Zbrani zapiski za priprave na računalniška tekmovanja

Šolsko leto 2022/23

Kazalo

Poglavje 1

Pisanje in branje

1.1 Vhod in izhod

Programi za svoje delovanje potrebujejo način za komunikacijo z uporabnikom. Kompleksnejši programi v ta namen uporabljajo ekran, miško in tipkovnico. Pri tekmovalnem programiranju pa najpogosteje uporabljamo najpreprostejši način za komunikacijo: pisanje in branje s *standardnega vhoda in izhoda*. Običajno to pomeni, da se nam ob zagonu programa odpre okno, kamor lahko pišemo programu in kamor program izpisuje stvari.
Ko želimo, da naš program kaj izpiše, uporabimo *funkcijo* `printf`.

Primer

```c
#include<stdio.h>

int main(){
    printf("Hello World!\n");
    return 0;
}
```

Primer vhoda in izhoda

```
Hello World!
```

`printf` - funkcija, ki ji v dvojnih narekovajih damo besedilo ali števila, ki jih želimo izpisati
`\n` - znak za novo vrstico

Stvari, ki jih mora vsebovati (skoraj) vsak program:

- `stdio.h` - *knjižnica* (datoteka), ki vsebuje funkcije, ki jih bomo uporabljali v programu (kot npr. `printf` in `scanf`)
- `#include<>` - ukaz, s katerim našemu programu povemo, katere knjižnice potrebuje
- `int main(){}` - telo našega programa - večino kode v programu napišemo med zavite oklepaje
- `return 0` - zadnja vrstica v programu, ki sporoča računalniku, da se je pravilno zaključil

V program lahko dodamo *komentarje*. To je takšno besedilo, ki je napisano v kodi, a vsebinsko ne vpliva na program.

- `//` - s tem zakomentiramo vse od poševnic do konca vrstice
- `/* */` - s tem lahko zakomentiramo več vrstic ali del znotraj vrstice

Primer

```
#include<stdio.h>
int main(){ //komentar do konca vrstice
    printf /*komentar znotraj vrstice*/("Zivjo svet!\n");
    /*
    komentar
    čez
    več
    vrstic
    */
    return 0;
}
```

Primer vhoda in izhoda

```
Zivjo svet!
```

Pogoste napake

Zadnji znak, ki ga program izpiše, mora biti `\n`.

Pogoste napake

Večina vrstic v programu se konča s podpičjem (`;`) (skoraj vse razen tistih, ki se končajo z oklepaji ali zavitimi zaklepaji). Brez tega program ne bo delal.

1.2 Branje

Program za branje stvari, ki mu jih sporočamo, uporablja funkcijo `scanf`.

Primer

```c
#include <stdio.h>

int main(){
    char ime[50];
    printf("Kako ti je ime?\n");
    scanf("%s", ime);
    printf("Zivjo %s!\n", ime);
    return 0;
}
```

Primer vhoda in izhoda

```
izhod:  Kako ti je ime?
vhod:   Tinka
izhod:  Zivjo, Tinka!
```

Če želimo, da program lahko kaj počne s podatki, ki jih je prebral, moramo najprej to shraniti na neko mesto v spominu. Temu mestu rečemo *spremenljivka*, saj lahko s programom spreminjamo, kaj je tam shranjeno. Spremenljivke v svojem programu poimenujemo, v našem primeru ji rečemo `ime`. Vsaki spremenljivki moramo določiti *podatkovni tip*, saj lahko beremo in pišemo več različnih vrst podatkov, npr. besede ali števila.

`char` - s tem povemo, da je naša spremenljivka besedilo oz. *niz*

`[...]` - številka v oglatih oklepajih za besedo pove največjo dolžino niza, ki ga lahko program prebere.

Pogoste napake

Ko določamo največjo dolžino niza, vedno vzamemo večjo številko, kot jo bomo potrebovali. O tem bomo več govorili v kasnejših poglavjih.

Funkciji `scanf` podamo dva ali več *parametrov*:

- kaj naj prebere, torej kakšne tipe spremenljivk. To podamo s *formatnikom* (v našem primeru `%s`, ki pomeni niz (*string*). `%s` prebere vse znake do prvega presledka ali nove vrstice)
- ostali parametri povejo, kam naj funkcija shrani stvari, ki jih je prebrala (torej v spremenljivke, ki smo jih naredili prej)

Do zdaj smo funkciji `printf` podali samo točno določeno besedilo, ki smo ga želeli izpisati. Izpisujemo pa lahko tudi spremenljivke, kot smo to naredili v tem zadnjem primeru. Znotraj besedila dodamo formatnike na mesta, kjer želimo, da so spremenljivke, potem pa izven narekovajev naštejemo imena spremenljivk, ki jih želimo izpisati (tako kot pri funkciji `scanf`).

1.3 Branje števil

Poleg nizov lahko programi delajo tudi s števili. Števila beremo in izpisujemo z istima funkcijama kot nize, vendar moramo uporabiti drug formatnik.

Primer

```c
#include<stdio.h>

int main(){
    int razred;
    printf("Kateri razred si?\n");
    scanf("%d", &razred);
    printf("%d. razred je najboljši.\n", razred);
    return 0;
}
```

Primer vhoda in izhoda

```
izhod:  Kateri razred si?
vhod :  7
izhod:  7. razred je najboljši.
```

`&` - znak, ki ga moramo dati pred ime spremenljivke vedno, kadar beremo števila.
`int` - podatkovni tip števil.
Pri številih za imenom spremenljivke ne povemo, kako velika so lahko.

Za branje in pisanje števil uporabimo formatnik `%d`.

Pogoste napake

Ko beremo števila, moramo pred ime spremenljivke dati znak `&`, česar pri branju nizov ne delamo. Prav tako tega ne delamo pri izpisovanju števil. Več o tem bomo povedali v kasnejših poglavjih.

Poglavje 2

Računske operacije

2.1 Seštevanje, odštevanje, množenje

Računalniki lahko s spremenljivkami počnejo veliko stvari. Najpreprostejše so operacije na številih, kot so seštevanje, odštevanje in množenje. Račune zapisujemo tako kot v šoli, z *operatorji*.

- `+` za seštevanje
- `-` za odštevanje
- `*` za množenje
- `/` za celoštevilsko deljenje
- `%` za ostanek pri deljenju

Primer

```c
#include<stdio.h>

int main(){
    int a, b, vsota, razlika, produkt;
    scanf("%d%d", &a, &b);
    vsota = a+b;
    razlika = a-b;
    produkt = a*b;
    printf("%d\n%d\n%d\n", vsota, razlika, produkt);
    return 0;
}
```

Primer vhoda in izhoda

```
3 7
```

```
12
-4
21
```

Negativna števila

Na meteorološki postaji Kredarica so leta 2014 izmerili povprečno januarsko temperaturo približno $-5\,^{\circ}\mathrm{C}$, povprečno avgustovsko pa približno $6\,^{\circ}\mathrm{C}$. Med tema meritvama je $11\,^{\circ}\mathrm{C}$ razlike.
Pozimi lahko izmerimo temperature, manjše od 0. Takšnim številom, kot je -5, rečemo *negativna števila*.
Lahko jih uporabimo tudi drugje, ne samo pri merjenju temperature.
S pozitivnimi števili lahko štejemo od 0 do ∞ (1, 2, 3, ...), z negativnimi pa do $-\infty$. (-1, -2, -3, ...).
Tako kot pozitivna števila jih lahko seštevamo in odštevamo:

Primer

5 - 11 = -6
-6 + 11 = 5
5 -(-6) = 11
-2 - 1 = -3
-2 - (-1) = -1 -2 + (-1) = -3

Pravila:
-(-6) = +6
+(-1) = -1
-(-(-(-1))) = -(-(+1)) = -(-1) = 1

Primer

2 * (-5) = -10
(-5) * (-5) = 25

Pravila: Če množimo dve pozitivni števili, je produkt pozitiven.
Če množimo eno pozitivno in eno negativno število, je produkt negativen.
Če množimo dve negativni števili, je produkt spet pozitiven.
O negativnih številih lahko razmišljamo kot: -3 = (-1) * 3

Primer

Računalniki z negativnimi števili računajo enako kot s pozitivnimi.

```c
#include<stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    printf("Vsota: %d\nRazlika: %d\nProdukt: %d\n", a + b, a - b, a * b);
    return 0;
}
```

Primer vhoda in izhoda

```
3 7
```

```
Vsota:  10
Razlika:  -4
Produkt:  21
```

2.2 Deljenje

Števila lahko tudi delimo. Za deljenje uporabljamo znak /.

Primer

```c
#include<stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d\n", a/b);
    return 0;
}
```

Primer vhoda in izhoda

```
16 8
```

```
2
```

Pogoste napake

Deljenje v jezikih C in C++ je celoštevilsko. Deljenje z ostankom lahko zapišemo po formuli $a = k * b + o$.

16/8: 16 = 2*8 + 0
Ostanek je 0, ker je 16 deljivo z 8.

16/5: 15 = 3*5 + 1
ostanek je 1.

Celoštevilsko deljenje pomeni, da nam program vrne samo k. Primer vhoda in izhoda za zgornji program, kjer števili nista deljivi:

Primer vhoda in izhoda

```
16 3
```

```
5
```

Tudi, če bi bil rezultat deljenja 7.9, tega program ne zaokroži na 8, temveč nam vrne 7.
Če manjše število delimo z večjim, zato vedno dobimo 0.

Operator `/` nam torej pri deljenju `a/b` vrne `k`. Lahko pa dobimo tudi ostanek `o`. Do njega pridemo z operatorjem `%` (*modulo*).

Primer

```c
#include<stdio.h>

int main(){
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d = %d*%d + %d\n", a, a/b, b, a%b);
    printf("Količnik: %d\nOstanek: %d\n", a/b, a%b);
    return 0;
}
```

Primer vhoda in izhoda

```
25 7
```

```
25 = 3*7 + 4
Količnik: 3
Ostanek: 4
```

Pogoste napake

Deljenje z 0 v matematiki ni definirano in prav tako ne v programiranju. Če neko število delimo z 0, se nam bo program sesul. Prav tako, če poskušamo izračunati ostanek pri deljenju z 0. Ta napaka se pogosto zgodi v programih, kjer delimo z več števili, zato moramo biti na to pozorni.

Poglavje 3

Pogojni stavki

Pogosto želimo, da računalnik izvaja drugačno kodo glede na vrednost ene ali večih spremenljivk, npr. da nam pokaže drugačno vsebino, če smo napisali pravilno ali napačno geslo, da računalo sešteva, če smo pritisnili gumb za seštevanje, oz. odšteva, če smo pritisnili gumb za odštevanje, ipd. Z drugimi besedami, želimo upravljati potek programa (torej izbrati, katera koda naj se izvede) glede na vrednosti spremenljivk. Angleško takemu upravljanju pravimo *control flow*, najpogosteje pa ga izvajamo s t.i. *pogojnimi* ali *if* stavki. Osnovna struktura je sledeča:

Primer

```
if (pogoj) {
    // koda, ki se izvede, ce pogoj velja
} else {
    // koda, ki se izvede, ce pogoj ne velja
}
```

Pogoj je nov pojem. Označuje neke vrste račun, katerega rezultat ni število, vendar *logična vrednost*. Tu sta možni vrednosti le dve: pravilno (angl. `true`) in napačno (angl. `false`). Če bo rezultat računa, navedenega v običajnih oklepajih v zgornjem `if` stavku, `true`, se bo izvedla koda znotraj prvih zavitih oklepajev, če pa je rezultat računa `false`, pa se bo izvedla koda v drugih zavitih oklepajih (tistih za besedo `else`). Drugega dela, t.j. `else` in oklepaji za njim, ni treba pisati, če ga ne želimo.
Kako pa zapišemo pogoj? Za to uporabimo posebne logične operatorje. Pri delu s številkami so nam na voljo naslednji:

- `==`: primerja dve številski vrednosti. Rezultat je `true`, če sta vrednosti enaki.
- `!=`: primerja dve številski vrednosti. Rezultat je `true`, če sta vrednosti različni.
- `<`: primerja dve številski vrednosti. Rezultat je `true`, če je vrednost na levi manjša od vrednosti na desni.
- `>`: deluje podobno kot `<`, le da v drugo smer; rezultat je `true`, če je vrednost na desni manjša od vrednosti na levi.
- `<=`: primerja dve številski vrednosti. Rezultat je `true`, če sta vrednosti enaki, ali če je vrednost na levi manjša od vrednosti na desni.
- `>=` deluje podobno kot `<=`, le da v drugo smer.

Primer

Poglejmo si primer uporabe pogojnega stavka.

```c
#include <stdio.h>

int main() {
    int a, b;
    scanf("%d%d", &a, &b);
    if (a == b) {
        printf("Stevili sta enaki.\n");
    }
    if (a != b) {
        printf("Stevili sta razlicni.\n");
    }
    if (a < b) {
        printf("Prvo stevilo je manjse od drugega.\n");
    }
    if (a > b) {
        printf("Prvo stevilo je vecje od drugega.\n");
    }
    if (a <= b) {
        printf("Prvo stevilo je manjse ali enako drugemu.\n");
    }
    if (a >= b) {
        printf("Prvo stevilo je vecje ali enako drugemu.\n");
    }
    return 0;
}
```

Primer vhoda in izhoda

```
12 12
```

```
Stevili sta enaki.
Prvo stevilo je manjse ali enako drugemu.
Prvo stevilo je vecje ali enako drugemu.
```

Primer vhoda in izhoda

```
3 7
```

```
Stevili sta razlicni.
Prvo stevilo je manjse od drugega.
Prvo stevilo je majnse ali enako drugemu.
```

Pogoste napake

Pri primerjavi moramo uporabiti dva enačaja (==). Če uporabimo samo en enačaj, kot v matematiki (torej =), se bo program sicer zagnal, vendar ne bo deloval pravilno. Enojnega enačaja nikoli ne uporabljamo v pogoju `if` stavka.

Pogoste napake

Taka primerjava deluje samo na številih. Če želimo primerjati dve besedili, za to potrebujemo posebno funkcijo `strcmp`, ki jo bomo podrobneje spoznali v sledečih poglavjih.

Primer

Primer uporabe stavka `else`. Program bo uporabnika vprašal za PIN, in mu napisal, če je bil PIN pravilen oziroma napačen.

```c
#include <stdio.h>

int main() {
    // Vprasaj uporabnika za PIN, in preveri, ce je pravilen
    int pin;
    scanf("%d", &pin);
    if (pin == 42) {
        printf("PIN je pravilen!");
    } else {
        printf("PIN je napacen!");
    }
    return 0;
}
```

Primer vhoda in izhoda

```
42
```

```
PIN je pravilen!
```

Primer vhoda in izhoda

```
7
```

```
PIN je napacen!
```

Primer

`if` stavke lahko tudi gnezdimo; torej vstavimo enega v drugega. Spodnji program od uporabnika sprejme naročilo v restavraciji, kjer ponujajo dve vrsti hrane; juhe in sendviče. Na voljo sta dve vrsti juhe, in dve vrsti sendvičev.

```c
#include <stdio.h>
int main() {
    // Uporabnik naj napise 1, ce zeli juho, in 2, ce zeli sendvic.
    int zelja;
    scanf("%d", &zelja);
    if (zelja == 1) {
        // Uporabnik naj napise 1, ce zeli govejo juho,
        // in 2, ce zeli paradiznikovo.
        scanf("%d", zelja);
        if (zelja == 1) {
            printf("Ena goveja juha. Dober tek!\n");
        } else {
            printf("Ena paradiznikova juha. Dober tek!\n");
        }
    } else {
        // Uporabnik naj napise 1, ce zeli sendvic s sunko,
        // in 2, ce zeli vegeterjanski sendvic.
        scanf("%d", &zelja);
        if (zelja == 1) {
            printf("En sendvic s sunko. Dober tek!\n");
        } else {
            printf("En vegeterjanski sendvic. Dober tek!\n");
        }
    }
    return 0;
}
```

Primer vhoda in izhoda

```
2
1
```

```
En sendvic s sunko. Dober tek!
```

Pogoste napake

Pozorni bodimo tudi na postavitev kode. Običajno kodo znotraj zavitih oklepajev `if` stavka pišemo tako, da je poravnana štiri presledke bolj desno od kode zunaj `if` stavka. V nekaterih programskih jezikih je taka poravnava obvezna, v C/C++ pa ne, vendar nezamaknjena koda že v zelo majhnih programih postane popolnoma nepregledna. Branje in popravljanje kode je veliko lažje, če del kode znotraj zavitih oklepajev zamaknemo za štiri presledke. Za to lahko uporabimo tudi tipko Tab, ki se na tipkovnici nahaja levo od tipke Q.

Tako **nikoli** ne pišemo:

```
#include <stdio.h>
int main() {
if (3 > 2) {
printf("Velja\n");
} else {
printf("Ne velja\n");
}
return 0;
}
```

3.1 Nizanje pogojev

Pogosto se srečamo s problemi, kjer je za rešitev potrebno upoštevati več kot en pogoj. V takih primerih želimo združiti več pogojnih stavkov tako, da se nek del kode izvede, če velja prvi pogoj, drugi del kode pa, če prvi pogoj ne velja, velja pa drugi pogoj. Z gnezdenjem stavkov lahko to v kodo vključimo na naslednji način:

```
if (prvi pogoj) {
    // koda, ki se izvede, če velja prvi pogoj
} else {
    if (drugi pogoj) {
        // koda, ki se izvede, če prvi pogoj ne velja,
        // velja pa drugi pogoj
    } else {
        // koda, ki se izvede, če ne veljata ne prvi ne drugi pogoj
    }
}
```

V takem primeru nam je na voljo bližnjica `else if`. Zgornja koda deluje popolnoma enako kot spodnja:

```
if (prvi pogoj) {
    // koda, ki se izvede, če velja prvi pogoj
} else if (drugi pogoj) {
    // koda, ki se izvede, če prvi pogoj ne velja,
    // velja pa drugi pogoj
} else {
    // koda, ki se izvede, če ne veljata ne prvi ne drugi pogoj
}
```

Prednost te bližnjice je, da je naša koda krajša in bolj razumljiva. Stavke `else if` lahko tudi verižimo; enemu `if` stavku lahko sledi poljubno mnogo stavkov `else if`. Pri tem bo računalnik pogoje preverjal po vrsti. Pri prvem veljavnem pogoju se bo ustavil in izvedel kodo v pripadajočih zavitih oklepajih,

za čimer ne bo več preverjal pogojev, temveč bo izvajanje nadaljeval za zaključkom vseh nanizanih stavkov.
Kakor nam v osnovnem `if` stavku ni bilo treba pisati dela z `else`, če ga nismo potrebovali, nam ga tudi pri uporabi `else if` ni treba.

Primer

Naslednji program prebere število in pove, če je večje, manjše ali enako 0.

```c
#include <stdio.h>

int main() {
    int stevilo;
    scanf("%d", &stevilo);
    if (stevilo < 0) {
        printf("Stevilo je manjse od nic.\n");
    } else if (stevilo == 0) {
        printf("Stevilo je enako 0.\n");
    } else if (stevilo > 0) {
        printf("Stevilo je vecje od 0.\n");
    }
    return 0;
}
```

Primer vhoda in izhoda

```
3
```

```
Stevilo je vecje od 0.
```

Primer

Pogoji se preverjajo po vrsti; izvede se samo koda pri prvem veljavnem pogoju, ne glede na to, koliko pogojev za njim je tudi veljavnih.

```
#include <stdio.h>

int main() {
    int a = 7;
    if (a < 3) {
        printf("a je manjsi od 3.\n");
    } else if (a == 7) {
        printf("a je 7.\n");
    } else if (a == 4) {
        printf("a je 4.\n");
    } else if (a > 1) {
        printf("a je vecji od 1.\n");
    } else {
        printf("Nic od nastetega ne velja.\n");
    }
    return 0;
}
```

Program izpiše le eno stvar - `a je 7.`, kljub temu, da velja tudi `a > 1`.

3.2 Logični vezniki

Osnovni operatorji za primerjavo pogosto niso dovolj, da izrazimo željen pogoj. Če na primer želimo pogledati, ali je neko število med dvema drugima, tega ne moramo narediti samo z eno primerjavo. Pogoje združujemo s t.i. *logičnimi vezniki*, ki jim včasih pravimo tudi *logični operatorji*. Poznamo tri osnovne veznike:

- `and` oz. `&&` združi dva pogoja tako, da združeni pogoj velja samo v primeru, da veljata oba hkrati.
- `or` oz. `||` združi dva pogoja tako, da združeni pogoj velja v primeru, da velja katerikoli od dveh, ali da veljata oba
- `not` oz. `!` sprejme samo en pogoj. Nov pogoj velja samo takrat, ko originalni pogoj ne velja.

Logična veznika `&&` (*in*) ter `||` (*ali*) lahko predstavimo z resničnostno tabelo.

Tabela 3.1: Resničnostna tabela za *in*

Leva vrednost	Desna vrednost	Vrednost veznika
resnica	resnica	resnica
resnica	neresnica	neresnica
neresnica	resnica	neresnica
neresnica	neresnica	neresnica

Tabela 3.2: Resničnostna tabela za *ali*

Leva vrednost	Desna vrednost	Vrednost veznika
resnica	resnica	resnica
resnica	neresnica	resnica
neresnica	resnica	resnica
neresnica	neresnica	neresnica

Primer

Če želimo preveriti, ali je dano število med dvema drugima, uporabimo logični veznik `&&`.

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    if (3 < n && n < 9) {
        printf("n je med 3 in 9.\n");
    }
    return 0;
}
```

Primer vhoda in izhoda

```
5
```

```
n je med 3 in 9.
```

Pogoste napake

V matematiki pogosto napišemo dvojno primerjavo: `a < b < c`. Če nekaj podobnega napišemo v C++ program, se bo le-ta sicer zagnal, vendar ne bo deloval pravilno. Kaj pričakujemo, da se zgodi, če v tako primerjavo zapišemo `3 < 2 < 1`? Kaj pa se dejansko zgodi?

Pri kombiniranju pogojev bodimo previdni glede pravil prednosti. Zanikanje (veznik `not` oz. `!`) ima namreč prednost pred primerjalnimi vezniki (`<`, `==`, ...), ter drugima ločnima veznikoma. Če v takem primeru uporabljamo zanikanje, moramo zanikan izraz postaviti v oklepaje.

Primer

Zanikanje se lahko v večini primerov zapiše tudi na krajši način.

Izraz	Ekvivalenten izraz	Komentar
`!(a == b)`	`a != b`	Če sta `a` in `b` enaka, bo prvi izraz veljaven.
`!(a < b)`	`a >= b`	
`!(a <= b)`	`a > b`	

Primer

Napišimo program, ki preveri, ali je uporabnik vpisal prestopno leto. Leto je prestopno, če je deljivo s 4; razen, če je hkrati deljivo s 100. Izjema so leta, deljiva s 400, ki so prestopna kljub temu, da so deljiva s 100.
Kako te pogoje zapišemo v program? Opazimo, da so leta, deljiva s 400, prestopna ne glede na druga pogoja. Če leto ni deljivo s 400, potem mora biti deljivo s 4 in ne sme biti deljivo s 100. Povedano krajše; leto mora biti deljivo s 400, ali pa s 4 in ne hkrati s 100. Tak pogoj lahko zapišemo z logičnimi vezniki.

```c
#include <stdio.h>

int main() {
    int leto;
    scanf("%d", &leto);
    if (leto % 400 == 0 || (leto % 4 == 0 && !(leto % 100 == 0))) {
        printf("Leto je prestopno.\n");
    } else {
        printf("Leto ni prestopno.\n");
    }
    return 0;
}
```

Poglavje 4

Zanke

4.1 Kako napišemo zanko?

V programiranju pogosto želimo nek del kode ponoviti, zato imamo *zanke*. Zanka, ki jo bomo najpogosteje uporabljali je *for* zanka, ki deluje na princip **začetka**, **pogoja** in **koraka**.

Primer

```c
#include <stdio.h>

int main() {
    // začetek pogoj korak
    for (int stevec=0; stevec < 3; stevec++) {
        // koda, ki se izvedce vsako zanko
        printf("nekaj\n");
    }
    return 0;
}
```

Primer vhoda in izhoda

```
nekaj
nekaj
nekaj
```

Poglejmo si, kako ta program deluje. Podpičja v vrstici
`for (int stevec=0; stevec < 3; stevec++) {`
razdelijo okrogle oklepaje na tri dele; začetek, pogoj in korak. Začetek se bo izvedel, ko se ta zanka začne. Vsakič preden se izvede koda v notranjosti zanke se preveri pogoj, če drži, se bo še enkrat izvedla koda v zanki, sicer se bo pa zanka končala. Korak je podoben začetku, le da se izvede na koncu vsake ponovitve zanke.
Tu začetek naredi novo številko *stevec* in jo nastavi na 0. Pogoj preveri, če je *stevec* manjši od 3. Korak `stevec++` je pa okrajšava za `stevec = stevec + 1`, torej poveča *stevec* za 1.

- Program se začne in pride do for zanke, najprej se izvede začetek `int stevec=0`.
- Zdaj se je začela zanka, preveri se pogoj `stevec < 3`. Ker je *stevec* za zdaj še 0, je pogoj izpolnjen. Izvede se vsebina zanke, torej program izpiše *nekaj*. Zdaj smo prišli do konca zanke, izvede se korak, *stevec* se poveča na 1, program pa skoči nazaj na začetek zanke.

- Ker smo na začetku zanke se preveri pogoj, `stevec < 3`, ker je *stevec* zdaj 1 je pogoj še vedno izpolnjen, zato se izvede vsebina zanke. Ko program še enkrat izpiše *nekaj* izvede korak, *stevec* poveča na 2 in skoči nazaj na začetek.
- Spet smo na začetku, zato se preveri pogoj, *stevec* je zdaj 2, kar je manjše od 3, zato se *nekaj* spet izpiše. Program poveča *stevec* na 3 in skoči nazaj na začetek.
- Ker smo spet na začetku se bo še enkrat preveril pogoj, a zdaj je *stevec* enak 3 in 3 ni manjše od 3, zato se for zanka konča. Ker je naslednji ukaz `return 0;` se bo program tam končal.

Vidimo, da bo res izpisal *nekaj* trikrat. Vredno je omeniti, da je naš števec zavzel vrednosti 0, 1, 2, kar se mogoče zdi čudno, glede na to, da bi ponavadi šteli do tri kot 1, 2, 3. Štetje od nič pa je v programiranju bolj naravno kot štetja od ena, kakor bomo videli v sledečih poglavjih.

4.2 Razni primeri uporabe for zanke

4.2.1 Spreminjanje dolžine for zanke

Zanka, ki smo jo napisali zgoraj, se bo vedno ponovila trikrat, kaj pa če hočemo da se zanka ponovi glede na neko število na vhodu? Seveda je tudi to mogoče in sicer tako, da vstavimo našo spremenljivko v pogoj for zanke. Poglejmo si primer:

Primer

Program, ki dobi število in nariše puščico te dolžine. Še en trik tu je, da printf-ja v for zanki ne končamo z \n, kar doseže to, da so v izhodu pomišljaji eden zraven drugega v isti vrstici in ne vsak v svoji vrstici.

```
#include <stdio.h>

int main() {
    int dolzina;
    scanf("%d", &dolzina);
    for (int stevec=0; stevec < dolzina; stevec++)
        printf("-");
    // zaviti oklepaji {} niso potrebni, če je v notranjosti
    // zanke le ena vrstica kode

    printf(">\n");
    return 0;
}
```

Primer vhoda in izhoda

```
4
```

```
---->
```

4.2.2 Branje števil v for zanki

Ena od moči računalnikov je zelo hitra obdelava velike količine podatkov, računalnik bo zlahka seštel 1000 števil, medtem ko bi bilo to početi na roko precej zamudno. Poglejmo si, kako bi napisali program, ki bi nekaj izračunal z več števili.

Primer

Najprej preberemo eno število, recimo mu `n`. Za tem moramo prebrati še `n` števil in jih sešteti.

```
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);
    int vsota = 0;
    for (int i=0; i < n; i++) {
        int sestevanec;
        scanf("%d", &sestevanec);
        vsota += sestevanec;
    }
    printf("%d\n", vsota);
    return 0;
}
```

Primer vhoda in izhoda

```
4
12
13
8
1
```

```
34
```

Najprej preberemo `n` in naredimo novo spremenljivko z imenom `vsota`, v katero bomo sešteli vsa dana števila.
Opazimo, da je v for zanki namesto `stevec` zdaj uporabljen `i`. To je tradicionalno ime za spremenljivke v zankah. V notranjosti zanke so zdaj trije ukazi. Najprej naredimo novo spremenljivko, ki jo poimenujemo `sestevanec` in preberemo naslednje število iz vhoda. Nato pa z okrajšavo `vsota += sestevanec` prištejemo spremenljivki `vsota` spremenljivko `sestevanec`. Na daljše bi to lahko napisali kot `vsota = vsota + sestevanec`.

4.2.3 For zanka z drugačnim korakom in začetkom

Do zdaj so vse naše for zanke imele obliko `for (int i = 0; i < 10; i++)`, torej so začele na nič in se nekajkrat ponovile. Zapis for zanke, ki ga imamo v C++, pa lahko naredi veliko več. Poglejmo si kot primer zanko, ki izpiše vsa soda števila med 1 in 100.

Primer

Pozorno poglejmo števila, ki jih moramo izpisati. Ker 1 ni sodo, bo prvo izpisano število 2. Število 3 prav tako ni sodo, tako da bomo izpisali 4, po tem pa 6, 8, 10 in tako dalje. Vidimo, da vsak korak povečamo izpisano število za 2.

```
#include <stdio.h>

int main() {
    for (int i=2; i<=100; i += 2) {
        printf("%d\n", i);
    }
    return 0;
}
```

Primer vhoda in izhoda

```
2
4
6
8
10
12
⋮
96
98
100
```

Ker se želena števila začnejo z dva, bomo v začetni del for zanke vpisali `int i=2`. Ker hočemo izpisati števila med 1 in 100 in ne med 1 in 99, bomo v pogojnem delu uporabili znak manjše ali enako `i <= 100` (pogoj `i < 100` je neresničen v primeru `i = 100`). Ker želimo povečati naše število za 2 v vsakem koraku, smo v polje za korak napisali `i += 2`. Edina stvar, ki jo naredimo v notranjosti napisane zanke pa je, da izpišemo trenutno vrednost spremenljivke `i`.

4.3 Zanka `while`

Poleg zanke `for` poznamo tudi zanko `while`, ki deluje podobno, vendar ima le pogoj.

Primer

Program prebere števila na vhodu in izpiše njihovo vsoto.

```c
#include <stdio.h>

int main() {
    int n;
    scanf("%d", &n);

    int vsota = 0;
    while (n > 0) {
        int t;
        scanf("%d", &t);
        vsota += t;
        n = n - 1;
    }
    printf("%d\n", vsota);
    return 0;
}
```

Poglavje 5

Nizi in besedilo

5.1 Uvod

Poleg dela s številkami od računalnika pogosto želimo, da nekaj naredi z nizi besedila. Primeri takšnih programov so npr. urejevalniki besedila, ki jih uporabljamo tako za pisanje "enostavnega" besedila (koda), kot tudi za razna obogatena besedila. Pravzaprav pa skoraj vsak računalniški program dela z besedilom; kadarkoli želimo uporabniku prikazati neke informacije, jih moramo namreč izpisati na zaslon. Ko smo delali s številkami, smo problem izpisovanja prepustili računalniku, ker je kodo za branje in izpisovanje številk k sreči napisal že nekdo drug. Za pisanje splošnih programov pa tovrstno znanje ne bo dovolj; zato si poglejmo osnove dela z besedili.
Pri slovenščini se naučimo, da je besedilo sestavljeno iz več odstavkov, odstavek iz več povedi, poved iz več stavkov, stavek iz več besed, besede pa iz več črk. Pri tem se moramo zavedati, da stavke ločimo z ločili (vejice, pike, klicaji, itd.), besede ločimo s presledki, posamične odstavke pa ločimo z zamikanjem prve povedi v desno. Za predstavitev v računalniku je tak model preveč zakompliciran, zato vzamemo bolj enostavnega. Besedila bomo predstavili z *nizi* (angl. *string*), ki bodo zaporedja več *znakov* (angl. *character*). Vse, kar bi si kadarkoli zaželeli izpisati, bomo proglasili za znak. Tako si bomo vse črke predstavljali kot znak, kjer bomo ločili tudi med velikimi in malimi črkami (saj vendar izgledajo drugače, če jih napišemo), prav tako bomo za znake proglasili tudi ločila, oklepaje in simbole za matematične operacije (`+, -, *, /`). Poleg tega bomo za znak proglasili tudi števke od 0 do 9, ker tudi njih izpišemo.
Nenazadnje bomo ustvarili še nekaj posebnih znakov, ki jih morda nebi pričakovali. Od teh bomo zdaj spoznali tri: znak za presledek, znak za novo vrstico in znak za konec besedila. Znak za presledek bomo uporabili, kjerkoli želimo imeti prostor med dvema besedama (torej presledek). Za razliko od slovenščine tudi presledke obravnavamo, kot da bi bili pravzaprav neke posebne črke. Znak za novo vrstico bomo uporabili tam, kjer želimo, da izpis našega programa skoči vrstico nižje; brez tega znaka nam bo program vse izpisal v eni zelo dolgi vrstici besedila. Ta znak označimo s posebno kodo `\n` (ker se v angleščini ta znak imenuje *newline*), opazimo pa, da smo ga pravzaprav že srečali čisto na začetku. Uporabili bomo tudi znak za konec besedila, ki ga označimo z `\0`, pogosto pa mu rečemo tudi *null*. Več o temu znaku bomo povedali kasneje.

5.2 Predstavitev znakov

Preden si pogledamo nize, moramo razumeti, kako delamo z znaki. V C++-u imamo za to poseben tip `char`, ki nam hrani en znak. Če želimo spremenljivki tipa `char` nastaviti vrednost, moramo želeni znak dati v enojne narekovaje

```
char crka = 'A';
```

Ta koda nam ustvari spremenljivko tipa `char`, ki hrani vrednost `'A'`, torej znak za veliko črko A. Tako kot števila lahko tudi znake pišemo in beremo; pri tem uporabimo `printf`, znotraj njega pa poseben formatnik `%c`, narejen za znake.

```
char crka;
scanf("%c", &crka);
printf("Tvoja crka: %c\n", crka);
```

Ker so računalniki narejeni za delo s številkami, moramo tudi znake predstaviti kot številke. To dosežemo s t.i. *kodnimi tabelami*, ki vsakemu znaku priredijo eno številko. Najpreprostejša kodna tabela je ASCII, ki lahko zakodira vse črke angleške abecede ter vse ostale zgoraj naštete znake, ne zmore pa zakodirati šumnikov. Le-teh se pri programiranju izogibamo, kar se le da. ASCII kode nekaterih pogostih znakov so prikazane v tabeli 5.1.

Znak	ASCII koda
null (`'\0'`)	0
Nova vrstica (`'\n'`)	10
Presledek (`' '`)	32
`'0'`	48
`'1'`	49
`'2'`	50
⋮	⋮
`'9'`	57
`'A'`	65
`'B'`	66
`'C'`	67
⋮	⋮
`'Z'`	90
`'a'`	97
`'b'`	98
`'c'`	99
⋮	⋮
`'z'`	122

Tabela 5.1: Del ASCII tabele

Opazimo lahko, da so številke in črke v tabeli zaporedno; številka `'0'` ima kodo 48, številka `'1'` 49, ..., črka `'A'` ima kodo 65, `'B'` ima kodo 66, itd. Opazimo tudi, da so velike črke od malih ločene, in da imajo male črke večje kode. Ta dejstva lahko uporabimo v programih tako, da črke preprosto obravnavamo, kot da bi bile številke. Črki `'a'` lahko npr. prištejemo neko številko, in tako dobimo črko, ki je toliko znakov naprej v abecedi; `'a' + 7` je na primer enako `'h'`. Poleg tega lahko znake med sabo primerjamo, kar bomo videli v prvem primeru.

Pogoste napake

ASCII koda je bila narejena v Ameriki specifično za angleško uporabo. Ker ne vsebuje šumnikov, le-teh ne moramo predstaviti v naših programih. Za delo s šumniki potrebujemo drugačne kodne tabele, ki jih tukaj ne bomo obravnavali.

Primer

Če poznamo kodne tabele, lahko na relativno enostaven način preverimo, če je neka črka velika ali majhna:

```
#include <stdio.h>

int main() {
    char crka;
    scanf("%c", &crka);
    if (crka >= 'A' && crka <= 'Z') {
        printf("To je velika crka\n");
    } else if (crka >= 'a' && crka <= 'z') {
        printf("To je majhna crka\n");
    } else {
        printf("To sploh ni crka!\n");
    }
    return 0;
}
```

Primer vhoda in izhoda

```
G
```

```
To je velika crka
```

Koda sprva prebere eno črko iz vhoda, nato pa preveri, če je vpisana črka med `'A'` in `'Z'`; če ni, potem preverimo še, če je črka med `'a'` in `'z'`.

Če dobro pogledamo v tabelo, vidimo, da koda 0 ne pripada števki `'0'`, pač pa znaku za konec besedila. To se morda na prvi pogled zdi nepričakovano, ampak ima svoj smisel; če je števka del besedila, o njej ne razmišljamo kot o številki, temveč pač o nekem znaku, ki ima v drugem kontekstu drugačen pomen. Če želimo to števko pretvoriti v številko, s katero lahko brez skrbi računamo, lahko uporabimo trik, kjer "odštejemo nič:"

```
char znak = '7';  // števka 7, napisana kot znak
int stevilka = znak - '0';

// NAROBE!
int to_pride_55 = znak - 0;
```

Pri tem triku moramo biti previdni, da odštejemo pravilno ničlo; če odštejemo številko `0`, se ne bo nič spremenilo; tako kot pri matematiki namreč odštevanje ničle številke ne spremeni. Če pa odštejemo *znak* `'0'` (v enojnih narekovajih), pa dejansko odštevamo številko 48, t.j. ASCII kodo znaka `'0'`. Praktično uporabo tega trika bomo pokazali v naslednjem razdelku.

5.3 Predstavitev nizov

Niz predstavimo kot zaporedje znakov. Ker o zaporedjih (oziroma natančneje o seznamih) nismo še ničesar povedali, moramo uvesti novo sintakso:

```
char niz_besedila[300];
```

Ta ukaz pove računalniku, naj ustvari spremenljivko z imenom `niz_besedila`, ki hrani *največ* 300 znakov. V tej spremenljivki bomo hranili naše zaporedje besedila. Če želimo nize brati ali pisati,

uporabimo funkciji, ki ju že poznamo, ter formatnik `%s`, tu pa je ena posebnost; za branje nizov *ne* napišemo znaka `&`. Če želimo neko spremenljivko nastaviti na niz, ki ga ne bomo prebrali, jo lahko nastavimo na običajen način z enačajem in uporabo dvojnih narekovajev, vendar lahko to storimo samo ob deklaraciji spremenljivke, in ne kasneje.

Primer

Da se navadimo nove sintakse, si poglejmo preprost primer. Spodnji program prebere uporabnikovo ime in ga pozdravi.

```
#include <stdio.h>

int main() {
    char uporabnikovo_ime[300];
    scanf("%s", uporabnikovo_ime);  // NE napišemo &
    printf("Zivjo %s!\n", uporabnikovo_ime);
    return 0;
}
```

Ko ustvarimo niz, računalniku povemo, kolikšna je njegova najdaljša možna dolžina. Nič pa nam ne preprečuje, da v to spremenljivko shranimo krajši niz. Kako pa potem računalnik ve, kje se naš niz dejansko konča? Pričakujemo namreč, da bomo za zapis kratkega niza uporabili nekaj mest za znake na začetku, potem pa se bo niz nekje končal; kaj je na neuporabljenih mestih zaporedja, nas ne zanima. Ravno iz tega razloga so nizi zgrajeni tako, da imajo na koncu dodaten znak za konec besedila; znak null, ki smo ga omenili na začetku. Ta znak pove računalniku, da se besedilo tu konča in da naj naprej ne gleda. Če vrednost niza nastavimo z enačajem ali niz preberemo s `scanf`, bo računalnik sam poskrbel, da bo ta znak napisan na pravo mesto; če pa z nizi delamo kaj bolj zapletenega, moramo za ta znak skrbeti sami. Zaradi tega znaka je dejansko število vidnih znakov, ki jih lahko shranimo v niz, za eno manjše od predpisane največje dolžine. Da se tovrstnim problemom izognemo, bomo od sedaj naprej vedno napisali nekaj večjo število za dolžino niza; če pričakujemo, da uporabnik vpiše največ 200 znakov, bomo za velikost niza dejansko napisali 201 (ali celo malo več).
Preden pridemo do primerov, moramo spoznati še indeksiranje. Vsako mesto za znak v nizu ima svoj zaporedni *indeks*, s katerim lahko enolično dostopamo do tistega mesta. Indeksi so zaporedna števila, ki se začnejo z 0; prvi znak v nizu ima indeks 0, drugi znak ima indeks 1, tretji ima indeks 2, itd. Indeksiranje od 0 se morda sprva zdi nesmiselno, vendar je za tem zelo dober razlog, ki ga bomo spoznali pri obravnavi kazalcev. Poglejmo si na primeru, kako deluje indeksiranje.

Znak	P	r	i	m	i	c	o	v	i	' '	J	u	l	j	i	null
Indeks	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

V zgornji tabeli je razpisan niz `"Primicovi Julji"`, pod vsakim znakom pa je napisan indeks, s katerim lahko dostopamo do tega mesta v nizu. Da zares dostopamo do teh znakov, uporabimo oglate oklepaje;

```
char niz[201] = "Primicovi Julji";
printf("Znak na prvem mestu: %c\n", niz[0]);  // P
niz[8] = 'a';
niz[14] = 'a';
printf("%s\n", niz);  // izpiše "Primicova Julja"
```

Ko zapišemo `niz[indeks]`, dobimo znak, ki ga lahko uporabljamo kot katerokoli drugo spremenljivko.

Primer

Za prvi primer si poglejmo, kako bi lahko izračunali dolžino niza.

```c
#include <stdio.h>

int main() {
    char niz[301];
    // uporabnik lahko napiše niz dolžine največ 300
    scanf("%s", niz);

    // da poiščemo dolžino, bomo dejansko poiskali indeks
    // znaka null; s tem bomo dobili točno število znakov pred njim
    int i = 0;  // trenutni indeks v nizu
    while (niz[i] != 0) {  // ASCII koda znaka null je 0
        i++;
    }
    // na koncu je i ravno indeks znaka null, in s tem enak
    // dolžini niza
    printf("Niz je dolg %d znakov\n", i);
    return 0;
}
```

Ta koda ni težka, vendar jo je pogosto neprijetno pisati, zato imamo boljšo alternativo; če na začetek programa dodamo `#include <string.h>`, lahko uporabljamo funkcijo `strlen`, ki nam ravno tako izračuna dolžino niza:

```c
#include <stdio.h>
#include <string.h>  // strlen

int main() {
    char niz[201];
    scanf("%s", niz);
    printf("Dolzina niza: %d\n", strlen(niz));
    return 0;
}
```

Funkcijo `strlen` uporabljamo v skoraj vsakem programu z nizi, zato je dobro, da se je čim prej navadimo. Poleg tega `strlen` dolžino dejansko izračuna hitreje kakor zgornja koda.

Primer

V naslednjem primeru bomo napisali kodo, ki pretvori besedilo v velike črke. Za to uporabimo eno od lastnosti ASCII tabele, ki smo jo omenili prej; namreč, da so črke napisane zaporedno, in da so velike črke pred malimi.

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

int main() {
    char niz[201];
    scanf("%s", niz);

    // Zamik med velikimi in majhnimi črkami je enak ne glede na to, katera
    // črka je to. V zanki bomo vsaki majhni črki odšteli zamik, s čimer jo
    // pretvorimo v veliko
    int zamik = 'a' - 'A';
    int dolzina = strlen(niz);
    for (int i = 0; i < dolzina; i++) {
        // če je črka majhna, jo moramo povečati
        // to moramo nujno preveriti, saj drugih znakov ne smemo spremeniti!
        if ('a' <= niz[i] && niz[i] <= 'z') {
            niz[i] = niz[i] - zamik;
        }
    }
    printf("%s\n", niz);
    return 0;
}
```

Primer

Za ta primer si poglejmo, kako bi pretvorili številko, zapisano z nizom, v številko, zapisano s številko. Prej omenjen trik z odštevanjem nič ne bo deloval, ker lahko z njim pretvarjamo le znake; lahko pa številko pretvorimo znak po znak. Pri tem pretvarjanju uporabljamo lastnosti desetiškega zapisa števil; namreč, da zaporedna mesta v zapisu predstavljajo vrednosti, ki se razlikujejo za faktor 10. Ko pretvorimo prvi del besedila, in želimo dopisati še eno števko, moramo že zapisani del "premakniti" eno mesto v levo, ter premaknjenemu številu prišteti novo števko. Premikanje dosežemo z množenjem z 10.

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

int main() {
    char niz[11];  // niz, ki bo hranil številko
    // premisli, zakaj je dolžina 11 že dovolj
    scanf("%s", niz);
    int dolzina = strlen(niz);
    int stevilo = 0;  // začnemo z 0
    for (int i = 0; i < dolzina; i++) {
        stevilo *= 10;
        stevilo += (niz[i] - '0');
        // niz[i] je znak, ki ga moramo pretvoriti v številko, da lahko
        // računamo z njim
    }
    printf("Stevilka je %d\n", stevilo);
    return 0;
}
```

5.4 Standardne funkcije

Izkaže se, da pri delu z nizi pogosto pišemo zelo podobne kose programa, kakor se je zgodilo pri primeru z izračunom dolžine. Namesto da večkrat napišemo skoraj enako kodo, so v knjižnici `string.h` dostopne razne funkcije, ki nam pogosto olajšajo delo.

5.4.1 Primerjava nizov

Za primerjavo dveh nizov ne uporabljamo dvojnega enačaja (`==`), temveč funkcijo `strcmp`. Funkcijo uporabimo tako, da ji v okrogle oklepaje napišemo dva niza; `strcmp(niz1, niz2)`. Če sta niza enaka, funkcija vrne rezultat `0`, sicer pa vrne drugačen rezultat. Spodnja koda preveri, če je uporabniku ime Filip:

```
char niz[101];
scanf("%s", niz);
if (strcmp(niz, "Filip") == 0) {
    printf("Ime ti je Filip\n");
} else {
    printf("Ni ti ime Filip\n");
}
```

Funkcija nam pravzaprav poda več informacij. Z njo lahko pogledamo, kakšna je *leksikografska ureditev* dveh nizov; preprosto povedano, kateri od nizov bi se, če bi bila oba niza besedi, pojavil prej v slovarju (leksikonu). Če bi se prvi niz pojavil prej, funkcija vrne negativno število. Če bi se drugi niz pojavil prej, pa funkcija vrne pozitivno število.

5.4.2 Kopiranje nizov

Če želimo eno spremenljivko prekopirati v drugo, lahko napišemo `b = a`. Na žalost pa to ne deluje za nize; namesto `niz2 = niz1` moramo napisati `strcpy(niz2, niz1)`. Funkcija `strcpy` prekopira drugi niz v prvega; na koncu bosta oba niza imela enako vsebino.
Če želimo nekemu nizu na konec dodati nek drug niz, lahko za to uporabimo funkcijo `strcat` (*string concatenate*). Ta funkcija prav tako sprejme dva niza; ko jo pokličemo, drug niz kopira na konec prvega.

5.4.3 Operacije na prvih n znakih

Včasih želimo kopirati ali primerjati le del niza. Za to imamo na voljo malce drugačne verzije zgoraj naštetih funkcij; če v imenih teh funkcij za `str` dodamo še `n` (torej `strncmp`, `strncpy`, ...), in funkciji kot zadnji argument podamo številko n, bo funkcija svoje delo opravila le na prvih n znakih; `strncmp(niz1, niz2, 3)` bo primerjal le prve tri znake, `strncpy(niz2, niz1, 7)` bo kopiral le prvih sedem znakov, ipd. Pri uporabi `strncpy` bodimo pozorni, da ne pozabimo ročno dodati znaka za konec niza, če ga potrebujemo.

Poglavje 6

Seznami

Ko si želimo podatke shraniti tako, da jih bomo lahko spreminjali in na koncu nekaj z njimi naredili (npr. da z njimi računamo, jih izpišemo, itd.), za to uporabimo spremenljivke. Vsaka spremenljivka hrani en podatek — eno številko, en znak, ..., razen nizov, ki lahko hranijo več zaporednih znakov. Nizi so posebni na zanimiv način. Med pisanjem programa nismo vedeli, točno kako dolg bo niz, ki ga bo uporabnik vnesel; rekli smo le, da mora biti krajši od neke največje dolžine niza, ki smo jo vnesli v program. Niz s kapaciteto 201 je tako lahko shranil do 200 znakov; namesto dvestotih spremenljivk smo vse te znake shranili v eno. Pogosto si želimo tudi števila shraniti tako, da bomo kasneje lahko dostopali do njih, ampak med pisanjem programa ne vemo točno, s koliko števili bo program moral delati. Problem rešimo s seznami.

6.1 Sintaksa

Osnovna sintaksa seznamov (angl. *array*) je zelo podobna sintaksi nizov. Da ustvarimo nov seznam, napišemo ime tipa (`int`, `long long`, itd.), ime spremenljivke in nato največjo možno dolžino seznama:

```
int seznam_stevil[300];
long long seznam_velikih_stevil[5000];
```

Prav tako kot pri nizih tudi do posamičnih elementov seznama dostopamo z oglatimi oklepaji:

```
// nastavi četrti (indeks 3) element na 7
seznam_stevil[3] = 7;

// izpiši element na petem mestu
printf("%lld\n", seznam_velikih_stevil[4]);
```

Opazimo da, prav tako kot pri nizih, tudi pri seznamih šteti začnemo pri 0. Spomnimo se, da smo pri nizih imeli posebni znak null, ki je programu sporočal, da se naš niz na tistem mestu konča. Za sezname števil takega znaka ne poznamo; ko delamo s števili, moramo drugače vedeti, kako dolg naš seznam dejansko je. Naloge so običajno narejene tako, da prvo število na vhodu pove, koliko števil (ki jih želimo shraniti v seznam) bo sledilo. V večini primerov je to ravno dolžina našega seznama, če pa delamo kakšne posebne trike, pa moramo za dolžino skrbeti sami.

Primer

Poglejmo si primer preproste naloge: Na vhodu je podano število N, ki mu sledi N števil. Program naj izpiše vsoto teh N-tih števil. Velja $N \leq 10^5$.

```
#include <stdio.h>

// seznam je malce daljši od 10^5 - glej spodaj
int seznam[100002];

int main() {
    int N;  // dolžina seznama
    scanf("%d", &N);
    for (int i = 0; i < N; i++) {
        // preberemo številko na i-to mesto seznama
        scanf("%d", &seznam[i]);
    }
    // v spremenljivki bomo hranili delno vsoto prvih
    // nekaj mest seznama
    int vsota = 0;
    for (int i = 0; i < N; i++) {
        vsota += seznam[i];
    }
    printf("%d\n", vsota);
    return 0;
}
```

Primer vhoda in izhoda

```
5 1 2 3 4 5
```

```
15
```

Pri programu opazimo nekaj posebnosti. Seznam števil smo postavili *zunaj* funkcije `main`. To je zmeraj dobro narediti, če uporabljamo zelo dolge sezname, kakor zgornji seznam je. Poleg tega smo največjo dolžino seznama nastavili na $10^5 + 2$. Lahko bi nastavili dolžino na točno 10^5, vendar je dobra ideja, da si pustimo malce praznega prostora, če se kje v programu zmotimo pri indeksiranju. Ta prazen prostor nam lahko v nekaterih primerih prepreči, da se program sesuje.

Primer

Poglejmo si malo bolj zanimiv primer. Naša naloga sedaj je, da uredimo seznam N števil ($N \leq 10^6$) po velikosti od najmanjšega do največjega. Podano imamo tudi informacijo, da bodo ta števila velika med vključno 0 in 100. Naloge se lotimo tako, da preštejemo, kolikokrat se neko število pojavi v danem seznamu, nato pa bomo seznam rekonstruirali tako, da bo urejen. Da preštejemo, kolikokrat se kakšno število pojavi, uporabimo nov seznam, kjer indeks pomeni številko, ki jo štejemo, shranjena vrednost pa kolikokrat smo to številko že prešteli.

```
#include <stdio.h>

// seznam iz vhoda
int seznam[1000003];
// na indeksu j je zapisano, kolikokrat smo že videli
// število j v seznamu
int stetje[101];
// nov, urejen seznam
int novseznam[1000003];

int main() {
    int N;  // velikost seznama
    scanf("%d", &N);
    for (int i = 0; i < N; i++)
        scanf("%d", &seznam[i]);

    // število na mestu i v seznamu je seznam[i]
    // v eni iteraciji zanke vidimo eno število; zato prištejemo
    // 1 na pravo mesto seznama za štetje
    for (int i = 0; i < N; i++)
        stetje[ seznam[i] ] += 1;

    // sedaj rekonstruiramo seznam
    // shranjujemo si indeks prvega elementa v novem seznamu,
    // ki ga še nismo nastavili
    int indeks = 0;
    for (int j = 0; j <= 100; j++) {
        // stetje[j]-krat moramo zapisati j v nov seznam
        for (int k = 0; k < stetje[j]; k++) {
            novseznam[indeks] = j;
            // povečamo indeks, ker smo ga ravno nastavili
            indeks++;
        }
    }
    // izpišemo nov seznam
    for (int i = 0; i < N; i++)
        printf("%d ", novseznam[i]);

    printf("\n");
    return 0;
}
```

Ta algoritem za urejanje je zelo znan; imenuje se urejanje s preštevanjem (angl. *counting sort*). Primeren je, kadar imamo zelo majhen razpon možnih vrednosti števil, kakor smo imeli tu ($0 - 100$).

6.2 Večdimenzionalni seznami

Videli smo, kako ustvariti seznam števil, kaj pa seznam seznamov? Takemu seznamu pravimo *dvodimenzionalen seznam*, ustvarimo pa ga tako, da napišemo dva zaporedna oglata oklepaja z velikostjo;

```
int seznam_seznamov_stevil[velikost1][velikost2];
```

Dvodimenzionalni seznami so uporabni, kadar moramo podatke predstaviti v tabeli. Do posamičnih elementov dostopamo z dvojnimi oglatimi oklepaji, tako kot pri inicializaciji spremenljivke; `tabela[i][j]`. Tabele si običajno predstavljamo tako, da nam prvi indeks pove zaporedno številko vrstice, drugi pa zaporedno številko stolpca.
Druga možna uporaba dvodimenzionalnih seznamov je ustvarjanje seznamov nizov. Sezname smo obravnavali kakor nekaj sorodnega nizom, dejansko pa je pravilna smer razmišljanja tu ravno obratna; nizi so pravzaprav le seznami znakov. Če želimo narediti seznam nizov, ustvarimo dvodimenzionalni seznam znakov, v katerem nam bo prvi indeks predstavljal indeks niza, drugi indeks seznama pa nam bo predstavljal indeks znaka v nizu.

Poglavje 7

Funkcije

Ko pišemo daljše programe, se pogosto zgodi, da večrat uporabimo nek podoben kos kode. Da si v takih situacijah olajšamo življenje, imamo *funkcije*. Že v zgodnjih sedemdesetih, ko se je začel razvijati jezik C so vedeli, da so funkcije zelo uporabne, zato je osnovna oblika tega jezika sestavljena iz njih. Res, funckijam se v C-ju ne da izogniti. Do zdaj smo brali in pisali podatke s funkcijama `scanf` in `printf`, preverjali dolžino nizov s `strlen` in vse smo napisali v funkciji `main`.
Funkcijo si lahko predstavljamo kot neko napravo. Napravi podamo nekaj *parametrov* (včasih se uporabi tudi beseda *argumentov*), ona nekaj melje in nam potem *vrne* neko vrednost. Poleg tega, da samo vrne neko vrednost, ima lahko funkcija tudi kakšne stranske učinke. Lahko kaj izpiše z uporabo `printf`, ali pa spremeni kakšno spremenljivko. Včasih so parametri ali vrnjena vrednost nepotrebni, zato lahko napišemo tudi funkcije brez parametrov in funkcije, ki ne vrnejo ničesar.

7.1 Kako napišemo svojo funkcijo

Najbolj osnovna oblika funkcije je takšna, ki ne sprejme nobenega parametra in ne vrne ničesar. Če hočemo, da ta funkcija ni neuporabna, bo morala imeti kakšen stranski učinek.

Primer

```c
#include <stdio.h>

void funkcija() {
    // koda, ki se bo izvedla, ko pokličemo funkcijo
    printf("Zdravo\n");
}

int main() {

    funkcija(); // pokličemo funkcijo

    return 0;
}
```

Komponente zapisa so naslednje:

- `void` - Posebna beseda, ki pove, da funkcija ne vrne ničesar.
- `funkcija` - Ime za funkcijo, kot pri spremenljivkah bi lahko tu napisali karkoli.
- `()` - Ime funkcije se more končati z (), v te oklepaje postavljamo parametre.
- `{ ... }` - Telo funkcije, v katerem je koda, ki se bo izvedla.

Kot napovedano, imajo lahko funkcije tudi parametre.

Primer

```c
#include <stdio.h>

void plusi(int n) {
    // izpišemo n plusov
    for (int i = 0; i < n; i++) {
        printf("+");
    }
    printf("\n"); // končamo z znakom za novo vrstico
}

int main() {
    plusi(12); // pokličemo funkcijo s parametrom 12
    return 0;
}
```

Zdaj smo v oklepaje za imenom spremenljivke postavili `int n`. Beseda `int` pove, da je ta parameter celo število, `n` je pa ime za parameter. Ime, ki smo si ga izbrali za parameter, uporabljamo v telesu funkcije, da dostopamo do vrednosti v parametru.
Če je naš parameter seznam, to napišemo tako:

Primer

```c
#include <stdio.h>

void izpisi_prve_tri(int seznam[]) {
    printf("%d %d %d\n", seznam[0], seznam[1], seznam[2]);
}

int main() {
    int stevila[7] = {7, 8, 6, 1, 2, 6, 3};
    izpisi_prve_tri(stevila); // izpiše "7 8 6"
    return 0;
}
```

Zdaj pa še funkcije, ki vrnejo kakšno vrednost:

Primer

```c
#include <stdio.h>

int vsota(int a, int b) {
    int a_plus_b = a + b;
    return a_plus_b;
}

int main() {
    int rezultat = vsota(3, 4);
    printf("%d\n", rezultat);
    return 0;
}
```

Zdaj smo v funkcijo dodali ukaz `return`, ki pove programu, da želimo vrniti spremenljivko, ki se pojavi takoj za njim. Spodaj pa vidimo, da moramo za dostop do vrnjene vrednosti nastaviti spremenljivko. Izraz `funkcija(3, 4)` lahko uporabimo kjerkoli, kjer bi lahko uporabili neko število, bodisi zapisano eksplicitno ali v spremenljivki. Prav tako lahko za ukazom `return` seštevamo števila. Zgornji program bi lahko krajše napisali kot:

Primer

```
#include <stdio.h>

int vsota(int a, int b) {
    return a + b;
}

int main() {
    printf("%d\n", vsota(3, 4));
    return 0;
}
```

7.2 Potek funkcije

Kot smo navajeni, se ukazi v telesu funkcije izvajajo po vrsti od zgoraj navzdol. V njej lahko uporabimo vse, kar smo se do zdaj naučili (lahko ustvarjamo nove spremenljivke, if stavke, zanke, ...). Vredno omembe je, da se funkcija konča, ko se izvede prvi `return`.

Primer

```
#include <stdio.h>

bool je_samoglasnik(char crka) {
    if (crka == 'a' || crka == 'e' || crka == 'i' || crka == 'o' || crka == 'u') {
        return true;
    }
    return false;
}

int main() {
    char c;
    scanf("%c", &c); // prebere eno črko

    if (je_samoglasnik(c)) {
        printf("To je samoglasnik!\n");
    }
    else {
        printf("To pa ni samoglasnik.\n");
    }
    return 0;
}
```

Tu funkcija vrača logično vrednost `bool`, ki je ena izmed `true` ali `false`, uporablja se jo lahko v if stavku. Deluje torej pravilno, saj če bo `crka` enaka a, e, i, o, u, se bo funkcija končala pri `return`-u v if stavku, sicer se bo pa nadaljevala do `return false;`.

7.3 Spremenljivke in funkcije

7.3.1 Globalne in lokalne spremenljivke

Spomnimo se, da lahko spremenljivke definiramo na dva načina. Tistim, ki so definirane na vrhu, izven funkcije rečemo *globalne spremenljivke*, tistim, ki so pa definirane v funkciji, pa *lokalne spremenljivke*. Globalne spremenljivke so vidne povsod, torej v vseh funkcijah, ki so definirane nižje od spremenljivke, medtem ko lahko lokalne spremenljivke uporabljamo samo v tisti funkciji, v kateri smo jih definirali.

Primer

```c
#include <stdio.h>

// tu so globalne spremenljivke
int g = 12;

void funkcija() {
    // v tej funkciji lahko dostopamo do g
    printf("%d\n", g);
    g += 3;
}

int main() {

    int n = 200;

    funkcija();

    // tu lahko dostopamo do g in n
    printf("%d, %d\n", g, n);

    return 0;
}
```

Primer vhoda in izhoda

```
12
15, 200
```

Seveda, če globalno spremenljivko nekje spremenimo, se bo spremenila tudi za vse druge funkcije. V zgornjem primeru je `funkcija` najprej izpisala `g` (12) in ga nato povečala na 15. Ko se je g izpisal drugič, je bil zato 15.

7.3.2 Spreminjanje parametrov

Če funkciji podamo neko spremenljivko kot parameter in poskušamo to spremenljivko v funkciji spreminjati, se obnaša na dva različna načina, odvisno od tega, če je spremenljivka seznam.
Če spremenljivka ni seznam (npr. `int`, `char`) se za funkcijo ustvari kopija te spremenljivke. Če jo v funkciji spremenimo, to ne bo spremenilo izvirne spremenljivke, saj v funkciji delamo s kopijo. Če je seznam, bodo pa spremembe na seznam v funkciji vplivale na izvirni seznam, saj se ta ne bo prekopiral.

Primer

Sprva ne vidimo smisla za drugačno obravnavo seznamov, a se izkaže, da je lahko ta lastnost tudi uporabna. Če bi želeli napisati funkcijo, ki sprejeme seznam in njegovo dolžino, potem pa obrne vrstni red elementov tega seznama, lahko to napišemo takole:

```c
#include <stdio.h>

void obrni(int seznam[], int dolzina) {
    // Obrnemo vrstni red prvih dolzina elementov seznama
    for (int i = 0; i < dolzina / 2; i++) {
        int vmesna = seznam[i];
        seznam[i] = seznam[dolzina-i-1];
        seznam[dolzina-i-1] = vmesna;
    }
}

int main() {
    int n;
    int seznam[1000];

    // preberemo seznam dolžine n
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &seznam[i]);
    }

    obrni(seznam, n);

    // izpišemo obrnjen seznam
    for (int i = 0; i < n; i++) {
        printf("%d\n", seznam[i]);
    }

    return 0;
}
```

Primer vhoda in izhoda

```
5
10 20 30 -3 7 1
```

```
1
7
-3
30
20
10
```

7.4 Rekurzija

Rekurzija je način pisanja programa, pri katerem funkcija kliče samo sebe. Recimo, da želimo sešteti vsa števila med 1 in nekim `n`. Z zanko bi to naredili tako, da si pripravimo neko spremenljivko `vsota` in potem napišemo for zanko od 1 do `n` in v `vsota` seštejemo vsa ta števila. Pri rekurziji je pa pristop drugačen.

Primer

```
int vsota(int n) {
    // osnovni pogoj
    if (n <= 0) {
        return 0;
    }

    // korak
    return n + vsota(n - 1);
}
```

Najprej napišemo *osnovni pogoj*, ki za vrednosti `n` manjše ali enake 0 vrne 0. Če ne bi napisali osnovnega pogoja, bi šli rekurzivni klici funkcije v neskončnost, `vsota(3)` bi poklicala `vsota(2)`, ki pokliče `vsota(1)`, potem `vsota(0)`, `vsota(-1)`, `vsota(-2)`, ... Po osnovnem pogoju moramo narediti *korak*, ki izračuna `vsota(n)` s pomočjo vrednosti funkcije v nekem manjšem številu, v našem primeru kar `vsota(n - 1)`.

Poglavje 8

Teorija števil

V programiranju se pogosto pojavijo razne matematične teme, ker nam dovoljujejo, da na smiseln način opišemo razne probleme, s katerimi se srečujemo. Ena od najbolj zanimivih vej matematike je teorija števil. Iz nje izhaja marsikateri algoritem, s katerim si lahko pri programiranju pomagamo; mi si bomo ogledali dva primera, ki izhajata že iz stare Grčije.

8.1 Evklidov algoritem

Evklidov algoritem je postopek, s katerim lahko hitro poiščemo največji skupni delitelj dveh števil. Spomnimo se; največji skupni delitelj števil a in b je *največje* število d, ki deli tako a kot b. V nadaljevanju ga bomo zapisali s krajšavo NSD ali kot $D(a, b)$.

Primer

NSD števil 30 in 75 je 15, ker sta tako 30 kot 75 deljiva s 15 (velja namreč $30 = 2 \cdot 15$ in $75 = 5 \cdot 15$), večje število, ki bi delilo oba, pa ne obstaja.

Primer

Vprašanje, kaj je *najmanjši* skupni delitelj dveh števil, je dolgočasno; najmanjši skupni delitelj dveh števil je vedno 1, ker 1 deli vsa števila.

Preprost algoritem, ki poišče največji skupni delitelj dveh števil, je sledeč;

```
Z zanko preiščemo vsa števila od d=1 do d=manjsi(a, b)
    preverimo, ali d deli a, ter ali d deli b.
    če deli oba, potem nastavimo NSD = d.
Ko se zanka zaključi, smo našli NSD.
```

Algoritem res pravilno deluje, ker bo zadnje število, ki bo delilo tako a kot b, ravno NSD, zato bo spremenljivka na koncu zanke pravilno nastavljena. Kljub pravilnemu delovanju pa tega algoritma ne želimo uporabljati, ker je zelo počasen. V pomoč nam priskoči drug algoritem, ki ga je opisal starogrški matematik Evklid v svoji knjigi *Elementi*. Ta algoritem bo prav tako vedno dal pravilni rezultat, za razliko od prejšnjega pa bo tudi zelo hiter.

Primer

Delovanje algoritma si poglejmo na primeru; izračunajmo, koliko je $D(54, 42)$. V prvem koraku izračunamo količnik in ostanek pri deljenju 54 s 42;

$$54 = 42 \cdot 1 + 12$$

Količnika v nadaljevanju ne bomo potrebovali, tako da ga lahko pozabimo. V drugem koraku prestavimo delitelja na mesto deljenca, ostanek pa na mesto delitelja, ter spet izračunamo količnik in ostanek;

$$42 = 12 \cdot 3 + 6$$

Postopek bomo ponovili še enkrat. V zadnjem koraku izračunamo količnik in ostanek pri deljenju 12 s 6; spet smo premaknili delitelja na mesto deljenca ter deljenca na mesto delitelja.

$$12 = 6 \cdot 2 + 0$$

Ko dobimo ostanek 0, se je algoritem zaključil (v naslednjem koraku bi morali deliti z 0, česar pa ne smemo početi). Takrat je največji skupni delitelj števil zapisan na mestu delitelja (oz. je ostanek, ki smo ga dobili v enem koraku prej). Na roko lahko preverimo, da je NSD števil 54 in 42 res 6; dobili smo pravi rezultat.

Opis algoritma je sledeč:

```
Izračunati želimo NSD števil a in b.
Izračunamo c = a % b (ostanek pri deljenju)
Ponavljamo, dokler je c različen od 0
    za novi a nastavimo star b
    za novi b nastavimo star c
    za novi c nastavimo (novi a) % (novi b)
Končni odgovor je b.
```

Tak opis lahko tudi enostavno zapišemo kot funkcijo v C++:

```
int nsd(int a, int b) {
    int c = a % b;
    while (c != 0) {
        a = b;
        b = c;
        c = a % b;
    }
    return b;
}
```

Če nas namesto največjega skupnega delitelja zanima najmanjši skupni večkratnik, lahko tega izračunamo po formuli:

$$D(a, b) \cdot v(a, b) = a \cdot b.$$

V tej formuli smo označili največji skupni delitelj z D in najmanjši skupni večkratnik z v. Za izračun najmanjšega skupnega večkratnika torej prvo izračunamo največji skupni delitelj s pomočjo Evklidovega algoritma, nato pa uporabimo formulo: `v = a / D * b`. Pomembno je, da prvo delimo, sicer bi lahko vmesni rezultat (produkt) postal prevelik za naš številski tip (`int` oziroma `long long`), in povzročil napako.

8.2 Eratostenovo rešeto

Praštevilo je tako število, ki je deljivo le z 1 in samo s seboj. Praštevil je neskončno, najmanjša od njih so $2, 3, 5, 7, 11, 13, \ldots$. Praštevila so v matematiki zelo pomembna, med drugim zaradi naslednje trditve, ki se ji reče *osnovni izrek teorije števil*.

Izrek: Vsako število lahko na enoličen način zapišemo kot produkt praštevil.

Primer

Izrek nam pove, da lahko vsakemu številu poiščemo praštevilski razcep. Poglejmo si praštevilski razcep števila 6552. To število je deljivo z 2, pri deljenju dobimo količnik 3276. Ta količnik lahko spet delimo z 2 in dobimo 1638. Ko še tretjič delimo z 2, dobimo rezultat 819, to število pa ni več deljivo z 2. Lahko ga še dvakrat delimo s 3; sprva dobimo 273, nato 91. Naposled lahko 91 delimo s 7 in dobimo 13. Praštevilski razcep števila 6552 je tako

$$6552 = 2^3 \cdot 3^2 \cdot 7 \cdot 13.$$

V primeru smo videli, da se v praštevilskem razcepu lahko kakšno praštevilo ponavlja, kakšno pa se sploh ne pojavi. Če želimo razcep poiskati s programom, ali pa narediti kaj drugega s praštevili, pa jih moramo prvo poiskati. Spoznali bomo en algoritem za iskanje praštevil, to je Eratostenovo rešeto.
Recimo, da želimo poiskati vsa praštevila do neke zgornje meje N. Sprva si napišemo vsa števila med 2 in N (1 namreč ni praštevilo);

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 ...

Vemo, da je število 2 praštevilo. Večkratniki števila 2 zato niso praštevila, ker so deljivi z 2, torej jih prečrtamo.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~ 21 ~~22~~ 23 ~~24~~ 25 ~~26~~ ...

Število 3 je praštevilo, torej tudi večkratniki 3 niso praštevila.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ 25 ~~26~~ ...

Prišli so do števila 4, ki je prečrtano, zato vemo, da ni praštevilo. Ker je deljivo z 2, smo vse večkratnike 4 (ki so tudi večkratniki 2), že prečrtali, torej nam ni treba posebej črtati večkratnikov 4.
Število 5 je praštevilo (ker ni prečrtano), torej moremo prečrtati njegove večkratnike.

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~ ~~21~~ ~~22~~ 23 ~~24~~ ~~25~~ ~~26~~ ...

Število 6 je prečrtano, tako da ga preskočimo.
Postopek bi lahko nadaljevali, dokler ne pridemo do 26 (oz. do želene zgornje meje), vendar nam ni treba. Velja namreč naslednje; če je število M produkt dveh števil, $M = AB$, potem je vsaj eno od teh števil manjše od ali enako veliko kot $\sqrt{M}$. Če bi bili obe števili večji kot $\sqrt{M}$, je njun produkt večji od M, to pa ne mora veljati, ker je njun produkt ravno M.
V našem primeru smo torej lahko iskanje ustavili že pri $\sqrt{26}$, kar je nekaj več kot 5, zato moramo preveriti tudi 6, več pa ne.

Primer

Možna implementacija algoritma za iskanje praštevil je podana spodaj. Zapisana koda ohranja še dodatno informacijo; za vsako število si zapomnimo enega od njegovih praštevilskih deliteljev.

```
int M;  // zgornja meja

// Tabela, ki hrani trenutno stanje rešeta
// če je na nekem mestu zapisana številka 0, to pomeni,
// da še nismo našli nobenega praštevilskega delitelja.
// Ko nekega delitelja najdemo, ga zapišemo
int reseto[M+1];

void poisci_prastevila() {
    // V spremenljivki i bomo hranili število, ki ga trenutno obravnavamo
    // začnemo pri 2, ker niti 0 niti 1 nista praštevili
    // Iteriramo samo do korena M
    for (int i = 2; i*i <= M; i++) {
        // Če je na mestu i ničla, do sedaj nismo našli nobenega praštevila,
        // torej mora biti število i nujno praštevilo
        if (reseto[i] == 0) {
            printf("Najdeno praštevilo %d!\n", i);

            // Sedaj "prečrtamo" vse večkratnike števila i
            for (int j = 2*i; j <= M; j += i) {
                // Zabeležimo si, da je i delitelj j
                reseto[j] = i;
            }
        }
    }
}
```

Ta implementacija nam dovoljuje, da hitro poiščemo nekega delitelja poljubnega števila `n`; preprosto pogledamo v `reseto[n]`. Če najdemo `0`, vemo, da je `n` praštevilo; sicer pa smo našli nekega delitelja.

Poglavje 9

Nekaj več o branju in pisanju

9.1 Scanf

Funkcija `scanf` lahko bere različne vrste podatkov:

- `%s` - beseda (vsi znaki razen praznih znakov (glej spodaj)) (`char`)
- `%c` - en (poljuben) znak (`char`)
- `%d` - število med -2^{31} in $2^{31} - 1$ (`int`)
- `%lld` - število med -2^{63} in $2^{63} - 1$ (`long long`)
- ...

Prazni znaki (whitespace) so znaki, ki jih ne vidimo:

- `\n` nova vrstica
- `\t` zamik
- `' '` presledek

S formatnikom `%s` funkcija `scanf` bere do prvega praznega znaka, ki sledi nepraznemu. Pri tem prebere tudi vse prazne znake pred prvim nepraznim, vendar jih ne shrani.

9.2 Branje do konca vrstice

Lahko določimo, da se `scanf` ne bo ustavil pri prvem praznem znaku, temveč šele pri koncu vrstice (ali kje drugje). To naredimo s pomočjo naslednjih formatnikov:
`%[...]` - med oklepaje pišemo navodila, kakšne znake lahko bere

- `%[a-z]` - beri male črke angleške abecede
- `%[a-zA-Z0-9]` - beri male in velike črke in številke

`%[^...]` - beri vse do znakov, ki sledijo strešici

- `%[^\n]` - beri vse do `'\n'`

Primer

```c
#include<stdio.h>

int main(){
    char a[50];
    scanf("%[^\n]", a); //ali scanf("%[^\n]%*c") in brez getchar()
    getchar();
    printf("%s\n", a);
    return 0;
}
```

Primer vhoda in izhoda

```
Beremo do konca vrstice.
```

```
Beremo do konca vrstice.
```

Pogoste napake

Funkcija `scanf` s `[^\n]` prebere vse do `\n`, tega pa ne prebere in se ustavi pred njim. Ko funkcijo pokličemo naslednjič, začne tam, kjer je nazadnje ostala, kar je v tem primeru točno pred znakom `'\n'`. Če jo torej ponovno pokličemo s parametrom `[^\n]`, se ne bo nikamor premaknila, saj je pred njo znak za novo vrstico.

`%c` - prebere en znak (`char`), ki je lahko karkoli - črka, številka, prazen znak...
`%*c` - prebere en znak, a ga ne shrani
Funkcija `getchar` dela podobno, vzame en znak in ga ne shrani.

Za razliko od tega s formatnikom `%s` preberemo vse prazne znake med dvema nepraznima nizoma in jih ne shranimo. Tudi, če bodo prazni znaki pred besedo, jih bo program ignoriral in poiskal prvi neprazen znak.

9.3 Branje do konca vhoda

Če vemo točno, koliko besed/številk/vrstic bomo imeli na vhodu, jih lahko preberemo s for zanko. Kako preberemo neznano količino podatkov na vhodu, tako da preberemo vse? Če napišemo `while(true)` ali `while(1)`, bomo sicer prebrali vse, a se program ne bo nikoli ustavil. Da zanko končamo natanko na koncu podatkov na vhodu, uporabimo kodo iz naslednjega primera.

Primer

```c
#include<stdio.h>

int main(){
    int a;
    while(scanf("%d", &a) != EOF){
        printf("%d\n", a*a); //izpisujemo kvadrate prebranega števila
    }
    return 0;
}
```

Primer vhoda in izhoda

```
1 2 3 4 5
```

```
1
4
9
16
25
```

Funkcija `scanf` vrne število formatnikov, ki jih je uspešno prebrala in shranila (če ji kot parameter podamo samo `%d`, bo vrnila 1, če je uspešno prebrala število, sicer pa 0). `EOF` (End of File) vrne, če na vhodu ni ničesar več za prebrati. Tedaj se bo zanka ustavila. Če programu vhodne podatke podajamo iz datoteke, se to zgodi avtomatsko ob koncu datoteke, če pa mu podatke podajamo na roko, konec vhoda sporočimo s `Ctrl+D (Linux in MacOS)` ali `Ctrl+Z` (Windows).

Primer

```c
#include<stdio.h>

int main() {
    int a, b, c;
    int r = scanf("%d%d%d", &a, &b, &c);
    printf("%d\n", r);
    return 0;
}
```

Primer vhoda in izhoda

```
5 8 100
```

```
3
```

Primer vhoda in izhoda

```
5 8 miha
```

```
2
```

9.4 Branje in pisanje v in iz niza

`scanf` uporabljamo za branje s standardnega vhoda, `printf` pa za pisanje na standardni izhod. Namesto tega lahko beremo in pišemo tudi drugače, npr. v in iz nizov. Za to uporabljamo funkciji `sscanf` in `sprintf`, ki delata podobno kot `scanf` in `printf`.

Primer

```c
#include<stdio.h>

int main(){
    int n;
    char a[10], b;
    char text[]="Slovenska 157 a";
    sscanf(text, "%s %d %c", a, &n, &b);
    sprintf(text, "%c %d %s", b, n, a);
    printf("%s\n", text);
    return 0;
}
```

Primer vhoda in izhoda

```
a 157 Slovenska
```

Pogoste napake

Medtem ko `%s` praznih znakov ne shrani, jih `%c` obravnava tako kot vse ostale. Če pogledamo prejšnji primer vidimo, da je v funkciji `scanf` pred `%c` presledek. Ker so med posameznimi podatki, ki jih želimo prebrati, presledki, bi `%c` pobral presledek, ne pa črke, ki mu sledi. Če med posamezne formatnike postavimo presledek, ta načeloma pobere prazne znake do naslednjega drugačnega znaka, vendar to v splošnem ni dobra praksa.

Funkciji `sscanf` podamo tri parametre (ali več):

1. ime niza, iz katerega naj bere
2. tip podatka, ki naj ga prebere (niz, število ...)
3. kam naj ta podatek zapiše (ime spremenljivke)

Funkciji `sprintf` prav tako podamo tri parametre (ali več):

1. ime niza, kamor naj piše
2. tip podatka, ki naj ga zapiše
3. kaj naj zapiše (ime spremenljivke ali podatek sam)

9.5 Branje in pisanje v in iz datoteke

Podobno kot v niz lahko pišemo in beremo tudi v in iz datotek s funkcijama `fscanf` in `fprintf`.

Primer

```c
#include<stdio.h>

int main(){
    int a;
    FILE *fr, *fw;
    fr = fopen("in.txt", "r");
    fw = fopen("out.txt", "w");
    while(fscanf(fr, "%d", &a) != EOF){
        fprintf(fw, "%d\n", a*a);
    }
    // iz datoteke in.txt preberemo vsa števila in
    // v out.txt izpišemo njihove kvadrate
    fclose(fr);
    fclose(fw);
    return 0;
}
```

Primer vhoda in izhoda

in.txt:

```
1 2 3 4 5 6 7
```

out.txt:

```
1
4
9
16
25
36
49
```

(Ta program ne bere s standardega vhoda in ne piše ne standardni izhod.)

`FILE`: tip podatka
`fopen`: funkcija, s katero odpremo datoteko, podamo ji ime datoteke, ki jo želimo odpreti in način `"r"` (*read*, za branje) ali `"w"` (*write*, za pisanje).
Datoteke, ki jih odpiramo za branje, morajo že prej obstajati, sicer se bo program sesul.
Za datoteke, v katere pišemo, ni nujno, da že obstajajo. Če še ne obstajajo, bo program ustvaril novo datoteko s tem imenom. Če že obstajajo, program ne bo pisal na konec te datoteke, temveč bo pobrisal vso prejšnjo vsebino. Če želimo obstoječi datoteki dodajati vsebino, moramo uporabiti način `"a"` (*append*, pripenjanje).
`fclose`: funkcija, ki zapre odprto datoteko.

Poglavje 10

Asimptotična notacija

10.1 Merjenje efektivnosti programa

Pogosto obstaja več možnosti, kako se lahko lotimo reševanja danega problema. Če želimo najti najmanjši element v seznamu, lahko pregledamo celoten seznam in si beležimo najmanjšega, ki smo ga našli do sedaj, lahko pa celoten seznam uredimo po vrsti in nato izberemo prvi element, na primer. Pričakujemo lahko, da se bodo različni algoritmi za reševanje istega problema razlikovali tudi po tem, kako hitro problem rešijo. Kako pa v računalništvu izmerimo hitrost? Če delamo samo na enem računalniku, lahko izmerimo, konkretno koliko časa je program potreboval, da je zaključil z delovanjem. Na ta način lahko na primerih demonstriramo, da je nek algoritem boljši od drugega; ko pa želimo naše rezultate deliti in primerjati z drugimi, pa se ne moramo zanašati, da bodo imeli enako močen računalnik kot mi, in da bodo njihovi testni primeri primerljivo zahtevni z našimi. Dejansko so težave pri tem še hujše; na hitrost delovanja našega programa ne vpliva samo strojna oprema računalnika (torej, kakšen procesor ima, koliko ima spomina itd.), temveč tudi ostali programi, ki jih imamo hkrati odprte. Če se želimo pogovarjati o hitrosti algoritmov, potrebujemo bolj abstraktno orodje. Na pomoč pride asimptotična zahtevnost.
Da določimo hitrost našega programa, moramo prvo določiti, katere spremenljivke vplivajo na čas delovanja, ter kako je čas od njih odvisen. Rezultat take analize zapišemo kot izraz v oklepaje, pred katere zapišemo veliko črko O: $O(\ldots)$ Poglejmo si primer.

```
int arr[100002];

int poisci_najmanjsega(int n) {
    // Poišči najmanjše število v seznamu, dolgemu n
    int min_idx = 0;
    for (int i = 0; i < n; i++) {
        if (arr[min_idx] > arr[i]) {
            min_idx = i;
        }
    }
    return arr[min_idx];
}
```

Funkcija `poisci_najmanjsega` sprejme število n, ki pove dolžino seznama `arr`. Po seznamu se nato enkrat sprehodi, in si ob tem beleži indeks najmanjšega elementa, ki ga je do sedaj našla.
Razmislimo, katere vse različne operacije program opravi.

- Večkrat med programom nastavimo neki spremenljivki novo vrednost.
- V vsaki iteraciji zanke prištejemo 1 spremenljivki `i`.
- Poleg tega v vsaki iteraciji zanke tudi primerjamo `i` z `n`,

- dvakrat dostopamo do nekega elementa v seznamu,
- ter ju primerjamo.
- Na koncu še enkrat dostopamo do elementa v seznamu, ter ga vrnemo.

Vse naštete operacije same po sebi *trajajo* $O(1)$ časa. To pomeni, da se vedno izvajajo enako hitro, neodvisno od parametrov, ki jim podamo. Rečemo tudi, da porabijo *konstantno mnogo* časa.
Kolikokrat pa izvedemo te operacije? Analizirajmo najslabši primer za naš program; če je seznam `arr` padajoče urejen. Tedaj bomo v vsaki iteraciji zanke enkrat primerjali `i < n`, dvakrat dostopali do elementov seznama, enkrat primerjali `arr[min_idx] > arr[i]`, enkrat nastavili `min_idx`, ter enkrat povečali `i`. Zunaj zanke bomo nastavili `min_idx` ter `i` na začetni vrednosti, ter še enkrat dostopali do elementa v seznamu. Zanka se vedno izvaja za natanko `n` iteracij; vedno vsak element pregledamo enkrat. Torej je celotna časovna zahtevnost našega programa $O(3 + 6n)$. Ker pa za velike n del zunaj zanke hitro postane nepomemben, ga ignoriramo. Poleg tega ignoriramo tudi faktor pred členom n – ker tako in tako ne moramo vedeti, kako hitre so operacije v zanki v primerjavi druga z drugo, te konstante ne moramo natančno določiti. Končna časovna zahtevnost našega programa je torej $O(n)$.
To je tudi najboljši možni algoritem za iskanje najmanjšega elementa v seznamu. Če bi nek algoritem namreč deloval v hitrejšem času kot $O(n)$, bi moral nekatera mesta v seznamu izpustiti; če tedaj algoritmu podamo seznam, ki ima najmanjši element ravno na takem mestu, ga algoritem ne bo našel, in bo podal napačen odgovor.
Pomembna opazka je, da hitrost našega algoritma ni odvisna od velikosti števil v seznamu, temveč le od velikosti seznama. Naslednji program prav tako poišče najmanjše število v seznamu, vendar je konkretno počasnejši od zgornjega:

```
int arr[100002];
int poisci_najmanjsega_slabsi(int n, int m) {
    // Poišči najmanjše število v seznamu, dolgemu n,
    // kjer je največje število veliko največ m
    for (int zelja = 0; zelja <= m; zelja++) {
        // zelja nam pove, kateri element si v tej iteraciji želimo
        // najti. Pogledati moramo še, da ta element dejansko je v seznamu;
        // ko pa najdemo enega, bo to najmanjši (ker zelja v vsaki iteraciji
        // narasca)
        bool je_v_seznamu = false;
        for (int i = 0; i < n; i++) {
            if (arr[i] == zelja)
                je_v_seznamu = true;
        }
        if (je_v_seznamu)
            return zelja;
    }
}
```

V tem programu imamo dve zanki; ena se sprehaja po vseh možnih vrednosti števil v seznamu, druga pa preverja, če je ta element dejansko v seznamu. Vsakič, ko se zunanja zanka izvede enkrat, se notranja izvede `n`-krat, zunanja zanka pa se izvede `m+1`-krat. Torej je zahtevnost $O((m+1) \cdot n)$, oziroma $O(mn + n)$. Člen n v vsoti pa je v vseh primerih manjši od člena mn ali njemu enako velik, zato ga izpustimo. Končna časovna zahtevnost drugega algoritma je torej $O(mn)$.
`int` lahko hrani števila, velika do približno dve milijardi — v najslabšem primeru je m torej približno $2 \cdot 10^9$. Če prvi algoritem na nekem računalniku potrebuje eno sekundo, da se konča, bi drugi algoritem v najslabšem primeru na istem računalniku potreboval več kot šestdeset let.

Primer

Naslednji program za vsako število v seznamu `arr` poišče število števil desno od njega, ki so večja.

```
for (int i = 0; i < n; i++) {
    int stevilo = 0;
    for (int j = i+1; j < n; j++) {
        if (arr[j] > arr[i]) {
            stevilo++;
        }
    }
    printf("%d\n", stevilo);
}
```

Notranja zanka se v prvi iteraciji izvede $(n-1)$-krat, v drugi iteraciji $(n-2)$-krat, v tretji $(n-3)$-krat, itd. V zadnji iteraciji se sploh ne izvede. Skupaj se koda znotraj druge zanke torej izvede $(n-1)+(n-2)+\ldots+1+0=\frac{n(n-1)}{2}$-krat. Spet ignoriramo konstanto $\frac{1}{2}$ ter člen samo z n, in pridemo do zahtevnosti $O(n^2)$.

Primer

Naslednji program preveri, če je število n praštevilo.

```
bool preklicano = false;
for (int i = 2; i * i < n; i++) {
    if (n % i == 0) {
        printf("%d ni prastevilo\n", n);
        preklicano = true;
        break;
    }
}
if (!preklicano)
    printf("%d je prastevilo\n", n);
```

Program ima eno zanko, ki se sprehaja toliko časa, da kvadrat spremenljivke i postane večji kot n, oz. dokler je $i \leq \sqrt{n}$. Zanka se torej izvede v $O(\sqrt{n})$.

10.2 Klasifikacija

Računalnik lahko v eni sekundi opravi približno 10^7 operacij. Da določimo, kako dober algoritem potrebujemo za rešitev neke naloge, lahko preverimo omejitve vhodnih podatkov. Spodnja tabela prikazuje nekaj pogostih časovnih zahtevnosti, ter pripadajoče največje omejitve. Z uporabo te tabele lahko vnaprej določimo, kakšno največjo časovno zahtevnost mora imeti naš program, da reši določeno nalogo.

Zahtevnost	Omejitev za n	Ime zahtevnosti
$O(1)$	brez	*konstantna*
$O(\log n)$	zelo visoka	*logaritemska*
$O(\sqrt{n})$	10^{14}	*korenska*
$O(n)$	10^7	*linearna*
$O(n \log n)$	10^6	
$O(n^2)$	10^4	*kvadratna*
$O(n^3)$	300	*kubična*
$O(2^n)$	20	*eksponentna*

Poglavje 11

Urejanje

11.1 Osnovno o urejanju

V programih pogosto želimo nek seznam števil urediti po vrsti. V ta namen lahko napišemo svojo funkcijo, ki implementira enega od znanih algoritmov za urejanje; npr. *bubble sort*, *insertion sort*, *quick sort*, ipd. Ker pa so učinkovite implementacije pogosto kompliciranе in se pri pisanju hitro zmotimo, je bolje, da uporabimo funkcije, ravno v ta namen vključene v standardno knjižnico. Za to bomo potrebovali na začetek programa dodati še dve vrstici:

```
#include <algorithm>
using namespace std;
```

Ukaz `#include` že poznamo, opazimo pa, da tokrat za spremembo nima končnice `.h`. To je zato, ker funkcije, ki smo jih uporabljali do sedaj, izvirajo iz jezika C, tokrat pa potrebujemo funkcijo, napisano posebej za C++. To razloži tudi drugo vrstico; vse funkcije v standardni knjižnici v C++ so vključene v imenski prostor `std`. Če jih želimo klicati, moramo pred ime funkcije vedno napisati `std::`, ali pa na začetek programa vključiti vrstico `using namespace std`.
Sedaj lahko uporabimo funkcijo `sort`, ki sprejme dva argumenta; začetek in konec predela spomina, ki ga želimo urediti. Poglejmo si enostavni primer.

```
#include <algorithm>
#include <stdio.h>
using namespace std;

int arr[1000003];

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    sort(arr, arr+n);
    for (int i = 0; i < n; i++) {
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

Program prebere število n, za njim pa še n števil, jih uredi naraščajoče, in jih izpiše. Funkcijo `sort` smo poklicali tako, da smo kot prvi argument podali seznam `arr`, kot drugi argument pa konec predela

seznama, ki ga želimo urediti; to zapišemo kot `arr+n`. Točen pomen tega izraza bomo spoznali v prihodnje, za sedaj pa bomo kot drugi argument vedno podali seznam plus njegovo dolžino.
Optimalna časovna zahtevnost algoritma za urejanje je $O(n \log n)$. To v praksi pomeni, da bo urejanje delovalo dovolj hitro za $n \leq 10^6$. Če imamo več podatkov kot toliko, bo urejaje trajalo predolgo in naša rešitev ne bo sprejeta.

11.2 Primerjalna funkcija

Če želimo urediti seznam padajoče namesto naraščajoče, lahko seznam prvo uredimo naraščajoče, in ga nato obrnemo. Ker s tem dobimo veliko dodatnega dela, je bolje, da funkciji `sort` podamo lastno primerjalno funkcijo. Le-ta mora sprejeti dva argumenta ter vrniti `bool`, in sicer; če mora biti prvi argument v urejenem seznamu levo od drugega, mora funkcija vrniti `true`, sicer pa `false`.
Če ne podamo tretjega argumenta, se `sort` obnaša tako, kot da bi podali naslednjo funkcijo:

```
bool compare(int a, int b) {
    return a < b;
}
```

Če želimo urediti seznam padajoče, moramo le podati nasprotno funkcijo:

```
#include <algorithm>
#include <stdio.h>
using namespace std;

int arr[1000003];

int compare_padajoce(int a, int b) {
    return a > b;
}

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    sort(arr, arr+n, compare_padajoce);
    for (int i = 0; i < n; i++) {
        printf("%d\n", arr[i]);
    }
    return 0;
}
```

11.2.1 Urejanje sestavljenih podatkov

Recimo, da imamo v nalogi dana imena tekmovalcev ter točke, ki so jih ti tekmovalci dosegli na tekmovanju, naš cilj pa je, da izpišemo imena tekmovalcev po vrsti glede na doseženo število točk. Če bomo prebrali točke in imena v različna seznama, ter uredili seznam točk, bo seznam imen ostal nespremenjen in ne bomo več vedeli, katero ime pripada katerim točkam.
Kako uredimo oba seznama hkrati? Bolj enostavna možnost je uporaba `struct`, ki pa ga še ne poznamo. Namesto tega si lahko pripravimo seznam indeksov, ki na začetku na `i`-tem mestu hrani številko `i`. Sestavili bomo funkcijo `compare` tako, da sprejme dva indeksa, ter ju uredi glede na vrednosti v tabeli s točkami na pripadajočih indeksih. Urejamo pa ne tabele s točkami, temveč novo tabelo indeksov. Na ta način se tabeli s točkami in z imeni ne bosta spreminjali, in bodo točke pripadale imenu na istem indeksu.

Primer

Primer implementacije opisane rešitve:

```
#include <algorithm>
#include <stdio.h>
using namespace std;

int tocke[100003];
char imena[100003][30];
int idxs[100003];

int compare(int i, int j) {
    return tocke[i] > tocke[j];
}

int main() {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; i++) {
        scanf("%s%d", imena[i], &tocke[i]);
    }
    // pripravimo tabelo indeksov
    for (int i = 0; i < n; i++)
        idxs[i] = i;

    sort(idxs, idxs+n, compare);

    for (int i = 0; i < n; i++) {
        int idx = idxs[i];
        printf("%s ", imena[idx]);
    }
    return 0;
}
```

Primer vhoda in izhoda

```
5
France 37
Gregor 34
Julija 38
Matija 29
Urska 8
```

```
Julija France Gregor Matija Urska
```

Poglavje 12

Spomin in kazalci

12.1 Računalniški spomin

Spomin, pomnilnik ali angl. RAM (*Random access memory*) je ključna komponenta v računalniku. Med tekom programa so v njem vse spremenljivke, ki jih program uporablja. S spremenljivkami lahko naredimo tudi več, kot smo se do sedaj naučili, vendar pa moramo za to razumeti, kako je spomin zgrajen.

Osnovna enota za merjenje količine informacij je *bit*. En bit informacij ustreza odgovoru na eno vprašanje tipa da ali ne – če nam nekdo pove, da so vrata zaprta, nam je podal en bit informacij, ker so lahko vrata bodisi odprta bodisi zaprta. Če imamo v omari dva para hlač, dve majici in dve kapi, lahko opišemo, kako smo oblečeni, s tremi biti informacije – za vsak kos oblačila porabimo en bit.

V računalništvu bite najpogosteje označujemo z ničlami in enicami. Običajno ničla predstavlja odgovor "ne" na dano vprašanje, enica pa odgovor "da". Bit pa je zelo majhna količina informacije, zato pogosto govorimo v večjih enotah, kot so *bajti*, *kilobajti*, *megabajti* itd. En bajt ustreza osmim bitom, kilobajt je tisoč bajtov, megabajt je tisoč kilobajtov, gigabajt je tisoč megabajtov, in terabajt je tisoč gigabajtov.

Pogoste napake

Pogosto, a napačno mišljenje je, da kilobajt ustreza 1024 bajtom. Ta mit izvira iz zgodnjih časov računalništva, ko so za hitrejše računanje nekateri programi med spominskimi enotami pretvarjali s to napačno številko; 1024 je namreč potenca 2, s katerimi računalniki pogosto lahko hitreje računajo. Če za pretvorbo uporabljamo faktor 1024, moramo za enote podati *kibibajte* (KiB), *mebibajte* (MiB), *gibibajte* (GiB), ipd., namesto običajnih SI predpon.

Računalniški spomin je sestavljen iz spominskih celic, ki so dolge en bajt. Vsaka od teh celic ima svoj *naslov* – številko, s katero lahko to celico ločimo od ostalih. Naslovi so zaporedne številke od 0 do velikosti pomnilnika, ki ga imamo nameščenega v računalniku. Celice so naraščajoče urejene po svojih naslovih – celica številka 150337 je sosednja celicama s številkama 150336 in 150338.

Upravljanje z računalniškim spominom je ena od nalog operacijskega sistema. Naši programi operacijski sistem med izvajanjem prosijo za neko količino spomina, operacijski sistem pa določi, katere spominske celice bo program prejel. Te celice tedaj pripadajo programu, dokler se ta ne zaključi, ali dokler tega spomina ne vrne operacijskemu sistemu na drugačen način. Med izvajanjem našega programa praviloma noben drug program nima dostopa do tega dela spomina.

Kako pa program ve, koliko spomina bo potreboval? Da to izračuna, se zanaša na tipe. Vsaka spremenljivka ima tip, vsak tip pa ima fiksno dolžino, ki jo zavzame v spominu. Dolžine pogostih tipov so sledeče:

- `int`: 4 bajti
- `long long`: 8 bajtov

- `char`: 1 bajt
- `bool`: 1 bajt

Spremenljivke, ki v spominu zavzamejo več kot 1 bajt, moramo shraniti v več kot eno spominsko celico. Celice, v katere te vrednosti zapišemo, so v spominu zaporedne; če imamo spremenljivko tipa `int`, bo zavzela 4 zaporedne celice.

12.2 Kazalci

Pogoste napake

Kazalci so pomemben koncept v programiranju, in razumevanje kazalcev je ključno za razumevanje bolj zapletenih podatkovnih struktur ter nekaterih algoritmov. S kazalci pa se lahko zelo hitro zmotimo, in razhroščevanje kode z veliko kazalci je pogosto zelo zapleteno. Zaradi tega in drugih razlogov se kazalcem izogibamo v tekmovalnem programiranju, razen če jih res nujno potrebujemo.

Ker so spominski naslovi številke, jih lahko shranjujemo, kakor shranjujemo ostale številke; za to pa imamo v C++ na voljo poseben tip, ki mu rečemo *kazalec* (angl. *pointer*). Pravzaprav kazalec ni sam svoj tip, ampak razširitev nekega drugega tipa; pravimo, da kazalec *kaže na drug tip*. Da ustvarimo nov kazalec, zapišemo ime tipa, na katerega želimo kazati, nato pa pred ime spremenljivke damo zvezdico `*`. Kazalcu lahko nastavimo vrednost tako, da vanj shranimo naslov neke spremenljivke, ki smo že ustvarili. Do naslova dostopamo z operatorjem `&`. Da dostopamo do vrednosti, shranjene v celici, na katero kazalec kaže, uporabimo operator `*` (ki ima drugačen pomen kot zvezdica v deklaraciji spremenljivke).

Primer

```c
#include <stdio.h>

int main() {
    int a = 7;
    int b = 3;

    // Ustvarimo kazalec na int, ki kaže na spremenljivko a
    // To pomeni, da bo v kazalcu shranjen naslov prve od štirih
    // celic, ki jih zavzema a.
    int *kazalec = &a;

    // Izpišemo vrednost, na katero kaže kazalec
    printf("%d\n", *kazalec);  // 7

    // Če spremenimo vrednost, na katero kaže kazalec, se spremeni
    // vrednost v spominu; torej tudi spremenljivka, ki je tam shranjena
    *kazalec = 15;
    printf("%d\n", a);  // 15

    // Če spremenimo lokacijo, kamor kaže kazalec, nismo spremenili vrednosti,
    // na katero je kazal prej
    kazalec = &b;
    printf("%d\n", *kazalec);  // 3
    printf("%d\n", a);  // 15

    return 0;
}
```

Vse, kar smo tu delali s kazalci, je bilo možno (in lažje) narediti tudi z običajnimi spremenljivkami. Kazalci, ki kažejo na spremenljivke, ki tako in tako že obstajajo, so bolj ali manj neuporabni. Kako pa naredimo kazalec, ki kaže na del spomina, v katerem ni nobene spremenljivke?
Če želimo storiti kaj takega, moramo prevzeti odgovornost za upravljanje spomina v našem programu. Do sedaj je za to skrbel prevajalnik, ki je v naš program na pravilna mesta zapisal ukaze, ki si spomin sposojajo od operacijskega sistema, ter ga vračajo, ko ga ne potrebujemo več. Bolj natančno; ko smo deklarirali spremenljivko, je prevajalnik poskrbel, da prosimo za natanko toliko spomina, kolikor ga za to spremenljivko potrebujemo (zato moramo za vsako spremenljivko zapisati tip), ter si njegov naslov zapomnil, ko pa spremenljivke nismo več potrebovali, je prevajalnik poskrbel, da ta del spomina vrnemo operacijskemu sistemu; temu pravimo *sprostitev* (angl. *deallocation*).
Za bolj sofisticirano uporabo spomina moramo ti vlogi prevzeti mi. Za to sta nam na voljo dve funkciji: `malloc` in `free`. Da ju uporabljamo, moramo vključiti `stdlib.h`. Oblika funkcij je naslednja:

```c
void *malloc(size_t size);
void free(void *ptr);
```

V obliki je posebnost, ki je še nismo omenili; ni namreč nujno, da ima vsak kazalec tip. Lahko imamo kazalce, ki kažejo na del spomina, mi pa (še) ne vemo, kaj je v tistem delu spomina shranjeno. Za take kazalce pravimo, da kažejo na `void`, kar pa ne pomeni, da ne kažejo na nič; na lokaciji v spominu, kamor kažejo, je nekaj shranjeno; mi samo ne vemo, kako naj te podatke interpretiramo.
Funkcija `malloc` sprejme en argument, in vrne kazalec na `void`. Ta argument je tipa `size_t`, ki je za naše potrebe skoraj enak tipu `unsigned long long`; to je torej številka. Pove, koliko bajtov spomina si želimo sposoditi od operacijskega sistema. Funkcija `malloc` nato vrne kazalec na prvi naslov znotraj bloka spomina, ki smo si ga ravno sposodili. Ker operacijski sistem ne ve, kaj bomo v ta spomin shranili,

nam funkcija `malloc` vrne `void*`, mi pa ga moramo pretvoriti v pravi tip kazalca. To storimo tako, da tik pred klic funkcije v oklepaje zapišemo želeni tip kazalca.
Funkcija `free` je ravno nasprotna od `malloc`; sprejme kazalec, ki ga nam je dal `malloc`, ter sprosti del spomina, na katerega kazalec kaže. Kazalec bo po klicu `free` še vedno obstajal, in bo še vedno kazal na isto mesto. Edina sprememba je, da del spomina, na katerega kaže, ne pripada več našemu programu, in ga ne smemo uporabljati.

Primer

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

int main() {
    int *kazalec = (int*)malloc(sizeof(int));
    *kazalec = 3;
    printf("%d\n", *kazalec);  // izpiše 3
    return 0;
}
```

Operator `sizeof` lahko uporabljamo, da si pomagamo pri določevanju velikosti tipov.

12.3 Kako delujejo seznami

Nič nas ne omejuje, da od operacijskega sistema zahtevamo zelo velik blok spomina, tudi po več sto tisoč bajtov. Pa imamo lahko kakšen utemeljen razlog, da si toliko spomina izposodimo? Da, ravno to stori prevajalnik, ko ustvarimo seznam. Poglejmo si, kako ustvarimo seznam samo s kazalci.

Primer

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

int main() {
    int dolzina_seznama = 100000;
    int *seznam = (int*)malloc(dolzina_seznama * sizeof(int));

    for (int i = 0; i < dolzina_seznama; i++) {
        scanf("%d", seznam+i);
    }

    // sedaj lahko spremenljivko ‘seznam‘ uporabljamo
    // kot katerikoli drugi seznam
    int vsota = 0;
    for (int i = 0; i < dolzina_seznama; i++) {
        vsota += seznam[i];
    }

    printf("%d\n", vsota);

    // ne smemo pozabiti sprostiti spomina
    free(seznam);

    return 0;
}
```

V zgornjem primeru uporabljamo dva nova operatorja na kazalcih; seštevanje in oglate oklepaje. Če kazalcu `seznam` prištejemo število v spremenljivki `i`, dobimo nov kazalec, ki kaže na mesto v spominu z naslovom `seznam + (velikost tipa) * i`, torej na idealno mesto, kamor zapišemo `i`-ti element seznama, če jih zapisujemo enega za drugega.
Drug novi operator so oglati oklepaji – ti se obnašajo popolnoma enako kot v seznamih. Oglati oklepaj `seznam[i]` je pravzaprav krajšava za zapis `*(seznam+i)`, torej za dostop do `i`-tega elementa v bloku spomina.
Tudi seznami, kakor smo jih spoznali prej, so dejansko kazalci na blok spomina, le da s tem spominom upravlja prevajalnik. Trik s prištevanjem števila h kazalcu deluje tudi za prištevanje števila k seznamu.

12.4 Podajanje po referenci

Opazimo, da smo operator `&` že srečali, in sicer čisto na začetku. Pri branju številk iz vhoda moramo v `scanf` zapisati ta operator pred imenom spremenljivke. Sedaj razumemo, zakaj je temu tako; `scanf` sprejme kazalce na spremenljivke, ki jih želimo prebrati, ter popravi vrednosti, na katere kažejo kazalci, s prebranimi vrednostmi.

Primer

`scanf` je tudi funkcija, le da je nismo zapisali mi. Zmožna pa je nečesa, česar naše funkcije niso sposobne; spremeniti vrednosti spremenljivk zunaj nje. Spodnji program se ne bo niti prevedel:

```
#include <stdio.h>

void f(int x) {
    y = 2 * x;
}

int main() {
    int y;
    int x = 3;
    f(x);
    printf("%d\n", y);
    return 0;
}
```

Naslednji program pa se bo prevedel, vendar bo morda izhod v nasprotju s pričakovanji:

```
#include <stdio.h>

void f(int x) {
    x = 2 * x;
}

int main() {
    int x = 3;
    f(x);
    printf("%d\n", x);
    return 0;
}
```

Primer vhoda in izhoda

```
3
```

Spremenljivke, ki jih deklariramo v funkciji, to je znotraj telesa funkcije, ali pa v seznamu argumentov, so lokalne k tej funkciji — zunaj nje sploh ne obstajajo. Če želimo, da funkcija popravi neko vrednost, ki jo uporabljamo tudi zunaj funkcije, smo do sedaj lahko to naredili samo tako, da smo spremenljivko naredili globalno, torej dostopno vsem funkcijam (tudi `main` je funkcija). Kaj pa, če želimo neko spremenljivko na tak način deliti samo med dvema funkcijama?

Da odgovorimo na to vprašanje, moramo razumeti, kako se argumenti podajajo v funkcije. Ko neko funkcijo pokličemo, se argumenti, ki jih funkciji podamo, *prekopirajo* v poseben del spomina, ki ji pripada. Ko smo znotraj ene funkcije, ne poznamo imena spremenljivk v drugih funkcijah; prav tako ne vemo, kje so te spremenljivke shranjene. Nič pa nam ne preprečuje, da spreminjamo spomin, ki našemu programu pripada, pa četudi je zunaj funkcije; razen tega, da ne vemo, kateri del spomina je naš, in kateri ni. Lahko si predstavljamo, da smo zabredli v spominsko džunglo, v kateri ne prepoznamo prave poti do spremenljivk, ki jih želimo popraviti. Če pa s seboj prinesemo zemljevid, bomo nenadoma to lahko naredili. Ta zemljevid je kazalec.

Funkcija lahko brez težav sprejme kazalec kot argument. Če to storimo, pravimo, da smo spremenljivki

vrednost podali *po referenci* (angl. *pass by reference*), namesto da bi argument podali običajno, čemur pravimo *podajanje po vrednosti* (angl. *pass by value*). Kazalec, ki smo ga podali, se bo še vedno prekopiral v del spomina, ki pripada funkciji; vrednost, na katerega kazalec kaže, pa bo ostala tam, kjer je. Tako lahko skozi kazalec spremenimo vrednosti spremenljivk zunaj funkcije.

Primer

Primera od zgoraj, narejena tako, da delujeta.

```
#include <stdio.h>

void f(int vrednost, int *kazalec) {
    *kazalec = 2 * vrednost;
}

int main() {
    int x = 3;
    int y;
    f(x, &y);
    printf("%d\n", y);
    return 0;
}
```

Kot vidimo, mora funkcija vedno sprejeti tudi argument, ki ga spremeni.

```
#include <stdio.h>

void f(int *kazalec) {
    *kazalec = 2 * (*kazalec);
}

int main() {
    int x = 3;
    f(&x);
    printf("%d\n", *x);
    return 0;
}
```