Iteracije i skokovi

Ciljevi:

- proučiti i shvatiti pojam iteracije
- proučiti i shvatiti pojam skoka
- navesti tipove iteracija
- shvatiti uporabu iteracija
- shvatiti zašto se skokovi više ne koriste u velikoj mjeri

Pregled lekcije

Iteracija

Vrlo često u programiranju imamo potrebu neki dio programskog koda ponoviti više puta. Kao konkretan primjer navedene situacije uzmimo sljedeći slučaj.

Uzmimo da želimo ispisati brojeve od 1 do 100 na ekran.

Prvi način, koji već znamo realizirati je da jednostavno ispišemo svaki pojedini broj koristeći naredbu za ispis 100 puta. Dakle, naše rješenje bi izgledalo ovako:

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

int main () {

	cout << "1" << endl;
	cout << "2" << endl;
	cout << "3" << endl;
	cout << "4" << endl;

	...

	cout << "98" << endl;
	cout << "99" << endl;
	cout << "100" << endl;

	system("pause");

	return 0;
}
```

Navedeno rješenje će u potpunosti izvršiti zadani zadatak, no ono što ćete s vremenom uvidjeti je da nije bitno samo da program radi ono što se od njega očekuje, već je bitno i koliko brzo to radi, da li je kod pregledan i realiziran na najbolji mogući način ili je bespotrebno dug i kompliciran te težak za održavanje i ispravljanje. Sve su to elementi koji su bitni kada govorimo o kvaliteti programa. U ovom slučaju to znači da mi želimo napisati takav progam u kojem nećemo morati navoditi naredbu ispisa za svaki pojedini broj.

Primjer 1:
Ulaz: -
Izlaz: Ispis prirodnih brojeva od 1 do 100

1. I = 1
2. Ispiši I
3. I = I + 1
4. Ako je I <= 100 idi na 2

Ovo je pseudokod koji je napravljen prema standardnom proceduralnom razmišljanju i on se može lako pretvoriti u programski kod:

Primjer 1.1
```
#include <iostream>
using namespace std;

int main () {
lb: int I = 1;
    cout << I << endl;
    if (I <= 100) goto lb;
    system("pause");
    return 0;
}
```

U ovom se programu koristi naredba skoka **goto**. Ova se naredba naziva naredbom bezuvjetnog skoka. Ona funkcionira tako da se negdje u progarmu navede oznaka (labela). Oznaka se mora nalaziti na početku linije i iza nje mora slijediti dvotočka. Kod imenovanja oznaka koriste se ista pravila kao i za identifikatore varijabli. Dakle, oznaka se može sastojati od brojki, slova i znaka "_", s time što prvi znak u oznaci ne smije biti brojka. Nakon same naredbe **goto**, slijedi naziv oznake ispred linije na koju želimo skočiti (bez dvotočke).

Kao što je to već rečeno prilikom uvođenja naredbe skoka `break`, zbog čitljivosti i strukturnosti programa naredbe skoka valja izbjegavati. Drugi način kojim se može riješiti ovaj problem, ne koristeći naredbu **goto** je korištenjem još jednog programskog konstrukta - iteracije. Naime već smo spomenuli što su to i koji su to programski konstrukti. Naše trenutno rješenje koristi samo jedan programski konstrukt, a to je sekvenca (slijed). U prošlom smo primjeru koristili selekciju i bezuvjetni skok. Sada ćemo po prvi puta koristiti iteraciju, odnosno petlju.

Iteracija je programski konstrukt koji omogućava da se dio koda izvede više puta.

U ovoj ćemo lekciji obraditi prvu vrstu iteracije – iteraciju s eksplicitnim brojačem, odnosno iteraciju tipa **for**.

Iteracija **for** je iteracija čija je karakteristika korištenje brojača i mogućnost saznanja o broju ponavljanja prije samog izvođenja iteracije. Brojač iz ove iteracije može biti iskorišten kao vrijednost unutar svakog pojedinog ponavljanja.

Sintaksa ove iteracije je sljedeća:

```
for(početna vrijednost; uvjet završetka; promjena){
   // programski kod iteracije ili programski blok
};
```

U slučaju našeg primjera **for** iteracija (petlja) je idealna jer nam je poznat broj koraka petlje (100 koraka tj. 100 ispisa). Unutar svakog pojedinog koraka možemo ispisati sam brojač **for** petlje i na taj način dodatno pojednostaviti kod. Ako iskoristimo ovu petlju dobivamo sljedeće rješenje.

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

int main () {

            for (int B = 1; B<=100; B=B+1)
                 cout << B << endl;

      system("pause");

      return 0;
}
```

DEKLARIRANA JE VARIJABLA B I NA POČETKU JE NJEZINA VRIJEDNOST 1

DAKLE, PETLJA ZAVRŠAVA U TRENUTKU KADA JE **B = 101**

B SE U SVAKOJ ITERACIJI POVEČAVA ZA 1 (OSIM PRVI PUTA)

Navedeni kod koristi **for** petlju koja prima 3 argumenta: varijabla koja služi kao brojač i njena početna vrijednost, uvjet završetka petlje i promjenu tj. korak povećanja/smanjenja (u našem slučaju brojač se u svakom koraku povećava za 1). U ovom je primjeru potrebno napomenuti jednu važnu stvar: varijabla B se deklarira u samoj glavi petlje. Naime, to je varijabla koju ne koristimo izvan petlje, pa je nije ni potrebno deklarirati izvan nje. S druge strane, ne možemo je deklarirati ni u tijelu petlje, jer će to dovesti do pogreške, koja će nam reći da se u glavi petlje koristi nedeklarirana varijabla. Stoga se varijabla mora deklarirati baš u glavi petlje.

Razlikujemo dva načina ulaska u glavu petlje: po prvi puta ili "odozgo" i povratak "odozdo". S obzirom na to jesmo li u glavu petlje ušli po prvi puta ili pak se radi o ponovnom povratku u glavu petlje izvršavat će se neki od triju izraza navedenih u glavi petlje. Prvi će se izraz, aritmetički izraz koji definira inicijalnu vrijednost, izvršava samo u prvom dolasku u glavu petlje. Drugi se izraz, logički izraz koji provjerava uvjet završetka petlje izvršava svaki puta, dok se posljednji izraz, koji definira promjenu vrijednosti brojača izvodi samo pri povratku u glavu petlje, ali ne i kod prvog ulaska u nju.

Možemo vidjeti da smo veliki dio koda od preko 100 programskih redova, korištenjem iteracije smanjili na svega nekoliko programskih redova. Dakle, kod je pregledniji, lakši i brži.

Pogledajmo još jedan primjer korištenja **for** petlje. Korisnik upisuje prirodni broj N i N decimalnih brojeva a_1, ..., a_N. Potrebno je ispisati sumu apsolutnih vrijednosti unešenih brojeva.

Pseudokod rješenja ovog problema je sljedeći

Primjer 1:

Ulaz: Prirodni broj N i N decimalnih rojeva a_1, ..., a_N.
Izlaz: suma apsolutnih vrijednosti brojeva a_1, ..., a_N.

1. Učitaj N
2. S = 0
3. Za I = 1..N radi
4. Učitaj A
5. Ako je A>=0
6. S = S + A
7. inače
8. S = S – A
9. Ispiši S

Ako se ovaj pseudokod izravno prevede u program, dobivamo sljedeći kod:

Primjer 1.1

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

int main () {
    unsigned short N;
    double A, S = 0;
    cout << "N = ";
    cin >> N;
    for (int B = 1; B <= N; B = B + 1) {
        cout << "A_" << B << "=";
        cin >> A;
        if (A >= 0)
           S = S + A;
        else
            S = S - A;
    }
    cout << "Suma apsolutnih vrijednosti je " << S << endl;
    system ("pause");
     return 0;
}
```

U prošloj smo lekciji obradili standardne binarne aritmetičke operatore, dok smo još ranije obradili osnovni osnovni operator pridruživanja vrijednosti varijabli. No, programski jezik C++ nastao je iz progamskog jezika jezika C, u kojemu je efikasnost važna komponenta. Ovaj se program može optimizirati na nekoliko mjesta. Naime, ako se pogleda izraz `S = S + A` i ako se razmili o načinu njegova izvršavanja, vidjet će se da on nije optimalan. Naime, da bi se on izvršio, čitaju se vrijednosti varijabli S i A, zbrajaju se i zapisuju u posebni prostor u memoriji, da i se nakon toga iz tog prostora vraćali u prostor koji zauzima varijabla S. Jasno je da za ovu operaciju nije trebalo sumu zapisivati u posebni memorijski prostor, već se mogla izravno smjestiti u varijablu S. Stoga se u programskom jeziku C++ ne koristi ovakav izraz već se on mijenja izrazom `S += A`, koji ne koristi pomoćni memorijski prostor za izvršenje ovog izraza.

Sada gornji program možemo optimizirati na sljedeći način:

Primjer 1.2

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

int main () {
     unsigned short N;
     double A, S = 0;
     cout << "N = ";
     cin >> N;
     for (int B = 1; B <= N; B += 1) {
          cout << "A_" << B << "=";
          cin >> A;
          if (A >= 0)
             S += A;
          else
              S -= A;
     }
     cout << "Suma apsolutnih vrijednosti je " << S << endl;
     system ("pause");
     return 0;
}
```

Ovakve pokrate postoje za svaki aritmetički operator koji smo u prošloj lekciji obradili. Sljedeća tablica prikazuje standardne izraze i njihove pokrate koje se koriste u programskom jeziku C++

Standardni izraz	C++ pokrata
`S = S + A`	`S += A`
`S = S - A`	`S -= A`
`S = S * A`	`S *= A`
`S = S / A`	`S /= A`
`S = S % A`	`S %= A`

Pogledajmo sada kako se izvršava izraz `B += 1`. U njemu se u pomoćni prostor u memoriji zapisuje broj 1, a nakon toga se vrijednost varijable B povećava za 1. Naravno da nam za to nije bilo potrebno broj 1 zapisivati u memoriju, već se vrijedbost varijable B mogla i izravno povećati za 1. Ova se operacija naziva inkrementacija vrijednosti varijable, dok se smanjenje vrijednosti varijable za 1 naziva dekrementacija vrijednosti varijable. Inkrementacija i dekerementacija su vrlo česte operacije u programiranju, pa za njih u programskom jeziku C++ postoji dodatna pokrata, koja još više optimizira njihovo izvođenje.

Operacija	C++ pokrata
`B += 1`	`B++` `++B`
`B -= 1`	`B—` `--B`

Tako se gornji primjer može dodatno optimizirati tako da se napiše

Primjer 1.3

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

int main () {
     unsigned short N;
     double A, S = 0;
     cout << "N = ";
     cin >> N;
     for (int B = 1; B <= N; B++) {
          cout << "A_" << B << "=";
          cin >> A;
          if (A >= 0)
             S += A;
          else
              S -= A;
     }
     cout << "Suma apsolutnih vrijednosti je " << S << endl;
     system ("pause");
     return 0;
}
```

Prije no što objasnimo operacije inkrementacije i dekrementacije, primijetimo da se u našem primjeru javlja selekcija koja se isključivo odnosi na aritmetički izraz. Sjetimo se da u tom slučaju može koristiti ternarni aritmetički operator koji će to obaviti još efikasnije. Dakle, na kraju će naš primjer zgledati kako slijedi.

Primjer 1.3

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

int main () {
     unsigned short N;
     double A, S = 0;
     cout << "N = ";
     cin >> N;
     for (int B = 1; B <= N; B++) {
          cout << "A_" << B << "=";
          cin >> A;
          S += A >= 0 ? A : -A;
     }
     cout << "Suma apsolutnih vrijednosti je " << S << endl;
     system ("pause");
     return 0;
}
```

Treba napomenuti da inkrementacija i dekrementacija nisu standardni operatori, iako se mogu naći u aritmetičkim izrazima. Oni se mogu javiti i kao sammostalni izrazi, kao što je to slučaj u našem primjeru. Ako se inkrementacija ili dekrementacija samostalna naredba onda ne postoji nikakva razlika pišemo li B++ ili ++B. No, javlja li se inkrementacija ili dekrementacija unutar aritmetičkog izraza, onda se oni razlikuju. Naime, prefiksni operator

inkrementacije ili dekrementacije (++B) će se izvršiti prije bilo koje druge operacije u izrazu, a cijeli će se izraz izračunavati s novom, već uvećanom vrijednosšću. S druge strane, postfiksna se inkrementacija, odnosno dekrementacija izvršava nakon što se izračuna cijeli izraz. Dakle, izraz će se izračunavati sa starom vrijednošću.

Pogledajmo sljedeći primjer:

Primjer 2:

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

int main () {
    int A = 1, B = 2, C = 3;
    int D = A * B++ + C;
    cout << "D = " << D << endl;
    cout << "B = " << B << endl;
    system ("pause");
    return 0;
}
```

```
D = 5
B = 3
```

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

int main () {
    int A = 1, B = 2, C = 3;
    int D = A * B++ + C;
    cout << "D = " << D << endl;
    cout << "B = " << B << endl;
    system ("pause");
    return 0;
}
```

```
D = 6
B = 3
```

U sljedećem primjeru upisujemo prirodne brojeve te ispisujemo najmanji i najveći od unešenih brojeva. Brojevi se unose sve dok se ne unese broj koji nije prirodan.

Pseudokod rješenja ovog problema glasi

Primjer 3:
Ulaz: Prirodni brojevi
Izlaz: Najmanji i najveći unešeni prirodni broj, kao i njihova pozicija

1. Učitaj A
2. MAX = MIN = A
3. Sve dok je A > 0 radi
4. Učitaj A
5. Ako je A > 0
6. Ako je A < MIN
7. MIN = A
8. Ako je A > MAX
9. MAX = A
10. Ispiši MIN, MAX, IMIN, IMAX

Primiijetimo da ovaj pseudokod sugerira korištenje petlje, no da se prva vrijednost unosi prije početka petlje. Naime, u petlji se svaka učitana vrijednost uspoređuje s dotada najmanjom i najvećom učitanom vrijednošću, te ako je manja od najmanje, odnosno veća od najveće dotad unešene vrijednosti, postaje najmanja, odnosno najveća unešena vrijednost.

No, prva se unešena vrijednost nema s čime usporediti. Stoga se ona unosi prije petlje, i automatski postaje i najmanja i najveća unešena vrijednost.

Ovaje se problem može također riješiti pomoću **`for`** petlje. Jedini je problem u tome što nam ovdje nije potreban brojač, a uvjet završetka se odnosi na vrijednost koja je unešena u prethodnom koraku. Fleksibilnost `for` petlje nam omogućuje da njome rješavamo i ovakve probleme. Naime, ako nam jedan ili više izraza koji se pojavljuju u glavi `for` petlje nije potreban, moguće ga je jednostavno izostaviti. Tako ćemo u ovom primjeru, u kojem nam nije potreban brojač, izostaviti prvi i treći izraz u glavi petlje.

Primjer 3.1:

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

int main () {
    int A;
    cout << "A = ";
    cin >> A;
    int MAX = A, MIN = A;
    for ( ; A > 0; ) {
        cout << "A = ";
        cin >> A;
        if (A > 0)
           if (A > MAX)
              MAX = A;
           if (A < MIN)
              MIN = A;
    }
    cout << "Najmanja vrijednost je " << MIN << endl;
    cout << "Najveca vrijednost je " << MAX << endl;
    system("pause");
    return 0;
}
```

Napišimo program kojim ćemo unijeti prirodni broj N i N decimalnih brojeva. Potrebno je napraviti sumu unešenih pozitivnih brojeva, dok se negativni zanemaruju.

Primjer 4:

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

int main () {
    int N;
    float A, S = 0;
    cout << "N = ";
    cin >> N;
    for (int i=1; i<=N; i++) {
        cout << "A = ";
        cin >> A;
        if (A <= 0) continue;
           S += A;
    }
    cout << "S = " << S << endl;
```

```
    system("pause");
    return 0;
}
```

U ovom smo primjeru koristili još jednu naredbu skoka – naredbu **continue**. Naredba **continue** se uvijek koristi unutar petlje i služi kako bi se prekinuo rad koraka petlje i odmah prešlo ponovo na glavu petlje, odnosno krenulo s sljedećim korakom petlje. Kao i sve narede skoka, i narednu **continue** treba izbjegavati. Ona se uvijek može zamijeniti selekcijom. Tako na primjer prethodni primjer možemo napisati kao

Primjer 4.1:

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

int main () {
    int N;
    float A, S = 0;
    cout << "N = ";
    cin >> N;
    for (int i=1; i<=N; i++) {
        cout << "A = ";
        cin >> A;
        if (A > 0) S += A;
    }
    cout << "S = " << S << endl;
    system("pause");
    return 0;
}
```

Osim stanardnih aritmetičkih operatora, u programskom jeziku C++ postoje i bitovni operatori – operatori čije se izvršenje temelji na binarnom zapisu cijelobrojnih vrijednosti u računalu. Sljedeća tablica daje popis bitovnih operatora u programskom jeziku C++.

Bitovna operacija	Primjer
Pomicanje za N bitova u lijevo	A << N
Pomicanje za N bitova u desno	A >> N
Bitovna konjunkcija	A & B
Bitovna disjunkcija	A | B
Bitovna ekskluzivna disjunkcija	A ^ B
Bitovna negacija	~A

Cjelobrojna je vrijednost zapisana u jednom dva ili četiri bajta na relativno jednostavan način. Prva dva bitovna operatora omogućujuju da se svaki bit binarnog zapisa cijelog broja pomakne za određeni broj bitova u lijevo, odnosno u desno, brišući pri tome one bitove koji tako ispadaju iz prostora predviđenog za zapis broja, a s druge strane puneći oslobođene bitove nulama. Pogledamo li što znači kada zapis nekog cijelog broja pomaknemo za N bitova u lijevo. Pomicanjem zapisa za jedan bit u lijevo dobit ćemo vrijenost koja je dva puta veća od vrijednosti koju je zapis prije toga sadržavao. Dakle, `A << N` je jednostavniji i mnogo efikasniji način da se zapiše `A * (int)pow(2., N)`. Slično tome, pomicanje zapisa

cjelobrojne varijable za N bitova u desno je isto kao cjelobrojno dijeljenje tog broja s 2^N, odnosno `A >> N` je isto što i `A * (int)pow(2., N)`.

Sljedeća četiri bitovna operatora predstavljaju standardne Boolovske logičke bitovne operatore. Kod ovih operatora međusobno djeluju bitovi na istoj poziciji prema sljedećim tablicama:

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

A	~A
0	1
1	0

Kao primjer korištenja uzmimo prikaz broja po bazi 4. Kako bismo dobili broj po bazi 4, moramo gledati po dvije znamenke broja i pretvarati ih u broj po bazi 4. Prvo što trebamo napraviti, jest izdvojiti dvije tražene binarne znamenke, a ostale postaviti na 0. Ovo se lako može napraviti binarnim operatorom konjunkcije. Naime,

```
  xxxxxxxxxxxxxxxx
& 0011000000000000
  00xx000000000000
```

Dakle, ako imam broj koji ima određene dvije binarne znamenke postavimo na 1, a ostale mu postavimo na 0 i ako tako dobiveni broj bitovno konjugiramo, dobit ćemo broj koji ima te dvije binarne znamenke jednake početnom broj, a ostale znamenke jednake 0. Pomaknemo li taj broj toliko udesno da te dvije znamenke postanu posljednje dvije u broju, dobit ćemo znamenku u zapisu broja po bazi 4. Još je jedino pitanje kako dobiti broj kojim trebamo konjugirati početnu vrijednost. No i to je jednostavno. Broj 3 je u binarnom zapisu 11. Uzmemo li broj 3 i pomaknemo mu bitove za 14 mjesta u lijevo, dobit ćemo broj 1100000000000000_2. U svakom sljedećem koraku samo taj broj pomičemo po dva bita u desno i konjugiranjem ćemo očitati sljedeća dva bita zadane vrijednosti. Na kraju. Pseudokod ovog programa je sljedeći:

Primjer 5:

Ulaz: Vrijednost A tipa unsigned short
Izlaz: Broj A u sustavu s bazom 4.

1. Učitaj broj A
2. F = 3 << 14
3. Za parne I = 14..0 radi
4. D = (A & F) >> I
5. Ispiši D
6. F = F >> 2

Zapišemo li to u programskom kodu, dobit ćemo

Primjer 5.1:

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

int main () {
    unsigned short A, F = 3 << 14;
    cout << "A = ";
    cin >> A;
    cout << "_4 = ";
    for (int i = 14; i >= 0; i -= 2) {
        unsigned short D = (A & F) >> i;
        cout << D;
        F >>= 2;
    }
    cout << endl;
    system ("pause");
    return 0;
}
```

U ovom primjeru, osim pojave samih bitovnih operatora vidi se da se oni, poput aritmetičkih operatora, mogu spajati s operatorom "=" u složene operatore pridruživanja. Tako da je izraz F >>= 2 zapravo optimizacija izraza F = F >> 2.

Zadaci za vježbu

1. Napišite program koji traži od korisnika unos jednog znaka i taj unos se ponavlja sve dok korisnik ne unese znak d.

2. Napišite program koji u for petlji ispisuje brojeve od 10 do 20 i nakon ispisa broja 15 pomoću naredbe `goto` skače na kraj programa.

3. Odgovorite na pitanje što je to iteracija.

4. Odgovorite na pitanje što je to skok.

Programski primjeri za laboratorijske vježbe

Zadatak 1.

Izradite program u kojem:

1. Korisnik unosi jedan cijeli broj. Nakon toga se, koristeći for petlju, ispisuju na ekran svi brojevi od 1 pa sve do unesene vrijednosti (za 3 se ispisuje 1, 2, 3).

2. Korisnik unosi 5 brojeva (mogu biti cijeli i decimalni) te se na ekran ispisuje njihova suma i aritmetička sredina. Rješenje realizirati pomoću for petlje.

3. Korisnik unosi jedan cijeli broj. Na ekran se potom ispisuje komplement unesenog broja u decimalnom obliku. Korisnik potom unosi drugi cijeli broj. Unesenom broju se bitovi pomiču za 5 mjesta ulijevo te se nakon toga isti broj ispisuje na ekran u decimalnom obliku.

4. Korisnik unosi 5 cijelih brojeva. Ukoliko je bilo koji od njih broj 3, na ekran se ispisuje poruka „Pogodak“ i program završava. Ukoliko korisnik unese svih 5 brojeva a među njima ne bude broj 3 tada se na ekran ispisuje proruka („Pogreška“) i program završava. U rješavanju zadataka koristiti for petlju, if uvjet i goto skok.

5. Korisnik unosi cijeli broj između 10 i 20 te se nakon toga u for petlji ispisuju svi parni brojevi od 1 do unesenog broja. Rješenje treba realizirati koristeći naredbu continue.