Név: Unger Tamás István
Neptun: FTD1YJ
Web: http://maxwell.sze.hu/~ungert

A projektfeladatom dokumentációja (Programozási alapismeretek)

Ebben a dokumentumban röviden és érthetően igyekszem összefoglalni a Programozási alapismeretek c. kurzus projektfeladatának megvalósítását. A program forráskódját az `UngerTamasIstvan.c` fájl tartalmazza. Mivel a projektfeladat keretei között megvalósítottam a fájlkezelést is, ezért a program futtatásához elengedhetetlen a `foglalasok.txt` nevű fájl, amelynek a forráskód jelenlegi állapotában a programmal megegyező mappában, helyen kell elhelyezkednie.

Az 546 soros program a `main ()` függvényen kívül összesen nyolc függvényt/eljárást tartalmaz, ezek nevei forráskód szerinti sorrendben:

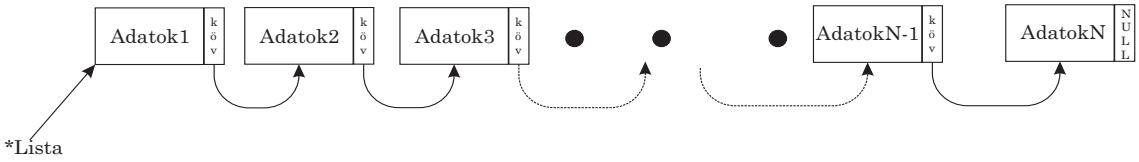
- `AdatBeolvasas`;
- `AdatMentes`;
- `DatumHasonlit`;
- `BemenetHiba`;
- `UjFoglalas`;
- `EddigiFoglalasok`;
- `FoglaloCegekNeve` és
- `FoglalasTorles`.

A forráskódot és a program működését célszerűen az egyes függvények szerint tagolva fogom bemutatni a továbbiakban. Az egyes részekben arra törekedtem, hogy bemutassam, mit és hogyan valósítottam meg. Ahol szükségesnek gondoltam, ott részletesen igyekszem megindokolni, hogy miért pont az adott típusú megvalósítás mellett döntöttem a munkám során.

Az alkalmazott adatstruktúra bemutatása

A forráskód 35. sorától kezdődő első függvény bemutatása előtt célszerű pár szót szólni az alkalmazott adatstruktúráról. A 12. sorban definiáltam az előfordító számára egy `ADATFAJL` nevű változót, melynek értéke az alkalmazott `foglalasok.txt` fájl neve. Ez célszerű a fájlnév esetleges megváltozásának könnyű, egy átírással történő kezelésének céljából. A foglalások adatainak tárolásához egyszeresen láncolt listát alkalmaztam, melynek lényege, hogy az egyes foglalások adatait tartalmazó listaelemek tartalmazzák annak a listaelemnek a címét, amelyek közvetlenül mellettük állnak a listában "jobbra", tehát a következő elem címét. A lista utolsó elemének azon rekeszét, amely a következő elemre mutatna, konvencionálisan `NULL` értékre állítom minden esetben. Innen fogjuk tudni, hogy

a lista végére értünk, amikor bejárjuk a listát. A lista sematikus ábrája az 1. ábrán látható. A lista legfontosabb része az ábrán `*Lista` jelölést kapott mutató, amely a lista első elemére mutat. Ez az úgynevezett kincstári mutató, ezzel fogjuk tudni megtalálni az adatainkat a memóriában. Ha elveszítjük vagy felülírjuk, az adataink elvesznek a memóriában. Nem törlődnek, csak soha nem fogjuk őket megtalálni.



1. ábra. Egyszeresen láncolt lista

A lista elemei speciális adatstruktúrák, amelyeket a feladatnak megfelelően alakítottam ki. A struktúra definiálása a 21. és 31. sorok között található, a neve pedig `Adatok`. Tekintsük ezt a struktúrát egy doboznak, amely a definíció szerint tartalmaz egy 20 hosszúságú karaktersztring tárolására alkalmas tömböt (ide kerül a cégek neve), 5 darab előjel nélküli egész szám tárolására alkalmas változót (a foglalás éve, hónapja és napja, valamint a foglalás kezdeti órája és végső órája) és egy ugyanilyen `Adatok` struktúrára mutató pointert, amelybe mindig a következő listaelem címét fogjuk eltárolni. Az említett `CegNev` tömbnek ebben az esetben 21 rekeszt szükséges kialakítani, hiszen számolni kell a karakterláncot lezáró `null terminator` jelenlétével is. Ezt a hosszt definiálja az előfordítónak a `MAXCEGNEV` változó, melynek értéke 21. Így ha később rájövünk, hogy hosszabb cégneveket szeretnénk alkalmazni, elég csak a kód 15. sorában átírni az értéket.

Az `AdatBeolvasas` függvény célja és működése

Az `AdatBeolvasas` függvény beolvassa a `foglalasok.txt` nevű fájlban elhelyezkedő foglalásokat, létrehozza a láncolt listát, feltölti azt a fájlból beolvasott foglalások adaival és visszaadja a lista elejére mutató kincstári mutatót. A 35. sorban található `Adatok* AdatBeolvasas()` definiálásból látszik, hogy nem kér bemenetet, kimenetként pedig egy `Adatok` típusú struktúrára mutató pointert fog visszaadni. Pár fontos kikötés:

- A `foglalasok.txt` fájl szerkezete kötött. Sorokból áll, minden egyes sor egy-egy foglalás adatait tartalmazza.
- Az egyes sorokban lévő foglalások adatai pontosvesszővel vannak elválasztva egymástól. Az adatok sorrendje is kötött: cégnév, foglalás éve, hónapja, napja, kezdő órája és végső órája.
- Az egyes sorok végén szintén pontosvessző van.
- A program feltételezi, hogy a fájlban lévő adatok minden szempontból helyesek és a benne lévő adatok már sorban vannak. Erre külön ellenőrzést a program nem végez. Azért így valósítottam meg, mert a program maga írja is a fájlt, nem csak olvassa. Amikor viszont írja, akkor a már sorbarendezett láncolt lista tartalmát írja bele, következésképpen az adatok sorban lesznek.

A függvényben létrehoztam egy `*FajlMutato` pointert, amely a fájlban történő barangoláshoz szükséges. Ezek után `fopen` segítségével olvasásra megnyitom a fájlt, majd egy `FoglalasokSzama` változó segítségével megszámolom, hogy hány sora van. Ehhez egy speciális format stringet alkalmaztam, amely illeszkedik a fájl egyes soraira.

Ezt a format stringet nagyjából úgy érdemes elképzelni, mint egy reguláris kifejezést UNIX-környezetben, csak nyilvánvalóan más a szintaktikája. A kifejezés megírásához egy internetes forrást használtam, amelyet a forráskód 42. sorában elhelyezett kommentben fel is tüntettem. Mivel a számolásnál nem szeretnénk elmenteni a beolvasott tartalmat, ezért minden egyes érték elé `*`-ot kell helyezni a specifikáció szerint. Így az `fscanf` függvény beolvassa ugyan a fájl tartalmát, de azt nem rendeli hozzá tömbhöz, változóhoz, stb. Ebben az esetben ugyanis csak a sorokat számoljuk meg.

Amennyiben a fájl üres (`FoglalasokSzama=0`), úgy nincs mit beolvasni, a függvény visszaadja a `NULL` értéket a lista elejére mutató pointer értékének és be is fejeződik az eljárás. Ha a fájlban már van adat, úgy a `rewind` segítségével visszapakolom a fájlmutatót a fájl elejére és elkezdem a tényleges beolvasást.

A láncolt lista elkészítéséhez létrehozunk három mutatót. Az `*ElsoElem` lesz a kincstári mutató, ezt adja majd vissza az eljárás. Az elemek összeláncolásához két mutató kell: az aktuális elemre mutató `*AktualisElem` és az előzőre mutató `*ElozoreMutato`. Kezdetben mindhárom mutató értékét `NULL`-ra inicializáljuk. Mivel már tudjuk, hány sora van a fájlnak, így bátran alkalmazhatunk `for` iterációt annyi lépéssel, ahány sorunk van. A láncolt lista elkészítésének lépései:

1. Az aktuálisan beolvasandó adatok számára létrehozunk egy `*AktualisElem` dobozt;
2. Ebbe a dobozba `fscanf` segítségével bepakoljuk az aktuálisan beolvasandó foglalási adatokat;
3. Amennyiben ez a doboz a lista első láncszeme, úgy beállítjuk a kincstári `*ElsoElem` mutatót ennek a doboznak az értékére;
4. Ha nem ez az első láncszem, akkor ennek a doboznak a címét beírjuk az előző doboz `kovetkezo` rekeszébe, így hozzácsapjuk dobozunkat a listához;
5. Megvizsgáljuk, hogy ez-e az utolsó doboz, és ha igen, akkor a `kovetkezo` rekeszbe beírjuk a `NULL`-t, ezzel zárjuk le a listát;
6. Az iteráció következő lépéséhez elmentjük az aktuális doboz címét a `*ElozoreMutato`-ba;
7. Kezdjük elölről a folyamatot addig, amíg le végig nem léptünk a fájl összes során. Ha végigmentünk, bezárjuk a fájlt és visszaadjuk a kincstári mutató értékét.

Így az adatokat beolvastuk, listába fűztük és visszaadtuk a lista kezdőcímét, tudunk vele tehát dolgozni a továbbiakban.

Az `AdatMentes` függvény célja és működése

Az `AdatMentes` függvény célja, hogy a rendezett láncolt listában található adatokat beleírja a kezelt fájlba az előre megkötött fájlstruktúra szerint. Ehhez ismét megnyitjuk a fájlt, de ezúttal írásra. Nagyon fontos, hogy ebben az esetben az `fopen` függvény argumentuma `"w"`, tehát nem hozzáírunk a fájlhoz, hanem kitöröljük annak tartalmát és újraírjuk az egészet. Így a legegyszerűbb megoldani a feladatot, hiszen amikor írjuk a fájlt, akkor úgyis mindig csak az aktuális, a láncolt listában található foglalásokat szeretnénk elmenteni bele. Az pedig könnyen előfordulhat, hogy időközben nemcsak hozzátoldottunk, hanem töröltünk is a foglalások listájából.

A fájlba íráshoz a `*SegedMutato` nevű munkamutatónknak értékül adjuk a kincstári mutatót, tehát a lista kezdőcímét, majd egy `fprintf` és a már ismertetett format string

segítségével addig (`while`) pakolgatjuk bele a fájlba az egyes foglalások adatait, amíg a munkamutatónk értéke `NULL` nem lesz, tehát amíg a lista végére nem érünk. Ehhez természetesen minden egyes lépésnél beállítjuk a munkamutatót a következő elem címére (`SegedMutato = SegedMutato->kovetkezo`). A ciklusból való kilépés esetén minden foglalás adatát beleírtuk a fájlba, becsukjuk tehát a fájlt, közöljük a felhasználóval, hogy a mentés sikeres és már végeztünk is.

A `DatumHasonlit` függvény célja és működése

A `DatumHasonlit` függvény célja eldönteni, hogy két foglalási dátum (időpont) közül melyik a nagyobb (melyik van időben később). A függvény bemenetként két `Adatok` típusú struktúrára mutató pointert kap, kimenetként pedig

- `1`-et ad vissza, ha az első argumentumnak megadott időpont a későbbi;
- `-1`-et ad vissza ha a második argumentumnak megadott időpont a későbbi, és
- `0`-át ad vissza, ha a két időpont azonos.

A függvényt négy darab `if`-páros segítségével valósítottam meg. Az első páros az évet vizsgálja meg. Ha bármelyik feltétel teljesül, a függvény visszaadja a megfelelő visszatérési értéket majd befejeződik. Ha nem ez történik, akkor az évek azonosak, ellenőrizhetjük a hónapokat ugyanezen módszer szerint, majd a napokat és a foglalások kezdeti óráit. Amennyiben egyik sem teljesül, akkor a két dátum azonos, tehát a függvény nullát ad vissza.

A `BemenetHiba` eljárás célja és működése

Adatok bekérése során előfordulhat, hogy a felhasználó nem olyan típusú adatot ad meg inputként, mint amilyet a program várna. Ilyen eset az, amikor számot várunk (tehát például egy `int` típusú változóba olvasunk be adatot) de a felhasználó karakter(sztringe)t ad be. Ezt az esetet célszerű lekezelni, erre való a `BemenetHiba` eljárás. Meghívása során tájékoztatjuk a felhasználót, hogy nem értelmezhető adattal örvendeztette meg a programot, majd a hibás inputot a 155. sorban található `scanf` segítségével "lenyeljük"'.

Érdemes megfigyelni, hogy mindezt sztringként tesszük, ami univerzális megoldás. Miként azt megtanultuk, számok is értelmezhetőek sztringként, ellenben máshogy kódolódnak, mintha ténylegesen számként kezelnénk őket. Ha az inputot sztringként kezeljük, biztos, hogy sikeres lesz a sztenderd inputon maradt szemét elnyelése. Ezért választottam ezt a megoldást.

Az `UjFoglalas` függvény célja és működése

Az `UjFoglalas` függvény bekéri a felhasználótól a foglalásának az adatait, majd megvizsgálja, hogy szintaktikailag helyesen adta-e meg. Ezek után megvizsgálja, hogy a dátum helyes-e, beleértve a szökőévek esetét is. Ezek után megvizsgálja, hogy szabad-e a kért időpontban a terem, és ha szabad, a foglalást elmenti, behelyezi a láncolt lista megfelelő helyére. Ez lényegében a program első menüpontjának feladatát fogja ellátni. Nézzük végig, hogyan.

Fontos, hogy ez a függvény nem a lista elejének a címét, hanem a címnek a címét kapja bemenetként (`Adatok **Lista`). Ez egy mutatóra mutató mutató (pointer to pointer), amelynek előnye, hogy így nemcsak használni tudjuk az értékét a bemenetként kapott mutatónak, hanem azt is tudjuk módosítani, hogy hová mutat. Gondoljuk meg, hogy

miért van erre szükség! Előfordulhat, hogy olyan foglalást kell rögzíteni, amely a láncolt lista elejére kell, hogy kerüljön. Ebben az esetben a beszúrás során módosítani kell a lista első elemének címét, hiszen az első elem címe az aktuálisan beszúrandó elem címe lesz. Az első elem címe pedig nem más, mint az ún. kincstári mutató. A kincstári mutatót kell tehát módosítani az új elem címére, ez pedig csak úgy lehetséges, ha nem közvetlenül a pointer értékét, hanem a pointer értékének a címét adjuk át a függvényünknek.

A függvény elején létre kell hozni ideiglenes változókat, ahová az adatok bekérését elvégezzük. Ezen kívül létrehozunk egy mutatót (`Adatok *UjFoglalas;`, 168. sor), amely sikeres foglalás esetén az új dobozunkra fog majd mutatni, amelybe a foglalás adatait pakoljuk el. Létrehozunk még egy `Foglaltsag` nevű 24 elemű tömböt és inicializáljuk úgy, hogy minden elemét nullára állítjuk be. Ez a tömb kulcsszerepet fog játszani a foglaltság ellenőrzésében, amelyet később mutatok majd be.

A bekérési interface a 182-198. sorok között található. Itt egyrészt bekérjük a felhasználótól a foglalás adatait, majd bekérés után közvetlenül megvizsgáljuk azt is, hogy formailag helyes-e a beadott érték. Erre szolgál a `JolAdtaMeg` indikátorváltozó és a korábban bemutatott `BemenetHiba` függvény. Kezdetben a `JolAdtaMeg` értéke 1. Először bekérjük a felhasználótól az évet (2000 és 2099 között). Ezt `scanf` segítségével tesszük meg, amely most egyszerre két szerepet is igyekszik betölteni. Egyrészt ténylegesen bekéri a felhasználótól az adatot a megfelelő ideiglenes változóba. Másrészt pedig a `scanf` függvény kimenetét eredményül adjuk a `JolAdtaMeg` változónak is.

Ez miért jó? A `scanf` függvény visszaadja a sikeresen beolvasott karakterek számát. Ez – amennyiben a beolvasás sikeres volt – egy nullától nagyobb érték kell, hogy legyen. Amennyiben a beolvasás sikertelen volt (értsd: nulla karaktert olvasott be sikeresen), úgy a függvény nullát ad visszatérési értékként. Ezt ellenőrzi a következő `if`-ág. Amennyiben ide bekerülünk, úgy a felhasználó biztos, hogy nem adta meg helyesen az adatot, meghívjuk a `BemenetHiba` függvényt, szólunk a felhasználónak, kitakarítjuk a standard inputot majd kidobjuk a felhasználót a menüpontból (`return 1`).

Ha idáig eljutottunk, következhet az adatok helyességének ellenőrzése. Megvizsgáljuk, hogy a felhasználó a lehetséges évintervallumból választott-e, valamint hogy a megadott hónap és nap létező dátum-e abban az évben, és természetesen megnézzük azt is, hogy az időpontként megadott szám tényleg létező óra-e, valamint hogy a foglalás vége később van-e, mint a foglalás eleje. Ellenőrizzük a szökőévek esetét is. Amennyiben a felhasználó február 28-nál később februári időpontra szeretne foglalni és az évszám néggyel való osztásának maradéka nem nulla (négyévente van szökőév), szólunk a felhasználónak, hogy hibásan adta meg a dátumot és kirúgjuk a főmenübe.

Amennyiben ez rendben van, eljutottunk arra a pontra, hogy tudjuk, hogy a felhasználó létező időpontra kíván foglalni. Következő lépés annak megvizsgálása, hogy a terem abban az időszakban foglalt-e. Ehhez készítünk egy munkamutatót, amelynek értékül adjuk a kincstári mutatót, tehát a láncolt lista elejére pakoljuk azt. Ezek után egy `while` segítségével végigfutunk a láncolt lista minden elemén, egyenként megvizsgáljuk őket, és ha úgy találjuk, hogy az a foglalás éppen ugyanarra a napra szól, mint az aktuális foglalási igény, elővesszük a `Foglaltsag` nevű indikátortömbüket, és a listában szereplő foglalás kezdeti időpontja és végidőpontja között egy `for` iteráció segítségével bebillentjük az összes olyan indikátort 0-ról 1-re, amelyhez tartozó órákban a terem már foglalt.

Ha ezzel az iterációval végeztünk, az indikátortömb tisztán és világosan fogja jelezni az aznapi foglaltsági helyzetet. Meg tudjuk tehát nézni, hogy a szóban forgó foglalási igény teljesíthető-e. Készítünk egy `for`-iterációt a foglalási igény kezdeti és a végidőpontja között, és amennyiben ezen a szakaszon bárhol találunk egyest az indikátortömbben, az azt jelenti, hogy a foglalási igény nem teljesíthető, a terem már foglalt. Ezt a nullára inicializált `MarFoglalt` változó egyre történő billentésével jelezzük. Amennyiben ezen indikátor értéke

1, közöljük a felhasználóval, hogy foglalt a terem és kidobjuk a főmenübe.

Ha ez nem történik meg, a foglalás teljesíthető, el kell tehát helyeznünk a láncolt listába. Lefoglaljuk neki helyet, elkészítjük a "dobozt" (`UjFoglalas = malloc(sizeof(Adatok));`), majd a doboz megfelelő rekeszeit megpakoljuk az aktuális foglalási igény adataival (284-289. sorok). A munkamutatónkat a lista elejére állítjuk, majd létrehozunk még egy segédmutatót, hogy a lista végigjárása során el tudjuk menteni az előző doboz címét. Ezt nullára inicializáljuk.

Ha ez megvan, lekezeljük azt az esetet, amikor nincsen a listában semmi. Ekkor az aktuális foglalás doboza lesz az első a listában. A doboz következő dobozra mutató címét `NULL`-ra állítjuk, a kincstári mutatónkat pedig beállítjuk ennek a doboznak a címére, hiszen mostantól ez lesz az első láncszem, innen fog kezdődni az adatsorunk. Pontosan ez a módosítás az, amiért kettős pointeraritmetikát kell alkalmazni. Ebben az esetben ugyanis nemcsak használjuk a kincstári mutatónkat, de még az értékét is módosítanunk kell.

Amennyiben nem az aktuális foglalás doboza a lista első eleme, úgy elindulunk végig a listán. Minden doboz esetén megnézzük, hogy az aktuálisan beszúrandó foglalás időpontja később van-e, mint az a foglalás, amelyet éppen vizsgálunk. Amennyiben ez teljesül, vándorolhatunk tovább a listán. Elmentjük az aktuális doboz címét és továbbpakoljuk a segédmutatót a következő dobozra. Amennyiben a következő doboz címe `NULL`, úgy a lista végére értünk, a végére kell beszúrni a dobozt. Ez speciális eset, hiszen a beszúrandó doboz következőre mutató pointerét `NULL`-ra kell állítani, valamint az előtte lévő doboz következőre mutató pointerét az aktuálisan beszúrandó doboz címére kell állítani. És ennyi, készen vagyunk, `break` utasítással kiléphetünk a teljes listavégigjárási folyamatból. Amennyiben ez nem teljesül, `continue` utasítással robogunk tovább a listán, előre.

Ha a forráskód 320. soráig eljutunk, az azt jelenti, hogy az első `if`-ágba nem léptünk be, tehát az új foglalás korábban van, mint az, amelyen éppen a munkamutatónk áll. Ekkor az aktuális foglalás dobozának következőre mutató címét a munkamutató aktuális címére állítjuk. Kérdés, hogy az a doboz, amely elé be kell szúrni az aktuális dobozt, a lista első eleme-e. Ezt ellenőrzi a 323. sorban lévő feltétel. Amennyiben nem, úgy nemes egyszerűséggel be kell állítani az előtte lévő doboz következő dobozra mutató címét az aktuálisan beszúrandó doboz címére és készen is vagyunk. Más a helyzet, ha az első elem elé kell beszúrnunk a dobozunkat. Ekkor a régi kincstári mutató értékét kell értékül adni a frissen beszúrandó doboz következő dobozra mutató címének, valamint módosítanunk kell a kincstári mutatónkat a frissen beszúrandó doboz címére. Ezzel becsatoltuk a lánc elejére az új dobozunkat, végeztünk.

Ha mindezekkel megvagyunk, a lista rendezett és kész, a foglalás el van mentve. Szólunk a felhasználónak és vége is a munkának.

Az `EddigiFoglalasok` eljárás célja és működése

Az `EddigiFoglalasok` eljárás bemenetként a kincstári mutatót kapja, melynek segítségével sorszámozva kiírja az eddig rögzített foglalások adatait a felhasználói felületre. Ez egy kifejezetten egyszerű eljárás, de a hónapok nevének kiiratását (szám/betű-konverzió) egy érdekes trükkel oldottam meg, amely érdemel pár szót.

A forráskód 17. sorában definiáltam egy 13 elemű, stringek tárolására alkalmas tömböt. A tömb nulladik elemének egy *dummy* "h"-betű abból a célból, hogy a 0-12-es indexelést kikerülve 1-12-ig tudjam használni és indexelni a tömböt a programban. Ennek a lényege, hogy ha a tömb első elemére hivatkozunk, akkor januárt ad vissza, ha a másodikra, akkor februárt és így tovább. Ez kiválóan alkalmazható a hónapok számának névre történő cseréléséhez, amit kihasznál az `EddigiFoglalasok` eljárás is. Az eljárás tulajdonképpen semmi mást nem csinál, csak a már megszokott módon bejárja a láncolt listát, majd sor-

számával együtt kiírja az egyes foglalások adatait egy `printf` segítségével. A trükk a `printf` argumentumában van, ahol előkerül az előbb ismeretett karaktertömb alkalmazása: `Honapok[(SegedMutato->Honap)]`. Látható, hogy minden kiiratás esetén az aktuális doboz hónap rekeszében található számával indexeljük a tömbünket, amely a fent említett okok miatt minden esetben a hónapok nevét fogja visszaadni.

Az `FoglaloCegekNeve` eljárás célja és működése

Az `FoglaloCegekNeve` eljárás bemenetként a kincstári mutatót kapja, melynek segítségével kiírja a felhasználói felületre a foglaló cégek neveit betűrendi sorrendben. Ez egy komplex feladat, melynek keretei között nemcsak végig kell járni a listánkat, hanem egy rendező algoritmust is meg kell valósítani. Az eljárás elején a `ListaMeret` változó segítségével megszámoljuk, hány foglalásunk van jelenleg a rendszerben. Ezek után megvizsgáljuk, van-e már foglalás a rendszerben, és ha nem, azonnal kilépünk az eljárásból, hiszen a semmit nem lehet rendezni, kiírni.

Ha van foglalás a rendszerben, akkor dolgunk van. Létrehozunk egy `CegNevek` tömböt, amelynek annyi rekesze lesz, ahány foglalás a rendszerben van. Ez a tömb trükkös, hiszen nem a cégek neveit fogja közvetlenül tartalmazni, hanem csak a láncolt listánk dobozainak címeit. Olyan címeket, amelyeken különböző cégnevek vannak. Mivel szélsőséges esetben előfordulhat, hogy minden egyes foglalás más céghez tartozik, fel kell készülnünk a legrosszabbra, ezért készítünk elő pontosan akkora tömböt, ahány foglalásunk a rendszerben van.

Ha ez megvan, ismét beállítjuk a munkamutatónkat a listánk elejére, majd elkezdünk végiggyalogolni a láncolt listánkon a szokásos módon. Mivel a `VanMar` változónkat nullára inicializáltuk, az első lépésnél egyből a 395. sorban kezdődő `if` ágba kerülünk, amely belerakja az aktuális dobozra mutató mutatót a `CegNevek` tömbbe, az `s` változót eggyel megnöveli és kocog tovább a láncolt listán. Ez világos, az első doboz cégneve még biztosan nem szerepel a cégnevek között, azt mindenképpen be kell rakni az egyedi cégnevek közé. Még egyszer célszerű hangsúlyozni: nem a cégnevet mentjük el, hanem csak a dobozra mutató mutatót, amellyel a cégnév is elérhető, természetesen.

Ha nem az első doboznál járunk, akkor minden egyes új doboznál meg kell vizsgálni, hogy az abban szereplő cégnév szerepel-e már a cégneves listánkban. Ezt a klasszikus `strcmp` függvény segítségével tesszük meg, és ténylegesen a `CegNevek` tömb rekeszeiben található címeken található cégneveket hasonlítjuk össze a munkamutatónk dobozának cégnevével. Amennyiben a munkamutató cégneve már szerepel a listában, úgy beállítjuk az indikátorváltozónkat egyre és egy `break` segítségével kiugrunk a `CegNevek` tömb rekeszeit bejáró `for`-ciklusból. Ha nem találunk egyezőt, akkor az indikátorváltozót nullára állítjuk és a már ismeretett módon (395-398. sorok) berakjuk a doboz címét a `CegNevek`tömbbe.

Ha ezzel végeztünk, megvannak azon dobozok címei, amelyek egyedi cégneveket tartalmaznak, de még nem névsorrendben. Rendezni kell őket tehát. Ehhez egy első végtelennek tűnő `while`-ciklust írtam (`while(1)`). Az iterációt úgy valósítottam meg, hogy egy idő után egészen biztosan egy olyan ágára fogunk futni, amely egy `break` utasítással kirúg minket a végtelen ciklusból. Az első ilyen ág az `s=1` esete. Ekkor csak egy cégnév van, nincs mit rendezni, végeztünk. Ha egynél több név van, akkor jön a tényleges rendezés. Egy `for` segítségével elindulunk a cégnevek mutatóit tartalmazó tömbünkön. Összehasonlítjuk az első két egymás mellett elhelyezkedő címen lévő cégnevet. Ha az első cégnév "nagyobb" (névsorban hátrébb van), mint a második, megcseréljük őket. Ehhez segítségként kell munkamutató, ahová elmentjük a csere idejére az első címet, majd az első helyére beírjuk a másodikat, végül pedig a munkamutatóból a másodikba beírjuk az elsőt. Mivel ebben a fázisban még nem sikerült végigmennünk a tömbbön csere nélkül, egy indikátorváltozóval

jelezzük, hogy még nincsen sorban a lista. Ezután megszakítjuk a `for`-ciklust, kezdődik elölről a végtelen `while`, ismét elindulunk a cégneves tömbbön, ha szükséges, akkor cserélünk és így tovább.

Mindezt addig folytatjuk, amíg a `for`-iteráció végig nem tud menni a tömbön csere nélkül. Ha ez összejött, akkor az indikátorváltozóval jelezzük, hogy sorban vannak a cégnevek, és ezek után egy speciálisan erre az esetre írt `if`-ággal kirúgjuk magunkat a végtelen ciklusból. Ezek után már csak ki kell írni a sorban elhelyezkedő cégneveket a cégnevek tömbben található mutatók segítségével.

A `FoglalasTorles` függvény célja és működése

A `FoglalasTorles` segítségével sorszám szerint lehetséges törölni az egyes foglalásokat az adatbázisból (a listából). Mivel ezúttal is előfordulhat, hogy a lista elejéről törlünk, ezért ismét mutatóra mutató mutatót szükséges alkalmazni.

A függvény első lépése a rendszerben lévő foglalások számának meghatározása a már ismeretett módon. Amennyiben úgy találjuk, hogy nincs foglalás a rendszerben, ezt közöljük a felhasználóval és kilépünk (visszaadunk 1-et, konvencionálisan). Ha van foglalás, bekérjük a felhasználótól törölni kívánt foglalás sorszámát. A `BemenetHiba` függvény segítségével a már ismertetett módon itt is megvizsgáljuk először, hogy formailag helyesen adta-e meg az inputot (értsd: számot adott-e meg). Ha ez rendben van, megnézzük, hogy van-e egyáltalán ilyen számú foglalás úgy, hogy megvizsgáljuk, nagyobb-e a beadott szám, mint a rendszerben lévő foglalások száma. Ha nagyobb, akkor nem jó értéket adott meg a felhasználó, szólunk neki és kirúgjuk a menüpontból.

Ha jó és értelmes sorszámot adott meg a felhasználó, a foglalást törölhetjük a listából. Ismét végigfutunk a szokásos módon a listánkon, és ha eljutottunk a törlendő foglaláshoz, törlünk. Először is közöljük a felhasználóval, hogy melyik foglalást töröljük, kiírjuk tehát a foglalás adait. Ezek után meg kell vizsgálnunk, hogy a törlendő foglalás a lista első eleme-e. Ha igen, módosítani kell a kincstári listamutatónkat a régi második/új első doboz címére. Ha nem az első elemet töröljük, akkor a törlendő dobozban található következő címet átmásoljuk a törlendő doboz előtt lévő doboz következő címébe (így iktatjuk ki a törlendő elemet a láncból). Ekkor még a törlendő adat a memóriában van, de mivel az már nem része a láncnak, nem tudnánk elérni. A felhasznált memóriaterületet a `free(SegedMutato);` paranccsal fel is szabadítjuk, szólunk a usernek, hogy a törlés sikeres volt és kirúgjuk magunkat a listavégigjáró ciklusból.

A `main ()` függvény működése

A `main ()` tulajdonképpen nem más, mint egy lehetőségválasztásos menü, amely a választástól függően meghívja a már ismertetett függvények és eljárások valamelyikét. A menü szerkezete annyira klasszikus, hogy én is egy interneten található verziót pofoztam át a saját feladatomra. A fórumot meg is hivatkozom a forráskód 512-es sorában. A függvény elején létrehozunk egy mutatót, amely alkalmas a definiált struktúránkra történő mutatásra. Ezt `NULL`-ra inicializáljuk, de rögtön utána az `AdatBeolvasas` függvény segítségével beolvassuk az adatfájl tartalmát és értékül adjuk neki a lista kezdőcímét. Ezek után bekérjük a felhasználótól, hogy mit szeretne csinálni. Lehetősége van választani 1-6 menüpontok közül. Itt is alkalmazzuk a már ismeretett ellenőrzést arra vonatkozóan, hogy számot ad-e meg a felhasználó. Ha nem, addig nem hagyjuk békén, amíg számot nem ad meg. A `switch-case` ágon pedig a bekért sorszámtól függően meghívjuk az adott feladat ellátására alkalmas függvényeket, eljárásokat.

Köszönetnyilvánítás

A programot a konzultációkon elhangoztak, valamint az eddigi programozási tapasztalataim alapján magam írtam, amely a legjobb tudásom szerint megfelelően működik. A program megvalósítása tartalmaz ugyanakkor egy-két olyan gyakorlatiasabb fogást, praktikát és megoldást, amelyek túlmutatnak az alapszintű programozási ismereteken. Elgondolásilag, nem szintaktikailag. Ezen megoldásokért (például a `BemenetHiba` függvény ötletéért, a foglaltság ellenőrzésére szolgáló segéd-indikátortömb használatának javaslatáért, valamint a fájl olvasásához/írásához használt `format string` alkalmazásáért) nagy köszönettel tartozom barátomnak és volt kollégámnak, Budai Tamásnak, akivel a projektfeladat elkészítése során többször is konzultáltam.

Budapest, 2017. november 4.