STDIN bekérés és input ellenőrzés getline() és fgets() függvényekkel

```
int getline(char s[], int lim) {
	int c, i;
	for (i = 0; i < lim && (c = getchar()) != EOF && c != '\n'; ++i)
		s[i] = c;
	s[i] = '\0';
	while (c != EOF && c != '\n')
		getchar();
	return(i);
}
```

[1. példa][1]

*A **getline*** alkalmazása kettő vagy több összefüggő bekérés esetén problémás, mivel üres sornál kilép a ciklusból, vagyis nem kezeli hibának az üres sort és tovább ugrik a következő bekérésre/ciklusra. Ezért az üres sor tesztelését, amennyiben szükséges (pl. 1. bekérés: üres sornál lépjen ki) rögtön az első bekérés ciklusa után kell végrehajtani és kiléptetni a programot [**(strlen(input1) == 0 >> break]** a bekéréseket összefogó főciklusból, az első **while(1)**-ból.

Ha adott (feladat szerint vagy külön bekéréssel) n-számú bekérést kell végrehajtanunk, akkor csak a bekérési ciklusonkénti üres soros tesztelés szükséges, ami a fenti példához hasonlóan szintén egy külső, feltételnélküli ciklus [**while(1)**] szerint történik. [**strlen(inputN) == 0 >> continue].** Itt azonban hibás bemenet miatt folytatni kell a bekérést.

Az inputokat C-ben szokás ellenőrizni is. A 2. és 3. példák az egész szám (integer) ellenőrzésre hoznak egy-egy példát.

```
int egesz_e(char s[]) {
	int i;
	if (strlen(s)!=1 && s[0] == 48) //0 lehet, de 0-val nem kezdődhet a szám
		return -1;
	for (i = 0; s[i] != '\0'; i++)
		if (!isdigit((unsigned char)s[i]))
			return -2;
	return 0;
}
```

[2. példa]

Egész szám ellenőrző függvény. Ha több számjegyből áll és az első 0, akkor hibás (opcionálisan). Ha 0, akkor helyes, illetve akkor is helyes, ha csak pozitív egész számjegyeket adtunk meg. Ha negatív egész számokat is elfogadunk akkor a következőképpen módosítható a függvény:

[1] Bauer Péter: Programozás I-II. – Universitas-Győr Nonprofit Kft. – Győr, 2010.

```c
int egesz_e(char s[]) {
	int i;
	if (strlen(s)!=1 && s[0] == 48) //0 lehet, de 0-val nem kezdődhet a szám
		return -1;
	if ((s[0] == '-' && strlen(s) > 1) || isdigit((unsigned char)s[0])) {
		for (i = 1; s[i] != '\0'; i++) {
			if (s[0] == '-'&& s[1] == 48)
				return -1;
			if (!isdigit((unsigned char)s[i]))
				return -2;
		}
	}
	else
		return -2;
	return 0;
}
```

[3. példa]

A fenti függvény (3. példa) megengedi a negatív egészeket is (negatív előjel és számjegy követi), ugyanakkor figyel a negatív előjel utáni esetleges 0-ákra is: [**s[0] == '-' && s[1] == 48]**

```c
	while (1) {
		printf("Adja meg az input1-et: ");
		while (getline(input1, BUF)) {
			if (egesz_e(input1) == -1) {
				printf("Nem kezdődhet 0-val a szám! Újat: ");
				continue;
			}
			else if (egesz_e(input1) == -2) {
				printf("Nem érvényes egész szám! Újat: ");
				continue;
			}
			else
				break;
		}
		if (strlen(input1) == 0)
			break;
		while (1) {
			printf("Adja meg az input2-őt: ");
			while (getline(input2, BUF)) {
				if (egesz_e(input2) == -1) {
					printf("Nem kezdődhet 0-val a szám! Újat: ");
					continue;
				}
				else if (egesz_e(input2) == -2) {
					printf("Nem érvényes egész szám! Újat: ");
					continue;
				}
				else
					break;
			}
			if (strlen(input2) == 0) {
				printf("Adjon meg egy érvényes értéket: ");
				continue;
			}
			else
				break; } }
```

[4. példa]

A getline-val történő összefüggő inputbekérés pszeudo kóddal:

- **while(1)**
  - **while(getline,input1)**
    - **if**
    - **else if**
    - **else break**
  - **if strlen(input1)==0 >> break**
  - **while(1)**
    - **while(getline,input2)**
      - **if**
      - **else if**
      - **else break**
    - **if strlen(input2)==0 >> continue**
    - **else break**

Ha kettőnél több összefüggő bekérést kell végrehajtani, akkor az input2-t illetve a keretciklusát [while(1)] kell tovább másolni. Ilyenkor figyelni kell, hogy input2-t átírjuk 3-ra 4-re stb. illetve a megfelelő ellenőrző függvények alkalmazását sem szabad figyelmen kívül hagyni!

***fgets** függvény használata* esetén, nem kell több feltétel nélküli keretciklus, mivel a függvény nem lép ki a ciklusból üres sorra, illetve csak akkor teszi ezt, ha kényszerítjük azáltal, hogy megadjuk a kilépés feltételeit. Az **fgets** függvény a puffer változóhoz egy extra lánczáró ’\0’-át rendel, ezért hogy a későbbiekben ne legyenek problémák a feldolgozandó karakterlánc hosszával, csonkoljuk őket: [**input[strlen(input1)-1]=’\0’]**. Amennyiben az első inputnál üres sorral szeretnénk kilépni, akkor még a csonkolás előtt kell tesztelni: **[input1[0] == ’\n’ >> goto end]**, mert azután már csak a **[strlen(input) == 0]**-val detektálhatjuk az üres sort. Összefüggő, többes bekérés esetén tehát az **fgets** előnye, hogy csak egy fő keretciklus szükséges, hátránya viszont, hogy a puffer karakterláncokat csonkolni kell, illetve a fő keretciklusból üres sor ütése után **goto**-val lehet csak kilépni. A **getline** előnye ezzel szemben az, hogy nem kell **goto**-t használni, hátránya viszont az, hogy a bekérési ciklusokon kívül kell detektálni az üres sort, ezért több keretciklusra is szükség van, ami pedig növeli a kód komplexitását.

```
while (1) {
	printf("Adja meg az input1-et: ");
	while (fgets(input1, BUF, stdin)) {
		if (input1[0] == '\n') {
			//n--;
			goto end;
		}
		input1[strlen(input1) - 1] = '\0';
		if (egesz_e(input1) == -1) {
			printf("Nem kezdődhet 0-val a szám! Újat: ");
			continue;
		}
		else if (egesz_e(input1) == -2) {
			printf("Nem érvényes egész szám! Újat: ");
			continue;
		}
		else
			break;
	}
```

```
		printf("Adja meg az input2-őt: ");
		while (fgets(input2, BUF, stdin)) {
			input2[strlen(input2) - 1] = '\0';
			if (strlen(input2) == 0) {
				printf("Adjon meg egy érvényes értéket: ");
				continue;
			}
			if (egesz_e(input2) == -1) {
				printf("Nem kezdődhet 0-val a szám! Újat: ");
				continue;
			}
			else if (egesz_e(input2) == -2) {
				printf("Nem érvényes egész szám! Újat: ");
				continue;
			}
			else
				break;
		}
		//n++;
		//if (n == 3)
			//break;
	}
	end:
```

[5. példa]

Az üres soros kilépés alternatívájaként a keretciklus végére, amennyiben bekérési limitet határoztunk meg [**n**], egy számlálót tehetünk [**n++**], mely segítségével egy bizonyos limit/kör után kiléptethetjük a programot. Amennyiben mindkét kilépési módszerre szükség van és az adatokat tömbökben tároljuk, a főciklusból való kiugrás előtt dekrementálni kell a számlálót [**n--**].

Az fgets-vel történő többszintű input bekérés pszeudo kóddal:

- **while(1)**
  - **while(fgets,input1)**
    - **if '\n' >> goto**
    - **strlen(input1)-1='\0'**
    - **if**
    - **else if**
    - **else break**
  - **while(fgets,input2)**
    - **strlen(input2)-1='\0'**
    - **if strlen(input2)==0 >> continue**
    - **if**
    - **else if**
    - **else break**

Karakterláncok tárolása dinamikus memóriafoglalással

```c
int main(){

	char **str, input[BUF];
	int n = 5, i = 0;

	//memória foglalás: hány karakterláncra lesz szükség
	str = malloc(sizeof(char*)*n);

	printf("Adjon meg karakterláncokat!\n");
	while (getline(input, BUF)) {
		/*validáló függvény helye
		                                                */
		//memória foglalás: hány karakterből fog állni
		str[i] = malloc(sizeof(char)*strlen(input) + 1);
		strcpy(str[i], input);
		i++;
		if (i == n)
			break;
	}
	if (i < n) //ha üres sorral korábban ugrik ki
		n = i;

	//teszt
	printf("A megadott karakterláncok:\n");
	for (i = 0; i < n; i++)
		printf("%s\n",str[i]);

	//memória felszabadítás
	for (i = 0; i < n; i++)
		free(str[i]);

	free(str);

	return 0;
}
```

[6. példa]

A fenti példában most az [**n**], vagyis hogy pontosan hány darab karakterláncnak szeretnénk a memóriában dinamikusan helyet foglalni, konstans értékként szerepel. A foglalandó/allokálandó memória meghatározása ugyanis történhet még táblázat sorok/oszlopok alapján (lásd 8-9. példa), vagy más futási időben kapott/számolt érték megadásával (egésszel visszatérő függvény, input, parancssori argumentum).

A fenti programban lehetőség van *<n* karakterlánc megadására is, hiszen pl. üres sor leütésével *<n* kör után ugrik ki a program a ciklusból. Ezt az [**if(i<n)n=i;**] kezeli. Erre azért van szükség, hogy a memória felszabadításánál, pontosan annyi felszabadítási parancs [**free()**] történjen mint amennyi foglalás **[malloc()**] történt. Ellenkező esetben a program hibával tér vissza. A fenti példától eltérően a feladathoz rendszerint tartozik legalább egy validáló függvény is, pl. a már bemutatott egész szám ellenőrző, de általában ennél bonyolultabbakról van szó mint pl. érvényes név-e, neptun-kód-e, dátum-e, rendszám-e stb.

A fenti megoldás ekvivalens a char *str[5] -vel. Ebben az esetben azonban csak a karakterláncokat alkotó karaktereknek szükséges memóriát foglalni [**str[i] = malloc(sizeof(char)*strlen(input) + 1);**]. Ez a megoldás csak akkor használható, ha előzetesen ismerjük a szükséges karakterláncok számát. Olyan esetekben viszont, amikor a karakterláncok száma és hossza is kérdéses a **str jöhet szóba.

Ugyanakkor, ha táblázatban szereplő karakterláncoknak szeretnénk memóriát foglalni már nem dupla, hanem tripla mutatóra van szükségünk: [***str]. Ezek a táblázatok rendszerint szöveges (.txt) vagy adatbázisokban rugalmasan kezelhető fájlokban (.csv) fordulnak elő. Illetve a szabvány bementről (stdin) kerülhetnek táblázatos formába, melyeket vagy a szabvány kimenetre, vagy egy fájlba ír ki a programunk (11-14. old.).

```c
#define BUF 500+1
#define SOR 4
#define OSZLOP 3

int main() {

	char ***str, *fnev = "c:\\temp\\viragbolt1.txt", fbuf[BUF], *rec;
	int i, j, sor = 0, oszlop = 0;
	FILE *fp;

	//fájl megnyitás
	fp = fopen(fnev, "r");
	if (fp == NULL) {
		perror("Hiba a fájl megnyitásakor: ");
		exit(1);
	}

	//memória foglalás soroknak
	str = malloc(sizeof(char**)*SOR);

	//memória foglalás oszlopoknak = egy sorban hány karakterlánc van
	for (i = 0; i < SOR; i++)
		str[i] = malloc(sizeof(char*)*OSZLOP);

	while (fgets(fbuf, BUF, fp) != NULL) {
		fbuf[strlen(fbuf) - 1] = '\0';
		rec = strtok(fbuf, "\t");
		while (rec != NULL) {
			//memória foglalás: hány karakterből fog állni
			str[sor][oszlop] = malloc(sizeof(char)*strlen(rec)+1);
			strcpy(str[sor][oszlop], rec);
			oszlop++;
			rec = strtok(NULL, "\t");
		}
		oszlop = 0;
		sor++;
	}

	//kiíratás
	for (i = 0; i < SOR; i++) {
		for (j = 0; j < OSZLOP; j++)
			printf("%s\t", str[i][j]);
		printf("\n");
	}
```

```
	//memória felszabadítás
	for (i = 0; i < SOR; i++) {
		for (j = 0; j < OSZLOP; j++) {
			free(str[i][j]);
		}
		free(str[i]);
	}
	free(str);
	str = NULL;

	//fájl bezárás
	fclose(fp);

	return 0;
}
```

[7. példa]

Ilyen, adott sor és oszlop szám esetében, ezzel ekvivalens megoldás a *str[SOR][OSZLOP], persze azokban az esetekben ha dinamikusan, változók segítségével kell memóriát foglalni, a tripla pointer a megoldás. A sorok illetve oszlopokok megmérésére önálló függvényeket is alkalmazhatunk, így például:

```
int sorok(FILE *fp) {
	int sor = 0, aktpoz = ftell(fp);
	char fbuf[BUF];
	fseek(fp, 0L, SEEK_SET);
	while (fgets(fbuf,BUF,fp)!=NULL)
		sor++;
	fseek(fp, aktpoz, SEEK_SET);

	return sor;
}
```

[8. példa]

Fontos, hogy mielőtt az fgets() függvényt meghívjuk, amit most fájlból való tartalom beolvasásra használtunk (*vegyük észre, hogy a harmadik argumentum nem stdin, hanem fp*), a fájlt az elejére forgassuk. Majd a ciklus lefutása után – ami a fájl végére „tekeri” a fájlmutatót – forgassuk vissza az eredeti pozíciójába [**aktpoz = ftell(fp)**], ami általában a fájl eleje. A függvény ahogy végig megy a fájl tartalmán a sortörések számával, azaz lényegében a sorok számával tér vissza.

```
int oszlopok(FILE *fp, char delim) {
	int check = 0, i, aktpoz = ftell(fp);
	char fbuf[BUF], *rec;
	fseek(fp, 0L, SEEK_SET);
	fgets(fbuf, BUF, fp);
	for (i = 0; fbuf[i] != '\0'; i++)
		if (fbuf[i] == delim)
			check++;
	fseek(fp, aktpoz, SEEK_SET);

	return check + 1;
}
```

[9. példa]

Az oszlopok számának meghatározásakor a sorok számának méréséhez hasonlóan járunk el. Azzal a különbséggel, hogy itt nem a sortöréseket számoljuk, hanem az egyes sorokon belül az elválasztó fehérkaraktereket [**char delim = '\t'**], ami esetünkben a '\t' tabulátor. A visszatérő egész értékhez hozzá kell adni egyet, mivel a sorban az utolsó karakterlánc után – ami egyben az utolsó oszlopot is reprezentálja – a sortörést jelölő lánczáró '\0' következik. Itt azonban hangsúlyozni kell, hogy szabályos táblázatból indulunk ki, vagyis olyan alakzatból, aminek minden sora egyenlő számú adatot vagy értéket -legyen akár 0 vagy „"- tartalmaz. Egyszóval minden sora azonos számú oszlopból áll. Éppen ezért a sorokat számláló függvényhez képest, itt elegendő csak az első sort vizsgálni: [while (fgets(fbuf,BUF,fp)!=NULL)] helyett [fgets(fbuf,BUF,fp);].

Tipikus táblázat beolvasó feladat a csv fájlok kezelése, ott az elválasztó fehérkarakter rendszerint a ';'.

Egész decimális számok (integer) tárolása táblázatos formában illetve tömbben

```
int main() {

	int **arr, i, j;
	//memória foglalás
	arr = (int*)malloc(sizeof(int*)*5); //sorok
	for (i = 0; i < 5; i++)
		arr[i] = (int**)malloc(sizeof(int) * 3); //oszlopok

	//értékadás, kiíratás
	for (i = 0; i < 5; i++) {
		for (j = 0; j < 3; j++) {
			arr[i][j] = j;
			printf("%d ", arr[i][j]);
		}
		printf("\n");
	}
	//memória felszabadítás
	for (i = 0; i < 5; i++)
		free(arr[i]);
	free(arr);

	return 0;
}
```

[10. példa]

Egész számokat tartalmazó táblázatoknak, mátrixoknak a fenti módon foglalhatunk memóriát, adhatunk értéket illetve szabadíthatjuk fel a memóriát a végén. Erre szintén akkor lehet szükség, ha a sorok illetve oszlopok számát csak futásidőben ismerjük meg. A fenti példában csak az egyszerűség kedvéért kaptak konstans értéket a mátrix paraméterei (5 sor és 3 oszlop). Amennyiben csak arra van szükségünk, hogy egész számokat tömbben tároljunk, és a tömb méretét futás időben, pl. egy változó értéke alapján kapjuk meg, úgy célszerűbb a mutató használata.

```
int main() {

	int *arr, i;
	//memória foglalás
	arr = (int*)malloc(sizeof(int)*5);

	//értékadás, kiíratás
	for (i = 0; i < 5; i++) {
		arr[i] = i;
		printf("%d ", arr[i]);
	}
	printf("\n");
	//memória felszabadítás
	free(arr);

	return 0;
}
```

[11. példa]

Adatok tárolása struktúrában

```c
typedef struct {
	char *nev;
	int id;
}ugyfel;

int main() {

	int i;

	ugyfel *tag = (ugyfel*)malloc(sizeof(ugyfel)*5);

	//értékadás az 5 db id-nek és a neveknek
	for (i = 0; i < 5; i++) {
		tag[i].id = i;
		tag[i].nev = (char*)malloc(sizeof(char)*strlen("noname")+1);
		strcpy(tag[i].nev, "noname");
		printf("id: %d nev: %s\n", tag[i].id, tag[i].nev);
	}
	printf("\n");

	//memória felszabadítás
	for (i = 0; i < 5; i++)
		free(tag[i].nev);

	free(tag);

	return 0;
}
```

[12. példa]

Amennyiben csak futásidőben ismerjük meg, hogy mennyi összefüggő adatra lesz szükség, úgy célszerű a típusdefiníció szerint eljárni [typedef struct] és dinamikusan lefoglalni a memóriát [ugyfel *tag = (ugyfel*)malloc(sizeof(ugyfel)*5);]. A struktúra tagoknak is foglalhatunk memóriát, ha arra van szükség [tag[i].nev = (char*)malloc(sizeof(char)*strlen("noname")+1);]. A példában az egyszerűség kedvéért konstans (5) érték szerepel, de persze a dinamikus memória foglalásnak az eddig ismertetetett esetekben akkor van értelme, ha futásidőben jutunk a lefoglalni kívánt memória nagyságának információjához. Ezzel ellentétben az alábbi példa akkor használható, ha minden adatnak ismerjük a nagyságát, hosszúságát.

```c
struct tabla {
	char nev[16];
	int id;
}ugyfel[5];

int main() {

	int i;

	//értékadás az 5 db id-nek és a neveknek
	for (i = 0; i < 5; i++) {
		ugyfel[i].id = i;
		strcpy(ugyfel[i].nev, "noname");
		printf("id: %d nev: %s\n", ugyfel[i].id, ugyfel[i].nev);
	}
	printf("\n");

	return 0;
}
```

[13. példa]

Láncolt listák

Egyirányú láncolt listák

Az első egyirányú láncolt lista példában, a program a szöveges fájl sorainak beolvasása után egy a lista elejére mutató horgonnyal tér vissza. Ezután nyomtatja ki a listát, abban a sorrendben, ahogy az a szöveges fájlban is szerepelt. A program végén a lefoglalt memória felszabadításra kerül.

```
typedef struct lista {
	char sor[MAX];
	struct lista *kov;
}lista;

FILE *fp;
char *fnev = "test0.txt", fbuff[MAX];

//fájl megnyitása, ellenőrzése
fp = fopen(fnev,"r");
if (!fp) {
	perror("Hiba a fájl megnyitásakor!\n");
	return fp;
}

//fájl soronként való beolvasása
Lista *uj, *horgony = NULL, *utolso = NULL;

while (fgets(fbuff,MAX,fp)!=NULL) {
	uj = (Lista*)malloc(sizeof(Lista));
	if (uj == NULL)
		break;
	char *poz;
	if ((poz = strchr(fbuff, '\n')) != NULL) //új sor karakter 'törlése'
		*poz = '\0';
	strcpy(uj->sor, fbuff);
	uj->kov = NULL;
	if (horgony == NULL)
		horgony = uj;
	else
		utolso->kov = uj;
	utolso = uj;
	}

fseek(fp, 0L, SEEK_SET); fclose(fp);

//lista kiíratása
while (horgony) {
	printf("%s\n", horgony->sor);
	horgony = horgony->kov;
}

//memória felszabadítása
while (horgony) {
	horgony = horgony->kov;
	free(horgony);
}
```

[14.példa]

A következő példa szintén egy egyszerű egyirányú lista, amely mindig a lista elejére szúrja az új elemet, ezért az eredeti elemsorrend fordítottját eredményezi. A változatosság kedvéért a beolvasandó adatsor legyen egy egészekből álló tömb. Amennyiben az alábbi program egy függvényt valósítana meg, úgy a visszatérési értéknek az eleje mutató kellene, hogy legyen! (… return eleje;)

```
typedef struct lista {
	int elem;
	struct lista *kov;
}lista;

int szamok[] = {4,91,23,19,88,71,29,16,18};

lista *eleje = NULL;
int i, len = sizeof(szamok) / sizeof(szamok[0]);

//lista feltölt
for (i = 0; i < len; i++) {
	lista *uj = (lista*)malloc(sizeof(lista));
	uj->kov = eleje;
	uj->elem = szamok[i];
	eleje = uj;
}
//return eleje;

//lista kiíratása

lista *iter;
for (iter = eleje; iter != NULL; iter = iter->kov)
	printf("%d ", iter->elem);
```

[15.példa]

Kétirányú láncolt lista

Az első lépés a mutatók inicializálása NULL-ra. Ennek értelmében három mutatóról van szó, az első elemre, az aktuálisra és a listában az előzőre mutatóról. A második lépésben az adatok láncolt listába való beolvasása történik. A harmadikban pedig a felhasználásuk, pl. kiíratásuk a képernyőre.

```
//struktúra láncolt listához
typedef struct akta {
	char nev[32];
	struct akta *kov;
}akta_t;

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

	char buf[BUF];
	int i = 0;
	akta_t *elso = NULL;
	akta_t *aktualis = NULL;
	akta_t *elozo = NULL;

	printf("Input [name] : ");
	while (getline(buf,BUF)) {

		aktualis = (akta_t*)malloc(sizeof(akta_t));
```

```
		if (elso == NULL)
			elso = aktualis;

		if (elozo != NULL)
			elozo->kov = aktualis;

		strcpy(aktualis->nev, buf);

		aktualis->kov = NULL;
		elozo = aktualis;

		printf("Input [name] : ");
	}
	aktualis = elso;
	while (aktualis != NULL) {

		printf("[%d]. %s\n", i, aktualis->nev);
		i++;

		elozo = aktualis;
		aktualis = aktualis->kov;

		free(elozo);
		elozo = NULL;
	}
	elso = NULL;

	return 0;
}
```

[16. példa]

Láncolt lista buborék rendezése

```
typedef struct lista{
	int szam;
	struct lista *kov;
}lista;

// Csere segédfüggvény
void swap(lista *a, lista *b) {
	int temp;
	temp = a->szam;
	a->szam = b->szam;
	b->szam = temp;
}
// Lista buborék rendezése
lista *bubble(lista *Lista) {
	int csere;
	lista *uj, *utolso = NULL;
	if (Lista == NULL)
		return Lista;

	do {
		csere = 0;
		uj = Lista;
		while (uj->kov != utolso) {
			if (uj->szam > uj->kov->szam) {
				swap(uj, uj->kov);
				csere = 1;
			}
```

```
                        uj = uj->kov;
                }
                utolso = uj;
        } while (csere);

        return(Lista);
}
// Lista létrehoz
lista *listafeltolt() {
        //mindegy, hogy lista eleje vagy vége, mert rendezve lesz!
        lista *eleje = NULL;
        int szamok[] = { 4,91,23,19,88,71,29,16,18 };
        int i, len = sizeof(szamok) / sizeof(szamok[0]);
        for (i = 0; i < len; i++) {
                lista *uj = (lista*)malloc(sizeof(lista));
                uj->kov = eleje;
                uj->szam = szamok[i];
                eleje = uj;
        }
        return eleje;
}

int main(){

        setlocale(LC_ALL, "hun");

        lista *l = listafeltolt();
        lista *iter;

        printf("\nA lista:\n");
        for(iter = l; iter != NULL; iter=iter->kov)
                printf("%d ", iter->szam);
        printf("\nA lista növekedő sorrendben:\n");
        lista *b0 = bubble(l);
        for (iter = b0; iter != NULL; iter = iter->kov)
                printf("%d ", iter->szam);
        printf("\n");

        return 0;
}
```

[17. példa]

Táblázatkezelés (stdin és FILE *fp)

Táblázat adatainak bekérése szabvány bemenetről fix sor és oszlop szám esetén

Táblázat adatait szabvány bemenetről (stdin) bekérhetjük struktúrába, illetve 3D tömbbe (egészek esetén: 2D). A struktúrában bekérés, csak annyiban egyszerűbb, hogy legfeljebb csak a struktúra tagok számának és hosszuknak kell memóriát foglalni (előbbi ekvivalens a sorok számával (elem *sor… typedef + malloc). Ebben az esetben azonban csak az utóbbira van szükség, hiszen adott sor és oszlopszámmal dolgozunk. [#define M 3 és #define N 3]. A következő megoldások annyiban lehetnek komplexek, hogy az elemeket a ciklus különböző lépéseiben (m=0…2) lehet csak validálni>>menteni (strcpy). Akkor alkalmazható jól, ha eltérő típusú vagy formájú adatokat akarunk validálni.

```
struct elem{
	char *oszlop0, *oszlop1, *oszlop2;
}sor[M];

int main() {
	while (1) {
		printf("Irja be a %d. sor első adatát: ", n + 1);
		while (1) {
			while (getline(input, BUF)) {
				//validáló függvény helye >> ha valid, akkor:
				if (m == 0) {
				sor[n].oszlop0 = malloc(sizeof(char)*strlen(input) + 1);
				strcpy(sor[n].oszlop0, input);
				}
				else if (m == 1) {
				sor[n].oszlop1 = malloc(sizeof(char)*strlen(input) + 1);
				strcpy(sor[n].oszlop1, input);
				}
				else if (m == 2) {
				sor[n].oszlop2 = malloc(sizeof(char)*strlen(input) + 1);
				strcpy(sor[n].oszlop2, input);
				}
				break;
			}
			if (strlen(input) == 0) {
				printf("Üres sor nem lehet! Kötelező adatot megadni: ");
				continue;
			}
			n++;
			if (n == N)
				break;
			printf("Irja be a %d. sor %d. adatát: ", m + 1, n + 1);
		}
		n = 0; m++;
		if (m == M) break;
	}
	//memória felszabadítás
	for (i = 0; i < N; i++) {
		free(sor[i].oszlop0);
		free(sor[i].oszlop1);
		free(sor[i].oszlop2);
	}
	return 0;	}
```

[18. példa]

A másik megoldás, ha 3D tömbökbe olvassuk be a táblázat tartalmát. Ezt megtehetjük tripla mutatók segítségével is (***adat), de ismert sor és oszlopok esetén célszerűbb 2D mutató tömb (*adat[M][N]) felhasználásával az adatok tárolását végrehajtani.

```
int main() {

	char *adat[M][N], input[BUF];
	int n = 0, m = 0, i, j;

	while (1) {
		printf("Irja be a %d. sor első adatát: ", n + 1);
		while (1) {
			while (getline(input, BUF)) {
				//validáló függvény helye, ha másféle adatok, ell.: m = 0…2
				//ha valid, akkor:
				adat[n][m] = (char*)malloc(sizeof(char)*strlen(input)+1);
				strcpy(adat[n][m],input);
				break;
			}
			if (strlen(input) == 0) {
				printf("Üres sor nem lehet! Kötelező adatot megadni: ");
				continue;
			}
			n++;
			if (n == N)
				break;
			printf("Irja be a %d. sor %d. adatát: ", n + 1, m + 1);
		}
		n = 0;
		m++;
		if (m == M)
			break;
	}

	//kiíratás
	printf("%s\t%s\t%s\n", "0.oszlop", "1.oszlop", "2.oszlop");
	for (i = 0; i < M; i++) {
		for (j = 0; j < N; j++)
			printf("%s\t",adat[i][j]);
		printf("\n");
	}

	//memória felszabadítás
	for (i = 0; i < M; i++)
		for (j = 0; j < N; j++)
			free(adat[i][j]);

	return 0;
}
```

[19. példa]

A tripla mutató (***adat) használata pedig akkor javasolt, ha a táblázat sor és oszlop paramétereit változó értékek szerint kaphatjuk meg, pl. szabvány bementről való bekéréssel, stb. A memória lefoglalása és felszabadítása a 7. példa alapján hajtható végre.

Táblázat adatainak beolvasása fájlból és frissítésük stdin-ről

Amikor a bekérés összefüggő adatok, táblázatok fájlból történő beolvasása után történik alapvető probléma a memóriafoglalás, ugyanis legtöbbször nem tudjuk előre, hogy mennyi adat fogadására kell felkészülni a szabvány bemeneten. Abban az esetben, ha csak az a feladat, hogy a fájl adatainak frissítése után kiírassuk az új táblázatot a képernyőre, nem kell egyetlen tömbben vagy struktúrában tárolni az adatokat, hanem a fájl tartalmat kétszer, a frissítés előtt és után íratjuk ki. (fgets + printf) - itt ugyanaz a buffer is felhasználható (felülírás történik).

```
printf("A fájl tartalmának kiíratása:\n");
while (fgets(fbuf, BUF, fp) != NULL)
	printf("%s", fbuf); //'\n' nem szükséges, mivel nem csonkoltuk az fbuf-t

/*
itt a bekérés helye, lásd pl. 2. oldal!
Ebben az esetben a bekérés kiegészül, a fájlba író függvénnyel, így pl.:
*/
fprintf(fp, "%s\t%s\n", adat1[i], adat2[i]);
i++;
/*, melynek a bekérés első ciklusának (1. while(1)) legvégén kell elhelyezkedni, az
inkrementációs utasítás fölött!
*/

printf("A fájl frissített tartalmának kiíratása:\n");
fseek(fp, 0L, SEEK_SET); //a fájlmutatót a fájl elejére kell visszaállítanunk!
while (fgets(fbuf, BUF, fp) != NULL)
	printf("%s", fbuf);
```

[20. példa]

Azonban vannak esetek, amikor arra van szükség, hogy a táblázat adatait egyetlen tömbben, struktúra tömbben tároljuk. Ilyen eset például az, amikor a frissített tartalmat időrendi, alfabetikus vagy valamilyen más sorrendben szeretnénk kiíratni a képernyőre. Problémát itt is az jelenti, hogy nem tudjuk előre meghatározni a szabvány bemenetről érkező adatok mennyiségét. Vagyis, ha nem adunk meg egy valamekkora limitet, akkor elvileg bármennyi adat érkezhet. Ezért a dinamikus memória foglalás, legfeljebb csak a reallokálás [realloc()] segítségével valósulhat meg. Azonban jobb megoldásnak tűnik, hogy konstans méretű mutató vagy struktúra tömbökkel dolgozunk.

```c
int main(void) {

	system("chcp 1250");
	setlocale(LC_ALL,"hun");

	char fbuf[BUF], *rec, *adat[M][N], input1[BUF], input2[BUF];
	int m = 0, n = 0, sor, i, j;
	FILE *fp;

	//fájl megnyitása
	fp = fopen("c:\\temp\\tablazat.txt","r+");
	if (fp == NULL) {
		perror("Hiba a fájl megnyitásakor: ");
		exit(1);
	}

	//fájl tartalmának a beolvasása és kiíratása
	printf("A fájl tartalmának a kiíratása:\n");
	while (fgets(fbuf, BUF, fp) != NULL)
		printf("%s", fbuf);


	//bekérés, a fájl tartalmának frissítése
	while (1) {
		printf("Input1: ");
		while (getline(input1, BUF)) {
			fprintf(fp, "%s\t", input1);
			break;
		}
		if (strlen(input1) == 0)
			break;
		while (1) {
			printf("Input2: ");
			while (getline(input2, BUF)) {
				fprintf(fp, "%s\n", input2);
				break;
			}
			if (strlen(input2) == 0) {
				printf("Nem lehet üres sor! ");
				continue;
			}
			else
				break;
		}
	}

	// a fájl frissített tartalmának a kiíratása
	printf("A fájl frissített tartalma:\n");
	fseek(fp, 0L, SEEK_SET);
	while (fgets(fbuf, BUF, fp) != NULL) {
		fbuf[strlen(fbuf) - 1] = '\0';
		rec = strtok(fbuf, "\t");
		while (rec != NULL) {
			adat[m][n] = (char*)malloc(strlen(rec) + 1);
			strcpy(adat[m][n], rec);
			//kiíratás
			printf("%s\t", adat[m][n]);
			n++;
			rec = strtok(NULL, "\t");
		}
		n = 0;
```

```
			m++;
			printf("\n");
		}

		//lefoglalt memória lefoglalása
		for (i = 0; i < m; i++)
			for (j = 0; j < N; j++)
				free(adat[i][j]);

		//fájl bezárása
		fclose(fp);

		return 0;
}
```

[21. példa]

A fenti megoldás tehát alapvetően három részre osztható. *Az első a fájl eredeti tartalmának a kiíratása*, *második a fájl új adatokkal való feltöltése* (bemenetről 1 sorban 2 adat!), és a *harmadik a fájl, új adatokkal frissített tartalmának a kiíratása* és 2D mutató tömbbe való elmentése. Ez utóbbira akkor lehet szükség, ha még valamilyen, pl. sorba rendező műveletet is el kell végeznünk a fájl adataival. A példában azonban ettől eltekintettünk, a buborék rendezés bemutatására később kerül sor.

Validáló függvények

A validáló vagy ellenőrző függvényeket általában közvetlenül az input bekérés után hívjuk meg, de alkalmazzuk még fájlok, tömbök, struktúrák, láncolt listák tartalmi ellenőrzésére is. Egyszerűbb validáló függvényekre láttunk már példát (2-3. példa). Azokhoz hasonló egyszerűbb függvények az alfanumerikus karaktereket, csak betűket és hosszúságot, hexadecimális értéket ellenőrző függvények.

```
int szoveg_ell(char s[]) {
	int i;
	const int maxlen = 6;
	for (i = 0; s[i] != '\0'; i++)
		if (!isalpha((unsigned char)s[i]) && !isspace((unsigned char)s[i]))
			return 1;
	if (strlen(s) > maxlen)
		return -1;
	return 0;
}
```

[22. példa]

A fenti kódrészlet egy egyszerű szövegellenőrző függvény, mely a szabvány bemenetről érkező szöveget ellenőrzi, aszerint hogy valóban csak betűket (magyar ábécé betűit is) és fehérkaraktereket (szóköz, tabulátor) tartalmaz-e illetve, hogy nem haladja-e meg a 6 karakter hosszúságot.

A bonyolultabb esetek közé tartoznak, a dátumot (szökőévvel), rendszámot, ip-címet vagyis valamilyen speciális, de meghatározott formájú azonosítókat validáló függvények.

```
//rendszám ellenőrző függvény BBB-SSS
int rendszam_e(char *s) {
	int i, len = strlen(s);
	const int ideal_len = 7;
	if (len != ideal_len)
		return 1;
	if (s[3] != '-')
		return -1;
	for (i = 0; s[i] != '\0'; i++) {
		if (i < 3) {
			if (!isalpha(s[i]) || !isupper(s[i]))
				return -2;
		}
		if (i > 3) {
			if (!isdigit(s[i]))
				return -3;
		}
	}
	return 0;
}
```

[23. példa]

A rendszám ellenőrző függvényben négy dolgot kell ellenőrizni. Az input hosszát, ami 7 karakter, azt hogy a középső karakter ’-’ legyen illetve, hogy az első 3 kis vagy nagy betű, az utolsó 3 karakter pedig szám legyen.

```
//dátum ellenőrző függvény
const char *valid_date(char *s) {
	//változók
	int i, n = 1, szam = 0, pont = 0, len = strlen(s), szokoev=0, honap;
	int napok[2][13] = { //szökőévvel
		{ 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
		{ 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
	};
	char *rec, *dest = malloc(sizeof(char)*len + 1);
	const int limit = 2;
	//formai ellenőrzés [ számokból és pontokból állhat ]
	for (i = 0; s[i] != '\0'; i++) {
		if (isdigit(s[i]))
			szam++;
		if (s[i] == '.')
			pont++;
	}
	if (len != (szam + pont) || pont != limit)
		return "Helytelen dátum formátum. A helyes formátum : ÉÉÉÉ:HH:NN!";
	//tartalmi ellenőrzés
	strcpy(dest, s);
	rec = strtok(dest, ".");
	while (rec != NULL) {
		//évszámellenőrzés
		if (n == 1) {
			if (atoi(rec) > 999 && atoi(rec) < 10000) { //YYYY.
				if (atoi(rec) % 4 == 0 && atoi(rec) % 100 != 0 || atoi(rec)
% 400 == 0) //szokoev-e?
					szokoev = 1;
			}
			else return "Helytelen évszám! [ YYYY. ]";
		}
		//hónap ellenőrzés
		if (n == 2) {
			if (strlen(rec) != 2) //HH.
				return "Helytelen hónap formátum! [ HH. ]";
			if (atoi(rec) <= 0 || atoi(rec) > 12)
				return "Helytelen hónap! <13";
			honap = atoi(rec);
		}
		//nap ellenőrzés
		if (n == 3) {
			if (strlen(rec) != 2) //NN.
				return "Helytelen nap formátum! [ NN ]";
			if (atoi(rec) <= 0 || atoi(rec) > napok[szokoev][honap])
				return "Helytelen nap!";
		}
		rec = strtok(NULL, ".");
		n++;
	}
	free(dest);
	return "Érvényes.";
}
```

[24. példa]

A dátum ellenőrző függvény nem annyira komplikált, mint amennyire hosszú. Először is a formai ellenőrzést kell elvégezni. A dátum 8 számjegyből és két pontból áll, vagyis összesen 10 karakter hosszú. A tördelő függvény azt ellenőrzi, hogy a pontok által határolt darabok [***rec**], megfelelnek-e a validációnak. Az első rész, 4 számjegyből kell, hogy álljon és vizsgálni kell azt is, hogy szökő év-e. A második a hónap (1-12-ig) és 2 karakter hosszú, a harmadik a nap, ami a hónap [n==2-ben] és szökőévtől [n==1-ben] függően 30-31 illetve 28 napos is lehet.

```
//ip-cím ellenőrző függvény
const char *valid_ip(char *s) {
	//változók
	int i, szam = 0, pont = 0, len = strlen(s);
	const int limit = 3;
	char *rec, *dest = malloc(sizeof(char) * len + 1);
	//formai ellenőrzés
	for (i = 0; s[i] != '\0'; i++) {
		if (isdigit(s[i]))
			szam++;
		if (s[i] == '.') {
			if(i != 0)
			if(!isdigit(s[i - 1])) //minden pont előtt szám van !!!
				return "Helytelen Ip-cím formátum. A helyes formátum 4 db
pontokkal elválasztott decimális egész számból áll!";
			pont++;
		}
	}
	if (len != (szam + pont) || pont != limit || !isdigit(s[i - 1])) //számok és
pontok + 3 pont van + az utolsó nem pont !
		return "Helytelen Ip-cím formátum. A helyes formátum 4 db pontokkal
elválasztott decimális egész számból áll!";
	//tartalmi ellenőrzés
	strcpy(dest, s);
	rec = strtok(dest,".");
	while (rec!=NULL) {
		if (atoi(rec) < 0 || atoi(rec) > 255)
			return "Az Ip-címben szereplő számoknak 0 és 255 közöttinek kell
lennie!";
		rec = strtok(NULL, ".");
	}
	free(dest); //itt is felszabadítható, mivel nem a dest változóval tér vissza a
függvény !!!
	return "Érvényes.";
}
```

[25. példa]

Az ip-cím ellenőrző függvény szintén egy-egy karakterlánccal/üzenettel visszatérő függvény, melyet egyébként a fő/main függvényben a strcmp(input,"Érvényes.")?=0 összehasonlító függvény segítségével kell validálni, a visszatérő érték (**const char**) okán hasonlóan a dátum ellenőrző függvényhez. Az ip-cím meghatározott számú pontokból (3) és számjegyekből (a többi) állhat. Fontos, hogy minden pont előtt szám van [if(i!=0)- nem az első >> if(!isdigit(s[i - 1]))] és az utolsó nem lehet pont [!isdigit(s[i - 1]) – itt az i már az utolsó karakter helyiértékét, a lánczáró 0-át fejezi ki!]. Végül ellenőrizni kell azt is, hogy a pontokkal elválasztott számok [rec = strtok(dest,".");] 0 és 255 közötti értékek-e. Hasonlóan a dátum ellenőrzéséhez, itt is részletes formai és tartalmi vizsgálatot kell végezni.

Pszeudo random szám generálás

```
int main() {

	time_t t;
	int r, i = 0, j = 0;
	const int tol = 1, ig = 35, mennyit = 7, szelveny = 2;

	srand((unsigned)time(&t));

	while (i < szelveny) {
		while (j < mennyit) {
			r = rand() % ig + tol;
			printf("%d, ", r);
			j++;
		}
		i++;
		j = 0;
		printf("\n");
	}

	return 0;
}
```

[26. példa]

A fenti példában egy két szelvényes (const int szelveny = 2) hetes/skandináv lottó húzás (const int tol = 1, ig = 35) szimulációját láthatjuk. A random szám generálásához szükséges változók és függvények:

```
#include <time.h>

time_t t;
int r;

srand((unsigned)time(&t));

r = rand() % ig + tol;
```

Ha nem adunk meg -tól értéket, akkor 0-tól kezdődik a számok generálása.

A fentebb látható számhúzást szimuláló kód hiányossága, hogy megengedi a generált számok ismétlődését, ezért a kód egy lehetséges kimenete:

```
3, 15, 30, 30, 27, 20, 20,
4, 20, 4, 25, 11, 25, 30,
```

A kiemeléssel jól látható, hogy az első szelvényen a 30 és a 20 ismétlődik, illetve a másodikon a 4 és a 25.

Ha valamilyen formai előírás megköveteli (pl. utolsó szám után ne legyen vessző!), akkor a kiíratást a következőképpen módosíthatjuk:

```
while (i < szelveny) {
		while (j < mennyit) {
			r = rand() % ig + tol;
			if(j==6)
				printf("%d\n", r);
			else
				printf("%d, ", r);
			j++;
		}
		i++;
		j = 0;
```

[27. példa]

Annak érdekében, hogy a számsorok ne tartalmazzanak ismétlődő elemeket, a kódot a következőképpen kell módosítanunk:

```
#define INTERVAL 35+1
#define MAX 7
#define N 2


int main() {

	time_t t;
	int r, i = 0, j = 0;
	bool arr[INTERVAL] = { 0 }; // #include <stdbool.h>

	srand((unsigned)time(&t));

	while (i < N) {

		while (j < MAX) {

			r = rand() % INTERVAL;

			if (!arr[r])
				printf("%d, ", r);
			else
				j--;

			arr[r] = 1;
			j++;
		}
		memset(arr, 0, sizeof(arr));
		i++;
		i = 0;
		printf("\n");
	}

	return 0;
}
```

[28. példa]

Arra is szükség lehet, hogy a számokat tömbökben tároljuk, pl. azért, hogy emelkedő sorrendbe állítsuk vagy a későbbiek során tömbindexként használjuk fel – pl. karakterlánc (táblázatos) kiíratásához. Ezért az előbbi esetben, mivel több szelvényről/adatsorról is szó van (N>2) a tomb[N][MAX] két dimenziós megoldást kell alkalmaznunk, míg az utóbbi esetben elég az egy dimenziós tömböt tomb[MAX] létrehoznunk a számok (potenciális indexek) tárolására. Nézzük előbb azt az esetet, amikor számsorok sorrendbe állítását kell végrehajtanunk. Megjegyzendő, hogy a fenti kódban még annyi módosítást is tettünk, hogy a tól-ig intervallumot definiáltuk (#define INTERVAL 35+1).

```
int main() {

	time_t t;
	int r, i = 0, j = 0, tomb[N][MAX], a = 0, temp = 0;
	bool arr[INTERVAL] = { 0 };
	srand((unsigned)time(&t));

	printf("Az ismetlodes nelkuli generalt szamok:\n");
	while (i < N) {

		while (j < MAX) {

			r = rand() % INTERVAL;

			if (!arr[r]) {
				tomb[i][j] = r;
				printf("%d, ", r);
			}
			else
				j--;
			arr[r] = 1;
			j++;
		}
		memset(arr, 0, sizeof(arr));
		i++; j = 0;
		printf("\n");
	}
	//emelkedő sorrendbe állítás [bubble sorting]
	while (a < N) {
		for (i = 0; i < MAX; i++)
			for (j = i + 1; j < MAX; j++)
				if (tomb[a][i] > tomb[a][j]) {
					temp = tomb[a][i];
					tomb[a][i] = tomb[a][j];
					tomb[a][j] = temp;
				}
		a++; //szelvény - sor
	}
	//a tömb tartalmának kiíratása
	printf("A generalt szamok emelkedo sorrendben:\n");
	for (i = 0; i < N; i++)
		for (j = 0; j < MAX; j++) {
			if(j==(MAX-1)) //formai korrekció
				printf("%d\n", tomb[i][j]);
			else
				printf("%d, ", tomb[i][j]);
		}

	return 0;	}
```

[29. példa]

A 26. példában a generáló kódban történt bővítések aláhúzással vannak jelölve, ezek a tömbbe másolást és az ismétlődések törlését hivatottak végrehajtani. A generáló kód után az emelkedő sorrendbe való rendezés (bubble sorting) és a kiíratás (+formai korrekció) következnek.

A következő példa egy karakterláncokat tartalmazó tömböt irat ki a képernyőre, random generált indexálással/sorrendben.

```
int main() {

	//magyar nyelvű konzol
	system("chcp 1250");
	setlocale(LC_ALL, "hun");

	char input[BUF], *nev[] = {"Tóth József","Kiss Ákos","Horváth Anna","Kovács
	Pista","Nagy Mari"};
	const int n = 5;
	int i = 0, j = 0, r, tomb[5];
	bool arr[5] = { 0 };
	time_t t;

	srand((unsigned)time(&t));

	//kiíratás
	printf("Nevek listázása megadási sorrendben:\n");
	for (i = 0; i < n; i++)
		printf("[%d.] név: %s\n", i, nev[i]);

	printf("Nevek listázása random sorrendben:\n");
	while (j < n) {
		r = rand() % n;
		if (!arr[r])
			tomb[j] = r;
		else
			j--;
		arr[r] = 1;
		j++;
	}
	memset(arr, 0, sizeof(arr));

	for (i = 0; i < n; i++)
		printf("[%d.] név: %s\n", i, nev[tomb[i]]);

	return 0;
}
```

[30. példa]

A nevek bekérése inkább a szabvány bemenetről (getline(,)||fgets(,,stdin)) vagy fájlból való kiolvasással (fgets(,,fp)) történik, de a szemléltetés kedvéért most egyszerűbb megoldást választottunk. A lényeg itt is az, hogy egy olyan random generált egész számokból álló tömböt hozzunk létre, melyben nincsenek ismétlődések és a tömbméret megegyezzen a nevek számával. Végül az utolsó for ciklussal a karakterlánc tömbindexei alapján történik a random sorrendű kiíratás [nev[tomb[i]]].

Ismétlődő karakterláncok kezelése

```c
//ismétlődő karakterláncok törlése tömbből
const char *str_ism(char **s, int n) {
	int i, j, k = 0, l, ism = 0;
	char **arr = malloc(sizeof(char*)*n);
	for (i = 0; i < n; i++) {
		for (j = 0; j < n; j++)
			if (strcmp(s[i], s[j]) == 0) { //*(s+i),*(s+j)
				ism++;
				if (ism == 2) {
					if (k > 0)
						for (l = 0; l < k; l++)
							if (strcmp(s[i], arr[l]) == 0)
								goto re;
					arr[k] = malloc(sizeof(char)*strlen(s[i]) + 1);
					strcpy(arr[k], s[i]);
					k++;
				}
			}
	re:
		ism = 0;
	}
	arr[k] = NULL;
	return arr;
}

int main() {

	//magyar nyelvű konzol
	system("chcp 1250");
	setlocale(LC_ALL, "hun");

	char *strs[] = { "hello","hello","world","yes","no","yes","world" };
	char *str = "world";
	int i, j = 0, check = 0;
	const int n = 7; //strs tömb mérete

	//többször szereplő elemek felsorolása
	printf("Többször szereplő elemek:\n");
	const char **ism = str_ism(strs,n);
	for (i = 0; ism[i] != NULL; i++)
		printf("%s\n",ism[i]);

	//előfordul-e a tömbben?
	for (i = 0; i < n; i++)
		if (strcmp(str, strs[i]) == 0)
			check++;
	//eredmény kiíratása
	printf("A %s %d-szer forul elő a tömbben.\n", str, check);
	check = 0; //ha pl. egy bekérő ciklusban van
	//memória felszabadítása
	free(ism);

	return 0;
}
```

[31. példa]

CSV fájlok kezelése/feldolgozása

```
int main() {

	char *fnev = "c:\\temp\\teszt.csv", *rec, fbuf[BUF];
	int i = 0, j = 0;
	FILE *fp;

	fp = fopen(fnev,"r");
	if (fp == NULL) {
		perror("Hiba a fájl megnyitáskor: ");
		exit(1);
	}

	//csv fájl beolvasása
	while (fgets(fbuf, BUF, fp) != NULL) {
		//extra lánczáró csonkolása
		fbuf[strlen(fbuf) - 1] = '\0';
		//;-vel elválasztott értékek
		rec = strtok(fbuf, ";");

		while (rec != NULL) {

			printf("%s;", rec);
			//ugyanitt történhet a memóriafoglalás az adatoknak [
			malloc(char*) ] és a változóba másolás [ strcpy() ].

			rec = strtok(NULL, ";");
			j++;
		}
		printf("\n");
		j = 0;
		i++;
	}

	fclose(fp);

	return 0;
}
```

[32. példa]

A fenti megoldást már alkalmaztuk, a dátum és ip-cím validálás során [21-22. példa], hiszen ott is karakterekkel elválasztott értékeket ellenőriztünk. A fenti kód egy csv fájl beolvasását végzi az fgets() és a strtok() függvények segítségével, majd a pontosvesszővel elválasztott értékeket (*Comma Separated Values*) kiíratja a képernyőre.

Az adatok beolvasása 3D tömbök, tripla mutatók vagy struktúrák segítségével történhet, ahogy azt a karakterlánc tárolást bemutató részben is láthattuk (6. oldal). A validáló programokhoz képest csupán annyi a különbség, hogy itt a csv fájlok esetében a ’;’ az elválasztó (delimiter) karakter.

Amennyiben a sorok és oszlopok száma változik, úgy a sor illetve oszlop számláló függvényeket célszerű alkalmazni, melyeket tripla mutatók segítségével történő dinamikus memóriafoglalás során lehet felhasználni. Adott/konstans sor és oszlop

számok esetén a *adat[SOR][OSZLOP] is alkalmazható, feltéve, hogy: #define SOR 8 és #define OSZLOP 13.

Tömbök/mutatók szerepe a .csv fájl feldolgozása során

```
char ***data; //*data[SOR][OSZLOP]; //#define SOR 8 #define OSZLOP 13

data = (char***)malloc(sizeof(char**)*sor);
	for(i=0;i<sor;i++)
		data[i] = (char**)malloc(sizeof(char*)*oszlop);

//A fenti kódban ez kerül a printf("%s;", rec); helyére !!!
data[i][j] = (char*)malloc(sizeof(char)*strlen(rec) + 1); //*data[SOR][OSZLOP];-
ra csak ez a sor vonatkozik, és a felszabadítás is csak erre vonatkozik >>
free(data[i][j]).
strcpy(data[i][j],rec);

//adatok kiíratása
for (i = 0; i < sor; i++) {
	for (j = 0; j < oszlop; j++)
		printf("%s;",data[i][j]);
	printf("\n");
}

//memória felszabadítás
for (i = 0; i < sor; i++) {
	for (j = 0; j < oszlop; j++) {
		free(data[i][j]);
	}
	free(data[i]);
}
free(data);
data = NULL;
```

[33. példa]

CSV táblázat részletének kiíratása

Az alábbi kódrészlet, csak annyiban tér el az előzőtől, hogy a harmadik (j==2) oszloptól az 2-7. sorig (i>0 && i<=6) veszi figyelembe az adatokat és csak ezeket íratja ki a képernyőre.

íme a módosított kódrészlet:

```
//adatok kiíratása
	for (i = 1; i < sor-1; i++) {
		for (j = 2; j < oszlop; j++)
			printf("%s;",data[i][j]);
		printf("\n");
	}
```

Amennyiben a feladat előírja, hogy a memóriafoglalás is kizárólag ezen részlet számára történjen, úgy új indexváltozók bevezetésére van szükség, melyek csak a megfelelő logikai kikötések után dekrementálhatók.

Időkezelés

A következőkben a <time.h> fejfájl bekapcsolásával elérhető időkezelő függvényekre hozok példákat. Dátum kiííratás és formázás, rendszeridő kiíratás és formázás, program futási idejének mérése, dátumok közt eltelt idő mérése.

Aktuális dátum, rendszeridő és másodpercekben eltelt idő kiíratása

```
time_t t;
time(&t);
struct tm aktualis_ido = *localtime(&t);

char buff[100];

// Kiírja a dátumot és a rendszeridőt a képernyőre
strftime(buff,64,"Dátum: %Y.%m.%d - rendszer idő: %H:%M:%S", &aktualis_ido);
printf("%s\n", buff);

// Másodpercekben eltelt idő <- 1970. január 1-je óta számolja
printf("%d másodperc telt el 1970 óta!\n", t);
printf("%d év telt el 1970 óta!\n", t / (60 * 60 * 24 * 365));
```

[34. példa]

A fenti kód először is az aktuális dátumot és rendszeridőt íratja ki a kimenetre Y-m-d és H-M-S formában. Majd a time_t t; -> time(&t) -> t alkalmazásával, az 1970 óta eltelt időt másodpercekben kaphatjuk meg. Amennyiben pl. az eltelt napokat szeretnénk megtudni: t/(60*60*24)-t kell megadnunk. Látható, hogy a példában években kértük az eltelt időt.

Programok futási idejének megmérése

1. difftime()

```
time_t kezd, veg;
time(&kezd);

//A program
for (int i = 0; i < 1000000000; i++);

time(&veg);

//Ciklus feldolgozása alatt eltelt másodpercek
double diff = difftime(veg,kezd);
printf("A program %.0f sec alatt futott le.\n", diff);
```

[35. példa]

A fenti kóddal megmérhetjük, hogy egy program, esetünkben egy milliárd lépést végrehajtó ciklus, hány másodperc alatt fut le. Segítségül a double difftime() függvényt hítvuk. Ez a függvény a végidőből vonja ki a kezdő időt és egy lebegőpontos értékkel tér vissza.

2. clock()

```
clock_t kezd = clock();

//A program
for (int i = 0; i < 1000000000; i++);

clock_t veg = clock();

//Ciklus feldolgozása alatt eltelt másodpercek
double diff = (double)(veg - kezd) / CLOCKS_PER_SEC;
printf("A program %.0f sec alatt futott le.\n", diff);
```

[36. példa]

Amennyiben pontos mérést szeretnénk, a fenti megoldás hasznosabb. A difftime() inkább nagyobb időközök, pl. napok megmérésére alkalmas. A alábbiakban erre hozok példákat.

Két dátum között eltelt idő mérése

```
//mettől?
int ev1 = 2001;
int ho1 = 11;
int na1 = 30;
//meddig?
int ev2 = 2020;
int ho2 = 3;
int na2 = 11;

time_t t, tt;
struct tm kezd = { 0 }, veg = { 0 };

//kezdő dátum
kezd.tm_year = ev1 - 1900;
kezd.tm_mon = ho1 - 1;
kezd.tm_mday = na1;

t = mktime(&kezd);

//vég dátum
veg.tm_year = ev2 - 1900;
veg.tm_mon = ho2 - 1;
veg.tm_mday = na2;

tt = mktime(&veg);

// A különbség kiíratása a képernyőre napokban számolva
double diff = difftime(tt/(60*60*24),t/(60*60*24));
printf("A két dátum között eltel napok száma: %.0f nap\n", diff);

// Ha kikötöttük, hogy egésszel kell visszatérnie a függvénynek, akkor
int kezd_ = t / (60*60*24);
int veg_ = tt / (60*60*24);
int diff_ = veg_ - kezd_;
printf("A két dátum között eltel napok száma: %d nap\n", diff_);
```

[37. példa]

A fenti példában a két dátumot tetszőlegesen adtuk meg, ugyanakkor számos esetben az aktuális időt is be kell vonnunk a kezdő vagy vég időként a számításba. Erre példa pl.

az életkor megállapítása a születési dátum megadásával, ilyenkor a vég idő az aktuális rendszeridő. Egy másik példa, ha azt keressük, hogy egy adott dátumig mennyi időnek kell eltelnie. Ezekre láthatunk példákat az alábbiakban.

Életkor meghatározása

```
int szev = 1987;
int szho = 01;
int szn = 7;
int e, h, n;

time_t t, tt;
time(&tt);
struct tm sznap = { 0 }, most = *localtime(&tt);

//aktuális idő
e = most.tm_year + 1900;
h = most.tm_mon + 1;
n = most.tm_mday;

//születésnap
sznap.tm_year = szev - 1900;
sznap.tm_mon = szho - 1;
sznap.tm_mday = szn;

t = mktime(&sznap);

// Kiíratás
double diff = difftime(tt,t) / (60*60*24*365); //különbség sec-ben évekre átszámítva!
printf("Ön %.0f éves!\n", diff);
```

Egésszel visszatérő függvénynek esetén a diff:

```
int rec = (int)diff;
int start = t / (60 * 60 * 24);
int veg = tt / (60 * 60 * 24);
int diff = veg - start;
```

[38. példa]

Mennyi nap van hátra karácsonyig?

```
int ev = 2020;
int ho = 12;
int nap = 24;

time_t t, tt;
time(&t);
struct tm most = *localtime(&t), akkor = { 0 };

// Karácsonykor
akkor.tm_year = ev - 1900;
akkor.tm_mon = ho - 1;
akkor.tm_mday = nap;
akkor.tm_hour = most.tm_hour; //pontosság miatt
akkor.tm_min = most.tm_min;
akkor.tm_sec = most.tm_sec;

tt = mktime(&akkor);

double diff = difftime(tt, t) / (60 * 60 * 24);
printf("%.0f nap van még karácsonyig!\n", diff);
```

[39. példa]

Dátum és idő formázása

```
time_t t;
time(&t);
struct tm most = *localtime(&t);

char buff[100];

strftime(buff,100,"Dátum/idő: %Y.%m.%d - %H:%M:%S",&most);

-> Dátum/idő: 2020.05.14 - 14:51:42

%A -> nap: pl.: csütörtök

%a -> nap pl.: cs

%c -> dátum és idő együtt: Dátum/idő: 2020. 05. 14. 14:58:15

A helyi beállítás setlocale() befolyásolja a kimenetet

%x -> ha nincs setlocale: HH/NN/EE, ha van: EEEE.HH.NN

%B -> hónap név: May, ha setlocale: május

%b -> hónap név: May, ha setlocale: máj.

%I -> 01-12 óra formátum

%H -> 00-23 óra formátum

%j -> az év hányadik napja 001-366
```

Hasznos függvények/megoldások

Bubble sorting [emelkedő i > j sorrend]

```
int main() {

	int arr[] = { 9,1,3,2,7,5,6,4,8,0 }, i, j = 0, temp;
	size_t size = sizeof(arr) / sizeof(arr[0]);

	//kiíratás eredeti sorrendben
	for (i = 0; i < size; i++)
		printf("%d ", arr[i]);
	printf("\n");

	//kiíratás emelkedő sorrendben
	for(i=0;i<size;i++)
		for(j=i+1;j<size;j++)
			if (arr[i] > arr[j]) { //emelkedő
				temp = arr[i];
				arr[i] = arr[j];
				arr[j] = temp;
			}
	for(i=0;i<size;i++)
		printf("%d ", arr[i]);
	printf("\n");

	return 0;
}
```

Karakterláncok ábécé sorrendben való kiíratása[2]

```
	char fbuf[BUF], **adat, *temp;
	sorsz = sorok(fp); //függvény alapján

	adat = (char**)malloc(sizeof(char*)*sorsz);

	//adatok tömbbe mentése - egy sor egy adatot tartalmaz
	while (fgets(fbuf, BUF, fp) != NULL) {
		fbuf[strlen(fbuf)-1] = '\0';
		adat[m] = (char*)malloc(strlen(fbuf)+1);
		strcpy(adat[m],fbuf);
		m++;
	}
	//ábécé sorrend
	for (i = 0; i < m; i++)
		for (j = i + 1; j < m; j++) {
			if (strcmp(adat[i], adat[j]) > 0) {
				temp = (char*)malloc(strlen(adat[i]) + 1);
				strcpy(temp, adat[i]);
				adat[i] = (char*)realloc(adat[i],strlen(adat[j])+1);
				strcpy(adat[i], adat[j]);
				adat[j] = (char*)realloc(adat[j], strlen(temp) + 1);
				strcpy(adat[j],temp);
				free(temp); }  }
```

A bubble sorting függvény pszeudo kóddal:

- **for(i++<size)**
  - **for(j++=i+1;<size)**
    - **if (i>j) || (i<j)** //emelkedő vagy csökkenő
      - **temp = i;**
      - **i=j;**
      - **j=temp**

Ismétlődő karakterek törlése karakterláncból

```
//ismétlődők keresése, törlése
const char *chdel(char *s) {
	int i, j, k;
	char dest[BUF];
	//char *dest = (char*)malloc(sizeof(char)*strlen(s) + 1);
	strcpy(dest, s);
	int len = strlen(dest);
	for (i = 0; dest[i] != '\0'; i++) {
		for (j = i + 1; j<len;) {
			if (dest[j] == dest[i]) {
				for (k = j; k<len; k++)
					dest[k] = dest[k + 1];
				strlen(dest) - 1;
			}
			else j++;
		}
	}
	return dest;
}
```

[2] A példában némelyik változó nem deklarált (vektoros), illetve a main és a fájl kezelő függvények sem szerepelnek.

Amennyiben a fenti függvényben a dest karakterláncnak dinamikusan foglalunk memóriát (lásd kikapcsolt sor: `(char*)malloc(sizeof(char)*strlen(s)+1);`), a memória felszabadítását a main()-ben kell végrehajtanunk, méghozzá úgy, hogy előtte szintén a main()-ben const char változóként deklaráljuk [`const char *cdel = chdel(str);`]. Lássuk is a példát:

```
int main() {

	char *str = "Hello Worldd!";
	const char *cdel = chdel(str);

	printf("%s\n", cdel);

	free(cdel);

	return 0;
}
```

Visszatérés a képernyőn: Helo Wrd!, melyből már a függvény törölte az ismétlődő karaktereket. A cdel konstanst csak ott tudjuk deklarálni, ahol már ismert a feldolgozandó karakterlánc. Amint szükségtelenné válik, felszabadíthatjuk a chdel() függvényben lefoglalt memóriát a main()-ben.

Jelek/karakterek számlálása fájlban

```
//jelek keresése fájlban
int jfreq(char ch, FILE *f) {
	int sum = 0, c;
	int pos = ftell(f);
	do {
		c = fgetc(f);
		if (c == ch)
			sum++;
	} while (c != EOF);
	fseek(f, pos, SEEK_SET);
	return sum;
}
```

A függvény a keresett karakter [`char ch`], adott fájlban [`FILE *f`] való előfordulásának értékével tér vissza.

Fájlméret meghatározása

```
#define VALTO 1024

//fájlméret
float fmeret(FILE *f) {
	float res;
	int pos = ftell(f);
	fseek(f, 0L, SEEK_END);
	res = (float)ftell(f);
	fseek(f, pos, SEEK_SET);
	return res /VALTO;
}
```

Ha nincs váltószám [VALTO], akkor bájtokat számol. Itt a váltó 1024, ezért a fájlméret Kilobájtban egy lebegőpontos érték (float), ami kettő tizedesjeggyel az alábbi módon iratható ki a kimenetre:

```
printf("A fájl mérete %.2f kB.\n", fmeret(fp));
```

A példákban felhasznált fejfájlok

```
#include <stdio.h> #include <stdlib.h> #include <string.h> #include <ctype.h> #include
<locale.h> #include <windows.h> #include <time.h> #include <stdbool.h>
```

A felhasznált típusok méretei

```
printf("sizeof_int: %d byte\n", sizeof(int)); >> 4
printf("sizeof_int*: %d byte\n", sizeof(int*)); >> 4
printf("sizeof_float: %d byte\n", sizeof(float)); >> 4
printf("sizeof_float*: %d byte\n", sizeof(float*)); >> 4
printf("sizeof_char: %d byte\n", sizeof(char)); >> 1
printf("sizeof_char*: %d byte\n", sizeof(char*)); >> 4
```

A mutatók mérete, ahol az indirekció *-gal jelezve van, mindig 4 bájt, ezen kívül még az integer és a floating point 4-4, illetve a char 1 bájt.

Magyar nyelvű konzol

```
system("chcp 1250"); //magyar nyelvű bemenet/bekérésnél
setlocale(LC_ALL, "hun"); //magyar nyelvű kimenet/tizedes vessző stb.
```

Vonatkozó/szomszédos adat kiíratása 2 oszlopos táblázatból

```
printf("Adjon meg sorokat: ");
	while (getline(input,BUF)) {
		fseek(fp, 0L, SEEK_SET); //fájlmutatót a fájl elejére!
		while (fgets(fbuf, BUF, fp) != NULL) {

			fbuf[strlen(fbuf)-1] = '\0';
			temp = (char*)malloc(strlen(fbuf) + 1);
			strcpy(temp, fbuf);

			if (strcmp(input, strtok(temp, "\t")) == 0) {
				printf("%s\n", strchr(fbuf, '\t') + 1 );
				check = 1;
			}
			free(temp);
		}
		if (check == 0)
			printf("Nincs egyezés!\n");
		else
			check = 0;
		printf("Adjon meg sorokat: ");
	}
```

A fenti kód szabvány bemenetről kér be sorokat, és ha a bemenet egyezik egy adott sor 0. oszlopában szereplő karakterlánccal, akkor kiíratja az 1. oszlop adott sorában levő karakterláncot. Amennyiben az 1. oszlop karakterlánca alapján szeretnék kiíratni a 0. oszlop értékét, úgy a strtok és strchr függvényeket fel kell cserélni. Ekkor ügyelni kell strchr-nál a +1-re (detektált hely után eggyel) és a „” és ’’

jelek különbségére! Továbbá fontos, hogy minden ciklus elején a fájlmutatót a fájl elejére programozzuk (fseek)!

Formázott táblázat kiíratása képernyőre(printf)/fájlba(fprintf)

Egy M sorból és N oszlopból álló táblázat formázott kiíratása során általában az oszlop nevekkel és az alájuk kerülő sorok adataival foglalkozunk. Ebben a konkrét esetben balra igazítunk 12 karakteren: [**%-12s = a b c d** _ _ _ _ _ _ _ _].

```
printf("%-10s%-12s%-12s%-12s\n", " ", "0.oszlop", "1.oszlop", "2.oszlop");
for (i = 0; i < 46; i++) //10 + 3*12
	printf("-");
printf("\n");
for (i = 0; i < N; i++) {
	printf("%d. adat: | ", i);
	for (j = 0; j < M; j++) {
			//if(j==0)
				//printf("%5s", adat[i][j]);
			//else
				printf("%-12s", adat[i][j]);
	}
	printf("\n");
}
```

Az első kinyomtatott oszlop (sárga) opcionális, csak úgy mint az egyes oszlopokra vonatkozó külön formázás (pl. j==0-nál legyen 5 karakteren jobbra tolt [_ _ **a b c**]).

Fájl kiterjesztésének ellenőrzése

```
//fájlnév érvényesség
int fval(char *s, char *kit) {
	char *ptr = strchr(s, '.') + 1;
	if (strcmp(ptr, kit) != 0)
		return 1;
	return 0;
}

int main() {

	//magyar nyelvű konzol
	system("chcp 1250");
	setlocale(LC_ALL, "hun");

	char *fnev = "nato.txt", *kit = "txt";

	if (fval(fnev, kit) == 0)
		printf("Helyes fájlnév!\n");
	else
		printf("Helytelen fájlnév\n");

	return 0;
}
```

A strchr() függvény a mutatóval [*ptr] onnan tér vissza, ahol a pontot detektálja, ahhoz hogy a visszatérő karakterlánc ne tartalmazza a pontot léptetnünk kell eggyel, ahogy a fenti példában az látszik. [strchr(s, '.') + 1]

Karakterlánc darabot elforgató függvény

```
const char *forgat(char s[], int a) {
	char *mit = malloc(sizeof(char)*strlen(s) + 1);
	int mennyivel = strlen(s) - a;
	const char *ptr = &s[mennyivel];

	strcpy(mit, ptr);
	s[mennyivel] = '\0';
	strcat(mit, s);

	strcpy(s, mit);

	return mit;
}
```

```
for (i = 0; i < FKOR; i++) { //hányszor forgassa el?
	char *frgt = forgat(input, a);
	printf("%s\n", frgt);
	free(frgt);
}
```

Ez a kód egy karakterláncot forgat el annak megadásával, hogy ezt hány karakterrel tegye (a végéről lecsíp és az elejére tesz). FKOR és a for ciklus meghatározza, hogy hányszor történjen meg az elforgatás.

Késleltetés n másodpercig

```
#include <windows.h>

Sleep(3000); //3 sec
```

Formázott rendszeridő/dátum kiíratása

```
//időkezelés
#include <time.h>
time_t current_time;
struct tm * time_info;
char timeString[48];  // space for "...HH:MM:SS\0"

//rendszeridő, dátum ÉÉ.HH.NN - ÓÓ:PP:SS //+1
time(&current_time);
time_info = localtime(&current_time);
strftime(timeString, sizeof(timeString), "idő: [%y.%m.%#d - %H:%M:%S]",
time_info);
printf(timeString);
```

Aktuális rendszeridő fájlba íratása

```
time_t aktido;
struct tm *ido_info;
char tbuff[64];

system("chcp 1250");

//aktuális rendszeridő és dátum
time(&aktido);
ido_info = localtime(&aktido);
strftime(tbuff,sizeof(tbuff),"%Y. %B %#d. %H:%M:%S",ido_info);
fseek(fp, 0L, SEEK_END);
fprintf(fp, "%s\n", tbuff);
```

Másodperc mérő függvény

```
//itt indul a mérés, a program elején
time_t t;
clock_t ora;
ora = clock();

//itt állítjuk meg és íratjuk ki az eltelt időt
ora = clock() - ora;
double ido = ((double)ora) / CLOCKS_PER_SEC; // in seconds
printf("\nA nevek begépelése %.2f másodpercet vett igénybe.\n", ido);
```

Fájl tartalmának másolása egy másik fájlba

```
#define LIMIT 200

int masolas(char input[], char output[]) {

	int i, c = 0;
	char buffer[BUF];
	FILE *in, *out;
	//fájlok megnyitása
	in = fopen(input,"r");
	if(!in)
		return 1;
	out = fopen(output,"w");
	if(!out)
		return 1;
	//másolás-beillesztés
	while (fgets(buffer, BUF, in)!=NULL) {
		buffer[strlen(buffer) - 1] = '\0';
		for (i = 0; buffer[i] != '\0'; i++) {
			fprintf(out,"%c",buffer[i]);
			c++;
			if (c == LIMIT) {
				fprintf(out, "\n");
				c = 0;
			}
		}
	}
	fclose(in);
	fclose(out);

	return 0;
}
```

Alkalmazható pl. parancssori argumentumként megadott fájlnevek esetében [masolas(argv[1], argv[2])] vagy közvetlen fájlnév megadással. A függvény először is megnyitja a forrás (in) és a célfájlt (out), majd az fgets() függvény segítségével beolvassa a forrás fájl tartalmát és karakterenként átmásolja a célfájlba [fprintf(out,"%c",buffer[i]);], amint eléri a limitet [#define LIMIT 200] új sort kezd és a számlálót nullázza, végzi ezt addig, amíg a forrás fájl végre nem ér [!=NULL].

Fájl tartalmának beolvasása fscanf() függvénnyel

Fájlok tartalmát a korábbiakban fgets(,,fp) függvény segítségével olvastuk be, és a sorok darabolását strtok() függvénnyel oldottuk meg. Az fscanf() függvény használata ezzel szemben, akkor célszerű, ha az adatokat (főleg karakterláncok) struktúrák segítségével olvassuk be, nem csv fájlról van szó és nem ismerjük a számukra foglalandó memória nagyságát. Csak nem pontos hosszúságú, fehérkarakterekkel elválasztott adatok esetén célszerű tehát a használata:

```
	typedef struct {
		char elso[16], masodik[16];
	}adat;

	adat *uj = (adat*)malloc(sizeof(adat)*sor);

	while (1) {
		if (fscanf(fp, "%s %s", uj[i].elso, uj[i].masodik) == EOF) break;
		i++; } free(uj);
```