoztes.pontszam lesz, valamint mivel láncolt listát használunk a kiíráshoz, szükséges egy Ranklist * next pointer is, ami majd a következő listaelemre mutat.

```
typedef struct {
   char **tomb;
   int meret;
}Palya;
```

Ez fogja a pályánkat tárolni, a kétdimenziós dinamikus tömbünket, és annak méretét. Így nem kell minden függvénynél külön átadni a tömböt és a méretet, hanem csupán a struktúrát. Ha a függvény módosítani akarja vagy a méretet vagy a mezőket, akkor pedig cím szerint adjuk át.

```
enum { GYOZ = 5 };
```

Ez egy globális változó, mellyel jelezzük, hogy a játéknak 5-ig kell mennie, vagyis akkor van vége, ha kigyűlik 5 egymás melletti azonos jel. (Célszerű enumot használni, hisz így nem kell a kódban mindenhová beírni az ötös számot, illetve elég azt csak egy helyen módosítani. Persze ezzel csak egész típusú számokat tudunk tárolni, de nálunk más nem is jöhet szóba).

A program működését vezérlő függvények

Felhasználóbaráti megjelenítést segítő függvények

```
void delay(double masodperc);
```

Paraméterként egy másodpercet vesz át, majd a program indításához mérten clock_t start_time-t beállítjuk, és átkonvertáljuk a gép által számolt milliszekundumot másodpercbe. Végül elszámolunk egy while ciklussal eddig, ezzel kis szünetet érünk el a program lefutása közben.

```
void loading();
```

Mintha szükség lenne arra, hogy a játékot betöltsük, ezért egy printf mutatványozással egy töltősávot idézünk elő.

```
void clrscr();
```

Az includeolt „econio.h” header által tartalmazott képernyőtörlést használjuk. Ennek hatására egy új, üres konzolt látunk magunk előtt.

```
bool menu();
```

Megjelenítjük a menüt az 5 menüponttal:

„1-re jatekszabalyt kiirja, es maradunk a menuben

2-vel ugrunk a mainben oda, ahol folytatodhat a befejezett jatek, egyedul itt terunk vissza trueval, mindenhol false

3-mal a main elejere ugrunk, kezdodik az uj jatek

4-kiirja a ranglistat filebol es maradunk a menuben

5-kilep ha a felhasznalo szeretne (es persze felszabaditja a tombot), kulonben maradunk a menuben"

A visszatérési értéket a 2-es menüpontban állítjuk be, ugyanis ha folytatni akarja a felhasználó egy korábban elmentett játékát, akkor return trueval térünk vissza a függvényből, és így a mainben nem foglalunk helyet a tömbüknek, nem kérdezzük meg újból a nevét, nem nullázzuk le az adatokat, hanem egyből a while ciklusba lépünk, ahol folytatódhat a lépések egymásutánisága.
Ha pedig új játékot kezdünk, akkor return false, és megtesszük az előbb leírt lépéseket.

A tömbkezelést végző függvények

```
void felszabadit(char **tomb);
```

Paraméterként átveszi a dinamikusan foglalt kétdimenziós tömbünket. Ahogy foglaltuk neki a helyet, a „leképezés és pointertömb" módszerrel, ugyanúgy szabadítjuk is fel: először az elemek tömbjét, utána pedig a pointertömböt szabadítjuk fel.

```
void kiir(Palya palya);
```

A palya struktúrát veszi át paraméterként, és kiírja a mezőket színesítve az „econio.h"-ban deklarált függvények segítségével. A pálya kiírásáért felelős fgv, -, |, space, + jelek segítségével íratja ki a pályát
Amely így négyzetrácsosként fog megjelenni, es a négyzet közepén van a rakott jel, X, O, vagy üres
A négyzet tetején es oldalan számozás található, hogy egyszerűbb legyen a mezők koordinátáit beazonosítani.

```
void nullaz(Palya * palya);
```

Átveszi cím szerint a palya struktúránkat, és az összes mezőt ' '-re cseréli. Ezt a függvényt a mainben akkor hívjuk meg, ha új játékot kezdünk, vagyis a menüben a hármas gombot nyomjuk meg.

```
bool szelso_ket_sor_oszlop(int kulsofor, int kulsofeltet, int belsofor, int belsofeltet,
                           char **tomb);
```

Mivel kétdimenziós tömböt kell bejárnunk, ezért két forciklus szükséges ahhoz, hogy átmenjünk az összes mezőn. A kulsofor es a kulsofeltet a tomb x koordinatait nezi, a belsofor es a belsofeltet pedig az y koordinatait. Ha van olyan mező, amelyik nem üres, vagyis van rajta X vagy O, akkor return true, egyebkent ha teljesen üres, akkor false.

bool peremterulet(Palya palya);
Ebben hívjuk meg a szelso_ket_sor_oszlop függvényt, és ellenőrizzük a négy irányt, felső-alsó 2 sor, bal-jobb 2 oszlop. Ha ezek közül egyiknél sincs true, akkor meghívjuk az atir függvényt.

void atir(Palya * palya);
Cím szerint veszi át a palyat, és ha az előző függvény trueval tér vissza, akkor megnöveljük a négyzetes pályánk méretét néggyel. Majd átmásoljuk a régi tömbnek a mezőit egyesével a forciklusban elcsúsztatva kettővel: vagyis a régi tömb 0:0 koordinátájú mezője az új tömb 2:2 mezőbe kerül. Ezzel egy olyan pályát hozunk létre, ahol megint a szélső 2 sor és oszlop üres. Végül az újonnan foglalt tömböt átadjuk a palya->mezonek, és felszabadítjuk a régi tömbünket.

A játék keretét adó függvények

bool vege(**char** **tomb, **int** * szamol, Jatek *vissza, **int** feltetx, **int** feltety, **int** i, **int** j);
Átveszi paraméterként a vizsgálni kívánt tömböt, vagyis a pályát és az irányt, amit a feltetelx es feltetely ad meg. Megnézzük hány db azonos jel van egymás mellett, és a folytonos meghívás alatt a ciklusban növeljük a szamolt. Addig térünk vissza trueval, ameddig megegyeznek az egymás mellett lévő mezők, ha ez nem teljesül akkor return false. Ha ez a szamol eléri az ötöt, vagyis a GYOZ-t, akkor a vissza.vege=true, es a vissza.karakter pedig a vizsgált mező karakterét kapja meg.

Jatek jatekvege(Palya palya);
Átveszi a pályát, és a 4 főirányra meghívja a vege függvényt, függőleges, vízszintes, és 2 átló. Ha valamelyik irányban is a vissza.vege truera változik, akkor a játéknak vége, és a győztes karaktert is eltároljuk ebben a struktúrában. Ez alapján tudjuk majd ellenőrizni a játék végét, ha a visszaadott jatek struktúra karaktere O akkor a felhasználó nyert, ha X akkor pedig a gép.

void lepj(Palya * palya, Gyoztes jatekos);
Átveszi paraméterként a pályát és a Felhasználó adatait kezelő struktúránkat. Ha ez az elso lépése lenne a játékosnak, akkor még nem lehet menteni. Ezért megnézzük, ha a pálya üres, akkor az az első lépése. Azért hogy gyorsítsuk ezt az ellenőrzést, ha a palya merete nagyobb mint a kezdeti, akkor biztosan nem az első lépés ez. Bekérjük a felhasználótól a lépésének az x y koordinátáit, és meghívjuk a szabade függvényt, valamint ellenőrizzük, hogy nem-e betűt, vagy rossz kombinációt adott meg. Ha minden teljesül, akkor rakunk egy O-t a kívánt mezőre. A jatekos-t csak azért vettük át, hogy nevén tudjuk szólítani a felhasználónkat a lépés előtt.

void elso(Palya * palya);
Ez a gépnek az első lépése lesz, ilyenkor még nincs szükség taktikázásra, ezért a paraméterként átvett palyan megnézzük, hogy hová rakott a felhasználó (hisz ugye a játékos kezd), és annak a mezőnek a szomszédos mezői közül sorsolunk egyet. Ehhez randomolunk egy számot 1-4 között, majd egy switch segítségével döntjük el, hogy melyik szomszédos mezőt válasszuk, és odarakjuk a jelet.

bool szabade(Palya * palya, **int** x, **int** y);
Paraméterként átveszi a módosítani kívánt pályát, és a felhasználó által megválasztott mező x y koordinátája. A függvény ellenőrzi, hogy a
az x és y a tömb méreteinek megfelel-e (a palya->meret es a 0-hoz viszonyítva). Illetve, ha olyan x y párt veszünk át, amelyik mező már foglalt, akkor oda nem lehet rakni, ilyenkor return false, ha pedig szabad a mező és jók a koordináták akkor return true.

Adatok tárolása fileban, és függvényei

void mentes(Palya *palya, Gyoztes jatekos);
Paraméterként átveszi a palyat és a jatekost. A lépés előtt a jatekos eldöntheti mit szeretne csinálni, menteni, folytatni, bezárni. A kis és nagybetű között nem teszünk különbséget. Ha pedig érvénytelen gombot nyom meg a felhasználó, hibaüzenetet írunk ki. Ilyenkor megnyitunk egy txt filet írásra, és beleírjuk a játék adatait (ebben a sorrendben): felhasználó neve, pontja, pálya mérete, és az egyes mezőknek a jele (0, 1, 2-vel = ’ ’, ’O’, ’X’). Minden egyes adat után entert nyomunk, ezzel elválasztva ezeket a txtben. Ha I vagy X-et nyom a felhasználó, akkor fel kell szabadítanunk a foglalt tömbünket, hogy ne legyen memóriaszivárgásunk.

void betolt(Palya * palya, Gyoztes *jatekos);
Ha a menüben a felhasználó a kettes pontot nyomta, akkor be kell töltenünk a filebol az előző állást. Ehhez megnyitjuk a filepointerünk segítségével a txt-nket és enterig olvassuk be a sorokat. Elsőnek a nevet, aztán a pontot (ezek kerülnek a játékos struktúrába), méretet. Ezzel a mérettel foglalunk egy új kétdimenziós dinamikus tömböt, és ebbe töltjük be a fileból a mezőket. És végül palya->tomb-ot rámutatjuk az újonnan foglalt tömbünkre.

void top10(**FILE** *fp);
Egy filepointert vesz át, és ebben összpontosul a Ranklista kezelő függvények összessége. Elejére szúrjuk a filebol beolvasott elemet a listának, rendezzük, végül kiíratjuk.

A Ranklist típus lényeges függvényei és szerepük

void freelista(Ranklist * eleje);
A függvény átveszi paraméterként a láncolt lista elejét, majd létrehozunk egy mozgo pointert, amellyel rámutatunk az elejére a listának, és ezzel bejárjuk az egészet. Mindig eltároljuk az adott mozgo->nextet, hogy ne veszítsük el ezt a pointert, és freezuk a mozgot végül.

void sorting(Ranklist *eleje);
Paraméterként szintén a lista elejét kapja, és buborékrendezést hajtunk végre az egész láncolt listán. Növekvő sorrendbe rakjuk a listát, ha a mozgonak a pontszama nagyobb mint a mellette lévőé, akkor megcseréljük őket a szokásos hármascserével. Közben a pointereket áttesszük, és az adatokat is átmásoljuk.

void elejere(Ranklist **eleje, **FILE** *p);
Első paraméterként egy kettős indirekcióval kapott láncolt listát veszünk át, második paraméterként pedig egy filepointert. Az elmentett filebol olvassuk be az adatokat EOF jelig, ahol minden tag enterrel van elválasztva. Ehhez megnyitjuk a filepointerünk segítségével a txt-nket és enterig olvassuk be a sorokat egy stringbe, majd sscanf segítségével kiolvassuk ebből a stringből a felhasználó nicknevet es a pontját, ezeket egy lista új elemében tároljuk. Ameddig van adat beolvassuk, és befűzzük a listánknak az elejére. Szóval így a láncolt listánk az egész txt-t tartalmazza, de ezt majd a sorting függvény rendezi, és a kiirlist függvényben pedig csak az első 10 (a 10 legkisebb) nevet és pontot írjuk ki).

void kiirlist(Ranklist *eleje);
Átveszi a rendezett láncolt listánknak az elejét, és az első 10 elemét kiirja.

A Gép játékát vezérlő függvényei és szerepük, az algoritmus

Az egymás melletti jelek összehasonlítása

Az első gondolat arra, hogy a gépnek, hová is kellene raknia a saját jelét talán egyértelmű, ahol a legtöbb azonos jel van egymás mellett. Azonban ezzel egy apró gond van. Ugye összehasonlításra használhatnánk valamiféle ciklust, és akkor, ha vízszintesen jobbra akarjuk nézni, akkor növeljük az x koordinátát, az y pedig stagnál, megszámoljuk, hány darab azonos bábú van, vesszük a maximumot, és odarakjuk a jelünket ennek a sornak a végére.

Viszont nem vettük figyelembe ekkor azt, hogyha például van egy lyuk a két jel között. Ekkor külön kellene megnézni vízszintesen jobbra, balra, és összeadni. De ez elég macerás, ha vízszintes mellett függőlegesen fel-le, jobbátló fel-le, balátló fel-le irányban is meg kellene nézni.

```
bool ellenoriz(int i, int j, int sori, int oszlopi, char **tomb, Mezo *vissza, int *szamol,
               bool *bennevan, int limit);

Mezo ujtaktika(Palya * palya, int limit, char babu);

void veszelyes(Palya * palya);
```

A fentebbi problémára az ötlet a következő: mintha megjósolnánk, hogy a felhasználó hova akarná rakni a következő jelet. Ezt úgy valósítjuk meg, hogy egy külső ciklusban a létező összes üres mezőre beállítunk egy felhasználói jelet, és ez alapján végezzük a vizsgálatot. Megnézzük, hogy így hány darab egymás melletti azonos jel van. (Megy egy dupla for az átmeneti beállításra, és megy egy dupla for magára a vizsgálatra.)

A négy főirányt fogjuk vizsgálni, függőlegesen le, vízszintesen jobbra, és kétszer átlósan le. A segédváltozónk ehhez a bennevan, csekk és a szamol. Ameddig a csekk true, addig maradunk a forciklusban, vagy ameddig el nem érünk ötig (hisz a mezőnek az 5 sugarú környezetében vizsgáljuk a mezőket). Ha nem azonosak az egymás melletti mezők, akkor a csekk az false. Közben számoljuk, hogy hány darab azonos jel van egymás mellett (a babu paraméterrel adjuk meg, hogy X vagy O-t keresünk). Viszont az átvett Mezo * vissza struktura tagba csak akkor írjuk bele ezt a szamolt, hogyha az ellenőrzes alatt az adott irányban szerepelt az átmenetileg ’X'-re vagy ’O’-ra állított mező is, és nem csak a pálya egy adott részén vannak ezek egymás mellett.

(Szoval ha pl az 1;1 mező az átmeneti X vagy O, és a fgv elkezdi vizsgálni a 4.sort: 4;1, 4;2, 4;3 stb, es mondjuk ezek mind 'X'-ek lennének, akkor ugye ezt veszélyesnek ítélné meg, mert hisz megegyeznek a mezők, es ezért van ott a feltétel:

```
if (sori == i && oszlopi == j) {

//Itt a sori és az oszlopi az átmeneti mező koordinátája, i és j pedig a vizsgált mezőé

    * bennevan = true;

}
```

Vagyis csak akkor tekinti ezt veszélyesnek, ha benne van az átmeneti 'X' vagy ’O’ is valahol a sorban, oszlopban, átlóban. Ha végeztünk egy átmeneti mező vizsgálatával, akkor ezt az átmeneti mezőt visszaállítjuk üresre, hogy ne zavarja a következő vizsgálat kimenetelét.

Ezzel a módszerrel végig megyünk az összes mezőn, és megnézzük, hogy melyik mezőnél lenne a legmagasabb a Mezo.mennyi, és az lesz a maximum, szóval arra a mezőre kell raknia a gépnek a jelet.

A gép prioritása a támadás, ezért ott már ha van egymás mellett 2 azonos jel, azt is veszélyesnek tartja, ellenben a védekezéssel, ahol egymás mellett 3 azonos jelnek kell szerepelnie (és természetesen, ennek a 2-nek és 3-nak tartalmaznia kell az átmeneti mezőt is). A függvény pedig visszaadja azt a Mezot, ahová kell raknia a gépnek. Kétszer hívjuk meg a függvényt, egyszer a védekezésre, egyszer a támadásra, majd összehasonlítjuk, hogy a két visszaadott érték közül melyiknél nagyobb a mennyi, amelyik a nagyobb azt fogja elvégezni a gép.

A veszelyes függvényben történik maga a lépés, védekezés vagy támadás. Az ujtaktikaban vagy a támadás vagy a védekezés ellenőrzése történik, a limit és a bábu paraméter tér csak el. Az ellenoriz függvényt hívja meg az ujtaktika négyszer, a négy fő irányra.

Felhasználói dokumentáció

A program egy konzolban vezérelt amőbajáték. A játék elindítása a menü betöltésével kezdődik, ahol öt menüpont közül választhatunk. A szám és enter lenyomása után betöltődik a kért pont. Ha nem létező számot, vagy betűt, hibás kombinációt nyomunk, akkor ezt hibaüzenettel jelzi a program.

Az 1-es menüpont a 'Játékszabály': Kiírja a képernyőre az amőba szabályait, illetve a játék menetét.

A 2-es menüpont: 'Előző játék betöltése': Ha volt korábbról elmentett játékunk, akkor azt tölti be a képernyőre, és folytathatjuk onnan, ahol befejeztük a legutóbb. Ilyenkor megmarad a felhasználónevünk, a pontjaink, és természetesen a lépéseink is. A program várja a lépésünket, és a gép minimális gondolkodás után lép szintén, és kiírja a pályára az állást minden képernyő törlés után.

A 3-as menüpont: 'Új játék': A nevéből adódóan egy teljesen új pályát generál, és megadhatjuk a nevünket. Majd ha ez megtörtént, kezdődhet az előzőhez hasonló módon a játék, mi kezdünk, majd a gép rak, és így tovább. A játéknak akkor van vége, ha valakinek kigyűlik az öt azonos jel egymás mellett. Ha hibás mezőre kívánunk rakni (foglalt, érvénytelen koordináta, rossz betűkombináció stb) a gép hibaüzenettel jelzi. Ha a szélső 2 sor vagy oszlop mezőjére raktunk, akkor a pályánk mérete automatikusan megnövelődik. Az első kör kivételével, minden lépés előtt lehetőségünk van arra, hogy elmentsük a játékot. Az 'I' gombbal mentünk és kilépünk, az 'N' gombbal folytatjuk a játékot, az 'X' gombbal mentés nélkül azonnal abbahagyjuk a játékot! (A program egyaránt kezeli a kis és nagybetűket, ebből hiba nem lehet!). Ha vége a játéknak, akkor kiíródik a képernyőre a győztes neve (ha a gép nyert, akkor RobotAI íródik ki, ha a felhasználó akkor az ő elején beállított nickneve). A neven kívül a pontszám is kiíródik, ami egyenlő a megtett lépések számával!

A 4-es menüpont: 'Dicsőséglista': A legjobb 10 játékosnak a nevét és pontját jelenítjük meg növekvő sorrendben. Minél kevesebb pontot ért el valaki, annál előrébb helyezkedik el a toplistán. Egy sikeres játék befejeztével automatikusan kiíródik ez a ranglista.

Az 5-ös menüpont: 'Kilépés': Az elején is lehetőségünk van a játékot befejezni ezzel a menüponttal. Azonban a játék elhagyása előtt még plusz kérdésként megkérdezi tőlünk a program, hogy biztosan el szeretnénk-e hagyni, ha nem, akkor visszakerülünk a főmenübe, ahonnan folytathatjuk az egészet.