Algoritmi i strukture podataka

Tutorial o datotekama

autor: Marko Čupić

SVE O DATOTEKAMA

Od trenutka kada je stvoreno prvo računalo, postoji potreba za permanentnom pohranom podataka. Uobičajene memorije za to nisu prikladne, jer nestankom napajanja cijeli se sadržaj gubi. Zbog toga su razvijeni mnogi uređaji i mediji za pohranu podataka, od bušenih kartica pa sve do danas modernih DVD uređaja. No ono što je do danas ostalo nepromjenjeno, jest način na koji se postupa sa podacima. Naime, podaci se grupiraju u cijeline - datoteke! Svaka datoteka se također jednoznačno razlikuje od svih ostalih svojim - imenom! Dakako, način na koji se ti podaci fizički zapisuju i čitaju ovisi o samom uređaju i mediju na koji su pohranjeni. Time se dakle bavi sam uređaj. Na malo višoj razini nalazi se onaj tko određuje gdje želi pohraniti koje podatke, kako će ih razlikovati, i zapravo onaj tko vodi računa o datotekama: *operativni sustav*, odnosno još preciznije: *datotečni sustav*! Naime, razlika između ovih termina postoji, jer datotečni sustav jest sastavni dio operativnog sustava čija je jedina namjena rad sa datotekama (dok operativni sustav ima daleko šire područje rada). Svaki datotečni sustav programeru nudi niz funkcija za osnovne operacije sa datotekama, i time ćemo se pozabaviti malo kasnije. No najprije treba istaknuti prednosti i nedostatke pohrane podataka u datoteke. Kao što smo već spomenuli, pohrana je (više-manje) permanentna (ako ne sjednete na vaš najdraži CD, ili nešto tome slično). To je, svakako, dobro. No do dana današnjeg pristupanje takvim podacima daleko je sporije od pristupa podacima u memoriji. Zato svaka aplikacija treba što pametnije koristiti memoriju i minimizirati pristup vanjskim memorijama (ukoliko je to naravno moguće). No pogledajmo sada koja je generalna ideja rada sa datotekama.

Da biste u računalu dobili komad memorije, morate za to zvati npr. funkciju malloc. Tek ako vam ona kaže da ste dobili traženi prostor, smijete ga koristiti. Sa datotekama je stvar identična! Nakprije treba pozvati funkciju koja će datoteku stvoriti, ili otvoriti postojeću. Nakon toga podatke iz datoteke možemo čitati, zapisivati; možemo čak i mijenjati veličinu datoteke. Po završetku rada datoteku treba zatvoriti! Isto kao i poziv funkcije free nakon što više ne trebate memoriju.

Vjerojatno ste primjetili da još nisam naveo ime niti jedne funkcije za rad sa datotekama. Razlog tomu je postojanje više različitih! Naime, postoje dva načina pamćenja podataka o otvorenim datotekama. Prvi način jest da od operativnog sustava dobijete pokazivač na strukturu koju popuni sam operativni sustav (FILE*), te taj pokazivač proslijeđujete dalje funkcijama za rad sa datotekama. Ovo je porodica *f* funkcija (*fopen*, *fclose*, ...), i svi prototipovi nalaze se u **stdio.h** zaglavlju. No, kako na ovaj način i vi imate pristupa do dotične strukture, mogli biste odlučiti nešto mijenjati po toj strukturi, a to se operativnom sustavu nikako ne bi dopalo. Zbog toga je razvijena nova serija funkcija koje datoteke pamte preko tzv. handlova, tj. brojeva. Na ovaj način operativni će vam sustav vratiti samo broj, a vi nećete imati pristupa do kritičnih struktura. Ovo je klasična serija funkcija čiji se prototipovi nalaze u **io.h** zaglavlju. Dodatno, ovih funkcija ima daleko više i pokrivaju sve operacije koje možete poželjeti raditi datoteci. No kako se rad sa datotekama u C-u na FER radi upravo preko f-funkcija, i mi ćemo se zadržati na njima. Tek ćemo tu i tamo pozvati upomoć poneku funkciju iz *io.h*.

STANDARDNE DATOTEKE. PRIMJER 1.

Pri pokretanju svakog programa, operativni sustav automatski otvara tri datoteke koje su u C-u dostupne preko imena:

stdin	standardni ulaz; obično tipkovnica
stdout	standardni izlaz; obično ekran
stderr	standardni izlaz za greške; obično ekran

Ove su datoteke FILE* tipa. To znači da ih direktno možete koristiti samo u f-funkcijama. Evo jednog kratkog primjera:

```
#include <stdio.h>

void main() {
  fprintf(stdout, "Hello world!\n");
}
```

Svi znate što radi printf() funkcija: ispisuje tekst na ekran. No, postoji inačica ove funkcije koja tekst ispisuje u zadanu datoteku! To je upravo funkcija *fprintf(FILE *datoteka, char* specifikacija, ...)*. U našem primjeru program u datoteku stdout ispisuje poruku 'Hello world!', a ovo se naravno preslika na ekran. Kao što vidimo iz primjera, u datoteku *stdout* se smije pisati, ali ne smijemo pokušati čitati! Isto vrijedi i za *stderr*. Za razliku od ove dvije datoteke, *stdin* je predviđen samo za čitanje, te u njega ne smijemo pisati!

KREIRANJE DATOTEKE. PRIMJER 2.

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

void main() {
  FILE *f;

  f = fopen("pozdrav.txt","wt");
  if( f == NULL ) {
    fprintf(stderr, "Ne mogu otvoriti datoteku pozdrav.txt\n");
    exit(1);
  }
  fputs("Hello, world!\n", f);
  fclose(f);
}
```

Evo prvog primjera u kojem imamo sve uobičajene korake pri radu sa datotekama. Deklarirali smo varijablu *f*, tipa *FILE**. Nakon toga pozivamo funkciju za otvaranje datoteke: *fopen*. Prvi parametar je ime datoteke koju želimo otvoriti, a drugi parametar govori što želimo raditi sa tom datotekom. Slovo **w** govori da želimo pisati u datoteku (write), dok slovo **t** govori da je datoteka tekstualna (text). Slovo t se može i ispustiti jer je podrazumjevano. Moguće je datoteku otvoriti i za čitanje (**r**), ili za dodavanje na kraj datoteke (**a**). Datoteka ne može istovremeno biti otvorena sa više od jednog načina rada. To znači da ne možemo napisati "**rw**", ili "**ra**". Ipak, datoteka se može otvoriti za istovremeni upis i ispis, ali se to postiže na drugi način. Za više detalja pogledajte donju tablicu. Ukoliko funkcija uspije otvoriti datoteku, vratiti će valjanu adresu strukture koju pohranjujemo u varijablu *f*. No ukoliko otvaranje ne uspije, funkcija će vratiti NULL pokazivač, o čemu treba voditi računa u daljnjem programu. U program je uključeno i *stdlib.h* zaglavlje zbog funkcije *exit()*.

NAČIN OTVARANJA	ZNAČENJE
r ili **rt**	Otvara datoteku za čitanje u tekstualnom modu. Ukoliko datoteka ne postoji, funkcija vraća grešku.
W ili **wt**	Otvara datoteku za pisanje u tekstualnom modu. Ukoliko datoteka ne postoji, biti će stvorena. Ukoliko postoji, sadržaj će biti izbrisan i datoteka biti skraćena na duljinu 0.
a ili **at**	Otvara datoteku za pisanje u tekstualnom modu. Ukoliko datoteka ne postoji, biti će stvorena. Ukoliko postoji, sadržaj će biti sačuvan i pisanje će se izvoditi na kraju datoteke.
r+ ili **at+**	Otvara datoteku za čitanje i pisanje u tekstualnom modu. Ukoliko datoteka ne postoji, funkcija vraća grešku. Ukoliko datoteka postoji, sadržaj ostaje sačuvan prilikom otvaranja.
w+ ili **wt+**	Otvara datoteku za pisanje i čitanje u tekstualnom modu. Ukoliko datoteka ne postoji, biti će stvorena. Ukoliko postoji, sadržaj će biti izbrisan i datoteka biti skraćena na duljinu 0.
x**b**, gdje je x={**r**,**w**}	Otvara datoteku kao što je gore opisano, samo što se otvaranje vrši u binarnom modu (umjesto u tekstualnom). Moguće su kombinacije: **rb**, **wb**, **rb+**, **wb+**

Ukoliko otvaranje nije uspjelo, grešku ispisujemo na *stderr* i prekidamo program. Inače u datoteku ispisuje poruku "Hello world!" (funkcija *fputs*), te zatvaramo datoteku (funkcija *fclose*). Iz gornje tablice vidimo da će naša datoteka gotovo sigurno biti otvorena. Naime, ako postoji, biti će otvorena; ako ne postoji, biti će stvorena! Jedino što bi se moglo dogoditi jest da nema dovoljno prostora za novu datoteku. Dodatno, pod unix-om bi se moglo dogoditi da nemamo ovlasti kreirati datoteku. Nadalje, došli smo do još jedne funkcije koja zna ispisivati u datoteku! Riječ je o funkciji *fputs()* koja se ponaša isto kao i njezina inačica koja tekst ispisuje na ekran: *puts*. Postoji još jedna razlika. Funkcija puts nakon što ispiše zadani tekst, ispisuje i '\n' znak, čime slijedeći ispis počinje u novom redu. Funkcija *fputs* ne dodaje '\n' znak! To je razlog što smo ga mi dodali ručno.

POKAZIVAČ TRENUTNOG POLOŽAJA. PRIMJER 3.

Kada otvorimo datoteku, operativni sustav mora znati sa koje pozicije želimo čitati, odnosno pisati u datoteku. Pamčenje dotične pozicije vrši se pomoću pokazivača trenutne pozicije, i o njemu se brine operativni sustav. No operativni nam sustav nudi funkcije kojima možemo doznati koliko iznosi ovaj pokazivač, ili pak funkcije za promjenu pokazivača.

Pretpostavimo da postoji datoteka znakovi.txt slijedećeg sadržaja:

```
Pero i stefica.
Ante.
```

Primjer slijedi:

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

void main() {
  FILE *f;
  long poz;
  int c;

  f = fopen("znakovi.txt","rt");
  if( f == NULL ) {
    fprintf(stderr, "Ne mogu otvoriti datoteku znakovi.txt\n");
    exit(1);
  }
```

```
  poz = ftell(f);
  while( (c = fgetc(f)) != EOF ) {
   printf("Na poziciji %6ld nalazi se znak %c.\n", poz, c);
   poz = ftell(f);
  }
  fclose(f);
}
```

Ispis programa na ekran:

```
Na poziciji      0 nalazi se znak P.
Na poziciji      1 nalazi se znak e.
Na poziciji      2 nalazi se znak r.
Na poziciji      3 nalazi se znak o.
Na poziciji      4 nalazi se znak  .
Na poziciji      5 nalazi se znak i.
Na poziciji      6 nalazi se znak  .
Na poziciji      7 nalazi se znak s.
Na poziciji      8 nalazi se znak t.
Na poziciji      9 nalazi se znak e.
Na poziciji     10 nalazi se znak f.
Na poziciji     11 nalazi se znak i.
Na poziciji     12 nalazi se znak c.
Na poziciji     13 nalazi se znak a.
Na poziciji     14 nalazi se znak ..
Na poziciji     15 nalazi se znak
.
Na poziciji     17 nalazi se znak A.
Na poziciji     18 nalazi se znak n.
Na poziciji     19 nalazi se znak t.
Na poziciji     20 nalazi se znak e.
Na poziciji     21 nalazi se znak ..
Na poziciji     22 nalazi se znak
.
```

Program započinjemo otvaranjem datoteke *znakovi.txt*. Zatim pamtimo trenutnu poziciju u varijablu poz. Nakon toga ulazimo u petlju gdje iz datoteke čitamo znak po znak funkcijom fgetc(). Ova funkcija se ponaša isto kao i već poznata funkcija getc(), samo što znakove čita iz datoteke. Ukoliko fgetc() uspije pročitati znak, vratiti će njegovu ascii vrijednost. U suprotnome će vratiti EOF što nam je signal da smo pročitali sve znakove iz datoteke. Numeracija znakova u datoteci počinje od nule. Zato se (prema očekivanjima) na poziciji 0 nalazi slovo 'P'. Sada uočite da nakon što je znak pročitan, trenutna pozicija se je pomaknula na slijedeću, za jedan veću! To je važno! Svako čitanje n-znakova iz datoteke pomiče trenutnu poziciju na za n veću! Isto vrijedi i za pisanje n znakova. Zbog toga čitanje slijedećeg znaka započinje sa pozicije 1, slijdećeg sa pozicije 2, itd. Obratite pažnju što se događa iza pozicije 14! Sa 14. pozicije čitamo znak '.'. Zatim sa 15. pozicije čitamo očito znak za prelazak u novi red, jer je pri ispisu točka završila u novom redu. No onda nastavljamo čitanje sa pozicije 17 ??? Što se je dogodilo? Kada u tekstualnom editoru (gdje je datoteka znakovi.txt stvorena) pritisnemo ENTER za prelazak u novi red, u datotekama pod DOS-om to se zapisuje kao slijed od dva znaka: ascii 13, te ascii 10, ili "\r\n". Kako je datoteka otvorena u tekstualnom modu, funkcija fgetc kada naiđe na ovaj slijed pročita oba znaka, i javi da je našla samo jedan: '\n'. Zbog toga se trenutna pozicija poveća za dva, a mi pročitamo samo jedan znak. Dodatno, pod UNIX-om ovo ne vrijedi, jer se tamo prelazak u novi red doista zapisuje samo '\n' znakom. Da je ova priča istinita, možemo se uvjeriti jednostavnom preinakom gornjeg programa. Umjesto za "rt", otvorimo datoteku za "rb" čime ćemo svim funkcijama reći da se čitanje vrši u binarnom modu, te nema muljaže oko pretvaranja slijeda "\r\n" u znak '\n'. Ispis tako modificiranog programa slijedi:

```
Na poziciji      0 nalazi se znak P.
Na poziciji      1 nalazi se znak e.
Na poziciji      2 nalazi se znak r.
Na poziciji      3 nalazi se znak o.
```

```
Na poziciji      4 nalazi se znak  .
Na poziciji      5 nalazi se znak i.
Na poziciji      6 nalazi se znak  .
Na poziciji      7 nalazi se znak s.
Na poziciji      8 nalazi se znak t.
Na poziciji      9 nalazi se znak e.
Na poziciji     10 nalazi se znak f.
Na poziciji     11 nalazi se znak i.
Na poziciji     12 nalazi se znak c.
Na poziciji     13 nalazi se znak a.
Na poziciji     14 nalazi se znak ..
.a poziciji     15 nalazi se znak
Na poziciji     16 nalazi se znak
.
Na poziciji     17 nalazi se znak A.
Na poziciji     18 nalazi se znak n.
Na poziciji     19 nalazi se znak t.
Na poziciji     20 nalazi se znak e.
Na poziciji     21 nalazi se znak ..
.a poziciji     22 nalazi se znak
Na poziciji     23 nalazi se znak
.
```

Sada vidimo i poziciju 15, i poziciju 16. Sa pozicije 15 pročitan je znak '\r' što se vidi po ispisu! Naime, prvo je ispisan tekst:
Na poziciji 15 nalazi se znak
te se je zatim ispisao znak '\r'. Kako je ovo specijalan znak, on se ne ispisuje već ima važnu funkciju: povratak na prvo slovo u trenutnoj liniji, što se i izvršava. Nakon toga još se ispisuje točka (očito preko slova N jer je znak '\r' poziciju slijedećeg ispisa postavio na početak reda!). Sa pozicije 16 čita se znak '\n' i ispisuje. Nakon toga se čitaju znakovi drugog reda, i ponovno kraj drugog reda: 22. pozicija '\r', 23. pozicija '\n'.

Ovdje smo upoznali funkciju koja vrača trenutnu poziciju u datoteci: *ftell()*. Funkcija koja mijenja trenutnu poziciju u datoteci jest funkcija *fseek()*.

PROMJENA TRENUTNE POZICIJE. PRIMJER 4.

Napisati ćemo program koji će u datoteci *znakovi.txt* svako pojavljivanje znaka *'e'* zamjeniti sa znakom *'E'*.

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

void main() {
  FILE *f;
  long poz,poz2;
  int c;

  f = fopen("znakovi.txt","rt+");
  if( f == NULL ) {
    fprintf(stderr, "Ne mogu otvoriti datoteku znakovi.txt\n");
    exit(1);
  }
  poz = ftell(f);
  while( (c = fgetc(f)) != EOF ) {
   if( c == 'e' ) {
    poz2 = ftell(f);
    fseek( f, poz, SEEK_SET );
    c = 'E';
    fputc(c,f);
```

```
    fseek( f, poz2, SEEK_SET );
   }
   poz = ftell(f);
  }
  fclose(f);
}
```

Program je opet vrlo sličan prethodnome. Prva razlika je mod otvaranja datoteke. Naime, kako moramo otvoriti datoteku za izmjene (upis), ali tako da stari sadržaj pri otvaranju ostane sačuvan, mod mora biti "rt+" (prema prvoj tablici), ili pak "rb+", trenutno je svejedno jer nam postupanje sa "\r\n" slijednom nije bitno. Nakon toga varijabla poz pamti trenutnu poziciju sa koje se čita znak u varijablu c. Nakon ovog čitanja pozicija se uvečava. Pročitani se znak uspoređuje sa traženim ('e'), i ukoliko je pronađen, u varijablu poz2 čitamo trenutnu poziciju na koju se nakon izmjene moramo vratiti (istina je da je poz2 jednako poz+1 jer smo pročitali jedan znak, ali demontracije radi...). Zatim se pozicioniramo na poziciju poz (jer smo sa nje pročitali znak 'e'), upisujemo novi znak funkcijom fputc, i ponovno se pozicioniramo na poziciju na kojoj smo stali sa čitanjem (poz2). Kada sve provjerimo i izmjenimo potrebno, zatvaramo datoteku.

Pogledajte kod još jednom, i uočite da između svakog čitanja (*fgetc*) i pisanja (*fputc*) dolazi (barem) jedan poziv funkcije *fseek*! Naime, bez ovoga program neće raditi ispravno. Dakle zapamtite pravilo:
PRELAZ SA ČITANJA NA PISANJE ILI OBRATNO MORA IĆI PREKO FUNKCIJE FSEEK!
Razlog za ovo leži u buferiranju datoteka, što ćemo spomenuti kasnije. Sada samo zapamtite pravilo!

Pogledajmo što nam sve nudi funkcija *fseek*. Ukoliko znamo točno na koju se poziciju želimo pozicionirati, koristiti ćemo poziv:
fseek(f, pozicija, SEEK_SET);
No ponekad je korisno pomaknuti se od trenutne pozicije za određen broj znakova! To možemo postići sa:
fseek(f, relativan_pomak, SEEK_CUR);
Npr. da bi se pomaknuli za jedan znak unatrag, pozvati ćemo fseek(f, -1L, SEEK_CUR); Sjećate li se pravila od malo prije? Što ako ste upravo nešto zapisali i sada želite čitati od pozicije gdje je pisanje baš stalo? To znači da ste već na pravoj poziciji pa funkciju fseek ne biste trebali niti zvati. Eh, da nema pravila! Ali Vi morate zvati fseek! Pa stvar je vrlo jednostavna: pozovite fseek(f, 0L, SEEK_CUR); dakle, pomaknite se za nula znakova od trenutne pozicije, a svejedno pozovete fseek! Treća mogućnost je pozicioniranje od kraja datoteke:
fseek(f, relativan_pomak, SEEK_END);
Npr. fseek(f, 0L, SEEK_END); pozicionirati će vas točno na kraj datoteke.

poziv funkcije: **`fseek( f, poz, od_kuda)`**	
od_kuda	*trenutna pozicija*
SEEK_SET	**trenutna pozicija = poz**
SEEK_CUR	**trenutna pozicija = trenutna pozicija + poz** (poz može biti i negativan)
SEEK_END	**trenutna pozicija = veličina_datoteke + poz**

USPOREDBA DATOTEKA. PRENOS BLOKA PODATAKA. PRIMJER 5.

Napisati ćemo jednostavnu funkciju koja će reći jesu li dvije datoteke jednake. U slučaju da su jednake treba vratiti 1, u slučaju da su različite, vratiti 0. U slučaju greške treba vratiti -1.

```
int UsporediDatoteke( char *ImePrve, char *ImeDruge ) {
  FILE *prva, *druga;
  int z1, z2;
  int jednake;

  prva = fopen(ImePrve, "rb");
  if( prva == NULL ) return -1;
  druga = fopen(ImeDruge, "rb");
  if( druga == NULL ) { fclose(prva); return -1; }
  jednake = 0;
  while( (z1 = fgetc(prva)) == (z2 = fgetc(druga)) ) {
   if( z1 == EOF ) { jednake = 1; break; }
  }
  fclose( prva );
  fclose( druga );
  return jednake;
}
```

Budući da nam je važan binarni sadržaj, datoteke otvaramo u binarnom modu. Čitamo znak po znak iz obje datoteke istovremeno, i tako dugo dok su pročitane vrijednosti iste, čitamo dalje. Ako se dogodi da su obje funkcije istovremeno vratile EOF, to nam je znak da su obje datoteke iste, jer su im svi znakovi prije toga isti. No ukoliko bi se pojavila dva različita znaka, ili pak jedna funkcija vrati znak a druga EOF, ispasti ćemo iz while petlje i varijabla *jednake* biti će postavljena na 0. Ovaj program radi. Savršeno. Osim što ima jednu manu. SPOROST!!! Naime, za svaki pojedini znak zovemo funkciju fgetc. Funkcija fgetc (ukoliko se izbjegne bufferiranje) za svaki znak pristupa uređaju na kojem se datoteka nalazi da bi pročitala taj znak. Dodatno, neprestano se provjerava je li funkcija otvorena i sl. I ovo sve za svaki pojedini znak! Ne bi li bilo bolje odjednom čitati po npr. 4096 podataka, pa ih onda usporediti u običnoj for petlji? Time bismo smanjili i pristupe disku, i pozive funkcija! Za ove svrhe postoje funkcije za čitanje i pisanje bloka podataka: fread i fwrite. Obje funkcije imaju iste parametre:
fread/fwrite (void *Buffer, int VelicinaObjekta, int BrojObjekata, FILE *f);
Prvi parametar je memorijska adresa od koje se podaci mogu čitati/upisivati. Funkcija vrača koliko je **objekata** uspjela pročitati.
Pogledajmo kako bismo isti posao ostvarili ovim funkcijama:

```
int UsporediDatoteke2( char *ImePrve, char *ImeDruge ) {
  FILE *prva, *druga;
  char z1[4096], z2[4096];
  int proc1, proc2, i;
  int jednake;

  prva = fopen(ImePrve, "rb");
  if( prva == NULL ) return -1;
  druga = fopen(ImeDruge, "rb");
  if( druga == NULL ) { fclose(prva); return -1; }
  jednake = 0;
  while( 1 ) {
   proc1 = fread(z1,1,4096,prva);
   proc2 = fread(z2,1,4096,druga);
   if(proc1 != proc2 ) break;
   if( proc1 == 0 ) { jednake = 1; break; }
   for( i = 0; i < proc1; i++ )
    if( z1[i] != z2[i] ) break;
   if( i < proc1 ) break;
  }
```

```
  fclose( prva );
  fclose( druga );
  return jednake;
}
```

Iz obje datoteke pokušavamo pročitati blokove od 4096 znakova, i pamtimo koliko smo zapravo objekata pročitali. Kod nas je objekt znak, pa je veličina objekta 1, broj objekata 4096. Koliko objekata pročitamo, biti će pohranjeno u proc1 odnosno u proc2. Ako smo pročitali različite količine podataka, datoteke su različite. Ako smo pročitali nula podataka iz obje datoteke, datoteke su iste. Inače provjeravamo pročitane znakove od 0-tog do proc1-1. Ako nađemo razliku, izaći ćemo iz for petlje te će 'i' biti manji od proc1; tada izlazimo i iz while petlje. Inače, ako su svi znakovi isti, čitamo slijedeći blok, itd.

Želio bih Vam skrenuti pažnju na jedan važan detalj. Uočite kako smo raspodijelili vrijednosti *VeličinaObjekta* i *BrojObjekata* pri pozivima funkcije *fread*. Nama je bilo važno da pročitamo određeni broj znakova. Stoga smo za *VeličinuObjekta* uzeli sizeof(char)=1, a za *BrojObjekata* dopustili smo maksimalno 4096 objekata. No uočite da ukoliko funkcija ne uspije pročitati svih 4096, već nešto manje, to ne znači grešku! Da smo za VeličinaObjekta postavili 4096, a za BrojObjekata 1, teoretski bismo čitali opet po 4096 znakova; no ukoliko funkcija ne uspije pročitati svih 4096 znakova, kao rezultat bi vratila nulu, što bi nama indiciralo grešku! Dakle, funkcija bi rekla da nije uspjela pročitati objekt. Ovakvo ponašanje funkcije u ovom nam slučaju ne odgovara, ali ono se da vrlo dobro iskoristiti u druge svrhe. Pretpostavimo da trebamo pročitati jednu strukturu koja je velika 200 bajtova (=znakova). Tada je bolje izabrati parametre VeličinaObjekta = 200, BrojObjekata = 1 nego obrnuto. Naime, funkcija fread vraća broj između nula i zadanog broja objekata. I to je ono što nama odgovara! Kako smo tražili čitanje jednog objekta, biti ćemo zadovoljni samo ako ga funkcija pročita cijeloga, i vrati broj 1. Da smo zamjenili vrijednosti parametara, dozvolili bismo funkciji da pročita od nula do 200 objekata (kod nas tada znakova). No što nama znači rezultat da je pročitano 173 znaka? Pa apsolutno ništa, jer mi želimo pročitati CIJELU strukturu, a ne samo dio strukture. Tada bi nam kao uvjet za uspješno čitanje bila komparacija rezultata funkcije fread sa 200, što želimo izbjeći. No promotrimo funkciju fread u još jednom primjeru.

ČITANJE STRUKTURA PODATAKA. PRIMJER 6.

U direktnoj neformatiziranoj datoteci "rezultati" nalaze se rezultati kolokvija. Rezultati su pohranjeni kao strukture sa članovima: matični broj (10 znakova), broj bodova prvog zadatka (integer), ..., broj bodova petog zadatka (integer). Treba napisati funkciju koja će izračunati koliko je studenata prošlo na kolokviju, ako se kao prolaz uzima broj bodova veći od p posto od maksimalnog broja bodova. Parametri p i maksimalni broj bodova predaju se funkciji. Ukoliko dođe do greške, kao rezultat vratiti -1.

```
int prosli( int p, int max ) {
 struct {
  char mbr[10];
  int bodovi[5];
 } zapis;
 int n, brbod;
 FILE *f;

 n = 0;
 f = fopen("rezultati", "rb");
 if( f == NULL ) return -1;
 brbod = max * p / 100;
 while( fread(&zapis,sizeof(zapis),1,f) ) {
  if( zapis.bodovi[0]+zapis.bodovi[1]+zapis.bodovi[2]+zapis.bodovi[3]+zapis.bodovi[4]
> brbod ) n++;
 }
 fclose( f );
 return n;
}
```

Kao prvo, za varijablu *zapis* definirali smo poseban tip podataka prema proloženoj specifikaciji. Jedno znakovno polje od deset znakova, te pet integer varijabli u nizu. Za ove varijable smo mogli navesti pet puta *int bodx*, gdje bi x prošetali od 1 do 5, ili je pak jednostavnije smisliti samo jedno ime, i dodijeliti mu jedno polje od 5 integera. Standardno otvaramo datoteku, te provjeravamo je li otvaranje uspjelo. Prije ikakvog danjeg posla pretpostavljamo da je kolokvij prošlo 0 ljudi (ah, taj optimizam). Sada uočite kako smo pametnom raspodjelom parametara *VeličinaObjekta* i *BrojObjekata* funkcije fread pojednostavnili while uvjet! Naime, kako smo rekli da čitamo samo jedan objekt, funkcija fread može ili ne pročitati objekt pa će vratiti 0, ili ga pročitati, pa će vratiti 1. No ovo je upravo što nam treba za while uvjet! Ili ćemo ući u tijelo (1), ili ćemo prekinuti petlju (0)! Zamislite da smo odabrali neku drugu kombinaciju za *VeličinaObjekta* i *BrojObjekata*. Recimo da je *VeličinaObjekta=1* i *BrojObjekata=sizeof(zapis)*. I ovo je sasvim u redu. Jedino što ćemo morati vršiti dodatna ispitivanja, tako da bi sada while uvijet trebao izgledati ovako:

```
while( fread(&zapis,1,sizeof(zapis),f) == sizeof(zapis) ) {
...
}
```

Dakle, osim što si kompliciramo život, uvodimo u igru i dodatna ispitivanja i komparacije. No vratimo se našem riješenju. Koliko dugo će se while petlja odvijati? Svakim čitanjem podataka približavamo se kraju datoteke. Jednom fread neće uspjeti pročitati novi zapis jer ovoga više neće biti, i vratiti će 0, što će prekinuti while petlju.

KOLIKO JE DATOTEKA VELIKA? PRIMJER 7.

Kada ljude pitate kako bi utvrdili koliko je datoteka velika, broj odgovora koje ćete dobiti postaje jako velik. Krenimo od realizacije nekih osnovnih ideja. Razgovarajući tako sa jednom osobom (X u nastavku), ideja koja je predložena jest slijedeća: otvorimo datoteku i čitajmo iz nje znak po znak. Svaki puta kada pročitamo znak, povečajmo brojač za jedan. Kada stignemo do kraja datoteke, brojač je jednak veličini datoteke.

```
long velicina1(char *ImeDatoteke) {
 long int n;
 FILE *f;

 n = 0;
 f = fopen(ImeDatoteke, "rb");
 if( f == NULL ) return -1;
 while( fgetc(f) != EOF ) n++;
 fclose( f );
 return n;
}
```

Pogledajte riješenje pa recite, što mislite? Ma mota mi se po glavi nekakav turbo, samo kakav? Ah, pa da. TURBO SPORO!!! Niti najgorem neprijatelju nemojte napisati ovakvo riješenje! Nikada! Za svaki znak zvati fgetc? Znate li vi koliko je to poziva! Koliko je to provjera? Ko zna što sve fgetc provjerava i na čemu gubi vrijeme! Ali nisam ja pričao za badava! Sjeti se X što sam ja govorio! Ako je znak po znak sporo, pročitajmo onda puno znakova odjednom! Eh...

```
long velicina2(char *ImeDatoteke) {
 long int n, t;
 char buffer[4096];
 FILE *f;

 n = 0;
 f = fopen(ImeDatoteke, "rb");
 if( f == NULL ) return -1;
 while( t = fread(buffer,1,4096,f) ) n+=t;
 fclose( f );
 return n;
}
```

Nema šta! Stvar leti! Ubrzali smo je u prosjeku samo 4096 posto (cca). Kao što vidite, uzeli smo buffer od 4096 bajtova, i kažemo funkciji fread da pokuša pročitati naravno svih 4096. Broj bajtova koliko uspije pročitati pospremiti ćemo u varijablu t. Ako je t jednak nula, tada se while prekida jer smo gotovi sa čitanjem. U protivnom, broj pročitanih bajtova pribrajamo n, i čitamo dalje. Naravno, i ovo riješenje ima daljnjih mogućih optimizacija. Naime, ako fread vrati broj manji od 4096, to jednostavno znači da u datoteci dalje nema bajtova, pa možemo odmah prestati sa čitanjem, a ne pozvati fread još jednom da nam ona vrati nula. Ali, to je trenutno nebitno. Zašto? Zato što je brzina (i) ove funkcije ovisna o veličini datoteke. Naime, više bajtova => više čitanja => više vremena! Kako je brzina obrnuto proporcionalna sa vremenom (ah, ta fizika!), više vremena => (čit. povlači) manju brzinu! I što ćemo sada? Pa ideja je jednostavna. Eliminirajmo utjecaj veličine datoteke na određivanje veličine datoteke (hm?). Da. Evo ideje.

```
long velicina3(char *ImeDatoteke) {
 long int n;
 FILE *f;

 f = fopen(ImeDatoteke, "rb");
 if( f == NULL ) return -1;
 fseek( f, 0L, SEEK_END );
 n = ftell(f);
 fclose( f );
 return n;
}
```

Malo kreativnosti i stvar je riješena! Pa već sam pokazao da funkciji fseek možemo reči da se pozicionira na kraj datoteke (preciznije, nula bajtova dalje od kraja datoteke). Zatim moramo samo funkciju ftell pitati koliko smo daleko od početka! I to je to. Zapravo, ne baš. Ima još jedan način, ako posegnemo u bogati fond funkcija iz *io.h*. Primjer slijedi.

```
#include <io.h>

long velicina4(char *ImeDatoteke) {
 long int n;
 FILE *f;

 f = fopen(ImeDatoteke, "rb");
 if( f == NULL ) return -1;
 n = filelength(fileno(f));
 fclose( f );
 return n;
}
```

Kao što vidite, u *io.h* postoji funkcija *filelength* koja vrača upravo veličinu datoteke! No, kako je to funkcija iz io.h, ona ne zna razlikova datoteke po *FILE** varijablama, već funkcija zahtijeva *handle* datoteke. Tu će nam u pomoć priskočiti jedan makro definiran u *stdio.h*, pod nazivom *fileno*. Naime, *fileno(FILE*f)* vrača handle datoteke *f*, koji proslijeđujemo dalje funkciji *filelength*.

Eto, štovane dame i gospodo, toliko od mene za ovaj puta!

Marko Čupić
Zagreb, 6. srpnja, 1999.