

**Maja Matetić**

**Skripta uz predmet Programiranje 1**

**Odjel za informatiku  
Sveučilište u Rijeci**

Rijeka, 2012.

## UVOD

Ova skripta je nastala kao nastavni tekst uz predmete Programiranje 1 i Programiranje koji se predaju u okviru studijskih programa preddiplomskog studija jednopredmetne informatike, dvopredmetne informatike, matematike, fizike i politehnike čiji je nositelj ili sunositelj Odjel za informatiku Sveučilišta u Rijeci. Skripta je dostupna studentima u okviru sustava za udaljeno učenje Mudri Sveucilišta u Rijeci (<http://mudri.uniri.hr/>).

Zbog zahtjeva kontinuiranog vrednovanja nastave u okviru Bolonjskog procesa ova skripta je nastala sa ciljem da studentima pomogne da čim brže i lakše počnu pratiti nastavne sadržaje u okviru ovog predmeta, te da se čim bolje pripreme za provjere znanja. Ona omogućava razne pristupe i aktivnosti u provjeri znanja. Primjerice, na temelju "Priprema za kviz" koje sadrže razne teoretske i praktične zadatke, na samim predavanjima studenti kroz aktivnost skupljaju bodove. Ti ih zadaci pripremaju i za teoretske kvizove. U okviru sustava za učenje na daljinu Mudri na forumima sa studentima se odvija diskusija o zadacima i njihovim rješenjima. U okviru vježbi koje prate predavanja, studenti sa asistentima rješavaju dodatne zadatke i zadaće koje nisu obuhvaćene ovom skriptom.

Skripta se temelji na suvremenoj literaturi koja se često koristi u predmetima vezanim uz programiranje. Zahvaljujem kolegama i studentima koji su kroz uporabu ove skripte svojim primjedbama i sugestijama doprinijeli da ona postane kvalitetnija. Dobrodošli su vaši daljnji prijedlozi i sugestije.

Autorica skripte

# Sadržaj

## UVOD 2

### 1. UVOD U PROGRAMIRANJE 6

RAČUNALNI SUSTAVI 6

SOFTVER 8

JEZICI VISOKE RAZINE 9

PREVODITELJI (KOMPJLERI) 9

### 2. PROGRAMIRANJE I RIJEŠAVANJE PROBLEMA 12

ALGORITMI 12

OBLIKOVANJE PROGRAMA 13

OBJEKTNO-ORIJENTIRANO PROGRAMIRANJE (OOP) 14

ŽIVOTNI CIKLUS SOFTVERA 14

UVOD U C++ 15

TESTIRANJE I ISPRAVLJANJE GREŠAKA (DEBUGGING) 19

### 3. UVOD U C++ 20

VARIJABLE I DODJELA 20

DEKLARACIJA VARIJABLI 21

NAREDBA DODJELE 22

INICIJALIZACIJA VARIJABLE 23

ULAZ I IZLAZ 23

TIPOVI PODATAKA I IZRAZI 26

ARITMETIKA 33

ARITMETIČKI IZRAZI 34

BIBLIOTEKA GOTOVIH FUNKCIJA 35

PRIPREMA ZA KVIZ 37

BONUS ZADATAK 39

### 4. JEDNOSTAVNA KONTROLA TIJEKA IZVOĐENJA PROGRAMA 41

JEDNOSTAVNI MEHANIZAM GRANANJA 41

LOGIČKI IZRAZI - SAŽETAK 44

PRIPREMA ZA KVIZ 47

### 5. UVOD U PETLJE 48

WHILE PETLJA 48

DO-WHILE PETLJA 50

INKREMENTIRANJE/DEKREMENTIRANJE 51

PRIPREMA ZA KVIZ 53

STIL PROGRAMA 54

PRIPREMA ZA KVIZ 55

BONUS ZADACI 57

### 6. NAPREDNIJI TIJEK KONTROLE IZVOĐENJA PROGRAMA 59

VREDNOVANJE LOGIČKIH IZRAZA 59

VREDNOVANJE IZRAZA TZV. KRATKIM SPOJEM 61

PRIPREMA ZA KVIZ 62

ENUMERACIJSKI TIPOVI 64

PRIPREMA ZA KVIZ 65

### 7. VIŠESTRUKO GRANANJE 65

VIŠESTRUKO GRANANJE UPORABOM IF-ELSE-NAREDBI 67

NAREDBA SWITCH 70

PRIPREMA ZA KVIZ 74

BONUS ZADATAK 77

## **8. VIŠE O PETLJAMA U C++-U 78**

**NAREDBA FOR** 80

**NAREDBA BREAK** 85

PRIPREMA ZA KVIZ 85

## **9. OBLIKOVANJE PETLJI 88**

**ZAVRŠAVANJE IZVOĐENJA PETLJE** 89

**UGNIJEŽĐENE PETLJE** 91

**“DEBUGGING” PETLJI (TRAŽENJE GREŠAKA)** 93

PRIPREMA ZA KVIZ 96

BONUS ZADACI 97

## **10. UVOD U POLJA 98**

**ZAŠTO TREBAMO POLJA?** 98

**INDEKSIRANE VARIJABLE** 98

**UPORABA [ ] SA POLJIMA** 98

**DODJELA VRIJEDNOSTI INDEKSIRANOJ VARIJABLI** 99

**PETLJE I POLJA** 100

**POLJA I MEMORIJA RAČUNALA** 101

**INDEKS POLJA IZVAN DOZVOLJENOG RASPONA** 102

**STRUKTURE** 104

**PRIMJERI PROGRAMA SA POLJIMA** 105

**SEKVENCIJALNO PRETRAŽIVANJE LISTE** 105

**BINARNO PRETRAŽIVANJE LISTE** 107

**SORTIRANJE U VALOVIMA – “BUBBLE SORT”** 108

**ZNAKOVNI NIZ** 111

PRIPREMA ZA KVIZ 114

## **11. PROCEDURALNA APSTRAKCIJA I FUNKCIJE KOJE VRAĆAJU 115**

### **VRIJEDNOST 115**

**OBLIKOVANJE OD VRHA PREMA DNU (“TOP DOWN”)** 115

**FUNKCIJE KOJE DEFINIRA PROGRAMER** 115

PRIPREMA ZA KVIZ 120

**PROCEDURALNA APSTRAKCIJA I FUNKCIJE** 120

PRIPREMA ZA KVIZ 125

**LOKALNE VARIJABLE** 125

**GLOBALNE KONSTANTE** 125

## **12. FUNKCIJE KOJE NEMAJU ODREĐEN TIP: VOID-FUNKCIJE 128**

**DEFINICIJA VOID-FUNKCIJE** 128

PRIPREMA ZA KVIZ 130

**PROSLJEĐIVANJE REFERENCE (REFERENTNI PARAMETRI)** 130

PRIPREMA ZA KVIZ 133

## **13. POLJA U FUNKCIJAMA 134**

**POLJA KAO ARGUMENTI FUNKCIJA** 134

PRIPREMA ZA KVIZ 138

## **14. TESTIRANJE FUNKCIJA 139**

**PREDUVJETI I UVJETI NA REZULTAT IZVRŠAVANJA FUNKCIJE** 139

**STRATEGIJE TESTIRANJA PROGRAMA** 143

**ZAMJENSKI PROGRAMSKI ELEMENT (STUB)** 145

**15. NADJAČAVANJE IMENA FUNKCIJA 147**

**PODRAZUMIJEVANI PARAMETRI FUNKCIJE 151**

**16. TEHNIKE PROGRAMIRANJA: REKURZIJA 152**

**REKURZIVNI ALGORITMI 152**

**PREDNOSTI I NEDOSTACI REKURZIJE U ODNOSU NA ITERACIJU 154**

**17. TEHNIKE PROGRAMIRANJA: DINAMIČKO PROGRAMIRANJE 154**

**TEMELJ REKURZIJE: «PODIJELI I SAVLADAJ» 155**

**PRIMJER: REKURZIVNA IZVEDBA BINARNOG PRETRAŽIVANJA 157**

**DINAMIČKO PROGRAMIRANJE 158**

**PRIMJER: REKURZIVNA IZVEDBA IZRAČUNAVANJA FIBONACCIJEVIH BROJEVA 158**

**PRIMJER: IZRAČUNAVANJE FIBONACCIJEVIH BROJEVA (DINAMIČKO PROGRAMIRANJE) 160**

**PRIMJER: «KNAPSACK» PROBLEM (PROBLEM NAPRTNJAČE) 161**

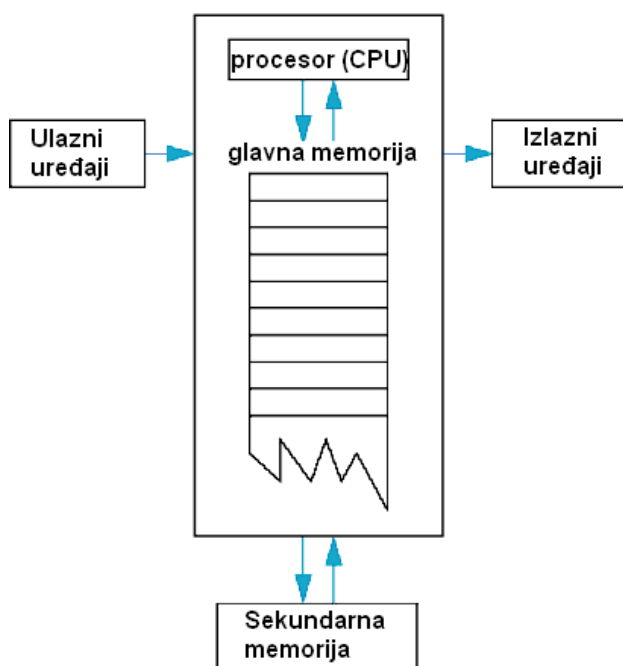
**LITERATURA: 164**

## 1. UVOD U PROGRAMIRANJE

### RAČUNALNI SUSTAVI

Skup instrukcija koje računoalo treba izvršiti zovemo program. Skup programa koje koristi računoalo je **softver** (programska podrška) tog računoala. Stvarni fizički stroj koji predstavlja računoalo zovemo **hardver**.

Glavni dijelovi računoala



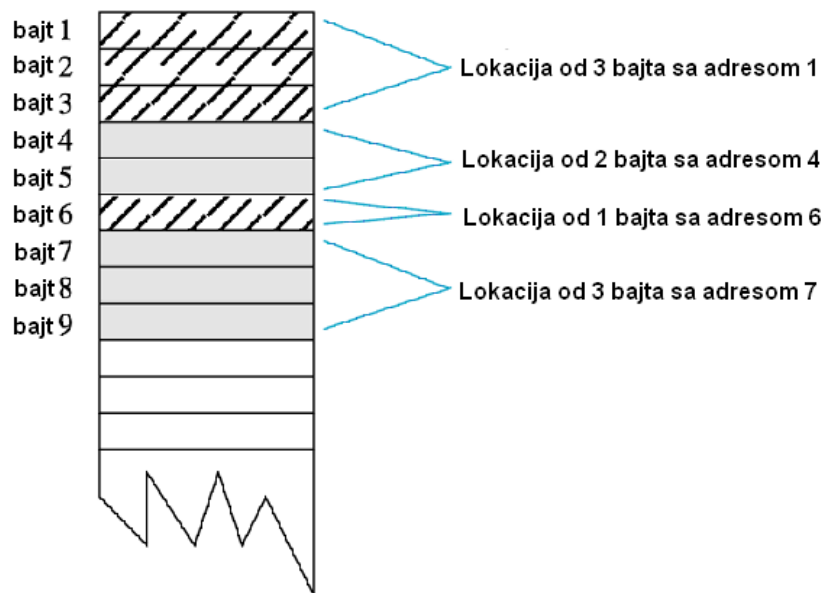
Slika 1: Dijelovi računoala

Računoalo se sastoji od pet glavnih dijelova: ulazni uređaji (npr. tipkovnica i miš), izlazni uređaji (monitor, pisač), procesor (CPU), glavna (radna) memorija i sekundarna memorija.

Da bi se mogli pohraniti ulazni podaci kao i međurezultati i rezultati računanja, računoalo raspolaže s memorijom. Program koji se izvršava je pohranjen u istoj memoriji. Računoalo ima dva oblika memorije: glavnu i sekundarnu memoriju. Program koji se trenutno izvršava je pohranjen u **glavnoj memoriji** – najvažnijoj memoriji računoala. Glavna memorija se sastoji od duge liste numeriranih lokacija – **memorijskih lokacija**. Broj memorijskih lokacija je različit od računoala do računoala: od nekoliko tisuća do nekoliko miliona ili čak  $2^{64}$  lokacija (2011. godina). Svaka memorijska lokacija sadrži niz nula i jedinica. Sadržaj lokacije je promjenjiv. Memorijska lokacija u većini računoala sadrži 8 bita (ili višekratnik od 8 bita). 8-bitni dio memorije čini **byte**. Zato numeriranu memorijsku lokaciju zovemo i bajt. Zbog toga možemo glavnu memoriju računoala zamisliti kao dugu listu numeriranih memorijskih lokacija. Broj koji predstavlja bajt zove

se adresa. Neki podatak kao što je broj ili slovo može se pohraniti u jednom od tih bajtova, a adresa tog bajta se kasnije koristi da bi se pronašao podatak kada je potrebno. Ako računalo treba raditi sa npr. velikim brojevima ili nekim drugim podacima koji ne stanu u jedan bajt, koristi se nekoliko susjednih bajtova za pohranu podatka. Korišteni dio memorije još uvijek se zove memorijska lokacija. Adresu ove velike memorijske lokacije predstavlja adresa njezinog prvog bajta. Praktično je dakle razmišljati o glavnoj memoriji kao o dugoj listi lokacija različitih veličina. Veličina svake memorijske lokacije se izražava u bajtima. Veličina memorijskih lokacija na slici mijenja se ovisno o programu koji se izvodi.

#### Memorijske lokacije



Slika 2: Memorijske lokacije

Glavna memorija se koristi samo kada se program izvršava. **Sekundarna memorija** (pomoćna, vanjska) se koristi za trajan zapis informacije. Informacije se na sekundarnoj memoriji čuvaju u **datotekama** različitih veličina. Npr. program je pohranjen u datoteci u sekundarnoj memoriji i kopira se u glavnu memoriju kada se program pokrene. U datoteku možemo pohraniti pismo, listu s podacima, program ili neke druge informacije. Različite vrste sekundarne memorije: tvrdi disk, disketa, CD, DVD, flash memorija.

Glavna memorija se zove i **RAM** (random access memory – **memorija slučajnog pristupa**), jer računalo može trenutno direktno pristupiti podatku na bilo kojoj lokaciji. Sekundarna memorija obično zahtijeva sekvencijalni pristup što znači da računalo mora pregledati određeni broj lokacija (obično veliki) dok ne pristupi traženom podatku.

"Mozak" računala je **procesor** (CPU – central processing unit). Procesor slijedi instrukcije programa i izvodi izračune specificirane programom (koje je predvidio programer).

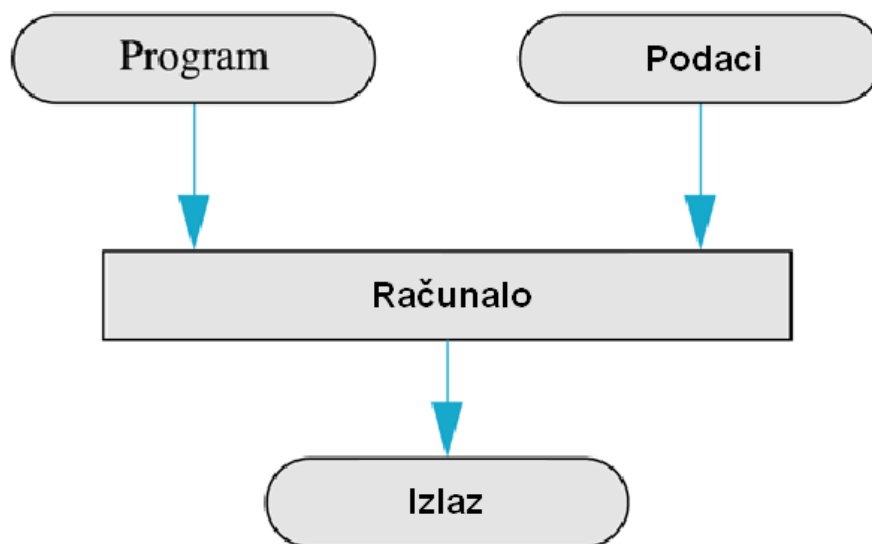
## SOFTVER

Sa računalom komuniciramo posredstvom operacijskog sustava. **Operacijski sustav** dodjeljuje resurse računala različitim zadacima koje računo mora izvršiti. Operacijski sustav je zapravo program, a ima ulogu našeg glavnog pomoćnika koji nadgleda rad svih programa i prenosi im naše zahtjeve. Ako želimo pokrenuti neki program, kažemo operacijskom sustavu ime datoteke koja ga sadrži i operacijski sustav pokrene program. Ako želimo uređivati neku datoteku, kažemo operacijskom sustavu ime datoteke i on pokrene editor za rad na datoteci. Za većinu korisnika operacijski sustav predstavlja samo računo. Većina korisnika nije vidjela računo bez operacijskog sustava. Najpoznatiji operacijski sustavi su UNIX, DOS, Linux, Windows, Mac OS i VMS.

**Program** je skup instrukcija koje računo treba slijediti.

### Pojednostavljeni prikaz izvršavanja programa

---



Slika 3: Prikaz izvršavanja programa

Ulaz računala čine dvije osnovne komponente: program i podaci. Računo slijedi instrukcije u programu i na neki način izvodi proces. Podaci su ono što smatramo ulazom programa. Npr. ako program zbraja dva broja, onda su ta dva broja podaci. Drugim riječima, podaci su ulaz programa, a podaci zajedno sa programom čine ulaz računala (putem operacijskog sustava). Kada računalu zadamo da izvrši program i osiguramo podatke za program, kažemo da smo **pokrenuli program** za te podatke, a za računo kažemo da **izvršava program** na podacima. Pojam podatak općenito označava svaku informaciju koja je računalu na raspolaganju.



## JEZICI VISOKE RAZINE

Za pisanje programa koriste se mnogi jezici. Mi ćemo programirati u programskom jeziku C++. C++ je jezik visoke razine kao i većina programskih jezika za koje ste čuli (C, Java, Pascal, Visual Basic, FORTRAN, COBOL, Scheme, Lisp, Ada, PROLOG, Haskell, Perl, Python, PHP). Jezici visoke razine sličje prirodnim jezicima u mnogo čemu. Oblikovani su tako da se čovjeku programeru olakša pisanje i čitanje programa. Jezici visoke razine kao što je C++ raspolažu sa instrukcijama koje su mnogo složenije od jednostavnih instrukcija koje može izvršiti procesor računala (CPU).

Jezik koji računalo može razumjeti zovemo jezik niske razine. U pojedinim detaljima se jezici niske razine razlikuju na pojedinim tipovima računala. Tipična instrukcija niske razine izgleda ovako:

ADD X Y Z

(Dodaj broj na memorijskoj lokaciji s imenom X broju na memorijskoj lokaciji s imenom Y i smjesti rezultat u memorijsku lokaciju Z)

Ova jednostavna instrukcija napisana je u assembleru. Iako je assembler gotovo isti kao jezik koji računalo razumije, potrebno je prevesti riječi u nizove jedinica i nula. Npr. ADD bi moglo biti 0110, X 1001, Y 1010, a Z 1011. Tako instrukcija koju računalo razumije ima oblik:

0110 1001 1010 1011

Instrukcije assemblera i njihovi prijevodi razlikuju se od stroja do stroja.

Za programe pisane pomoću 0 i 1 kažemo da su napisani u **strojnom jeziku**. To je verzija programa koju računalo može čitati i izvršavati. Assembler i strojni jezik su skoro isti jezici i za nas razlika između njih nije bitna. Važna je razlika između strojnog jezika i jezika visoke razine kao što je C++: Svaki program napisan na jeziku visoke razine mora se prevesti u strojni jezik prije da bi ga računalo moglo razumjeti i izvršiti.

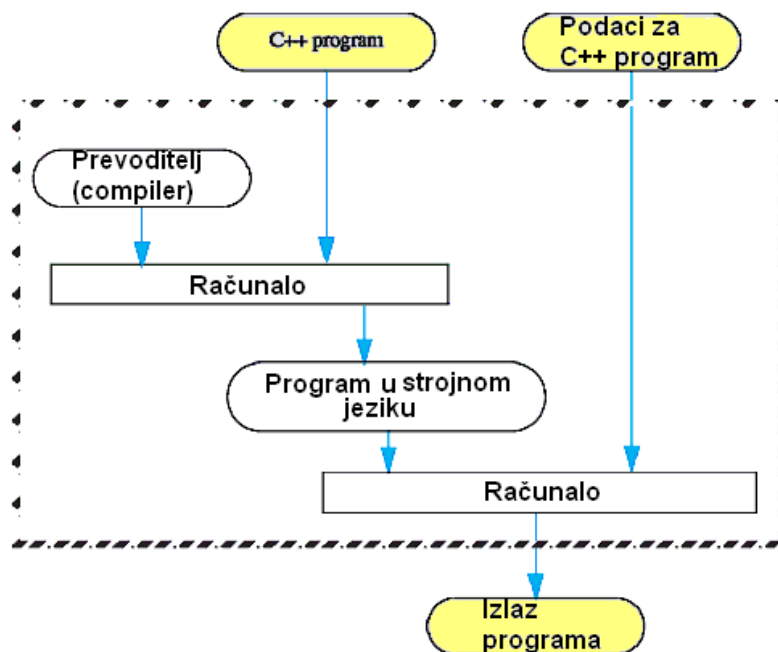
## PREVODITELJI (KOMPJLERI)

Program koji prevodi jezik visoke razine kao što je C++ u strojni jezik zovemo **prevoditelj** (kompajler). Prevoditelj je posebna vrsta programa koji na ulazu ima jedan program, a na izlazu drugi program. Ulazni program se obično zove izvorni kod ili izvorni program, a prevedenu verziju koju kreira prevoditelj zovemo objektni (ciljni) program ili objektni (ciljni) kod. Riječ **kôd** označava program ili dio programa i tu riječ često koristimo kada govorimo o objektnom programu (objektni kod).

Prevoditelj izvorni kod tretira kao dugi niz znakova, a na izlazu daje drugi dugi niz znakova – ekvivalent izvornom programu u strojnom jeziku. Kada pokrenemo objektni program u strojnom jeziku, kao izlaz dobivamo ono što smatramo izlazom C++ izvornog programa.

## Prevođenje (compiling) i izvršavanje (running) C++ programa (osnovni prikaz)

---



Slika 4: Prevođenje i izvršavanje programa

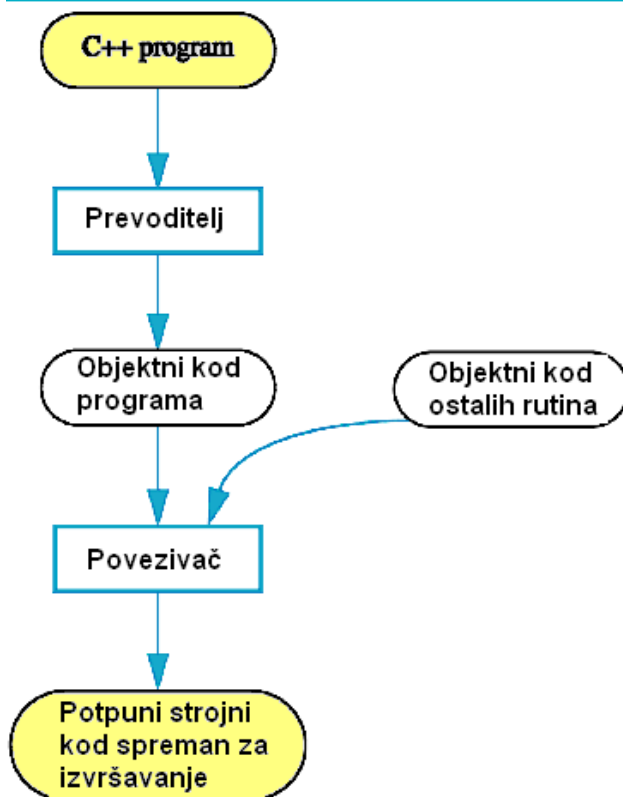
U stvarnosti se isto računalo koristi dva puta (prevođenje, izvršavanje).

**Prevoditelj (kompajler)** - program koji prevodi program napisan na jeziku visoke razine, kao što je C++ u program u strojnom jeziku koji računalo direktno razumije i izvršava.

Potpuni proces prevođenja i izvršavanja C++ programa je nešto složeniji. Svaki C++ program koristi neke operacije (npr. ulazno i izlazne procedure) koje su već isprogramirane. Te su procedure već i prevedene i njihov objektni kod se samo kombinira sa objektnim kodom našeg programa da bi dobili potpuni program u strojnom jeziku koji se može izvršiti na računalu. Poseban program koji zovemo **povezivač** (linker) kombinira objektni kod ovih dijelova sa objektnim kodom koji prevoditelj daje na temelju C++ programa.

## Priprema C++ programa za izvršavanje

---



Slika 5: Priprema programa za izvršavanje

Postupak kombiniranja objektnog koda zovemo **povezivanje** (linkanje), a izvodi ga poveziavač.

Postupak povezivanja se obično izvodi automatski pokretanjem prevoditelja.

Kviz:

1. Nabroji 5 glavnih dijelova računala.
2. Koji su ulazni podaci programa za zbrajanje dvaju brojeva?
3. Koji bi mogli biti ulazni podaci programa za dodjeljivanje ocjena studentima?
4. Koja je razlika programa u strojnom jeziku i jeziku visoke razine?
5. Koja je uloga prevoditelja?
6. Što je izvorni program? Što je ciljni program?
7. Što je operacijski sustav?
8. Koju ulogu ima operacijski sustav?
9. Koji operacijski sustav koristi tvoje računalo?
10. Što je povezivanje?
11. Da li prevoditelj za jezik C++ koji koristiš automatski izvodi povezivanje?

## 2. PROGRAMIRANJE I RIJEŠAVANJE PROBLEMA

### ALGORITMI

Kada počinjete učiti vaš prvi programski jezik dobivate dojam da je najteži dio rješavanja problema na računalu prevođenje vaših ideja u specifični jezik kojim komunicirate s računalom. To nije ni izdaleka tako kako se čini. Najteži dio rješavanja problema na računalu je otkrivanje postupka rješavanja problema. Kada ste kreirali postupak rješavanja problema, rutinski je postupak prevesti postupak rješavanja u traženi jezik, C++ ili neki drugi jezik. Zato je korisno privremeno zanemariti programski jezik i koncentrirati se na formulaciju koraka rješenja i zapisati ih na prirodnom jeziku, kao da instrukcije pišemo za čovjeka, a ne za računalo. Niz instrukcija izraženih na ovaj način zovemo **algoritam**.

**Algoritam je niz preciznih instrukcija koji vodi do rješenja.**

Instrukcije možemo izraziti u programskom jeziku ili na prirodnom jeziku. Algoritme ćemo izražavati na hrvatskom jeziku, engleskom jeziku i u programskom jeziku C++. Računalni program je algoritam izražen u jeziku koji računalo razumije. Pojam algoritma je puno širi od pojma programa.

Pogledajmo primjer tipičnog algoritma:

Algoritam koji određuje koliko puta se ime pojavljuje u listi imena:

1. Učitaj listu imena.
2. Učitaj ime koje treba pronaći u listi.
3. Postavi brojač na nulu.
4. Napravi slijedeće za svako ime u listi:  
    Usporedi ime u listi sa imenom koje se traži i ako je ime u listi jednako  
    povećaj brojač za 1.
5. Kao rezultat objavi vrijednost brojača.

Ovaj algoritam se često koristi kada pretražujemo listu prema određenom ključu. Npr. u listi pobjedničkih sportskih momčadi tražimo određenu momčad i provjeravamo koliko puta se pojavila u listi – broj ostvarenih pobjeda.

Instrukcije od 1 do 5 se izvršavaju redoslijedom kojim su navedene (ako nije drugačije naznačeno). Najzanimljiviji algoritmi ipak specificiraju neku promjenu poretka izvršavanja – obično ponavljanje neke instrukcije kao što je instrukcija 4 u primjeru.

Riječ algoritam ima dugu povijest. Nastala je od imena perzijskog matematičara i astronoma po imenu al-Khowarizmi. Danas se riječ algoritam može primijeniti na široki raspon instrukcija za manipuliranje sa simboličkim i numeričkim podacima. Da li skup instrukcija predstavlja algoritam ovisi o prirodi instrukcija, a ne o podacima s kojima one

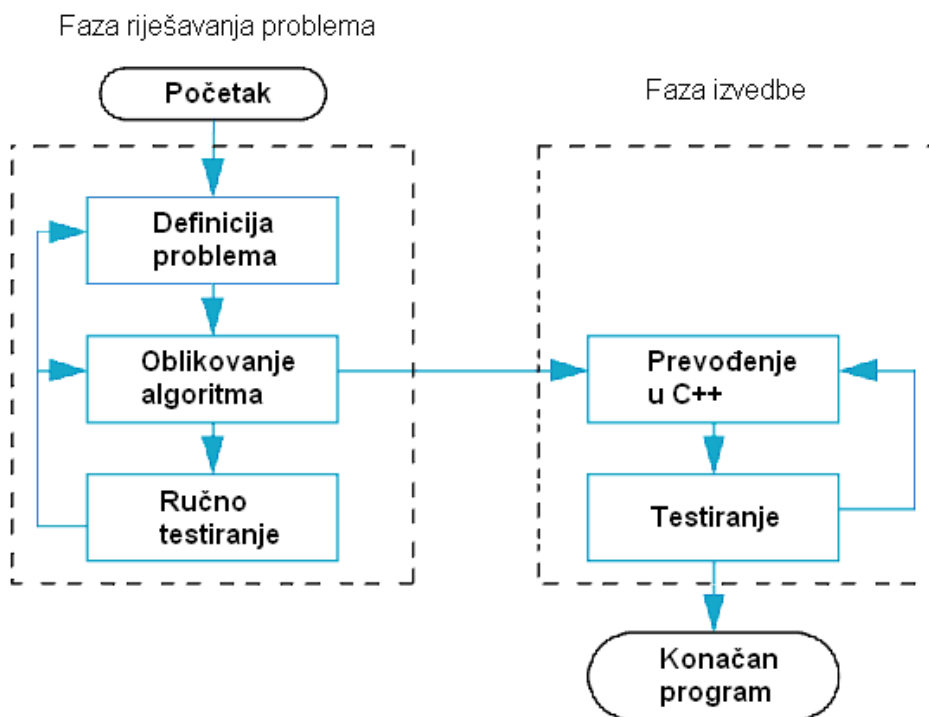
manipuliraju. Da bi ga mogli smatrati algoritmom skup instrukcija mora potpuno i jednoznačno specificirati korake postupka i redoslijed koraka.

## OBLIKOVANJE PROGRAMA

Oblikovanje programa je često težak zadatak. Nema potpunog skupa pravila o pisanju programa. Pisanje programa je kreativan proces. Ipak možemo slijediti određeni plan rada:

### Postupak oblikovanja programa

---



Slika 6: Postupak oblikovanja programa

Proces oblikovanja možemo podijeliti u dvije faze:

1. Faza rješavanja problema
2. Faza izvedbe

Rezultat faze rješavanja problema je algoritam za rješavanje problema, izražen na prirodnom jeziku.

Da bi proizveli program u programskom jeziku C++, algoritam prevodimo u programski jezik. Izrada konačnog programa na temelju algoritma je faza izvedbe.

Najprije moramo provjeriti da je zadatak koji program mora izvršiti potpuno i precizno specificiran. Ako niste sigurni što je izlaz vašeg programa, mogao bi vas rezultat

programa iznenaditi. Provjerite kakvi su ulazni podaci vašeg programa i kakve odgovarajuće izlazne podatke trebate dobiti za njih. Programu treba dostaviti sve potrebne podatke da bi mogao izvršiti zadani postupak. Ulazne i odgovarajuće izlazne podatke obično pripremamo ručno i zovemo ih test podacima. Test podaci trebaju biti potpuni da bi mogli provjeriti rad programa za sve moguće slučajeve ulaznih podataka.

Mnogi programeri početnici ne razumiju da je potrebno oblikovati algoritam prije pisanja programa u programskom jeziku kao što je C++, pa preskaču fazu rješavanja problema ili samo definiraju problem. To međutim ne štedi vrijeme, već naprotiv vodi do dugotrajnije izrade ispravnog programa. Čak i za jednostavne programe pravilnim pristupom možemo uštedjeti i do ukupno pola dana posla i nekoliko frustrirajućih dana traženja grešaka u programu kojeg slabo razumijemo.

Faza izvedbe nije trivijalan korak. Jednom kada se naviknete na C++ ili neki drugi programski jezik, prijevod algoritma sa prirodnog jezika u programski jezik postaje rutina. Stjecanje rutine najčešće se navodi kao garancija uspješnog programiranja.

Testiranje se dakle provodi u dvije faze. Ručno testiranje algoritma možemo provesti misaonim putem ili uz uporabu olovke i papira (za velike programe). Testiranje C++ programa provodimo prevođenjem i pokretanjem programa na računalu za pripremljene test podatke. Prevoditelj će nas obavijestiti putem poruka o kojem se tipu sintaktičke greške radi. Da bi otkrili druge tipove grešaka moramo dobro pripremiti test podatke.

## **OBJEKTNO-ORIJENTIRANO PROGRAMIRANJE (OOP)**

Program ne mora uvijek biti predstavljen algoritmom za manipulaciju sa podacima. Suvremeno programiranje se temelji na OOP-u. Sa stajališta OOP-a program čine objekti u interakciji. Svaki od objekata posjeduje algoritme koji opisuju njegovo ponašanje u različitim situacijama. Dakle, oblikovanje algoritama je zamijenjeno oblikovanjem objekata i njihovih algoritama.

Glavne značajke OOP-a su:

- klasa – vrsta podatkovnog tipa koja kombinira podatke i algoritme
- enkapsulacija – oblik skrivanja informacija ili apstrakcija
- nasljeđivanje – vezano je uz ponovnu uporabivost koda (reusability), omogućeno je definiranjem klasa na temelju postojećih klasa
- polimorfizam – jedno ime može imati razna značenja ovisno o kontekstu nasljeđivanja

## **ŽIVOTNI CIKLUS SOFTVERA**

Dizajneri velikih softverskih sustava kao što su kompajleri i operacijski sustavi, dijele proces razvoja softvera na šest faza koje se jednim imenom zovu životni ciklus softvera.

1. Analiza i specifikacija zadatka (definicija problema)
2. Oblikovanje softvera (oblikovanje algoritma)
3. Izvedba (kodiranje)
4. Testiranje

5. Održavanje i daljnji razvoj sustava
6. Zastarijevanje

## UVOD U C++

Programski jezik C++ razvio se iz jezika C, a jezik C iz jezika B, koji je nastao iz jezika BCPL.

Shema razvoja programskih jezika:

[http://www.oreilly.com/news/graphics/prog\\_lang\\_poster.pdf](http://www.oreilly.com/news/graphics/prog_lang_poster.pdf)

Programski jezik C razvijen je u Bellovom laboratoriju 1970-tih u svrhu razvoja i održavanja operacijskog sustava UNIX. C je općenamjenski jezik koji se može koristiti za pisanje raznih vrsta programa, ali svoju popularnost veže uz sustav UNIX. Programi koji se razvijaju za pokretanje na UNIX-u pišu se gotovo svi u C-u. C je zbog svoje popularnosti doživio verzije i za druge operacijske sustave.

C je jezik visoke razine sa mnogim karakteristikama jezika niske razine. Ima svoje prednosti i nedostatke. Kao i assembleri, C omogućava manipulaciju memorijom, dok mu značajke jezika visoke razine omogućavaju pisanje programa koji se lakše pišu i čitaju u usporedbi sa assemblerom. Izvrstan je izbor za pisanje sistemskih programa, dok za ostale programe i nije najbolji izbor jer nema ugrađene automatske provjere kao neki drugi jezici visoke razine.

Da bi se prevladali ovi nedostaci C-a Bjarne Stroustrup iz Bellovog laboratorija razvio je C++ u ranim 1980-ima. Veći dio C-a je podskup C++-a, tako da je i većina programa napisanih u C-u ujedno napisana i u C++-u (obratno ne vrijedi). C++ omogućava objektno orijentirano programiranje.

Primjer C++ programa:

```
#include <iostream>
using namespace std;
// MJENJACNICA
int main()
{
    float iznos_kune, tecaj_eura, iznos_euri;
    cout << "Stisni ENTER nakon unosa.\n";
    cout << "Unesi iznos u kunama:\n";
    cin >> iznos_kune;
    cout << "Unesi tečaj_eura:\n";
    cin >> tecaj_eura;
    iznos_euri = iznos_kune / tecaj_eura;
    cout << "Za ";
    cout << iznos_kuna;
    cout << " kuna\n";
    cout << "uz tečaj eura ";
    cout << tecaj_eura;
```

```
cout << ",\n";  
cout << "dobiti ćete ";  
cout << iznos_euri;  
cout << " eura \n";  
return 0;  
}
```

Primjer dijaloga (korisnički ekran)

Stisni ENTER nakon unosa.

Unesi iznos u kunama:

100

Unesi tečaj eura:

**7.23**

Gornji primjer predstavlja jednostavan program u C++-u i prikaz na ekranu za vrijeme interakcije korisnika sa programom prilikom pokretanja programa. Dakle, osoba koja pokreće program zove se korisnik. Podebljani tekst unio je korisnik, za razliku od teksta koji ispisuje program. Na stvarnom ekranu oba teksta izgledaju jednako. Osoba koja piše program zove se programer. Korisnik i programer mogu i ne moraju biti ista osoba. U slučaju profesionalne izrade programa to su obično različite osobe.

Početak i kraj našeg jednostavnog programa sadrži neke detalje kojima ćemo se baviti kasnije. Program počinje slijedećim linijama:

```
#include <iostream>  
using namespace std;  
// MJENJACNICA  
int main( )  
{
```

To je samo kompliciran način za označavanje početka programa koji ćemo objasniti kasnije.

Program završava sa slijedećim linijama:

```
return 0;  
}
```

Za ovaj jednostavan program to znači da program tu završava.

Linije između početka i kraja programa čine srce programa. Kratko ćemo ih opisati.

```
int iznos_kune, tecaj_eura, iznos_euri;
```



Navedena linija programa opisuje **deklaraciju varijabli**.

Deklaracija varijabli kaže računalu da će iznos\_kune, tecaj\_eura i iznos\_euri biti korišteni kao imena triju varijabli. Varijable koristimo da bi dali ime brojevima, odnosno memorijskim lokacijama na koje će se brojevi spremiti. Riječ int kaže računalu da se radi o cijelim brojevima (npr. 1, 6, -5, 339, -465 i sl.). Preostale linije programa su instrukcije koje kažu računalu da izvrši nešto. Ove instrukcije zovemo naredbe ili izvršne naredbe. U našem primjeru u svakoj liniji se nalazi jedna naredba.

cin i cout su ulazna i izlazna naredbe. Strelice << i >> označavaju smjer premještanja podataka (operator "insert" i "extract").

U naredbi programa

```
cout << "Stisni ENTER nakon unosa.\n";
```

- ☐ cout (si-aut) koristimo za izlaz na monitor
- ☐ "<<" šalje "Stisni...unosa.\n" na monitor
- ☐ cout možemo shvatiti kao ime monitora
  - ☐ "<<" pokazuje odredište podataka
- ☐ '\n' uzrokuje prijelaz u novi red na monitoru

Naredba programa

```
cin >> iznos_kune;
```

- ☐ cin (si-in) koristi se za ulaz sa tipkovnice
- ☐ ">>" uzima podatke sa tipkovnice
- ☐ cin možemo shvatiti kao ime tipkovnice
  - ☐ ">>" pokazuje od tipkovnice prema varijabli u koju se podatak sprema

Naredba programa

```
iznos_euri = iznos_kune / tecaj_eura;
```

- Izvodi izračun
- '/' se koristi za dijeljenje
- '=' uzrokuje da **iznos\_kune** dobije novu vrijednost temeljenu na izračunu na desnoj strani znaka '='

Naredba programa

```
cout << iznos_euri;
```

- šalje vrijednost varijable **iznos\_euri** na monitor.

Izgled programa

Prevoditelj prihvaća skoro svaki uzorak prijelaza u novi red i uvlačenja redaka.

Programeri oblikuju programe tako da ih je lako čitati.

- ☐ Smjesti otvorenu vitičastu zagradu '{' i zatvorenu vitičastu zagradu '}' same u liniju.
- ☐ Uvuci naredbe
- ☐ Upotrijebi samo jednu naredbu po liniji

Varijable se deklariraju prije nego što se koriste.

- Obično se varijable deklariraju na početku programa
- Naredbe (a ne linije) završavaju sa ';' (točka-zarez)

Direktiva include

```
#include <iostream>
```

- ☐ Kaže prevoditelju gdje može naći informaciju o elementima korištenim u programu
- ☐ iostream je biblioteka koja sadrži definicije za cin i cout

**using namespace std;**

- Kaže prevoditelju da koristi imena u iostream na "standardan" način

Započinjanje glavne funkcije programa:

```
int main()  
{
```

Završavanje glavne funkcije:

```
return 0;  
}
```

- ☐ Glavna funkcija završava sa naredbom return.

Pokretanje programa

- C++ izvorni kod pišemo pomoću uređivača (editora) teksta.
- Prevoditelj vašeg sustava pretvara izvorni kod u objektni kod.
- Povezivač kombinira objektni kod u izvršni program.

Primjer programa:

```
#include <iostream>  
using namespace std;  
int main()  
{
```

```
cout<<"Proba 1 2 3\n";
```

```
    return 0;  
}
```

Primjer dijaloga:

Proba 1 2 3

Zadatak:

- Upiši kod iz gornjeg primjera
- Prevedi kod
- Ispravi greške koje javlja prevoditelj i ponovo prevedi kod
- Pokreni program
- Sada znaš kako se pokreće program!

## TESTIRANJE I ISPRAVLJANJE GREŠAKA (DEBUGGING)

Greška (bug - buba) je pogreška u programu

Postupak ispravljanja grešaka (debugging) uključuje

- ☐ Uklanjanje grešaka iz programa
- ☐ Izraz koji se koristi nakon što je moljac uzrokovao grešku u radu računala Mark I na Harvardu.

Grace Hopper je zapisala u dnevnik: "Prvi stvarni slučaj da smo pronašli bubu."

Vrste grešaka u programu:

- Sintaktičke greške
  - ☐ Povreda gramatičkih pravila jezika
  - ☐ Otkriva ih prevoditelj
    - Poruke o greškama ne pokazuju uvijek pravu poziciju grešaka
  - ☐ Primjer: pogrešno napisana naredba ili izostavljanje ";" iza naredbe
- Greške prilikom izvođenja (Run-time errors)
  - ☐ Otkriva ih sustav za vrijeme izvođenja programa
  - ☐ Primjer: pokušaj dijeljenja s nulom, računanje drugog korijena za neg. broj
- Logičke greške
  - ☐ Greške u algoritmu na kojem se temelji program
  - ☐ Najteže ih je otkriti
  - ☐ Ovaj tip grešaka računalo ne može otkriti
  - ☐ Primjer: U postupku se umjesto zbrajanja pojavljuje oduzimanje – dobivamo rezultat koji nije očekivan prema test podacima

Okruženje za razvoj programa Microsoft Visual C++ ima ugrađene alate za "debugging" što uključuje praćenje mijenjanja vrijednosti varijabli za vrijeme izvođenja programa u posebnom prozoru, mogućnost postupnog izvođenja programa i druge mogućnosti.

### 3. UVOD U C++

#### VARIJABLE I DODJELA

C++ kao i ostali programski jezici visoke razine koristi varijable za imenovanje i pohranu podataka. Objasnit ćemo uporabu varijabli na slijedećem programu:

```
1. #include <iostream>
2. using namespace std;
3. int main( ){
4.     int broj_cokoladica;
5.     double tezina_cokoladice, ukupna_tezina;
6.     cout << "Unesi broj cokoladica u kutiji i tezinu jedne c. u gramima.\n";
7.     cout << "pa pritisni ENTER.\n";
8.     cin >> broj_cokoladica;
9.     cin >> tezina_cokoladice;
10.    ukupna_tezina = tezina_cokoladice * broj_cokoladica;
11.    cout << broj_cokoladica << " cokoladica\n";
12.    cout << tezina_cokoladice << " grama svaka\n";
13.    cout << "Ukupna tezina je " << ukupna_tezina << "grama. \n\n";
14.    cout << "Probaj za drugu vrstu cokoladica.\n";
15.    cout << "Unesi broj cokoladica u kutiji i tezinu jedne c. u gramima.\n";
16.    cout << "pa pritisni ENTER.\n";
17.    cin >> broj_cokoladica;
18.    cin >> tezina_cokoladice;
19.    ukupna_tezina = tezina_cokoladice * broj_cokoladica;
20.    cout << broj_cokoladica << " cokoladica\n";
21.    cout << tezina_cokoladice << " grama svaka\n";
22.    cout << "Ukupna tezina je " << ukupna_tezina << "grama.\n\n";
23.    cout << "Radije pojedite jednu jabuku!.\n";
24.    return 0;
25. }
```

Ograničiti ćemo se na varijable koje sadržavaju numeričke podatke. Svaka takva varijabla uvijek sadrži neku vrijednost, pa makar i slučajnu. Broj koji varijabla sadrži nazivamo **vrijednost varijable**. U gornjem programu broj\_cokoladica, tezina\_cokoladice i ukupna tezina su **varijable**. Kada program pokrenemo (prikaz dijaloga sa korisnikom) broj\_cokoladica dobiva vrijednost 5 naredbom:

```
cin >> broj_cokoladica
```

Kasnije se vrijednost varijable broj\_cokoladica mijenja na 10 nakon što se izvrši slijedeća kopija iste naredbe.

#### Primjer dijaloga s programom (unos je podebljan)

Unesi broj cokoladica u kutiji  
i tezinu jedne cokoladice u gramima.  
Pritisni ENTER.

**5**

**20**

5 cokoladica

20 grama svaka

Ukupna teжина je 100grama.

Probaj za drugu vrstu cokoladica.

Unesi broj cokoladica u kutiji

i tezinu jedne cokoladice u gramima.

Pritisni ENTER.

**10 10**

10 cokoladica

10 grama svaka

Ukupna teжина je 100grama.

Radije pojedite jednu jabuku!

Varijablama su dodijeljene memorijske lokacije. Prevoditelj dodjeljuje memorijsku lokaciju svakoj varijabli programa. Nije nam važno koje adrese je prevoditelj dodijelio varijablama. O memorijskim lokacijama razmišljamo kao da su označene imenima varijabli.

### Imena – identifikatori

Primjećujemo da su imena koja se koriste kao imena varijabli dulja nego što je to uobičajeno u matematici. Da bi program bio čitljiviji koristimo asocijativna imena koja imaju neko značenje. Ime varijable zovemo identifikator. Identifikator mora početi ili sa slovom ili znakom za podcrtavanje, a ostali znakovi moraju biti slova, znamenke ili znak za podcrtavanje. U imenima varijabli razlikuju se mala i velika slova.

Ključne riječi (zovu se i rezervirane riječi) koje koristi jezik C++ imaju unaprijed određenu ulogu, te se moraju koristiti kako je definirano u programskom jeziku i ne mogu se koristiti kao identifikatori.

### DEKLARACIJA VARIJABLI

Prije uporabe varijable se moraju **deklarirati**. Deklaracija kaže prevoditelju tip podatka koji se sprema u varijablu.

Primjeri:    **int** broj\_cokoladica;  
              **double** teжина\_cokoladice, ukupna\_tezina;

**int** je kratica za integer (cjelobrojni tip), a varijabla toga tipa može pohraniti 3, 102, 3211, -456, itd.

Kažemo da je varijabla broj\_cokoladica **tipa** integer (cjelobrojna). Tip **double** predstavlja realne brojeve. Može pohraniti 1.34, 4.0, -345.6, itd. Varijable teжина\_cokoladice i ukupna\_tezina su **tipa** double.

Svaka varijabla programa mora se deklarirati prije uporabe. Deklaracija varijabli može se nalaziti na dva mjesta.

Neposredno prije uporabe:

```
int main()
{
    ...
    int zbroj;
    zbroj = pribr1 + pribr 2;
    ...
    return 0;
}
```

Na početku funkcije:

```
int main()
{
    int zbroj;
    ...
    zbroj = pribr1 + pribr2;
    ...
    return 0;
}
```

**Primjeri deklaracije:**

**double prosjek, m\_bodovi, ukupni\_bodovi;**

**double udaljenost\_mjeseca;**

**int dob, broj\_studenata;**

**int auti\_na\_cekaju;**

## NAREDBA DODJELE

Najdirektniji način za promjenu vrijednosti varijable je uporaba **naredbe dodjele**. Npr. naredba u liniji 19. prethodnog programa je naredba dodjele:

```
ukupna_tezina = tezina_cokoladice * broj_cokoladica;
```

Ovom naredbom se kaže računalu da postavi vrijednost varijable `ukupna_tezina` na vrijednost koja se izračunava kao rezultat umnoška vrijednosti varijabli na desnoj strani operatora dodjele. Zvezdica označava operator množenja. Naredba dodjele završava točka-zarezom. Varijabla koja se mijenja uvijek se nalazi s lijeve strane operatora dodjele '='.

Na desnoj strani operatora dodjele može se nalaziti:

- ☐ Konstanta: `dob = 21;`
- ☐ Varijabla: `moja_cijena = tvoja_cijena;`

❑ Izraz:            opseg\_kruznice = promjer \* 3.14159;

Broj kao što je 21 u gornjoj naredbi dodjele zovemo **konstanta** jer se za razliku od varijable njezina vrijednost ne može mijenjati.

Promotrimo dodjelu:

```
broj_cokoladica = broj_cokoladica + 3;
```

Operator dodjele ima značenje različito od jednakosti sa stajališta algebre. U naredbi dodjele prvo se izračunava izraz desno od znaka jednakosti. Nakon toga se rezultat izračuna sprema u varijablu čije je ime navedeno na lijevoj strani znaka jednakosti. Na taj način je moguće da se ista varijabla pojavi na obje strane operatora dodjele kao u gornjem izrazu.

## INICIJALIZACIJA VARIJABLE

Deklariranjem varijabla ne dobiva vrijednost. **Inicijalizacijom** varijable ona dobiva svoju prvu vrijednost. Varijable se inicijaliziraju u naredbama dodjele:

```
double z;        // deklaracija varijable  
z = 26.3;       // inicijaliziraj varijablu
```

**Deklaracija i inicijalizacija se mogu kombinirati na dva načina:**

**1. način: double z = 26.3, area = 0.0 , volumen;**

**2. način: double z(26.3), area(0.0), volumen;**

Prilikom odabira imena varijabli savjetuje se da se odabiru imena sa značenjem tako da samo ime opisuje značenje varijable. Usporedimo dodjele:

```
x = y * z;  
i  
put = brzina * vrijeme;
```

Objе naredbe dodjele obavljaju isti postupak, ali se druga naredba lakše razumije.

## ULAZ I IZLAZ

**Tok podataka** je niz podataka obično u obliku znakova ili brojki. Ulazni tok predstavljaju podaci koje program koristi. Izvor je obično tipkovnica ili datoteka. Izlazni tok je izlaz programa. Odredište je obično monitor ili datoteka.

### Izlaz uporabom cout

**cout** je **izlazni tok** koji šalje podatke na monitor. **Operator umetanja** "<<" umeće podatke u cout.

Primjer:

```
cout << broj_cokoladica << " cokoladica\n";
```

Navedena linija šalje dva ispisa na ekran

- ❑ Vrijednost *broj\_cokoladica*
- ❑ Niz znakova u navodnicima " cokoladica\n"
  - Uočimo razmak prije c u riječi cokoladica
  - '\n' uzrokuje prijelaz u novi red nakon riječi cokoladica
  - Za svaki pojedini ispis koristi se novi operator umetanja "<<"

Primjeri uporabe cout:

Slijedeća dva ispisa daju isti rezultat kao i prethodni primjer.

```
cout << broj_cokoladica ;
cout << " cokoladica\n";
```

U cout naredbi može se izvoditi aritmetika:

```
cout << "Ukupna cijena je " << (cijena + porez) << " kn";
```

Nizovi su zatvoreni dvostrukim navodnicima ("Cijena"). Ne koriste se jednostruki navodnici (') za označavanje nizova.

Prazno mjesto (razmak) se može umetnuti i pomoću

```
cout << " " ;
```

Npr:

```
cout << prvi_broj << " " << drugi_broj;
```

## Direktiva include

Direktive include dodaju datoteke iz biblioteke našem programu. Da bi definicije naredbi cin i cout bile dostupne programu navodimo:

```
#include <iostream>
```

Uporaba direktiva uključuje skup definiranih imena. Da bi imena cin i cout bila dostupna programu pišemo:

```
using namespace std;
```

Ova direktiva using kaže da program koristi standardni imenik std. To znači da imena koja koristite imaju značenje definirano u imeniku std.

## Escape sekvence

Escape sekvence kažu prevoditelju da tretira znakove na poseban način. '\ ' je escape znak.

**Da bi u izlazu došlo do prijelaza u novi red koristimo:**  
 \n – cout << "\n" ili noviju alternativu: cout << endl;



**Ostale escape sekvence:**

<code>\t</code>	-- tab
<code>\\</code>	-- backslash znak
<code>\"</code>	-- dvostruki navodnik

Primjer prijelaza u novi red pri ispisu:

Da bi prešli u novi red možemo `\n` umetnuti u niz znakova kao u sljedećem primjeru:

```
cout<< "Ovo je program koji izracunava\n"  
    << "godisnji prosjek temperature.\n";
```

Novu liniju možemo započeti i sa `endl`. Slijedi ekvivalentan način za zapis gornje naredbe `cout`:

```
cout << "Ovo je program koji izracunava << endl  
    << "godisnji prosjek temperature" << endl;
```

Dobra je navika završiti program sa prijelazom u novi red.

**Formatiranje realnih brojeva**

Realni brojevi (tip `double`) mogu se ispisati na razne načine:

```
double cijena = 78.5;  
cout << "Cijena je " << cijena << " kn" << endl;
```

Izlaz može biti bilo što od sljedećeg:

```
Cijena je 78.5 kn  
Cijena je 78.500000 kn  
Cijena je 7.850000e01 kn
```

Vjerojatno se neće ispisati:

```
Cijena je 78.50 kn
```

`cout` uključuje specifikacije izlaza tipa `double`. Da bi osigurali da izlaz bude u željenom obliku program treba sadržavati određene instrukcije koje kažu računalu kako da ispiše brojeve.

Ako želimo dvije znamenke nakon decimalne točke koristimo sljedeće specifikacije:

Specifikacija zapisa fiksne točke: `setf(ios::fixed)`

Specifikacija: decimalna točka će biti uvijek prikazana: `setf(ios::showpoint)`

Specifikacija: dva decimalna mjesta će uvijek biti prikazana: `precision(2)`

<b>Primjer:</b>	<code>cout.setf(ios::fixed);</code> <code>cout.setf(ios::showpoint);</code> <code>cout.precision(2);</code>
-----------------	---

```
cout << "Cijena je "  
      << cijena << endl;
```

**Rezultat ispisa:**  
**Cijena je 78.50 kn**

## Ulaz uporabom cin

cin je ulazni tok koji dobavlja podatke s tipkovnice

Primjer:

```
cout << "Unesi broj čokoladica u paketu\n";  
cout << " i težinu u gramima za jednu čokoladicu.\n";  
cin >> broj_cokoladica;  
cin >> tezina_cokoladice;
```

Ovaj kôd traži od korisnika unos podataka i prihvaća dva podatka od cin

- ☐ Prva pročitana vrijednost se sprema u broj\_cokoladica
- ☐ Druga pročitana vrijednost se sprema u tezina\_cokoladice
- ☐ Podaci se pri unosu odvajaju razmakom

Pri unosu više podataka odvajamo ih razmakom. Podaci se učitavaju tek kada stisnemo tipku enter. Prije pritiska na enter možemo po želji ispravljati podatke.

**Primjer:**

```
cin >> v1 >> v2 >> v3;
```

**Traži se unos tri podatka odvojena razmakom**  
**Korisnik može unijeti**  
**34 45 12 <tipka enter>**

## Oblikovanje ulaza i izlaza

Upozorimo korisnika kakav ulaz se traži. Naredba cout ispisuje uputu:

```
cout << "Unesi svoju dob: ";  
cin >> dob;
```

Ispis podataka sa prikazom učitano (eho). Dajemo korisniku priliku da provjeri što je unio:

```
cout << " Unijeli ste: " << dob << endl;
```

## TIPOVI PODATAKA I IZRAZI

- 2 i 2.0 nisu isti brojevi. Cijeli broj kao npr. 2 je tipa *int*. Broj 2.0 je tipa *double* jer ima decimalni dio (čak iako je decimalni dio jednak 0). Matematika koja se koristi u računalnom programiranju se razlikuje u neki segmentima od onoga što ste naučili na satovima matematike. Brojevi tipa *int* spremaju se kao točne vrijednosti i ponašaju se kao što i očekujemo. Brojevi tipa *double* mogu se spremati kao približne vrijednosti

zbog ograničenja na broj značajnih znamenki koje mogu biti predstavljene. Računalo pohranjuje brojeve tipa double kao aproksimacije. Preciznost s kojom su spremljene vrijednosti tipa double razlikuju se od računala do računala. Obično točnost double vrijednosti iznosi 14 i više znamenki što je za većinu primjena dovoljno. Ipak u nekim osjetljivim situacijama ta mala nepreciznost može stvarati probleme, pa je bolji odabir tip int u slučaju da radimo sa cijelim brojevima.

## Pisanje cjelobrojnih i realnih konstanti

Numeričke konstante tipa double i tipa int pišu se različito. Konstanta tipa int ne sadrži decimalnu točku. Konstanta tipa double može se pisati u oba oblika – s točkom i decimalnim dijelom ili bez točke i decimalnog dijela (u slučaju da decimalnog dijela nema). U svakom slučaju brojevi se ne pišu sa zarezom!

<b>Primjeri cjelobrojnih konstanti:</b>	<b>34 45 1 89</b>
<b>Primjeri realnih konstanti:</b>	<b>34.1 23.0034 1.0 89.9</b>

Zapis sa pomičnim zarezom (znanstveni zapis) obično se koristi za vrlo velike brojeve i jako male decimalne brojeve:

<b>Primjeri:</b>	<b>3.41e1</b>	<b>znači</b>	<b>34.1</b>
	<b>3.67e17</b>	<b>znači</b>	<b>367000000000000000.0</b>
	<b>5.89e-6</b>	<b>znači</b>	<b>0.00000589</b>

Broj lijevo od e ne mora sadržavati decimalnu točku. Eksponent ne može sadržavati decimalnu točku

Različiti numerički tipovi imaju različite zahtjeve na memoriju. Veća preciznost zahtijeva više bajtova memorije. Vrlo veliki brojevi zahtijevaju više bajtova memorije (1e38). Vrlo mali brojevi zahtijevaju više bajtova memorije (npr 1e-38)

## Neki numerički tipovi (1. dio)

Ime tipa	Veličina upotrebljene memorije	Raspon veličine	Preciznost
short (zove se i <i>short int</i> )	2 bytes	–32,767 to 32,767	(nije primjenjivo)
int	4 bytes	–2,147,483,647 to 2,147,483,647	(nije primjenjivo)

## Cjelobrojni tipovi

Imena long int i int predstavljaju različita imena za isti tip čije vrijednosti zauzimaju 4 bajta memorijskog prostora.

#### Ekvivalentni načini za deklariranje vrlo velikih cijelih brojeva

```
long veliki_r;  
long int veliki_r;
```

Imena `short int` i `int` predstavljaju različita imena za isti tip čije vrijednosti zauzimaju 2 bajta memorijskog prostora.

#### Ekvivalentni načini za deklariranje malih brojeva

```
short mali_t;  
short int mali_t;
```

### Tipovi s pomičnim zarezom

Tipovi za brojeve sa decimalnom točkom (npr. `double`) zovu se **tipovi s pomičnim zarezom**. Razlog tome je što računalo pohranjuje broj napisan na uobičajen način (npr. 546.964) da bi se najprije obavila konverzija tipa u oblik sličan e zapisu (5.46964). Prilikom ove konverzije decimalna točka se pomiče ("plovi") na novu poziciju. Ako nam je potrebna veća preciznost za prikaz realnog broja koristimo tip `long double`: `long double` (često 10 bajta)

#### Deklariranje brojeva s pomičnim zarezom sa do 19 značajnih znamenki

```
long double veliki_broj;
```

Ako nam je dovoljna manja preciznost za prikaz realnog broja, a želimo uštedjeti na memorijskom prostoru koristimo tip `long double` (npr ako su vrijednosti tipa `float` elementi matrice sa velikim brojem elemenata tad dolazi stvarno do racionalnog korištenja memorijskog prostora):  
`float` (često 4 bajta)

#### Deklariranje brojeva s pomičnim zarezom sa do 7 značajnih znamenki

```
float nije_jako_veliki_broj;
```

### Znakovni tip

Računala ne obrađuju samo numeričke podatke već obrađuju i **znakovne podatke**. Tip *char* kratica je za character, a vrijednosti ovog tipa su znakovi kao slova, znamenke ili interpunkcijski znakovi – bilo koji znak sa tipkovnice. Primjer deklaracije varijable tipa `char`:

```
char slovo;
```

Vrijednost varijable `slovo` smije biti bilo što od sljedećeg:

```
'a', '9', '+', 'A'
```

Primijetimo da se mala i velika slova smatraju različitim **znakovnim vrijednostima**.

## Znakovne konstante

Primijetimo da znak stavljamo između jednostrukih navodnika za razliku od nizova znakova koje ispisujemo, npr. "\nDanas je lijepo vrijeme.", i stavljamo ih u dvostruke navodnike. Dakle, "a" smatra se nizom znakova, za razliku od zapisa 'a' koji predstavlja znakovnu vrijednost. Kada znakovnu vrijednost unosimo sa tipkovnice **ne unosimo** jednostruke navodnike već samo znak.

**Dakle, znakovne konstante su zatvorene u jednostruke navodnike:**

**char slovo = 'a';**

**Nizovi znakova, čak i kada se sastoje od samo jednog znaka zatvaraju se u dvostruke navodnike**

**"a" je znakovni niz koji sadrži samo jedan znak**

**'a' je vrijednost znakovnog tipa (znak)**

## Čitanje znakovnih podataka

Naredba cin preskače praznine i prijelome redaka tražeći podatke. Slijedeći kod čita dva znaka, ali preskače sve razmake između njih

```
char simbol1,simbol2;
```

```
cin >> simbol1 >> simbol2;
```

Korisnik obično razdvaja podatke razmakom

```
J D
```

Rezultat je isti kao da podaci nisu razdvojeni razmakom

```
JD
```

Primjer:

```
#include <iostream>
using namespace std;
int main( )
{
    char simbol1, simbol2, simbol3;

    cout << "Unesi inicijale bez tocke:\n";
    cin >> simbol1 >> simbol2;

    cout << "Inicijali su:\n";
    cout << simbol1 << simbol2 << endl;

    cout << "Ponovo s razmakom:\n";
    simbol3 = ' ';
    cout << simbol1 << simbol3 << simbol2 << endl;
```

```
cout << "To bi bilo sve.\n";

return 0;
}
```

### **Primjer dijaloga (unos je podebljan)**

**Unesi inicijale bez tocke:**

**J K**

**Inicijali su:**

**JK**

**Ponovo s razmakom:**

**J K**

**To bi bilo sve.**

Funkcije za rad sa znakovnim tipom podataka

Funkcija koja pretvara malo slovo u ASCII kod istog velikog slova: `int toupper(int ch)`

Funkcija koja pretvara veliko slovo u ASCII kod istog malog slova: `int tolower(int ch);`

Slijedeće funkcije vraćaju 0 ako znak `c` nema traženo svojstvo, a vrijednost različitu od 0 ako ima traženo svojstvo (djeluju kao logičke funkcije i mogu se koristiti u IF naredbi za provjeru).

- `int isdigit(int c);`      znamenka (0-9)
- `int isalpha(int c);`      slovo (A-Z ili a-z)
- `int isalnum(int c);`      slovo (A-Z ili a-z) ili znamenka (0-9)
- `int isspace(int c);`      praznina
- `int islower(int c);`      slovo (a-z)
- `int isupper(int c);`      slovo (A-Z)

Koristi se međunarodni standard: 7-bitni ASCII kod (American Standard Code for Information Interchange). U programskom jeziku C++ znakovi su pohranjeni kao brojevi koji predstavljaju ASCII vrijednost (kod) navedenog znaka.

Najvažnije ASCII vrijednosti:

- 0 - znak NULL (`'\0'`)
- 32 - praznina (`' '`)
- 48 - 57 - znamenke `'0'-'9'`
- 65 - 90 - velika slova `'A' do 'Z'`
- 97 - 122 - mala slova `'a' do 'z'` ( $97 - 65 = 32$  - razlika izmedju malog i velikog slova!)

## Tip bool (logički tip)

Bool je novi dodatak jezika C++. Predstavlja kraticu za boolean. Vrijednost tipa boolean može biti ili true ili false.

Deklariramo varijablu tipa bool:

```
bool dovoljno_star;
```

Izrazi logičkog tipa (logički izrazi) izračunavanjem daju jednu od vrijednosti logičkog tipa: true ili false. Logički izrazi se koriste u naredbama grananja i u petljama kojima se nadzire tijek izvođenja programa.

## Klasa *string*

Da bi i sa nizovima znakova mogli obavljati jednostavan unos, i ispis i operaciju povezivanja nizova u C++ je dodana klasa string. Da bi koristili ovu klasu navodimo na početku programa da koristimo biblioteku u kojoj je definirana ova klasa:

```
#include <string>
```

<na početku programa se mora nalaziti i slijedeća direktiva:

```
using namespace std;
```

Kada unosimo vrijednost tipa klase string primjećujemo da ne možemo unijeti vrijednosti koje sadržavaju razmak. Ovaj problem riješit ćemo kasnije.

Primjer:

```
//Program 2-4
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string nadimak, ime_ljubimca;
    string alter_ego_ime;

    cout << "Unesi svoj nadimak i ime svojeg ljubimca.\n";
    cin >> nadimak;
    cin >> ime_ljubimca;

    alter_ego_ime = ime_ljubimca + " " + nadimak;

    cout << "Ime vaseg alter ega je ";
    cout << alter_ego_ime << "." << endl;

    return 0;
}
```

## Kompatibilnost tipova

Opće je pravilo da ne možemo vrijednost jednog tipa spremiti u varijablu drugog tipa. Općenito spremamo vrijednosti u varijable istog tipa.

Slučaj neodgovaranja tipova:

```
int int_varijabla;  
int_varijabla = 2.99;
```

Ako vaš prevoditelj ovo dozvoljava, int\_varijabla će vjerojatno sadržavati vrijednost 2, a ne 2.99

Problem se javlja zbog neodgovaranja tipa. Konstanta 2.99 je tipa double, a varijabla int\_varijabla je tipa int. Na žalost ne reagiraju svi prevoditelji na naredbu dodjele iz navedenog primjera na isti način. Neki će ovu dodjelu prijaviti kao grešku, a čak ako naredbu i prihvate neće broj zaokružiti na 3. Zbog toga ne bi trebali koristiti ovakve dodjele.

#### **int → double (1)**

Varijable tipa double ne bi trebale biti dodjeljivane varijablama tipa int

```
int int_varijabla;  
double double_varijabla;  
double_varijabla = 2.00;  
int_varijabla = double_varijabla;
```

Ako je dozvoljeno, int\_varijabla sadrži 2, a ne 2.00

#### **int → double (2)**

Cjelobrojne vrijednosti se mogu spremati u varijable tipa double.

```
double double_varijabla;  
double_varijabla = 2;
```

double\_varijabla će sadržavati 2.0

#### **char ← int**

Slijedeće akcije su moguće, ali općenito nisu preporučljive:

Moguće je spremiti znakovne vrijednosti u cjelobrojne varijable

```
int vrijednost = 'A';
```

Vrijednost će sadržavati cijeli broj koji predstavlja 'A'

Moguće je spremiti int vrijednosti u znakovne varijable

```
char slovo = 65;
```

#### **bool ← int**

Slijedeće akcije su moguće ali općenito se ne preporučuju:

Vrijednost tipa bool može biti dodjeljena int varijablama.

True je spremljena kao 1



**False je spremljena kao 0**  
**Vrijednosti tipa int mogu biti dodjeljene varijablama tipa bool.**  
**Svaki cijeli broj različit od 0 se sprema kao true.**  
**Nula se sprema kao false.**

## ARITMETIKA

U C++ programima možemo kombinirati varijable i brojeve uporabom aritmetičkih operatora.

**Aritmetika se izvodi uporabom operatora:**

**+ za zbrajanje**  
**- za oduzimanje**  
**\* za množenje**  
**/ za dijeljenje**

- Primjer: spremanje umnoška u varijablu `ukupna_tezina`:

```
ukupna_tezina = tezina_cokoladice * broj_cokoladica;
```

### Rezultat operacije

Aritmetički operatori se mogu koristiti sa svakim numeričkim tipom. Operand je broj ili varijabla na koju djeluje operator. **Rezultat operacije ovisi o tipu operandada.**

**Ako su oba operandada tipa int, rezultat je int.**

**Ako je bar jedan od dva operandada tipa double, rezultat je tipa double.**

### Dijeljenje double tipova

Dijeljenje u kojem je bar jedan **operand tipa double (djelitelj ili djeljenik)** proizvodi očekivane rezultate:

```
double djeljenik, djelitelj, kvocijent;  
djeljenik = 5;  
djelitelj = 3;  
kvocijent = djeljenik / djelitelj;
```

kvocijent = 1.6666...

Rezultat je isti ako je jedan od operandada tipa int.

### Dijeljenje cjelobrojnih tipova

**Ovo je vrlo česti izvor problema u programima (ne samo za programere početnike!). Oprezno koristite operator dijeljenja!**

**int / int daje cjelobrojni rezultat**  
**(vrijedi za varijable i numeričke konstante)**

```
int djeljenik, djelitelj, kvocijent;
djeljenik = 5;
djelitelj = 3;
kvocijent = djeljenik / djelitelj;
```

Vrijednost kvocijenta je 1, a ne 1.666...

Cjelobrojno dijeljenje ne zaokružuje rezultat, decimalni dio se odbacuje!

### Ostatak cjelobrojnog dijeljenja

**Operator % daje ostatak cjelobrojnog dijeljenja**

```
int djeljenik, djelitelj, ostatak;
djeljenik = 5;
djelitelj = 3;
ostatak = djeljenik % djelitelj;
```

Vrijednost ostatka je 2

## ARITMETIČKI IZRAZI

Da bi izrazi bili čitljiviji koristite razmake

☐ Koji se izraz lakše čita?

$x+y*z$  ili  $x + y * z$

**Prioritet izvršavanja operatora je isti kao i u matematici (operacije istog prioriteta se izvršavaju s lijeva na desno!).**

Koristite zagrade da bi promijenili redosljed izvršavanja operacija ako je potrebno.

$x + y * z$  (y se množi prvo sa z)

$(x + y) * z$  (x i y se prvo zbrajaju)

### Aritmetički izrazi

Matematička formula

C++ izraz

$b^2 - 4ac$

$b*b - 4*a*c$

$x(y + z)$

$x*(y + z)$

$\frac{1}{x^2 + x + 3}$

$1/(x*x + x + 3)$

$\frac{a+b}{c-d}$

$(a + b)/(c - d)$

### Operatori obnavljajućeg pridruživanja

Za neke izraze koji se koriste često, C++ raspolaže skraćenim operatorima. Svi aritmetički operatori mogu se koristiti na slijedeći način:

	Primjer dodjele:	Ekvivalentna dodjela:
+=	<b>brojac = brojac + 2;</b>	<b>brojac += 2;</b>
*=	<b>bonus = bonus * 2;</b>	<b>bonus *= 2;</b>
/=	<b>vrijeme = vrijeme / faktor;</b>	<b>vrijeme /= faktor;</b>
%=	<b>ostatak = ostatak % (cnt1+ cnt2);</b>	<b>ostatak %= (cnt1 + cnt2);</b>

## BIBLIOTEKA GOTOVIH FUNKCIJA

C++ posjeduje vlastitu biblioteku gotovih funkcija koje možemo pozivati u okviru izraza. Da bi znali koristiti funkciju potrebno je poznavati ime funkcije, broj, tip i redoslijed parametara funkcije te povratni tip funkcije. Na početku programa potrebno je navesti direktivu prevoditelju da koristi biblioteku funkcija u kojoj je funkcija definirana. O povratnom tipu funkcije ovisi kako ćemo funkciju pozvati.

Primjer: funkcija sqrt

korijen = sqrt(9.0);

vraća, ili računa, kvadratni korijen broja

Broj 9, zovemo argument (ili stvarni parametar). Korijen će dobiti vrijednost 3.0

### Pozivi funkcije

sqrt(9.0) je **poziv funkcije**

Poziv funkcije pokreće funkciju sqrt za stvarni parametar koji može biti varijabla ili izraz.

Poziv funkcije može biti dio izraza

bonus = sqrt(prodaja) / 10;

cout << "Stranica kvadrata s površinom " << p

<< " je "

<< sqrt(p);

### Sintaksa poziva funkcije (pravilo pozivanja funkcije)

Ime\_funkcije (lista\_parametara)

Lista\_parametara je lista sa stvarnim parametrima odvojenim zarezom:

(Parametar\_1, Parametar\_2, ... , Parametar\_zadnji)

Primjer:

stranica = sqrt(povrsina);

cout << "2.5 na 3.0 je "

<< pow(2.5, 3.0); // 2.5<sup>3.0</sup>

U bibliotekama se nalaze već gotove funkcije. Biblioteka mora biti uključena (included) u program da bi mogli koristiti funkcije. Direktiva include kaže prevoditelju koju bibliotečnu zaglavnu datoteku da uključi.

Da bi uključili matematičku biblioteku sa funkcijom sqrt():

```
#include <cmath>
```

Novije standardne biblioteke, kao cmath, traže i *using namespace std;*

### Ostale već definirane funkcije

abs(x), x je int vrijednost, abs(-8); Vraća apsolutnu vrijednost parametra x. Vraćena vrijednost je tipa int. Parametar je tipa x. Nalazi se u biblioteci cstdlib ili cmath.

fabs(x), x je double vrijednost, fabs(-8.0); Vraća apsolutnu vrijednost parametra x. Vraćena vrijednost je tipa double. Parametar je tipa double. Nalazi se u biblioteci cmath.

### Pretvorba tipova (type casting)

Rješavamo slijedeći problem: dijelimo zadani broj\_bombona na zadani broj\_djece gdje su naravno obje vrijednosti cijeli brojevi. Imamo problem sa cjelobrojnim dijeljenjem: int broj\_bombona = 9, broj\_djece = 4;

```
double bombona_po_djetetu;  
bombona_po_djetetu = ukupno_bombona / broj_djece;
```

Problem je u tome što cjelobrojno dijeljenje u ovom slučaju daje cjelobrojni rezultat:

bombona\_po\_djetetu = 2, a ne 2.25! Pretvorba tipova daje vrijednost jednog tipa na temelju vrijednosti drugog tipa :

static\_cast<double>(ukupno\_bombona) daje vrijednost tipa double na temelju cjelobrojne vrijednosti varijable ukupno\_bombona. Kako je jedan operand promijenio tip u double rezultat dijeljenja je isto tipa double.

```
int ukupno_bombona = 9, broj_djece = 4;  
double bombona_po_osobi;  
bombona_po_osobi = static_cast<double>(ukupno_bombona) / broj_djece;
```

Sada je bombona\_po\_djetetu 2.25! Može se pisati i ovako:

```
bombona_po_djetetu = ukupno_bombona / static_cast<double>(broj_djece);
```

Neuspjeli primjer:

```
bombona_po_djetetu = static_cast<double>(ukupno_bombona / broj_djece);
```

### Stari način pretvorbe tipa

C++ je jezik koji se razvija. Ovaj stariji način pretvorbe tipa mogao bi biti napušten u budućim verzijama jezika C++.

```
bombona_po_osobi = double(ukupno_bombona)/broj_djece;
```

#### Some Predefined Functions

Name	Description	Type of Arguments	Type of Value Returned	Example	Value	Library Header
sqrt	square root	<i>double</i>	<i>double</i>	sqrt(4.0)	2.0	cmath
pow	powers	<i>double</i>	<i>double</i>	pow(2.0,3.0)	8.0	cmath
abs	absolute value for <i>int</i>	<i>int</i>	<i>int</i>	abs(-7) abs(7)	7 7	cstdlib
labs	absolute value for <i>long</i>	<i>long</i>	<i>long</i>	labs(-70000) labs(70000)	70000 70000	cstdlib
fabs	absolute value for <i>double</i>	<i>double</i>	<i>double</i>	fabs(-7.5) fabs(7.5)	7.5 7.5	cmath
ceil	ceiling (round up)	<i>double</i>	<i>double</i>	ceil(3.2) ceil(3.9)	4.0 4.0	cmath
floor	floor (round down)	<i>double</i>	<i>double</i>	floor(3.2) floor(3.9)	3.0 3.0	cmath

Slika 7: Neke već definirane matematičke funkcije koje koristimo iz biblioteke

## Priprema za kviz

1. U sljedećem kodu varijabla x dobiva vrijednost 3. (Da/Ne)  
`int x=3`
2. Cijeli broj 0 smatra se logičkom vrijednošću true. (Da/Ne)
3. Dozvoljeno je deklarirati više od jedne varijable u jednoj naredbi za deklaraciju. (Da/Ne)
4. Svaka linija u programu mora imati komentar. (Da/Ne)
5. << se zove operator \_\_\_\_\_.
6. Koji identifikator je dozvoljen?  
A) 3\_com B) 3-com  
C) 3com D) tri\_com

7. Koju vrijednost ima x nakon izvođenja slijedećih naredbi?

```
int x, y, z;  
y = 10;  
z = 3;  
x = y * z + 3;
```

- A) 33                                      B) 30  
C) slučajnu vrijednost                      D) 60

8. Koju vrijednost ima x nakon izvođenja slijedećih naredbi?

```
int x;  
x=x+30;
```

- A) 33                      B) 30  
C) 0                      D) slučajnu vrijednost

9. Kakav izlaz daje slijedeći kod?

```
float vrijednost;  
vrijednost = 33.5;  
cout << vrijednost;
```

- A) slučajna vrijednost                      B) 33  
C) 33.5                                      D) vrijednost

10. Koja od slijedećih linija uzrokuje čitanje vrijednosti s tipkovnice i njezino spremanje u varijablu myFloat?

- A) cin >> "myFloat";    B) cin << myFloat  
C) cin >> myFloat      D) cin >> myFloat <<endl;

11. Drugi način za pisanje vrijednosti 3452211903 je

- A) 3.452211903e-09                      B) 3452211903e09  
C) 3.452211903x09                      D) 3.452211903e09

12. Koja od slijedećih naredbi nije dozvoljena?

- A) char ch='b';                      B) char ch='0'  
C) char ch="cc"                      D) char ch=65

13. Kakvu vrijednost ima x nakon izvršavanja slijedećih naredbi?

```
float x;  
x = 15/4
```

- A) 3.75                      B) 60  
C) 4.0                      D) 3.0

14. Kakvu vrijednost ima x nakon izvršavanja sljedećih naredbi?

```
int x;  
x = 15/4;
```

- A) 15                      B) 3.75                      C) 3                      D) 4

15. Kakvu vrijednost ima x nakon izvršavanja sljedećih naredbi?

```
int x;  
x = 15%4;
```

- A) 15                      B) 3.75                      C) 3                      D) 4

16. Kakvu vrijednost ima x nakon sljedećih naredbi?

```
float x;  
x = 3.0/4.0 + (3+4)/5;
```

- A) 1.75                      B) 5.75                      C) 2.15                      D) 1.0

17. Kakvu vrijednost ima x nakon sljedećih naredbi?

```
double x;  
x = 0;  
x += 3.0;  
x -= 2.0;
```

- A) 1.0    B) 5.0    C) 2.0    D) 0.0

18. Odredi vrijednost varijable d:

```
double d = 11 / 2;
```

19. Odredi vrijednost koja se dobije izračunom izraza:

```
pow(2,3)                      fabs(-3.5)                      sqrt(pow(3,2))  
7 / abs(-2)    ceil(5.8)                      floor(5.8)
```

20. Napiši izraze prema sintaksi jezika C++.

## Bonus zadatak

1. Napiši program za izračun gustoće morske vode na površini mora koristeći zadane formule. Gustoća morske vode na površini mora izračunava se na temelju saliniteta (S) i temperature (t) uporabom sljedeće formule:

$$\rho(S, t, 0) = \rho_w + (8.24493 \times 10^{-1} - 4.0899 \times 10^{-3}t + 7.6438 \times 10^{-5}t^2)$$

$$\begin{aligned}
& - (8.2467 \times 10^{-7} t^3 + 5.3875 \times 10^{-9} t^4) S \\
& + (-5.72466 \times 10^{-3} + 1.0227 \times 10^{-4} t - 1.6546 \times 10^{-6} t^2) S^{3/2} \\
& + 4.8314 \times 10^{-4} S^2
\end{aligned} \tag{2}$$

gdje je  $\rho_w$ , dano sa:

$$\begin{aligned}
\rho_w = & 999.842594 + 6.793952 \times 10^{-2} t - 9.095290 \times 10^{-3} t^2 \\
& + 1.001685 \times 10^{-4} t^3 - 1.120083 \times 10^{-6} t^4 + 6.536332 \times 10^{-9} t^5
\end{aligned} \tag{3}$$

Test podaci su:  
za  $S=35 \text{ ‰}$  (uvrsti 35),  $t=5^\circ \text{C}$  gustoća iznosi  $1027.675465 \text{ kg m}^{-3}$  (rezultat se mora podudarati na decimalu točno!)



## 4. JEDNOSTAVNA KONTROLA TIJEKA IZVOĐENJA PROGRAMA

Programi koje smo do sada vidjeli sastoje se od jednostavne liste naredbi koje se izvode danim redoslijedom. Kažemo da se naredbe u tim programima izvode **sekvencijalno**. Redoslijed izvođenja naredbi zovemo **tijekom izvođenja programa**. Upoznati ćemo dva načina za kontrolu tijeka izvođenja programa. Prvi, **mehanizam grananja omogućava** prilikom izvođenja programa **izbor između dva alternativna postupka**, pri čemu izbor jednog od njih ovisi o vrijednostima varijabli. Upoznati ćemo i **mehanizam petlje** koji omogućava ponavljanje postupka određeni broj puta.

### JEDNOSTAVNI MEHANIZAM GRANANJA

Problem:

Pretpostavimo da oblikujemo program za izračun tjedne plaće zaposlenika koji je plaćen po satu. Neka je prekovremeni sat plaćen kao jedan i pol sat u granicama norme. Dakle, ako zaposlenik radi 40 ili više sati plaća se izračunava kao:  
 $\text{cijena\_sata} * 40 + 1.5 * \text{cijena\_sata} * (\text{sati} - 40)$  gdje varijabla *sati* predstavlja broj odrađenih sati.

Ako zaposlenik radi manje od 40 sati tjedno, npr. 10 sati ova formula se ne može primijeniti (za 10 sati i cijenu od 30 kn po satu dobivamo -50 kn). Za zaposlenika koji radi manje od 40 sati tjedno primijenit ćemo izraz:  
 $\text{cijena\_sata} * \text{sati}$

Ako se mogu očekivati i slučaj plaćanja ispod i preko norme program će morati odabrati između dvije formule.

Dakle ako je ( $\text{sati} > 40$ ) istina

izvršava se slijedeća dodjela:

$\text{tjedna\_placa} = \text{cijena\_sata} * 40 + 1.5 * \text{cijena\_sata} * (\text{sati} - 40);$

a ako to nije istina računa se:

$\text{tjedna\_placa} = \text{cijena\_sata} * \text{sati};$

### Izvedba grananja

Za izvedbu grananja se u jeziku C++ koristi naredba if-else. Ova naredba bira između dva alternativna postupka. Izračun plaće iz opisanog problema rješavamo slijedećom C++ naredbom:

```
if (sati > 40)
    tjedna_placa = cijena_sata * 40 + 1.5 * cijena_sata * (sati - 40);
else
    tjedna_placa = cijena_sata * sati;
```

Slijedi kompletan program.

```

//PRIKAZ 2.7 Naredba if-else
#include <iostream>
using namespace std;
int main( )
{
    int sati;
    double tjedna_placa, cijena_sata;

    cout << "Unesi cijenu sata: kn ";
    cin >> cijena_sata;
    cout << "Unesi broj odradjenih sati,\n"
        << "zaokruženo na cijeli broj sati: ";
    cin >> sati;

    if (sati > 40)
        tjedna_placa = cijena_sata*40 + 1.5*cijena_sata*(sati - 40);
    else
        tjedna_placa = cijena_sata*sati;

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Sati = " << sati << endl;
    cout << "Cijena sata = kn " << cijena_sata << endl;
    cout << "Tjedna placa = kn " << tjedna_placa << endl;
    return 0;
}

```

Dijalog s korisnikom (podebljan je unos korisnika):

Primjer 1:  
 Unesi cijenu sata: kn **70**  
 Unesi broj odradjenih sati,  
 zaokruženo na cijeli broj sati: **60**  
 Sati = 60  
 Cijena sata = kn 70.00  
 Tjedna placa = kn 4900.00

Primjer 2:  
 Unesi cijenu sata: kn **60**  
 Unesi broj odradjenih sati,  
 zaokruženo na cijeli broj sati: **37**  
 Sati = 37  
 Cijena sata = kn 60.00  
 Tjedna placa = kn 2220.00

Općenita sintaksa if-else naredbe dana je slikom (Slika 8: Sintaksa naredbe if-else). Opisana su dva oblika ove naredbe. U prvom obliku uočavamo da dvije naredbe mogu biti bilo koje izvršne naredbe. Logičkim izrazom provjeravamo da li je uvjet zadovoljen. Kada program naiđe na if-else naredbu izvršiti će se samo jedan od alternativnih postupaka. Npr. logički izraz *sati > 40* izračunavanjem daje rezultat true ili false. Ako je rezultat izračuna true (ako je uvjet zadovoljen) tada se izvodi Da\_Naredba, a ako je izračunom logičkog izraza dobivena vrijednost false (nije zadovoljen) izvršava se Ne\_Naredba. Logički izraz se po sintaksi C++-a mora nalaziti u okruglim zagradama. Operatori uspoređivanja koji se koriste u logičkim izrazima dani su na slici Operatori uspoređivanja.

#### Sintaksa naredbe if-else

##### Jedna naredba u svakoj grani

```
if (Logički_izraz)
    Da_Naredba
else
    Ne_Naredba
```

##### Niz naredbi za svaku granu:

```
if(Logički_izraz)
{
    Da_Naredba_1
    Da_Naredba_2
    ...
    Da_Naredba_Zadnja
}
else
{
    Ne_Naredba_1
    Ne_Naredba_2
    ...
    Ne_Naredba_Zadnja
}
```

Slika 8: Sintaksa naredbe if-else

#### if-else kontrola tijeka (1) - sažetak

```
if (logički izraz)
    Da_Naredba_
else
    Ne_Naredba
Kada je logički izraz istinit izvodi se samo Da_Naredba
Kada je logički izraz lažan izvodi se samo Ne_Naredba
```

#### if-else kontrola tijeka (2) - sažetak

```
if (logički izraz)
{
    Da_Naredbe
}
else
{
```

Ne_Naredbe
}

Kada je logički izraz istina izvode se samo Da\_Naredbe unutar { }

Kada je logički izraz laž izvode se samo Ne\_Naredbe unutar { }

## LOGIČKI IZRAZI - SAŽETAK

Logički izrazi su izrazi koji kao rezultat izračuna daju vrijednost *istina* ili *laž*.

**Operatori uspoređivanja** kao što je '>' (veće od) koriste se za uspoređivanje varijabli i/ili brojeva; dva izraza koji se uspoređuju najprije se vrednuju (izračunavaju se) pa se nakon toga dobivene vrijednosti uspoređuju

(**sati > 40**) uključujući zagrade, je logički izraz iz prethodnog primjera

Nekoliko operatora uspoređivanja koji koriste dva simbola (Praznina nije dozvoljena između simbola!):

>=    veće od ili jednako

!=    različito

==    jednako

Operatori uspoređivanja

Matematički simbol	Značenje	C++ zapis	C++ primjer	Matematički ekvivalent
=	jednako	==	x + 7 == 2*y	$x + 7 = 2y$
≠	različito od	!=	odg != 'n'	$\text{odg} \neq 'n'$
<	manje od	<	brojač < m + 3	$\text{brojač} < m + 3$
≤	manje od ili jednako	<=	vrijeme <= granica	$\text{vrijeme} \leq \text{granica}$
>	veće od	>	vrijeme > granica	$\text{vrijeme} > \text{granica}$
≥	veće od ili jednako	>=	starost >= 21	$\text{starost} \geq 21$

Slika 9: Operatori uspoređivanja

Dva izraza uspoređivanja možemo kombinirati uporabom operatora "and" koji se u C++-u piše &&. Npr. slijedeći logički izraz je istinit (uvjet je zadovoljen) ako je vrijednost varijable x veća od 2 i x je manji od 7:

(2 < x) && (x < 7)

Kada su dva izraza uspoređivanja povezana sa && cjelokupni izraz je istinit samo ako su oba izraza istinita. Ako izraze povežemo sa "OR" (|| u C++-u) dovoljno je da je jedan

podizraz istinit da bi cjelokupan izraz bio istinit. Tablica istinitosti opisuje rezultat izračuna za pojedine logičke operatore.

### Logički tipovi i operatori - pregled

Logički operatori:

!x                      logička negacija ("NOT")  
 x && y                logički i ("AND")  
 x || y                 logički ili ("OR")

#### Tablica istinitosti

a	b	! a	a && b	a    b
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

### AND – sažetak

Logički izrazi se mogu kombinirati u složenije izraze uporabom operatora AND: &&

Daje istinu ako su oba izraza istina

Sintaksa: (LogičkiIzraz\_1) && (LogičkiIzraz\_2)

Primjer: if ( (2 < x) && (x < 7) )

Istina samo ako je x između 2 i 7

Zagrade nisu obavezne ali naglašavaju značenje

### OR - sažetak

|| -- Operator OR (nema razmaka!)

Istina ako je bar jedan od izraza istinit

Sintaksa: (Izraz\_1) || (Izraz\_2)

Primjer: if ( ( x == 1) || ( x == y) )

Istina ako x ima vrijednost 1

Istina ako x ima istu vrijednost kao y

Istina ako su obje usporedbe jednake istini

### NOT - sažetak

! negacija logičkog izraza

!( x < y)

Rezultat izračunavanja izraza je Istina ako x NIJE manje od y

!(x == y)

Istina ako x NIJE jednak y

Zbog operatora negacije (!) izraze je često teško čitati pa ih treba koristiti samo kada je to opravdano.

### Problem: Nejednakosti

Oprezno prevodite matematičke nejednakosti u C++. Matematički izraz  $x < y < z$  se prevodi u

```
if ( ( x < y ) && ( y < z ) )  
    a ne  
if ( x < y < z )
```

### Problem: Uporaba = umjesto ==

'=' je operator dodjele

Koristi se za dodjelu vrijednosti varijablama

Primjer: `x = 3;`

'==' je operator uspoređivanja na jednakost

Koristi se za uspoređivanje vrijednosti

Primjer: `if ( x == 3 )`

Prevoditelj neće javiti grešku za:

```
if ( x = 3 )
```

nego će spremiti 3 u x umjesto da uspoređi x i 3

Kako je rezultat jednak 3 (različit od 0), izraz je istina

### Redoslijed (prioritet) izvođenja operacija - pregled

```
++      uvećaj prije  
--      umanji prije  
+ - !    unarni operatori  
* / %    množenje, dijeljenje, ostatak cjelobrojnog dijeljenja  
+ -      zbrajanje, oduzimanje  
< > <= >= poredbeni operatori  
== !=    operatori jednakosti  
&&       logički i  
||        logički ili  
= *= /= += -= %= pridruživanja
```

### Složene naredbe

Složenu naredbu (blok naredbi) čini više od jedne naredbe koje su zatvorene u { }

U granama if-else naredbe obično se izvršava više od jedne naredbe

```
Primjer:  if (logički izraz)  
           {  
             Naredbe_istina  
           }  
           else  
           {  
             Naredbe_laž  
           }
```

Složena naredba se tretira kao jedna naredba i može se koristiti bilo gdje na mjestu na kojem se može pojaviti jedna naredba. pogledajmo složenu naredbu na primjeru:

Primjer: Naredba if-else sa blokovima naredbi u granama

```
if(moji_bodovi>tvoji_bodovi)  
{
```

```
        cout<<"Moja pobjeda!\n";  
        ukupno=ukupno+100;  
    }  
    else  
    {  
        cout<<"Da su to bar bodovi u golfu!\n";  
    }
```

## Priprema za kviz

1. Napiši if\_else naredbu u kojoj se ispisuje riječ VISOKO ako je vrijednost varijable rezultat veća od 100, odnosno ispisuje NISKO u slučaju da je vrijednost varijable rezultat najviše 100. Varijable su tipa int.
2. Napiši if-else naredbu u kojoj se ispisuje riječ Upozorenje u slučaju da je ili vrijednost varijable temperatura veća ili jednaka 100 ili je vrijednost varijable tlak veća ili jednaka 200 ili oboje. Inače, se ispisuje riječ OK. Varijable su tipa int.
3. Objasni pojmove kontrola tijeka i grananje (navedi primjer grananja u pseudokodu).
4. Što su logički izrazi i koje operatore koriste. Navedi tablice istinitosti za logičke operatore.
5. Sintaksa i značenje naredbe if-else (objasniti na konkretnom vlastitom primjeru).

## 5. UVOD U PETLJE

Programi obično uključuju neki postupak koji se ponavlja određeni broj puta. Dio programa koji ponavlja naredbu ili skupinu naredbi zovemo **petlja**. U C++ se petlja može izvesti na nekoliko načina. Dva konstrukta koja omogućavaju izvedbu petlju su **naredba while** ili **while petlja** i **petlja do-while** ili **naredba do-while**.

### WHILE PETLJA

Značenje while petlje naznačeno je značenjem engleske riječi while (dok). Izvršavanje (izvođenje) petlje se ponavlja dok je vrijednost logičkog izraza u okruglim zagradama jednaka true. U slijedećem primjeru naredbe koje su zatvorene u **vitičaste zagrade** zovemo **tijelo while petlje**. Tijelo petlje predstavlja postupak koji se ponavlja. Naredbe u tijelu petlje izvršavaju se redom kako su navedene, a zatim se njihovo izvršavanje ponavlja sve dok izvršavanje petlje ne završi. U prvom primjeru dijaloga s korisnikom tijelo petlje se izvršava tri puta prije **izlaska iz petlje**. Svako ponavljanje izvršavanja while petlje zove se **iteracija** (ili ponavljanje, ciklus) petlje.

```
int odbrojavanje=3;
while (odbrojavanje > 0)
{
    cout << "Hello ";
    odbrojavanje -= 1;
}
```

Izlaz:      Hello Hello Hello

Svaki put kad se dio programa (gornji primjer) izvede ispiše se "Hello", a varijabla *odbrojavanje* se smanjuje za 1. Nakon tri izvođenja tijela petlje varijabla odbrojavanje se smanjila na 0 i logički izraz više nije zadovoljen (ima vrijednost *false*).

#### Kompletan program:

```
#include <iostream>
using namespace std;
int main( )
{
    int odbrojavanje;

    cout << "Koliko pozdrava zelis? ";
    cin >> odbrojavanje;

    while (odbrojavanje > 0)
    {
        cout << "Hello ";
```



```

    odbrojavanje = odbrojavanje - 1;
}

cout << endl;
cout << "Nema vise!\n";

return 0;
}

```

### Primjer dijaloga 1

Koliko pozdrava zelis? **3** // Tijelo petlje se izvršava **3** puta.

Hello Hello Hello

Nema vise!

### Primjer dijaloga 2

Koliko pozdrava zelis? **1** // Tijelo petlje se izvršava **1** put.

Hello

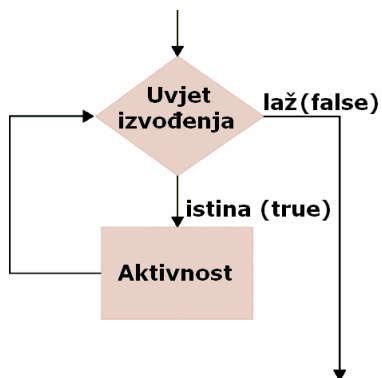
Nema vise!

### Primjer dijaloga 3

Koliko pozdrava zelis? **0** // Tijelo petlje se izvršava **0** puta.

Nema vise!

## Logika izvršavanja petlje



Slika 10: Logika izvršavanja petlje

Najprije se vrednuje logički izraz. Ako je njegova vrijednost *false*, program se nastavlja izvršavati od linije koja slijedi iza *while* petlje. U slučaju vrijednosti *true*, izvršava se tijelo petlje. Za vrijeme izvođenja, mijenjaju se neke varijable iz logičkog izraza (odbrojavanje u gornjem primjeru). Nakon izvođenja tijela petlje, logički izraz se ponovo provjerava i ponavlja cijeli postupak sve dok izraz ne postane *false*.

Čim logički izraz postane jednak *false* dolazi do izlaska iz petlje. While petlja se neće izvršiti niti jednom ako logički izraz ima vrijednost *false* na početku (prije) prvog izvođenja petlje.

## Sintaksa while petlje

Logički izrazi koje smijemo koristiti u while petlji su po sintaksi isti izrazi koje koristimo u if-else naredbi. Kao i kod if-else naredbe izraz mora biti zatvoren u okrugle zagrade.

Imamo dva slučaja sintakse while petlje:

### 1. Tijelo petlje sačinjava više naredbi

Tijelo petlje moramo označiti vitičastim zagrada. Kada vitičaste zagrade ne bi koristili, samo prva naredba petlje bi se ponavljala unutar petlje, a ostale naredbe slijedile bi iza petlje – mogući izvor logičke greške programa.

```
while (logički izraz)
{
    naredbe koje se ponavljaju
}
```

Točka-zarez se koristi samo nakon naredbi u petlji za njihovo odvajanje.

### Tijelo petlje sadrži samo jednu naredbu

Ne moramo koristiti vitičaste zagrade za označavanje tijela petlje.

```
while (logički izraz)
    naredba koja se ponavlja
```

## Do-While Petlja

do-while petlja je slična while petlji osim što se tijelo do-while petlje izvodi najmanje jednom budući se uvjet petlje nalazi na kraju. Najprije se izvršava tijelo petlje, a zatim se provjerava uvjet izvođenja petlje.



Slika 11: Logika izvršavanja do-while petlje

Tijelo do-while petlje s nekoliko naredbi:

```
do
{
    Naredba_1
    Naredba_2
    ...
    Naredba_zadnja
} while (Logički_izraz);
```

Petlja do-while s jednom naredbom

```
do
Naredba
while (Logički_izraz);
```

## INKREMENTIRANJE/DEKREMENTIRANJE

Unarni operatori zahtijevaju samo jedan operand

- + ispred broja kao: +5
- ispred broja kao: -5
- ++ operator inkrementiranja

Vrijednost varijable povećava se za 1:

```
    x ++;
je ekvivalentno    x = x + 1;
```

-- operator dekrementiranja

Vrijednost varijable umanjuje se za 1

```
    x --;
je ekvivalentno    x = x - 1;
```

### Primjer programa (while petlja)

Stanje na bankovnom računu je 400 kn. Mjesečne kamate iznose 2%. Za koliko mjeseci će iznos na računu prijeći 800 kn?

Stanje nakon 1 mjeseca:  $400 + 2 * 400 / 100 = 458$  kn  
Stanje nakon 2 mjeseca:  $458 + 2 * 458 / 100 = 467.16$  kn  
Stanje nakon 3 mjeseca:  $476.5 + 2 * 476.5 / 100 = 486.3$  kn ...

```
// Program: Kreditna kartica
#include <iostream>
using namespace std;
```

```

int main( )
{
    double racun = 400.00;
    int brojac = 0;

    cout << "Program odredjuje koliko je\n"
        << "potrebno da bi se skupio iznos od"
        << "800 kn, pocevsi sa iznosom od \n"
        << "400 kn na racunu.\n"
        << "Kamata je 2% mjesečno.\n";

    while (racun < 800.00)
    {
        racun = racun + 0.02 * racun;
        brojac++;
    }

    cout << "Nakon " << brojac << " mjeseci,\n";
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "vas je iznos " << racun << " kn"<< endl;

    return 0;
}

```

### Primjer dijaloga

Program odredjuje koliko je potrebno da bi se skupio iznos od 800 kn, pocevsi sa iznosom od 400 kn na racunu.  
 Kamata je 2% mjesečno.  
 Nakon 141 mjeseci,  
 vas je iznos 815.82 kn

### Primjer programa (do-while petlja)

```

//do-while petlja
#include <iostream>
using namespace std;
int main( )
{
    char odg;

    do
    {

```

```

cout << "Dobar dan!\n";
cout << "Zelite li jos jedan pozdrav?\n"
    << "Stisni d za da, n za ne,\n"
    << "i pritisni ENTER: ";
cin >> odg;
} while (odg == 'd' || odg == 'D');

cout << "Dovidjenja!\n";

return 0;
}

```

### Primjer dijaloga

```

Dobar dan!
Zelite li jos jedan pozdrav?
Stisni d za da, n za ne,
i pritisni ENTER: d
Dobar dan!
Zelite li jos jedan pozdrav?
Stisni d za da, n za ne,
i pritisni ENTER: D
Dobar dan!
Zelite li jos jedan pozdrav?
Stisni d za da, n za ne,
i pritisni ENTER: n
Dovidjenja!

```

### Beskonačna petlja

Petlje koje nikada ne završe s ponavljanjem su beskonačne petlje. Tijelo petlje bi trebalo sadržavati liniju koja će u jednom trenutku uzrokovati da logički izraz postane false.

Primjer: Ispiši neparne brojeve manje od 12

```

x = 1;
while (x != 12)
{
    cout << x << endl;
    x = x + 2;
} // LOGIČKA GREŠKA!

```

Bolje je koristiti usporedbu: while ( x < 12). x nikada neće imati vrijednost 12.

### Priprema za kviz

1. Kakav izlaz daje kôd ako je x tipa int?

```
x = 10;
while ( x > 0)
{
    cout << x << endl;
    x = x - 3;
}
```

2. Kakav je izlaz prethodnog koda ako usporedbu  $x > 0$  zamijenimo sa  $x < 0$ ?

## STIL PROGRAMA

Ako pazimo na stil programa, program se lakše čita, lakše ispravlja i lakše mijenja.

Uvlaka

Dijelovi programa koje smatramo cjelinom trebaju izgledati kao cjelina. Preskačemo linije između logičkih grupa naredbi. Uvucimo naredbu unutar druge naredbe :

```
if (x == 0)
    naredba;
```

Vitičaste zagrade { } kreiraju grupe naredbi (blokove). Uvlačite naredbe unutar vitičastih zagrada da bi naglasili grupu. Vitičaste zagrade izdvojene u posebne linije lakše se lociraju.

## Komentari

// je simbol za jednolinijski komentar. Komentari su zabilješke – objašnjenja programa za programera. Sav tekst koji slijedi iza // prevoditelj zanemaruje

Primjer:     // izračun tjedne plaće  
              tjedna\_placa = cijena\_sata \* sati;

/\* i \*/ zatvaraju višelinijski komentar

Primjer:     /\* Ovo je višelinijski komentar  
              - na srednjoj liniji nema oznake  
              \*/

## Konstante

Numeričke konstante koje se koriste u programu je teško pronaći i promijeniti kada je potrebno. Konstante omogućavaju imenovanje numeričkih konstanti i istovremeno mijenjanje vrijednosti svakog pojavljivanja konstante u programu mijenjanjem

vrijednosti konstante na jednom mjestu. Ako se promijeni vrijednost neke konstante koju program koristi npr. iznos PDV-a, dovoljno je vrijednost konstante promijeniti na jednom mjestu u programu.

const je ključna riječ za deklaraciju konstante

Primjer:

```
const int MINIMUM = 10;  
deklarira konstantu s imenom MINIMUM
```

Vrijednost konstante se ne može mijenjati u programu kao vrijednost varijable. U imenima konstanti obično su sva slova velika.

#### **Primjer (konstante)**

```
//PRIKAZ 2.15 Komentari i konstante s imenom  
//Ime datoteke: zdravlje.cpp  
//Autor: Ovdje upisite vase ime i prezime.  
//Email adresa: vi@vas_server.bla.bla  
//Redni broj zadatka: 2  
//Opis: Program koji dijagnosticira bolest  
//korisnika.  
//Zadnja promjena: 23 rujana, 2006
```

```
#include <iostream>  
using namespace std;  
int main( )  
{  
    const double NORMAL = 37.0;  
        //stupnjeva Celzijusa  
    double temperatura;  
  
    cout << "Unesite vasu temperaturu: ";  
    cin >> temperatura;
```

#### **Primjer dijaloga**

```
Unesite vasu temperaturu: 36  
Nemate povišenu temperaturu.  
Mozete dalje uciti.
```

Priprema za kviz

1. Objasniti sintaksu i značenje petlje **while** (objasniti na konkretnom vlastitom primjeru i nacrtati dijagram tijeka).
2. Objasniti sintaksu i značenje petlje **do-while** (objasniti na konkretnom vlastitom primjeru i nacrtati dijagram tijeka).
3. Objasni problem beskonačne petlje. Navedi konkretni primjer i objasni.
4. Koji su elementi dobrog stila programiranja i zašto je važan?
5. Kakav izlaz daje sljedeći dio programa (x je tipa int)?

```
x=10;
do
{
    cout<<x<<endl;
    x=x-3;
}while(x>0);
cout<<x;
```

6. Kakav je izlaz sljedećeg koda (x je tipa int)?

```
x=-42;
do
{
    cout<<x<<endl;
    x=x-3;
}while(x>0);

cout<<x;
```

7. Koja je najvažnija razlika između while i do-while naredbe?
8. Sljedeća if-else naredba je ispravna. Napišite je tako da poštuju pravila stila programa.

```
if(x<0) {x=7;cout<<"x je sada pozitivan";} else
{x=-7;cout<<"x je sada negativan."};}
```

9. Napišite kompletan program koji traži od korisnika da unese iznos u galonima i ispisuje ekvivalentan iznos u litrama. Galon ima 3.78533 litara. Koristi deklariranu konstantu.



## Bonus zadaci

1. Prirodan broj je djeljiv sa 3 ako mu je zbroj znamenaka djeljiv sa 3. Napišite program koji će unositi troznamenkasti prirodan broj i zbrajati mu znamenke sve dok ga ne svede na jednoznamenkasti, te će na osnovi toga reći da li je broj djeljiv sa 3.

Primjer:

Ulaz: 996

Ispis: Broj je djeljiv sa 3.

Komentar:  $9+9+6=24$ ,  $2+4=6$

2. Napiši program koji učitava n cijelih brojeva i ispisuje sumu svih brojeva većih od 0, sumu svih brojeva manjih ili jednakih 0 i sumu svih brojeva (pozitivnih, negativnih ili nule). Korisnik unosi n brojeva jednog po jednog u bilo kojem poretku. Program ne smije tražiti da se posebno unose pozitivni a posebno negativni brojevi.

Drugi dio zadatka:

Nadogradi program tako da se uz svaku sumu ispisuje i prosjek brojeva koji su ušli u tu sumu. Na kraju treba ispisati prosjek svih unesenih brojeva.

Primjer:

Ulaz: n: 11; Brojevi: 1, 3, 0, -4, 3, 0, -7, 2, 0, -2, 9

Izlaz: Suma pozitivnih: 18

Prosjek pozitivnih: 3.6

Suma negativnih: -13

Prosjek negativnih (i nula): 2.16

Suma svih brojeva: 5

Prosjek svih brojeva: 0.4545

3. Babilonski algoritam za izračun kvadratnog korijena broja n ima slijedeće korake:

1. Prvi pokusaj =  $n/2$

2. Računa se  $r = n/\text{pokusaj}$

3. Postavimo  $\text{pokusaj} = (\text{pokusaj} + r) / 2$

4. Vraćamo se na korak 2 koliko god puta je potrebno. Što se više puta ponove koraci 2 i 3. pokusaj će biti bliži kvadratnom korijenu od n.

Napiši program koji na ulazu ima cijeli broj n i ponavlja Babilonski algoritam dok god se vrijednost pokusaja ne razlikuje za 0.001 od prethodnog pokusaja. Konačni pokusaj se ispisuje kao realna vrijednost (double).

Primjer:

Računamo  $\sqrt{2}=1.41421356$ , znači  $n=2$

$\text{pokusaj}=2/2=1$

$r=n/\text{pokusaj}=2/1=2$

$\text{pokusaj}=(\text{pokusaj}+2)/2=(1+2)/2=1.5$

$r=2/1.5=4/3=1.333$

$\text{pokusaj}=(1.5+1.333)/2=1.41666$

$r=2*/1.41666=1.4117$

$p=(1.41666 + 1.4117)/2=1.41418$

...



## 6. NAPREDNIJI TIJEK KONTROLE IZVOĐENJA PROGRAMA

**Tijek izvođenja programa** je redoslijed izvođenja naredbi programa. Do sada smo upoznali slijedeće načine za specificiranje tijeka izvođenja programa:

if-else naredba  
while naredba  
do-while naredba

Nastavljamo sa upoznavanjem novih načina primjene spomenutih naredbi, a u ovom poglavlju uvodimo i dvije nove naredbe:

switch naredbu  
for naredbu

Djelovanje if-else naredbe, while naredbe i do-while naredbe nadzire **logički izraz**. Zato ćemo se njime na početku poglavlja detaljnije baviti.

### Uporaba logičkih izraza

Logički izraz je izraz koji rezultira vrijednošću *true* ili *false*. Do sada smo logičke izraze koristili na mjestu uvjeta u if-else naredbi i na mjestu uvjeta u while i do-while petlji. U jeziku C++ definiran je logički tip `bool` koji omogućava deklaraciju varijabli koje mogu imati samo vrijednosti *true* ili *false*.

Logički se izraz vrednuje (izračunava) kao i aritmetički izraz. Jedina je razlika u tome što aritmetički izraz koristi operacije kao `*`, `/`, `+`, `-` i kao rezultat se izračunava broj, dok se logički izrazi izračunavaju uporabom operacija uspoređivanja kao što su `==`, `<` i `>=` koje daju logičku vrijednost i logičkih operacija kao što su `&&`, `||` i `!` čiji je rezultat isto logička vrijednost. Uočimo da se operacije kao što su `==`, `<` i `>=` (relacijski, poredbeni operatori) mogu primijeniti za uspoređivanje bilo kojih istovrsnih ugrađenih tipova podataka (`int` sa `int`, `float` sa `float`, `double` sa `double`, `char` sa `char`, `bool` sa `bool`) da bi rezultat izračuna bila logička vrijednost *true* ili *false*. Moguće je usporediti i različite tipove podataka za koje je definirana konverzija između tipova.

### VREDNOVANJE LOGIČKIH IZRAZA

Promotrimo logički izraz:

`!((y < 3) || (y > 7))`

koji se može pojaviti na mjestu uvjeta u if-else ili u while naredbi. Pretpostavimo da vrijednost od `y` iznosi 8. U ovom slučaju (`y < 3`) se izračunava kao *false* i (`y > 7`) se izračunava kao *true*, pa je gornji izraz ekvivalentan izrazu

!(false || true)

Logički izrazi se vrednuju na temelju tablice istinitosti. Tako se vrijednost u zagradama izračunava kao true, a cijeli izraz se izračunava kao false. Rezultat izračunavanja početnog izraza je dakle false. Kako relacijski izrazi imaju viši prioritet od logičkih mogli smo u početnom izrazu izostaviti dio zagrada i napisati ekvivalentno:

!( y < 3 || y > 7 )

No zbog čitljivosti programa i naglašavanja prioriteta ove zagrade se ipak preporučuje koristiti.

Kada zagrade uopće ne koristimo, pri izračunu se poštuje prioritet izvođenja operacija, tj dijelovi izraza se grupiraju prema pravilima prioriteta za aritmetičke i logičke operatore.

a	b	! a	a && b	a    b
F	F	T	F	F
F	T	T	F	T
T	F	F	F	T
T	T	F	T	T

Slika 12: Tablica istinitosti

Prioritet operatora	
Unarni operatori +, -, ++, --, !	Najviši prioritet (izvršava se prvo) ↓ Najniži prioritet (izvršava se zadnje)
Binarni aritmetički operatori *, /, %	
Binarni aritmetički operatori +, -	
Relacijski operatori <, >, <=, >=	
Relacijski operatori ==, !=	
Logički operatori &&	
Logički operatori	

Slika 13: Prioritet operatora

Najprije se izvode operatori sa višim prioritetom. Binarni operatori sa jednakim prioritetom se izvode s lijeva na desno.

int b=6 + 3 – 2 + 4 – 10 .....1

Unarni operatori jednakog prioriteta izvode se s desna na lijevo

int a=9; !--a;.....false

### Primjer primjene prioriteta izvođenja operacije

Izrazi obično uključuju i aritmetičke i logičke operatore kao u slijedećem primjeru:

Izraz

$$(x+1) > 2 \ || \ (x + 1) < -3$$

je ekvivalentan izrazu:

$$((x + 1) > 2) \ || \ ((x + 1) < -3)$$

Kako  $>$  i  $<$  imaju viši prioritet od  $||$  ekvivalentan je i izrazu:

$$x + 1 > 2 \ || \ x + 1 < -3$$

Kako se vrednuje izraz:

$$x + 1 > 2 \ || \ x + 1 < -3 \ ?$$

Uporabom opisanih pravila za prioritet izvođenja operacija:

Najprije primijeni unarni operator –

Zatim primijeni operatore +

Slijedi primjena operatora  $>$  i  $<$

Na kraju primijeni  $||$

### VREDNOVANJE IZRAZA TZV. KRATKIM SPOJEM

Neki logički izrazi se ne moraju vrednovati u svim svojim dijelovima. Ako je  $x$  negativan, vrijednost izraza

$$(x \geq 0) \ \&\& \ (y > 1)$$

može se odrediti vrednovanjem samo podizraza  $(x \geq 0)$ .

Kažemo da C++ koristi vrednovanje izraza kratkim spojem. Ako vrijednost krajnje lijevog podizraza određuje konačnu vrijednost izraza, ostatak izraza se ne vrednuje.

### Primjer primjene

Vrednovanje kratkim spojem može se koristiti za sprječavanje grešaka pri izvođenju programa (“run-time errors”). Promotrimo naredbu `if`:

```
if ((djeca != 0) && (bomboni / djeca >= 2) )  
    cout << "Svako dijete može dobiti bar dva bombona!";
```

Ako je vrijednost varijable djeca jednaka 0, vrednovanje kratkim spojem sprječava vrednovanje podizraza (bomboni / 0 >= 2). Dijeljenje s 0 bi uzrokovalo grešku pri izvođenju programa.

Između tipova bool i int obavlja se automatska konverzija tipa. C++ koristi cijele brojeve kao da su logičke vrijednosti.

Svaki broj različit od nule (obično 1) je true; true se pretvara u 1  
0 (nula) je false; false se pretvara u 0

### Problemi sa logičkom negacijom !

Pretpostavimo da želimo provjeriti da li je isteklo vrijeme igre da bi prekinuli igru.

Ukoliko vrijeme nije isteklo želimo da se igra nastavi. Izraz

(! vrijeme > granica ), uz granica = 60,

vrednuje se kao

(! vrijeme) > granica

Ako je vrijeme cjelobrojna varijabla koja ima vrijednost 36, koliku vrijednost ima izraz

!vrijeme? False ili nula, jer se 36 najprije pretvara u 1 i onda računa negacija. Izraz se

dalje vrednuje kao

0 > granica

false

Kako izbjeći ovaj problem?

U prethodnom izrazu se vjerojatno mislilo slijedeće

( ! ( vrijeme > granica ) )

što se vrednuje kao

( ! ( false ) )

true

Kao što negacija u prirodnom jeziku ne olakšava razumijevanje, operator negacije ! čini C++ izraze teškim za razumijevanje. Prije uporabe operatora negacije (!), provjeri da li možeš izraziti istu ideju jasnije bez operatora negacije.

### Priprema za kviz

1. Koji logički operator opisuje slijedeća tablica istinitosti?

A	B	Operacija
True	True	True
True	False	True
False	True	True
False	False	False

A) and

B) not

C) or

D) ništa od navedenoga

2. Kakva je vrijednost slijedećeg izraza?  
 (true && (4/3 || !(6)))  
 A) 0      B) false      C) true      D) nedozvoljena sintaksa
3. Koji od izraza je ekvivalentan izrazu  $!(x < 15 \ \&\& \ y \geq 3)$ ?  
 A)  $(x > 15 \ || \ y < 3)$   
 B)  $(x > 15 \ \&\& \ y \leq 3)$   
 C)  $(x \geq 15 \ || \ y < 3)$   
 D)  $(x \geq 15 \ \&\& \ y < 3)$   
 E) C and D
4. Odredi vrijednost logičkih izraza (pretpostavi da je brojac = 0 i granica = 10):  
 a)  $(\text{brojac} == 0) \ \&\& \ (\text{granica} < 20)$   
 b)  $!(\text{brojac} == 12)$   
 c)  $(\text{granica} < 0) \ \&\& \ ((\text{granica} / \text{brojac}) > 7)$
5. Odredi vrijednost logičkih izraza, uz pretpostavku da je vrijednost varijable brojac jednaka 0, a vrijednost varijable granica je 10. Rezultat izrazite kao true ili false.
- a)  $(\text{granica} < 0) \ \&\& \ ((\text{granica}/\text{brojac}) > 7)$   
 b)  $(5 \ \&\& \ 7) + !(6)$   
 c)  $!(((\text{brojac} < 10) \ || \ (x < y)) \ \&\& \ (\text{brojac} \geq 0))$   
 d)  $((\text{granica} / \text{brojac}) > 7) \ || \ (\text{granica} < 20)$
6. Promotri slijedeći program:

```
#include<iostream.h>
int main() {
  int n; cin >> n;
  if (n>=100) cout<<"C";
  if (n>=10) cout<<"B";
  else cout<<"A";
  if (n<0) cout<<"-";
  else if (n>0) cout<<"+";
  cout << endl;
  return(0);
}
```

Kakav je izlaz programa, ako je unesena vrijednost za n:

- a) -1 izlaz: \_\_\_\_\_  
 b) 0 izlaz: \_\_\_\_\_  
 c) 1 izlaz: \_\_\_\_\_  
 d) 10 izlaz: \_\_\_\_\_

e) 100 izlaz: \_\_\_\_\_

7. Kakav izlaz daje slijedeći kod koji se nalazi u nekom programu?

```
int x=2;
cout << "Pocetak\n";
if (x <=3)
    if (x != 0)
        cout << "Pozdrav iz drugog if-a.\n";
    else
        cout << "Pozdrav iz else-a.\n";
cout << "Kraj\n";

cout << "Ponovo: " << endl;

if (x > 3)
    if (x != 0)
        cout << "Pozdrav iz drugog if-a.\n";
    else
        cout << "Pozdrav iz else-a.\n";
cout << "Ponovo kraj\n";
```

## ENUMERACIJSKI TIPOVI

Enumeracijski tip je tip čije su vrijednosti definirane listom konstanti tipa int. Sličan je listi deklariranih konstanti.

Primjer:

```
enum DuljinaMjeseca{ SIJ= 31, VELJ = 28, OZU = 31, TRA=30, SVI=31, LIP=30,
SRP=31, KOL=31, RUJ=30, LIST=31, STU=30, PRO = 31};
```

U ovom primjeru vidimo da istu vrijednost može imati nekoliko konstanti ovog tipa. Ako ne naznačimo numeričke vrijednosti tada identifikatori (imena) u definiciji enumeracijskog tipa dobivaju uzastopne vrijednosti koje počinju od 0 (podrazumijevane, "default" vrijednosti).

```
enum Smjer { SJEVER = 0, JUG = 1, ISTOK = 2, ZAPAD = 3};
```

je ekvivalentno kao

```
enum Smjer {SJEVER, JUG, ISTOK, ZAPAD};
```

Ako nije drugačije specificirano, vrijednost dodijeljena enumeracijskoj konstanti je za 1 veća od prethodne konstante.

```
Enum MojEnum{JEDAN = 17, DVA, TRI, CETIRI = -3, PET};
```

rezultira slijedećim vrijednostima



JEDAN = 17, DVA = 18, TRI = 19, ČETIRI = -3, PET = -2

## Priprema za kviz

8. Za danu definiciju enumeracijskog tipa, koja je vrijednost od SAT?
- ```
enum myType{SUN=3,MON=1,TUE=3,WED,THUR,FRI,SAT,NumDays};
```
- A) 6
  - B) 5
  - C) 7
  - D) 8
  - E) nepoznata

## 7. VIŠESTRUKO GRANANJE

Konstrukt jezika koji bira između dva ili više postupaka zovemo **mehanizam grananja**. if-else naredba bira između dva postupka. U ovom poglavlju se bavimo načinima za odabir između više od dvaju postupaka.

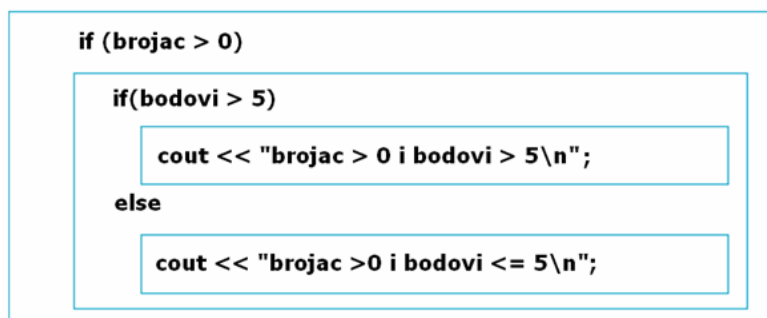
Mehanizam grananja odabire jednu od alternativnih akcija. Naredba if-else je mogući izbor mehanizma grananja. Mehanizam grananja može biti dio nekog drugog mehanizma grananja. Jedna naredba if-else može uključivati drugu if-else naredbu kao svoj dio.

Naredba koja je dio druge naredbe je ugniježđena naredba (nested). Kada pišemo ugniježđenu naredbu uobičajeno je da uvučemo svaku razinu gniježđenja:

Primjer: 

```
if ( x < y)
    cout << x << " je manje od " << y;
    else
    cout << y << " je manje od " << x;
```

### if-else naredba koja sadrži drugu if-else naredbu



Slika 14: Ugniježđena if-else naredba

Primjer:

Oblikujmo if-else naredbu sa ciljem da se vozač obavijesti da je vozilo ostalo bez goriva ako goriva ima malo, a u slučaju da je spremnik skoro pun savjetujemo vozača da prođe kraj crpke bez zaustavljanja. U drugim slučajevima nema nikakvih upozorenja.

Zadatak:

Program traži od korisnika da unese količinu goriva u spremniku i ispisuje poruku ovisno o količini goriva:

Ako je količina goriva  $> 3/4$  spremnika ispisuje se  
Razina goriva preko  $3/4$ . Ne zaustavljajte se!

Ako je količina goriva  $< 1/4$  spremnika ispisuje se  
Oprez, nema goriva!

inače se ne ispisuje ništa.

Pseudokod: ako je mjerač goriva ispod  $3/4$  tada:  
    ako je mjerač goriva ispod  $1/4$  tada:  
        upozori vozača  
    inače (mjerač  $> 3/4$ ) tada:  
        obavijest da ne mora puniti gorivo

Prvi pokušaj – ugniježđeni if:

Prevođenje prethodnog pseudokoda u C++ moglo bi izgledati (ako nismo pažljivi)

```
if (ocitanje_mjeraca_goriva < 0.75)
    if (ocitanje_mjeraca_goriva < 0.25)
        cout << "Goriva vrlo malo. Oprez!\n";
    else
        cout << "Goriva preko 3/4. Vozite dalje!\n";
```

Kod se uredno prevodi i izvršava, ali ne daje zadovoljavajući rezultat. Prevoditelj smatra da je else vezan uz najbliži if koji mu prethodi.

Vitičaste zagrade u ugniježđenim naredbama su kao okrugle zagrade u izrazima. Vitičaste zagrade kažu prevoditelju kako su grupirani dijelovi koda. Koristite vitičaste zagrade za označavanje početka i kraja podnaredbe ilustrira uporabu vitičastih zagrada u ugniježđenim zgradama.

```
// Ilustrira važnost vitičastih zagrada
#include <iostream>
using namespace std;
int main()
{
    double kolicina_goriva;
```

```

cout<< "Unesi količinu goriva: ";
cin>>kolicina_goriva;

cout<<"Sa zgradama:\n";
if(kolicina_goriva < 0.75)
{
    if(kolicina_goriva < 0.25)
        cout<<"Oprez, nema goriva!\n";
}
else
{
    cout<<"Razina goriva preko ¾.";
    cout<<" Ne zaustavljajte se!\n";
}
cout<<"Sada bez zagrada:\n";
if(kolicina_goriva < 0.75)
    if(kolicina_goriva < 0.25)
        cout << "Oprez, nema goriva!\n";
else
    cout<<"Razina goriva preko ¾.";
    cout<<" Ne zaustavljajte se!\n";

return 0;
}

```

Unesi količinu goriva: **0.1**

Sa zgradama:

Oprez, nema goriva!

Sada bez zagrada:

Oprez, nema goriva!

Unesi količinu goriva: **0.5**

Sa zgradama:

Sada bez zagrada:

Razina goriva preko ¾. Ne zaustavljajte se!

## VIŠESTRUKO GRANANJE UPORABOM IF-ELSE-NAREDBI

Naredba if-else ima dvije grane. Tri, četiri (ili više) grana može se postići uporabom ugniježđenih if-else naredbi.

Primjer:

Igra pogađanja brojeva sa brojevima redom pohranjenim u varijabli broj, a pokušajima u varijabli pokusaj. Kako usmjeravamo pokušaje igrača?

Slijedeće ugniježdene naredbe izvode usmjeravanje igrača u igri pogađanja brojeva.

```
if (pokusaj > broj)
    cout << "Previsoko.";
else
    if (pokusaj < broj)
        cout << "Prenisko.";
    else
        if (pokusaj == broj)
            cout << "Pogodak!";
```

Uočite kako se kôd na prethodnoj stranici raširio preko stranice ostavljajući sve manje prostora. Koristimo slijedeću alternativu za uvlačenje nekoliko ugniježđenih if-else naredbi:

```
if (pokušaj > broj)
    cout << "Previsoko.";
else if (pokusaj < broj)
    cout << "Prenisko.";
else if (pokusaj == broj)
    cout << "Pogodak!";
```

Kada se uvjeti koji se testiraju u ugniježđenoj if-else naredbi uzajamno isključuju, zadnji if-else može se izostaviti.

Prethodni primjer se može napisati:

```
if (pokusaj > broj)
    cout << "Previsoko.";
else if (pokusaj < broj)
    cout << "Prenisko.";
else // (pokusaj == broj)
    cout << "Pogodak!";
```

Općenito, razgranata if-else naredba ima oblik:

```
if(Logicki_Izraz_1)
    Naredba_1
else if ( Logicki_Izraz_2)
    Naredba _2
...
else if (Logicki_Izraz_n)
    Naredba _n
else
    Naredba _Za_Sve_Ostale_Slučajeve
```

**Primjer programa: Izračun poreza**

Napiši program koji izračunava porez prema slijedećoj raspodjeli kamatne stope:

1. Nema poreza na prvih 1500 kn
2. Računa se 15% poreza na svaku kunu od 1501 do 2500
3. Računa se 25% poreza na svaku kunu od 2500 na više

```
//Program koji racuna porez na dohodak.
#include <iostream>
using namespace std;

// Program izracunava iznos poreza koji se izracunava kako slijedi:
// Od poreza je oslobođen dio place manji ili jednak 1500 kn.
// Na dio dohotka izmedju 1500 kn i 2500 kn obracunava se porez
// od 15%, a na dio veci od 2500 kn porez iznosi 25 %.

int main( )
{
    int dohodak;
    double porez;
    double porez_od_15_posto, porez_od_25_posto;
    cout << "Unesi dohodak (zaokruzen na kune): ";
    cin >> dohodak;

    if (dohodak <= 1500)
        porez = 0;
    else if ((dohodak > 1500) && (dohodak <= 2500))
        //15% od iznosa veceg od 1500 kn i manjeg ili jednagog od 2500
        porez = (0.15*(dohodak - 1500));
    else //dohodak > 2500 kn
    {
        //porez_od_15_posto = 15% od iznosa izmedju 1500 kn i 2500 kn.
        porez_od_15_posto = 0.15*1000;
        //porez_od_25_posto = 25% dijela dohotka koji prelazi 2500 kn.
        porez_od_25_posto = 0.25*(dohodak - 2500);
        porez = (porez_od_15_posto + porez_od_25_posto);
    }
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Dohodak = " << dohodak << " kn"<< endl
        << "Porez = " << porez << " kn" <<endl;
    return 0;
}
```

Uočimo da liniju

```
else if (( dohodak > 1500 && dohodak <= 2500))
```

možemo zamijeniti sa

```
else if (dohodak <= 2500)
```

Ova linija neće se izvršiti osim ako je već zaključeno da je dohodak > 1500, pa je dovoljno provjeriti da li je dohodak manji ili jednak 2500 kn.

## NAREDBA SWITCH

Naredba switch je alternativa za izvedbu višestrukog grananja.  
Primjer: Određuje se izlaz temeljen na ocjeni izraženoj slovom.

```
//Program ilustrira uporabu switch naredbe.
#include <iostream>
using namespace std;

int main( )
{
    char ocjena;

    cout << "Unesi ocjenu iz kolokvija i pritisni ENTER: ";
    cin >> ocjena;

    switch (ocjena)
    {
        case 'A':
            cout << "Odlicno. "
                  << "Ne moras polagati konacni ispit.\n";
            break;
        case 'B':
            cout << "Vrlo dobro. ";
            ocjena = 'A';
            cout << "Tvoja ocjena iz kolokvija je sada "
                  << ocjena << endl;
            break;
        case 'C':
            cout << "Prolazno.\n";
            break;
        case 'D':
        case 'F':
            cout << "Nije dobro. "
                  << "Idi uciti.\n";
            break;
```

```

default:
    cout << "Takva ocjena ne postoji.\n";
}
cout << "Kraj programa.\n";
return 0;
}

```

**Ocjene 'A', 'B' i 'C' imaju po jednu granu. Ocjene 'D' i 'F' koriste istu granu. Ako se unese nedozvoljena ocjena, koristi se grana s oznakom default.**

### Sintaksa naredbe switch

```

switch (izraz odluke)
{
    case Konstanta_1:
        Niz_naredbi_1
        break;
    case Konstanta _2:
        Niz_naredbi _2
        break;
    ...
    case Konstanta _n:
        Niz_naredbi _n
        break;
    default:
        Podrazumijevani_ Niz_naredbi
}

```

Izraz odluke u switch naredbi mora vratiti jedan od tipova: logička vrijednost, enum konstanta, cjelobrojni tip, znak.

Vraćena vrijednost se uspoređuje sa konstantnim vrijednostima nakon svake pojedine riječi case. Kada je vrijednost pronađena, izvršava se kod za taj slučaj.

Naredba break završava izvođenje naredbe switch. Izostavljanjem naredbe break izvršavanje se nastavlja sa naredbama iza slijedeće riječi case! Izostavljanje naredbe break može se iskoristiti da bi se za isti dio koda navelo više case vrijednosti (labela):

```

case 'A':
case 'a':
    cout << "Odlično.";
    break;

```

Isti kod se izvršava i za 'A' i za 'a'

Ako se ni u jednoj case labeli ne pojavljuje konstanta koja odgovara izrazu odluke, izvršavaju se naredbe iza riječi default. Ako nema default dijela, ništa se ne događa za vrijeme izvođenja naredbe switch. Dobra je navika koristiti default dio.

Ugniježdene if-else naredbe su podložnije greškama nego switch naredba. Naredba switch doprinosi jasnoći koda. Izbornik je prirodna primjena za naredbu switch. Slijedi primjer programa uz kojem se korisniku nudi izbornik:

```
// Izbornik
//Program koji daje informacije o zadacama.
#include <iostream>
using namespace std;

int main( )
{
    int izbor;

    do
    {
        cout << endl
            << "Odaberi 1 da bi pogledao slijedecu zadacu.\n"
            << "Odaberi 2 da bi pogledao sto si dobio za proslu zadacu.\n"
            << "Odaberi 3 za upute uz zadacu.\n"
            << "Odaberi 4 za izlaz iz ovog programa.\n"
            << "Unesi svoj izbor i pritisni ENTER: ";
        cin >> izbor;
        switch (izbor)
        {
            case 1:
                //kod za prikaz slijedece zadace na ekran.
                break;
            case 2:
                //kod koji trazi unos maticnog broja studenta i ispisuje
                //ocjenu.
                break;
            case 3:
                // kod za prikaz uputa uz slijedecu zadacu
                break;
            case 4:
                cout << "Kraj programa.\n";
                break;
            default:
                cout << "Nedozvoljeni unos.\n"
                    << "Ponovi unos.\n";
        }
    }
}
```



```

    }
    }while (izbor != 4);

    return 0;
}

```

Naredbe switch i if-else omogućavaju izvršavanje više naredbi u jednoj grani. Zbog više naredbi u granama switch ili if-else naredba teže se čitaju. To se može riješiti uporabom funkcijskih poziva (u gornji primjeru treba ih umetnuti kod komentara koda) umjesto niza naredbi – olakšava se čitanje switch ili if-else naredbe.

Svaka grana naredbe switch ili if-else je zaseban podzadatak. Ako je postupak u grani prejednostavan da bi se koristio funkcijski poziv, koristi se niz naredbi između vitičastih zagrada. Blok je dio koda zatvoren u vitičaste zagrade. Varijable deklarirane u bloku su lokalne za blok ili kažemo da imaju doseg unutar bloka (nisu vidljive izvan bloka). Imena varijabli deklariranih u bloku mogu se koristiti izvan bloka za druge varijable.

Blok naredbi je blok koji nije tijelo funkcije ili tijelo glavnog programa (funkcije main). Blokovi naredbi mogu biti ugniježđeni u drugim blokovima naredbi. Gniježđenje blokova naredbi otežava preglednost koda. Općenito je bolje kreirati funkcijske pozive nego ugniježđivati blokove naredbi.

Ako je isti identifikator (ime) deklariran kao varijabla u svakom od dva bloka, koji su ugniježđeni, tada su to dvije različite varijable s istim imenom. Jedna od varijabli postoji samo u unutarnjem bloku i ne može joj se pristupiti izvan unutarnjeg bloka. Druga varijabla postoji samo u vanjskom bloku i ne može joj se pristupiti u unutarnjem bloku.

```

// Blok sa lokalnom varijablom
//Program racuna cijenu kupljenih proizvoda sa i bez PDV-a .
#include <iostream>
using namespace std;
const double PDV = 0.22; //22% PDV-a.

int main( )
{
    char tip_prodaje;
    int broj_proizvoda;
    double jedinica_cijena, ukupno;
    cout << "Unesi jedinicu cijenu: ";
    cin >> jedinica_cijena;
    cout << "Unesi broj kupljenih proizvoda: ";
    cin >> broj_proizvoda;
    cout << "Unesi B ako je cijena bez PDV-a.\n"
        << "Unesi P ako je cijena sa PDV-om.\n"
        << "Zatim pritisni ENTER.\n";
}

```

```

cin >> tip_prodaje;
if ((tip_prodaje == 'B') || (tip_prodaje == 'b'))
{
    ukupno = jedinica_cijena * broj_proizvoda;
}
else if ((tip_prodaje == 'P') || (tip_prodaje == 'p'))
{
    double ukupno1;
    ukupno1 = jedinica_cijena * broj_proizvoda;
    ukupno = ukupno1 + ukupno1 * PDV;
}
else
{
    cout << "Pogresan unos!.\n";
}
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout << broj_proizvoda
    << " proizvoda po cijeni od "
    << jedinica_cijena << " kn" << endl;
cout << "ukupno iznosi " << ukupno << " kn" << endl;
if ((tip_prodaje == 'P') || (tip_prodaje == 'p'))
    cout << " ukljucujuci PDV.\n";
return 0;
}

```

## Priprema za kviz

1. Odredi izlaz za slijedeći kod:

```

{
    int x = 1;
    cout << x << endl;
    {
        cout << x << endl;
        int x = 2;
        cout << x << endl;
    }
    cout << x << endl;
}

```

2. Kakav izlaz daje kod ako x ima vrijednost 15?

```

if(x < 20)
    if(x < 10)

```

```
    cout << "manje od 10 ";  
else  
    cout << "veliki broj\n";
```

- A) nema izlaza, sintaktička greška      B) manje od 10  
C) ništa      D) veliki broj

3. Što nije u redu sa slijedećom switch naredbom?

```
int ans;  
cout << "Unesi y za yes a n za no\n";  
cin >> ans;  
switch (ans)  
{  
case 'y':  
case 'Y': cout << "Rekao si yes\n"; break;  
case 'n':  
case 'N': cout << "Rekao si no\n"; break;  
default: cout << "nedozvoljen odgovor\n";  
}
```

- A) ništa  
B) Nema break naredbi u 2 slučaja.  
C) ans je int.  
D) break; je nedozvoljena sintaksa.

4. Kakvu vrijednost ima x nakon izvršavanja koda?

```
int x = 10;  
if(x ++ > 10)  
{  
    x = 13;  
}
```

- A) 11      B) 9      C) 13      D) 10

5. Ako želimo da se petlja prestane izvršavati za  $x < 10$  i  $y > 3$ , koji je pravilan uvjet petlje?

- A)  $(x < 10 \ \&\& \ y > 3)$       B)  $(x > 10 \ || \ y < 3)$   
C)  $(x >= 10 \ || \ y <= 3)$       D)  $(x >= 10 \ \&\& \ y <= 3)$

6. Ako pišete do-while petlju koja provjerava da li korisnik unosi broj u rasponu od 2 do 5 (uključujući 2 i 5) i traži od korisnika ponovni unos sve dok ne unese dozvoljenu vrijednost, kako treba glasiti uvjet petlje?

- A)  $(2 > \text{number} \parallel \text{number} > 5)$
- B)  $(2 < 5 < \text{number})$
- C)  $(2 \leq \text{num} \leq 5)$
- D)  $(2 \leq \text{number} \&\& \text{number} \leq 5)$
- E)  $(2 > \text{number} \&\& \text{number} > 5)$

7. Što se ispisuje na ekran?

```
int i1=2, i2=4, i3=3, i4=5; double d1=2.3, d2=25.5;
/*1.1.*/ cout<<i1-i2-i3-i4/i2<<endl;_____
/*1.2.*/ cout<<sqrt(18%10+(i2+i3)%i1)<<endl; _____
/*1.3.*/ cout<<floor(d1) + d2 <<endl; _____
/*1.4.*/ cout<<-(-(-fabs(d2)))<<endl; _____
/*1.5.*/ cout<<d1+d2/i4<<endl; _____
```

8. Promotri slijedeći program:

```
#include<iostream.h>
int main() {
char z;
cin >> z;
if (z>='a' && z<='z')
if(z<'m') cout<<"+++";
else cout<<"---";
if (z<'A' || z>'Z' )
cout<<"???";
return 0;}
```

9. Kakav je izlaz programa, ako je unesena vrijednost za z:

z:'a' izlaz: \_\_\_\_\_  
z:'m' izlaz: \_\_\_\_\_  
z:'Z' izlaz: \_\_\_\_\_  
z:'8' izlaz: \_\_\_\_\_  
z:'A' izlaz: \_\_\_\_\_

10. Napisati program koji unosi cijele brojeve x i y. Program na temelju unesenih vrijednost izračunava vrijednost varijable z tipa double prema definiciji:

za  $0 < x < 20$ ,  $0 < y < 10$   $z = x - y$   
za  $x \geq 20$  ili  $x \leq 0$ ,  $y \geq 10$  ili  $y \leq 0$   $z = 2x + y$   
inače je  $z = x - 2y$

|       |      |      |       |      |       |
|-------|------|------|-------|------|-------|
| Ulaz  | 10 5 | 30 5 | -5 15 | 22 5 | -5 -5 |
| Izlaz | 5    | 20   | 5     | 12   | -15   |

11. Nadopuniti prethodni program tako da se isti izračun obavlja za niz parova cijelih brojeva sa oznakom kraja (999,999). Kada korisnik unese oznaku kraja dolazi do izlaska iz petlje i program završava s izvođenjem. (koristi petlju while)

## Bonus zadatak

1. Napišite program koji simulira igru **papir-kamen-škare**. Program traži da unesete svoje ime i u toku igre vas cijelo vrijeme oslovljava sa imenom. Igrač - korisnik igra protiv računala te unosi znak P, K ili S. U programu se generira odgovarajuća slučajna vrijednost P, K ili S. Program u nastavku zaključuje o ishodu igre i objavljuje pobjednika kao i razne poruke. Primjeri poruka (svakako budite maštovitiji):

Papir obavlja kamen.

Kamen razbija škare.

Škare režu papir.

ili

Nema pobjednika.

Omogućite igraču unos i malih i velikih znakova (koristiti funkciju toupper za pretvorbu). **Program treba omogućavati igraču da ponavlja igru dok god to želi.** Na kraju određenog broja igara (kada igrač odluči završiti sa igrom) treba ispisati statistiku pobjeđivanja (npr. računalo: 5 pobjeda, Marko: 3 pobjede) i ispisati prigodnu poruku.

Za generiranje slučajnog znaka koristite funkciju random(). Budući trebate generirati slučajnu vrijednost od tri moguće, neka se generira cijeli broj 1, 2 ili 3, pa ga zatim pretvorite u odgovarajući znak (P, K ili S). Generirani znak treba ispisati na ekran.

Budite kreativni, komentirajte opširno program i ne zaboravite zalijepiti na kraj programa kao komentar izlazni ekran sa primjerima igre – bar 20 interakcija sa konačnom statistikom!

## 8. VIŠE O PETLJAMA U C++-U

Ponovimo: Petlja je programska konstrukcija koja ponavlja naredbu ili niz naredbi određeni broj puta. **Tijelo petlje** čine naredbe(a) koje se ponavljaju. Svako ponavljanje petlje je jedna **iteracija**.

Dva osnovna pitanja koja se postavljaju prilikom oblikovanja petlje su:

1. Što uključiti u tijelo petlje?
2. Koliko puta se trebaju izvršiti naredbe u tijelu petlje (koliki je broj iteracija)

Ponovimo koja je osnovna razlika između *while* i *do-while* petlje: Kod izvođenja *while* petlje logički uvjet se provjerava na početku petlje prije izvođenja naredbi tijela petlje. Ako se logički izraz izračuna kao *false*, tada se naredbe u tijelu petlje uopće ne izvršavaju. Pri izvođenju *do-while* petlje tijelo petlje se najprije izvršava, a logički izraz se izračunava nakon izvršavanja naredbi u tijelu petlje. Na taj se način naredbe u tijelu *do-while* petlje izvršavaju najmanje jednom. Ako izuzmemo opisanu razliku između ovih dviju petlji, one se ponašaju vrlo slično. Nakon svakog ponavljanja petlje provjerava se logički uvjet: ako je njegova vrijednost *true*, petlja se ponavlja. U trenutku kada se vrijednost izraza promijeni u *false* dolazi do izlaska iz petlje, petlja završava s izvođenjem.

### Operatori inkrementiranja i dekrementiranja

Podsjetimo se kako smo koristili operator inkrementiranja na jednostavnom primjeru:

```
int broj = 41;
broj++; // varijabla broj se poveća za 1
cout << broj; // ispisuje se vrijednost 42
```

Operator inkrementiranja može se pojaviti u složenijem izrazu, npr:

```
int broj = 5;
int rezultat = 2 * (broj++);
```

Izraz *broj++* najprije vraća vrijednost varijable *broj* (5) koja se množi sa 2, a zatim se vrijednost varijable *broj* poveća za 1 (postaje 3). Ako operator inkrementiranja stavimo ispred varijable *broj*, tada se najprije poveća vrijednost varijable *broj* i ta nova vrijednost varijable se koristi u izrazu.

### **broj++ nasuprot ++broj**

(*broj++*) vraća trenutnu vrijednost varijable *broj*

Izraz u kojem se pojavljuje (*broj++*) koristi vrijednost varijable *broj* PRIJE povećavanja njezine vrijednosti.

(*++ broj*) povećava prvo *broj* i zatim vraća novu vrijednost varijable *broj*

Izraz u kojem se pojavljuje ( $++broj$ ) koristi vrijednost varijable broj NAKON povećanja njezine vrijednosti.

*broj* ima istu vrijednost i nakon *broj++* i nakon  $++broj$

Primjeri sa  $++$ :

```
int broj = 2;
int rezultat = 2 * (broj++);
cout << rezultat << " " << broj;
izlaz: 4 3
```

```
int broj = 2;
int rezultat = 2 * (++broj);
cout << rezultat << " " << broj;
izlaz: 6 3
```

Sve što je rečeno vrijedi i za operator dekrementiranja, samo što se vrijednost varijable smanjuje za 1 umjesto da se povećava.

Primjeri:

```
int broj = 8;
int rezultat = broj--;
cout << rezultat << " " << broj;
```

izlaz: 8 7

```
int broj = 8;
int rezultat = --broj;
cout << rezultat << " " << broj;
```

izlaz: 7 7

Dakle,  $broj--$  najprije vraća vrijednost varijable broj, a zatim smanjuje vrijednost varijable broj za 1.  $--broj$  najprije smanjuje broj, a zatim vraća vrijednost varijable broj. Opisane operatore  $++$  i  $--$  smijemo koristiti samo na jednoj varijabli. Slijedeći izrazi nisu dopušteni:

$(x+y)++$ ,  $--(x+y)$ ,  $5++$

Primjer programa:

```
//Operator inkrementiranja u izrazu
//Program za brojanje kalorija.
#include <iostream>
using namespace std;

int main( )
```

```

{
    int broj_podataka, brojac,
        k_cal, ukupno_kcal;

    cout << "Koliko ste razlicitih namirnica pojeli danas? ";
    cin >> broj_podataka;

    ukupno_kcal = 0;
    brojac = 1;
    cout << "Unesite broj kalorija za svaku od\n"
        << broj_podataka << " pojedenih namirnica:\n";

    while (brojac++ <= broj_podataka)
    {
        cin >> k_cal;
        ukupno_kcal = ukupno_kcal + k_cal;
    }

    cout << "Energetska vrijednost namirnica iznosi= "
        << ukupno_kcal << " kcal"<<endl;
    return 0;
}

```

#### **Dijalog s korisnikom:**

Koliko ste razlicitih namirnica pojeli danas? **5**

Unesite broj kalorija za svaku od

5 pojedenih namirnica:

**250**

**344**

**100**

**500**

**80**

Energetska vrijednost namirnica iznosi= 1274 kcal

## **NAREDBA FOR**

Naredba for (for petlja) je još jedan mehanizam petlje u jeziku C++. Ova naredba je namijenjena za uobičajene zadatke kao što je zbrajanje brojeva u danom rasponu. Numerički izračuni često započinju sa 1, pa se nastavljaju sa 2, 3 itd. dok se ne postigne neka konačna vrijednost. Npr. da bi izračunali zbroj prirodnih brojeva od 1 do 10 potrebno je da računalo izvrši slijedeću naredbu 10 puta, gdje je pri prvoj iteraciji vrijednost n jednaka 1 i u svakoj slijedećoj iteraciji se poveća za 1:

```
suma = suma + n;
```



Evo kako bi ovaj problem riješili pomoću while petlje:

```
suma = 0;
n = 1;
while(n <= 10) // zbrajaj brojeve od 1 - 10
{
    suma = suma + n;
    n++;
}
```

Iako smo problem uspješno riješili pomoću while petlje, namjena for petlje je upravo rješavanje ovakvih problema. Evo rješenja uz uporabu for petlje:

```
suma = 0;
for (n = 1; n <= 10; n++) // zbrajaj brojeve od 1 - 10
    suma = suma + n;
```

Obje petlje su sastavljene od istih dijelova koji su različito složeni. Obje počinju sa naredbom dodjele koja postavlja varijablu suma na 0. U oba slučaja ova dodjela je učinjena prije početka same petlje. Naredbe petlje se u oba slučaja sastoje od sljedećih dijelova:

```
n = 1; n <= 10; n++; i suma = suma + n;
```

U obje petlje ovi dijelovi imaju istu funkciju. For petlja je nešto kompaktnija od while petlje, a ima iste dijelove kao while petlja. For petlju ćemo obično koristiti za izvođenje petlji čije izvođenje nadzire jedna varijabla. U našem primjeru to je varijabla n.

## Sintaksa for petlje

Pogledajmo sintaksu for petlje:

```
for(Inicijalizacija;Logički_izraz;Izraz_promjene)
    Tijelo petlje
```

Primjer for petlje koja ispisuje cijele brojeve od 100 do 0:

```
for(broj = 100; broj >= 0; broj --)
    cout << broj<<endl;
```

Izlaz

```
100
99
.
.
.
```

Ekvivalentna while petlja (ekvivalentna sintaksa) navedena je u sljedećem primjeru:

```
Inicijalizacija;
while (Logički_izraz)
{
    Naredba petlje;
    Izraz_promjene;
}
```

Funkcionalno ekvivalentni primjer while petlje koja ispisuje cijele brojeve od 100 do 0:

```
broj = 100;
while (broj >= 0)
{
    cout << broj<<endl;
    broj--;
}
```

Pogledajmo kakvo značenje imaju izrazi koji se navode u zaglavlju for petlje (prvoj liniji petlje):

Inicijalizacija – izraz koji opisuje inicijalizaciju varijabli

Logički izraz – koristi se za provjeru završetka petlje

Izraz\_promjene – opisuje kako se kontrolna varijabla petlje mijenja nakon svake iteracije petlje

U primjeru:

for (n = 1; n <= 10; n++)

n = 1 ..... n je inicijalizirano na 1

n <= 10 ..... petlja se ponavlja sve dok vrijedi da je n manje ili jednako 10

n++ ..... n se povećava za 1 nakon svake iteracije petlje (nakon svakog izvršavanja naredbi u petlji)

Primijetimo da se u slučaju for petlje uvjet ispituje na početku petlje tako da se može dogoditi kao kod while petlje da se naredbe u tijelu petlje ne izvedu niti jednom.

U sljedećem primjeru kontrolna varijabla for petlje inicijalizirana je i pri tome deklarirana u okviru for petlje. Ovakvu inicijalizaciju obično koristimo ako tu varijablu ne koristimo izvan petlje. Ako međutim tu varijablu želimo koristiti i izvan petlje, tada je trebamo deklarirati izvan petlje (naravno, prije petlje). Različiti kompajleri različito tretiraju varijable deklarirane u for petlji – neki ih ne smatraju lokalnima. Provjerite kako

reagira vaš kompajler na takve deklaracije. Ako želite da vaš kod bude prenosiv, nemojte očekivati da će svaki prevoditelj tretirati takve varijable kao lokalne za petlju!

```
// Program izračunava sumu brojeva
// od 1 do 10.
#include <iostream>
using namespace std;

int main( )
{
    int suma = 0;
    // Uocite da je varijabla n lokalna
    // varijabla tijela for petlje prema ANSI C++ standardu!
    for (int n = 1; n <= 10; n++)
        suma = suma + n;

    cout << "Suma brojeva od 1 do 10 je "
         << suma << endl;
    return 0;
}
```

Primjer dijaloga:

Suma brojeva od 1 do 10 je 55

Deklaraciju varijabli u dijelu za inicijalizaciju kao u primjeru:

```
for (int n = 1; n <= 10; n++)
```

možemo tumačiti:

Kreiraj varijablu n tipa int i inicijaliziraj je sa 1

Nastavi s ponavljanjem tijela petlje sve dok je n <= 10

Povećaj n za 1 nakon svake iteracije

Inicijalizacija i izraz promjene for petlje mogu sadržavati i složenije izraze. Izraz promjene nije ograničen na zbrajanje i oduzimanje. Evo nekih primjera:

```
for (n = 1; n <= 10; n = n + 2) // n se nakon svake iteracije povećava za 2
```

```
for(n = 0 ; n > -100 ; n = n -7) // n se nakon svake iteracije smanjuje za 7
```

```
for(double x = pow(y,3.0); x > 2.0; x = sqrt(x) )
// x se inicijalizira na vrijednost
// koja se dobiva izračunavanjem izraza pow(y,3.0), i nakon svake iteracije se
// izračunava kao x = sqrt(x)
```

Tijelo for petlje može sadržavati više od jedne naredbe. Tada tijelo petlje moramo označiti vitičastim zagradama.

Primjer:

```
for(int broj = 1; broj >= 0; broj--)  
{  
    // naredbe tijela petlje  
}
```

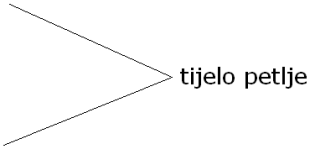
ilustrira sintaksu for petlje sa više naredbi u tijelu petlje.

Petlja for sa više naredbi u tijelu

---

Sintaksa

```
for (Inicijalizacija; Logički_izraz; Izraz_promjene)  
{  
    Naredba_1  
    Naredba_2  
    .  
    .  
    Naredba_zadnja  
}
```



Primjer

```
for (int broj = 100; broj >= 0; broj--)  
{  
    cout << broj  
        << "boca pive na polici.\n";  
    if (broj > 0)  
        cout << "Uzmi jednu i proslijedi dalje.";  
}
```

**Slika 15: Sintaksa for petlje**

### Problem s točka-zarezom u for petlji

Ako u programu pri odvajanju dviju naredbi stavimo dva puta točku-zarez to će se protumačiti kao prazna naredba. Ona se prevodi ali se ništa određeno ne događa.

```
cout << "Dobar" << endl;  
    ;  
cout << "dan" << endl;
```

Problem nastaje kada točku-zarez stavimo nakon okrugle zagrade zaglavlja for petlje, Na taj način smo kreirali tijelo petlje sa praznom naredbom:

Primjer:

```
for(int brojac = 1; brojac <= 10; brojac++);  
    cout << "Hello\n";
```

U gornjem primjeru ispisuje se jedan "Hello", ali ne u petlji! U ovom slučaju prazna naredba sačinjava tijelo petlje. Naredba `cout << "Hello\n";` ne nalazi se u tijelu petlje, već nakon petlje!

### Koju petlju koristiti?

Odabir tipa petlje treba napraviti na kraju oblikovanja postupka. Najprije oblikujemo petlju uporabom pseudokoda. Prevodimo pseudokod u C++. Na taj način olakšavamo izbor petlje. **While** petlja se obično koristi u slučajevima kada se može očekivati da se naredbe u tijelu petlje neće izvršiti niti jednom. **Do-while** petlja se koristi u slučajevima kada se očekuje da će se naredbe u tijelu petlje izvršiti bar jednom. **For** petlja se obično odabire kada se radi numerički izračun u kojem se koristi varijabla koja mijenja vrijednost za jednaki iznos u svakoj iteraciji petlje.

### Problemi: Neinicijalizirane varijable i beskonačne petlje

Problemi koje smo spomenuli vezano uz `while` i `do-while` petlju javljaju se i kod `for` petlje. Logičke greške često su posljedica neinicijalizacije varijabli prije petlje (npr. početna vrijednost sume ili produkta nije definirana pa se kao rezultat dobiva slučajna vrijednost). Zbog loše odabranog uvjeta ili logičke pogreške u petlji moguća je i pojava beskonačne petlje.

### NAREDBA BREAK

Ponekad je potrebno izaći iz petlje prije njenog završetka (koji ovisi o uvjetu izvođenja ili zaglavlju kod `for` petlje). Ako se u petlji detektira da je unos podataka nepravilan, što bi pokvarilo izračun, često je bolje prekinuti izvođenje petlje. Naredba `break` može se koristiti za izlaz iz petlje prije predviđenog završetka. Oprezno s ugniježđenim petljama! `Break` uzrokuje samo izlaz iz petlje u kojoj se pojavio.

### Priprema za kviz

1. Odredi izlaz pri izvršavanju koda:  

```
for(int brojac = 1; brojac < 5; brojac++)  
    cout << (2 * brojac) << " ";
```
2. Koji tip petlje bi bio najbolji izbor u slijedećim slučajevima?  
Zbrajanje članova niza  $1/2 + 1/3 + 1/4 + \dots + 1/10$ ?  
Čitanje liste ispitnih ocjena za jednog studenta?  
Testiranje funkcije za različite vrijednosti njezinih parametara (vrijednosti unosi korisnik)
3. Koliko puta se "Hi" ispiše na ekran?  

```
for(int i = 0; i < 14; i ++ );
```

```
cout << "Hi\n";
```

- A) 13            B) 1            C) 14            D) 15

4. Koja je konačna vrijednost varijable i (nakon petlje)?

```
int i;  
for(i = 0; i <= 4; i++)  
{  
    cout << i << endl;  
}
```

- A) 5            B) 4            C) 0            D) 3

5. Ako želimo da se petlja prestane izvršavati za  $x < 10$  i  $y > 3$ , koji je pravilan uvjet petlje?

- A)  $(x < 10 \ \&\& \ y > 3)$  B)  $(x > 10 \ || \ y < 3)$   
C)  $(x >= 10 \ || \ y <= 3)$  D)  $(x >= 10 \ \&\& \ y <= 3)$

6. Što nije u redu sa for petljom?

```
for(int i = 0; i < 10; i--)  
{  
    cout << "Hello\n";  
}
```

- A) jedan put se manje/više izvršava  
B) beskonačna petlja

- C) i nije inicijaliziran.  
D) ne može se koristiti for petlja u ovom slučaju

7. Kolike su vrijednosti varijabli i, j i k nakon izvođenja svake od petlji?

```
int i, j, k;
```

```
i=j=k=0;
```

```
for (i=-10; i<0; i++) { j=i; k++; }    i= _____ j= _____ k= _____
```

```
i=j=k=0;
```

```
for (i=10; i<0; i--) { j=i; k++; }    i= _____ j= _____ k= _____
```

```
i=j=k=0;
```

```
for (i=0; i<10; i++) ; { j=i; k++; }  i= _____ j= _____ k= _____
```

## 9. OBLIKOVANJE PETLJI

Oblikovanje petlje uključuje oblikovanje tijela petlje, naredbi inicijalizacije i uvjeta za završetak izvođenja petlje.

Bavit ćemo se sa dva uobičajena problema koji se rješavaju uporabom petlje i pokazati kako se oblikuju tri spomenuta elementa petlje.

### Petlje za izvedbu sume i umnoška

Rješavanje mnogih problema uključuje učitavanje liste brojeva i izračunavanje njihove sume. Ako je poznato koliko ima brojeva u listi, taj problem jednostavno rješavamo slijedećim postupkom.

Pseudokod:

```
zbroj = 0;
ponavljaj slijedeće ovoliko puta
    cin >> slijedeci;
    zbroj = zbroj + slijedeci;
kraj petlje
```

Varijabla zbroj se mora inicijalizirati prije tijela petlje! Vrijednost varijable *ovoliko\_puta* je broj brojeva koje treba zbrojiti. Zbroj se akumulira u varijabli zbroj. Ovaj pseudokod se može izvesti uporabom for petlje.

For petlja za računanje zbroja mogla bi izgledati ovako:

```
int zbroj = 0;
for(int brojac=1; brojac <= ovoliko_puta; brojac++)
{
    cin >> slijedeci;
    zbroj = zbroj + slijedeci;
}
```

Budući varijabla zbroj mora imati početnu vrijednost već pri prvom izvođenju petlje, ona mora biti inicijalizirana na neku vrijednost prije početka izvođenja petlje. Da bi odredili ispravnu početnu vrijednost za varijablu zbroj, razmislimo što zapravo želimo da se dogodi nakon prve iteracije. Nakon pribrajanja prvog broja iz liste zbroju, vrijednost varijable zbroj treba biti jednaka upravo tom broju. Zbog toga prije petlje varijabla zbroj treba biti postavljena na 0.

### Ponavljaj "ovoliko puta"

Pseudokod:

```
ponavljaj slijedeće "ovoliko puta":
```



### tijelo\_petlje

For petlja se obično koristi kada je unaprijed određen broj iteracija.

Ekvivalentna for petlja:

```
for(int brojac = 1; brojac <= ovoliko_puta; brojac++)  
    tijelo_petlje
```

### For petlja za umnožak

Formiranje umnoška (produkta) je vrlo slično računanju zbroja iz ranijeg primjera.

```
int umnozак = 1;  
for(int brojac=1; brojac <= ovoliko_puta; brojac++)  
{  
    cin >> slijedeci;  
    umnozак = umnozак * slijedeci;  
}
```

Umnožak mora biti inicijaliziran prije tijela petlje. Uoči da je umnožak inicijaliziran na 1, a ne na 0! Što bi se dogodilo da umnožak inicijaliziramo na 0?

## ZAVRŠAVANJE IZVOĐENJA PETLJE

Četiri su osnovna načina za završavanje petlje pri unosu podataka:

1. Unaprijed je poznat broj podataka koji se unose. Unosi ga korisnik ili se može izračunati.
2. Postavimo pitanje korisniku prije slijedeće iteracije. Pitamo korisnika prije svake iteracije da li se petlja treba još ponoviti.
3. Lista podataka završava s oznakom kraja. Koristi se zadana vrijednost za označavanje kraja liste. Pročitani su svi podaci.
4. Koristi se eof funkcija za označavanje kraja datoteke.

### Primjer 1: Poznat je broj podataka koji se unose

For petlje koje smo do sada vidjeli osiguravaju prirodnu izvedbu završetka petlje, na temelju metode unaprijed poznatog broja elemenata.

Primjer:

```
int n;  
cout << "Koliko ima podataka u listi?";  
cin >> n;  
for(int brojac = 1; brojac <= n; brojac++)  
{  
    int broj;  
    cout << "Unesi broj " << brojac;  
    cin >> broj;  
    cout << endl;  
    // Naredbe za obradu podatka broj  
}
```

### Primjer 2: Pitanje korisniku prije sljedeće iteracije

While petlja se koristi za izvedbu završetka petlje uporabom metode postavljanja pitanja korisniku prije iteracije.

```
zbroj = 0;
cout << "Da li ima podataka u listi (D/N)?";
char odg;
cin >> odg;
while (( odg == 'D') || (odg == 'd'))
{
    //naredbe za čitanje i obradu podataka
    cout << "Ima li još podataka(D/N)? ";
    cin >> odg;
}
```

### Primjer 3: Lista podataka završava s oznakom kraja

While petlja se obično koristi za završetak izvođenja petlje metodom koja koristi oznaku kraja liste (sentinel value).

```
cout << "Unesi listu nenegativnih cijelih brojeva.\n"
      << "Nakon elemenata liste unesi negativnu vrijednost.\n";
zbroj = 0;
cin >> broj;
while (broj > 0)
{
    //naredbe za obradu broja
    cin >> broj;
}
```

Uoči da je oznaka kraja učitana, ali nije obrađena.

### Primjer 4: Pročitani su svi podaci

While petlja se tipično koristi za izvedbu petlje koja završava s izvođenjem nakon što su učitani svi podaci.

```
ifstream ulazna_datoteka;
ulazna_datoteka.open("data.dat");
while (! ulazna_datoteka.eof() )
{
    //čitaju se i obrađuju podaci iz datoteke
}
```

Tri opća postupka za kontrolu izvođenja svake petlje su:

1. Petlja čije izvođenje kontrolira brojač
2. Kontrola ponavljanja petlje postavljanjem pitanja korisniku prije slijedeće iteracije
3. Izlaz iz petlje pri zadovoljenju uvjeta

Petlje koje kontrolira brojač su petlje za koje se broj ponavljanja može odrediti prije početka petlje. Npr. unos elemenata liste, kojemu prethodi unos veličine liste, je primjer petlje koju kontrolira brojač.

Petlje se mogu završiti kada je zadovoljen određeni uvjet. Varijablu koja mijenja vrijednost da bi se naznačilo da se dogodio neki događaj zovemo zastavica.

Primjeri izlaza uz zadovoljenje uvjeta za unos vrijednosti:

1. Lista koja završava s oznakom kraja
2. Iscrpljen je niz ulaznih vrijednosti

### **Problem pri izlazu iz petlje na temelju zastavice**

Pogledajmo petlju koja traži studenta koji ima 90 ili više bodova

```
int n = 1;
bodovi = racunaj_bodove(n);
while (bodovi < 90)
{
    n++;
    bodovi = racunaj_bodove(n);
}
cout << "Maticni broj " << n
      << " broj bodova " << bodovi << endl;
```

Problem: Petlja iz prethodnog primjera se možda neće završiti do kraja liste studenata, slučaju da ni jedan student nema bar 90 bodova. Dobra je ideja koristiti drugu zastavicu da bi osigurali da se petlja završi kad su obrađeni svi studenti. Slijedi poboljšano rješenje:

Slijedeći kod rješava problem koji se pojavljuje ako ni jedan student nema ocjenu 90 ili više:

```
int n=1;
bodovi = racunaj_bodove(n);
while ((bodovi < 90) && (n < broj_studenata))
{
    // isto kao prije
}
if (bodovi > 90)
    // isti izlaz kao prije
else
    cout << "Ni jedan student nema 90 ili više bodova.";
```

### **UGNIJEŽĐENE PETLJE**

Tijelo petlje može sadržavati bilo kakve naredbe, uključujući drugu petlju.

Kada su petlje ugniježdene sve iteracije unutarnje petlje se izvršavaju za svaku iteraciju vanjske petlje

Umjesto unutarnje petlje preporučuje se koristiti funkcijski poziv da bi program bio jasniji.

Primjer:

```
// Program koji za broj_studenata racuna prosjek ocjena za svakog studenta.
// broj_predmeta za svakog studenta je razlicit.
// Pretpostavimo da je broj predmeta za svakog studenta razlicit od 0,

#include <iostream>

using namespace std;
int main()
{
    int broj_studenata=1;
    int i,j,ocjena;
    double zbroj;
    char odg;
    cout<<"\n Program racuna prosjek ocjena za svakog studenta.";
    cout<<"\nUnesi broj studenata: ";
    cin>>broj_studenata;

    for(i=1;i<=broj_studenata;i++)
    {

        cout<<"\nUnesi ocjene za "<<i<<". studenta: ";
        zbroj=0;
        j=1;
        do
        {
            cout<<"\n"<<j<<". ocjena: ";
            cin>>ocjena;
            j++;
            zbroj+=ocjena;

            cout<<"\nZelis li unijeti slijedecu ocjenu? ";
            cin >>odg;
        }while(odg=='d' || odg=='D');
        cout<<"\nProsjek za "<<i<<". studenta je "<<zbroj/(j-1)<<endl;

    }

    return 0;
}
```

```
/*
Program racuna prosjek ocjena za svakog studenta.
Unesi broj studenata: 3
Unesi ocjene za 1. studenta:
```

```
1. ocjena: 2
Zelis li unijeti sljedeću ocjenu? d
2. ocjena: 5
Zelis li unijeti sljedeću ocjenu? d
3. ocjena: 3
Zelis li unijeti sljedeću ocjenu? n
Prosjek za 1. studenta je 3.33333
Unesi ocjene za 2. studenta:
1. ocjena: 5
Zelis li unijeti sljedeću ocjenu? d
2. ocjena: 2
Zelis li unijeti sljedeću ocjenu? n
Prosjek za 2. studenta je 3.5
Unesi ocjene za 3. studenta:
1. ocjena: 2
Zelis li unijeti sljedeću ocjenu? d
2. ocjena: 3
Zelis li unijeti sljedeću ocjenu? n
Prosjek za 3. studenta je 2.5
*/
```

## “DEBUGGING” PETLJI (TRAŽENJE GREŠAKA)

Uobičajene greške sa petljama uključuju slučajeve:

- Petlja se izvršava jedan put previše ili jedan put manje
- Beskonačne petlje su obično rezultat greške u logičkom izrazu koji kontrolira petlju

Rješavanje problema: Petlja se izvršava jedan put previše ili jedan put manje.

Provjeri usporedbu:

da li treba u izrazu pisati:

< ili <= ?

Provjeri da li se u inicijalizaciji koristila ispravna vrijednost

Da li petlja predviđa slučaj u kojem se tijelo petlje ne izvršava niti jednom?

Rješavanje problema: beskonačna petlja

Provjeri da li relacijski operator gleda na pravu stranu:

< ili > ?

Testiraj izraz sa < ili > radije nego sa (=)

Imajte na umu da su realne vrijednosti samo aproksimacije

Još savjeta:

- Provjerite da li je greška stvarno u petlji
- Pratite vrijednosti varijabli da bi uočili kako se varijable mijenjaju
- Praćenje varijable je promatranje njezine vrijednosti za vrijeme izvršavanja
- Mnogi sustavi uključuju alate za pomoć u traženju grešaka
- Naredba cout se može koristiti za praćenje vrijednosti varijabli

## Primjer “debugginga”

Slijedeći kod treba izračunati vrijednost varijable umnozak tako da sadrži umnožak brojeva od 2 do 5

```
int slijedeci = 2, umnozak = 1;
while (slijedeci < 5)
{
    slijedeci++;
    umnozak = umnozak * slijedeci;
}
```

Praćenje varijabli

Dodajemo privremeno naredbe cout za praćenje vrijednosti varijabli.

```
int slijedeci = 2, umnozak = 1;
while (slijedeci < 5)
{
    slijedeci++;
    umnozak = umnozak * slijedeci;
    cout << "slijedeci = " << slijedeci
    << "umnozak = " << umnozak
    << endl;
}
```

Prvo poboljšanje gornjeg koda:

Naredbe cout dodane u petlju nam pokazuju da se množenje sa 2 nikada ne događa. Riješimo problem premještanjem naredbe slijedeci++

```
int slijedeci = 2, umnozak = 1;
while (slijedeci < 5)
{
    umnozak = umnozak * slijedeci;
    slijedeci++;

    cout << "slijedeci = " << slijedeci
    << "umnozak = " << umnozak
    << endl;
}
```

Još postoji problem!

Drugo poboljšanje:

Ponovno testiranje petlje pokazuje da se sada petlja nikada ne množi sa 5. Ispravljamo grešku uporabom <= umjesto < na mjestu uspoređivanja.

```
int slijedeci = 2, umnozak = 1;
while (slijedeci <= 5)
{
```

```
        umnozak = umnozak * slijedeci;  
        slijedeci++;  
    }
```

### Upute za testiranje petlje

Svaki put kada se program mijenja, mora se ponovo testirati.  
Promjena jednog dijela programa može zahtijevati promjenu drugog dijela programa.

Svaka petlja treba se testirati barem za ulazne vrijednosti koje uzrokuju:

- 0 iteracija tijela petlje
- 1 iteraciju tijela petlje
- Za 1 manje od maksimalnog broja iteracija
- Maksimalni broj iteracija

Ponekad je učinkovitije odbaciti program sa greškama i početi pisati program iz početka. Novi program će biti lakše čitati. Novi program će vjerojatno sadržavati manje grešaka. Radnu verziju novog programa razvit ćete brže nego da ispravljate loš kod. Iskustvo u ispravljanju lošeg koda pomoći će vam u bržem oblikovanju novog programa.

## Priprema za kviz

1. Predvidi izlaz slijedećeg koda sa ugniježđenim petljama:

```
int n,m;
for(n = 1; n <= 5; n++)
    for(m = 5; m >= 1; m--)
        cout << n << " puta " << m
            << " = " << n*m << endl;
```

2. Odredi izlaz koji daje kod:

```
int n=5;
for(int i=1;i<=n;i++)
{
    for(int j=1;j<=n;j++)
    {
        cout<<i*j<<"\t";
    }
    cout<<endl;
}
```

3. Pronađi grešku u kodu. Izlaz programa treba biti isti kao u prethodnom primjeru.

```
int n=5;
for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)
    {
        cout<<i*j<<"\t";
    }
    cout<<"\n";
```

4. Što se ispisuje u slijedećoj petlji?

```
int i = 1;int sum; int j;
while(i <= 5)
{
    sum = 0;
    j = 1;
    while (j <= i)
    {
        sum = sum + j;
        j++;
    }
    cout << sum << ' ';
    i++;
}
```

5. Objasni problem beskonačne petlje.  
6. Koji su elementi dobrog stila programiranja i zašto je važan?



7. Navedi primjer i objasni vrednovanje izraza kratkim spojem.
8. Kako se može u programu realizirati višestruko grananje?
9. Navedi primjer ugnježđenih if-else naredbi. Objasni značenje.
10. Sintaksa i značenje naredbe **switch** (**konkretna primjer**).
11. Što je blok i zašto se koristi? Navedi primjer.
12. Objasni pravilo dosega za ugnježdene blokove.
13. Sintaksa i značenje naredbe **for** (objasniti na konkretnom vlastitom primjeru).
14. Objasni načine završavanja izvođenja petlje.
15. Objasni postupke za kontrolu izvođenja petlje.
16. Što možemo poduzeti da bi našli grešku u petlji?

## Bonus zadaci

1. Za broj ćemo reći da je gotovo prost ako nije prost, ali ispuštanjem točno jedne (neke) njegove znamenke dobijemo prost broj. Napišite program koji će unositi prirodan broj  $n$  i provjeravati je li uneseni broj gotovo prost.  
Test: Ulaz:543    Izlaz:Broj je gotovo prost.
2. Za broj ćemo reći da je gotovo jako prost ako nije prost, ali ispuštanjem bilo koje (svake) njegove znamenke dobijemo prosti broj. Napišite program koji će unositi prirodan broj  $n$  i provjeravati je li gotovo jako prost.  
Test: Ulaz:117    Izlaz:Broj je gotovo jako prost.
3. Za broj ćemo reći da je jako prost ako je prost, ali ako je i broj koji dobijemo ispuštanjem bilo koje njegove znamenke također prost. Napišite program koji će učitavati prirodan broj  $n$  i provjeravati da li je uneseni broj jako prost.  
Test: Ulaz:197    Izlaz:Broj je jako prost.
4. Za broj ćemo reći da je superprost ako je prost, ali ako i ispuštanjem bilo koliko posljednjih znamenaka dobijemo prosti broj. Napišite program koji će unositi prirodan broj  $n$  i provjeravati je li uneseni broj superprost.  
Test: Ulaz:3793    Izlaz:Broj je superprost.

## 10. UVOD U POLJA

Polje se koristi za obradu **kolekcije podataka istog tipa** kao što su primjerice lista imena i lista temperatura. Bavit ćemo se pitanjima definiranja i uporabe polja u jeziku C++ i upoznati ćemo temeljne algoritme za rad sa poljima.

### ZAŠTO TREBAMO POLJA?

Zamislimo da spremamo u memoriju rezultate (bodove) za 5 testova i rješavamo slijedeći problem. Želimo odrediti najveći broj bodova za sve testove i nakon toga odrediti razliku za svaki test posebno u usporedbi sa tim najvećim brojem bodova. Da bi mogli izvesti opisani postupak svih pet podataka treba biti istovremeno smješteno u memorijski prostor. U tu svrhu možemo deklarirati pet različitih varijabli za smještanje podataka. Međutim, problem nastaje kada se broj testova poveća na 100 testova ili možda 1000 testova. Kako ćemo dati imena tolikim različitim varijablama? Kako ćemo obraditi svaku od tih varijabli? Odgovor na to pitanje je odabir polja kao strukture podataka koja omogućava smještanje velikog broja podataka istog tipa uz omogućavanje direktnog pristupa podacima uporabom indeksiranih varijabli.

Polje se ponaša kao lista varijabli sa istim imenom i deklarira se u jednoj liniji koda. Prilikom pristupanja elementima polja ime polja je uvijek isti. Mijenja se samo indeks preko kojega pristupamo elementu polja.

Primjer: Deklariranje polja

Polje, s imenom bodovi, koje sadrži pet varijabli tipa int možemo deklarirati kao

```
int bodovi[ 5 ];
```

Zapravo smo time deklarirali 5 varijabli tipa int:

```
bodovi[0], bodovi[1], ... , bodovi[4] // indeksirane varijable
```

Vrijednost u zagradama zovemo indeks.

### INDEKSIRANE VARIJABLE

Varijable koje sačinjavaju polje zovemo **indeksirane varijable** ili **elementi polja**. Broj indeksiranih varijabli u polju je jednak veličini polja u deklaraciji – **deklarirana veličina polja**. Prilikom deklaracije polja veličina polja je dana u uglatim zagradama nakon imena polja. **Najveći indeks je za jedan manji od veličine, a indeks prvog elementa jednak je 0.** U gornjem primjeru indeksirane varijable su tipa int. Općenito, tip indeksirane varijable može biti bilo koji tip. Sve indeksirane varijable u polju uvijek su istog tipa, a tip indeksirane varijable zovemo **temeljni tip polja**. Indeksirana varijabla se može koristiti na mjestu gdje inače koristimo običnu varijablu istog tipa.

### UPORABA [ ] SA POLJIMA

U deklaraciji polja uglate zagrade [ ] sadrže veličinu polja kao što je ovo polje sa 5 cijelih brojeva:

```
int bodovi [5];
```

Kada referiramo na jednu od indeksiranih varijabli, [ ] sadrže broj preko kojeg identificiramo indeksiranu varijablu.

Primjer: bodovi[3] je indeksirana varijabla sa indeksom 3.

Vrijednost u [ ] može biti i cjelobrojni izraz sa rezultatom od 0 do (veličina -1).

## DODJELA VRIJEDNOSTI INDEKSIRANOJ VARIJABLI

Da bi dodijelili vrijednost indeksiranoj varijabli, koristimo operator dodjele:

```
int n = 2;  
bodovi[n + 1] = 99;
```

U ovom primjeru varijabli bodovi[3] se dodjeljuje vrijednost 99. Izrazi n+1 i 3 su u ovom slučaju ekvivalentni jer se izračunom n+1 dobiva vrijednost 3. Dakle, pristup indeksiranoj varijabli možemo izvesti preko indeksa čija se vrijednost izračunava, odnosno predstavljen je cjelobrojnim izrazom. U primjeru programa koji slijedi na taj se način bodovi učitavaju i obrađuju na način koji smo opisali.

```
//Program koristi polje  
//Učitavaju se postignuti bodovi za 5 natjecatelja i prikazuje koliko  
// se svaki pojedini rezultat razlikuje od najviseg postignutog broja bodova.  
#include <iostream>  
  
int main( )  
{  
    using namespace std;  
    int i, bodovi[5], max;  
  
    cout << "Unesi bodove za 5 natjecatelja:\n";  
    cin >> bodovi[0];  
    max = bodovi[0];  
    for (i = 1; i < 5; i++)  
    {  
        cin >> bodovi[i];  
        if (bodovi[i] > max)  
            max = bodovi[i];  
        //max je najveća vrijednost u polju bodovi[0],..., bodovi[i].  
    }  
  
    cout << "Najveći bodovi: " << max << endl  
        << "Bodovi i njihova razlika\n"  
        << "razlike od najvećeg broja bodova:\n";  
    for (i = 0; i < 5; i++)  
        cout << bodovi[i] << " se razlikuju za "  
            << (max - bodovi[i]) << endl;  
  
    return 0;  
}
```

```
/*  
Unesi bodove za 5 natjecatelja:  
3  
66  
44  
88  
100  
Najveci bodovi: 100  
Bodovi i njihova razlika  
razlike od najveceg broja bodova:  
3 se razlikuju za 97  
66 se razlikuju za 34  
44 se razlikuju za 56  
88 se razlikuju za 12  
100 se razlikuju za 0  
*/
```

## PETLJE I POLJA

Uobičajeni način za obilazak polja je obilazak izveden uz uporabu for petlje. For petlja je idealan odabir za manipulaciju za elementima petlje. U slijedećem primjeru izveden je ispis elemenata polja:

```
for (i = 0; i < 5; i++)  
{  
    cout << bodovi[i] << endl;  
}
```

Prilikom pristupanja elementima polja moramo biti pažljivi. Naime, ako polje ima primjerice deklariranu veličinu 5, element sa indeksom 5 nije definiran pa pristup takvom nepostojećem elementu može prouzročiti pad sustava ili bar logičku grešku pri izvođenju programa.

### Konstante i polja

Prilikom deklariranja veličine polja preporučuje se koristiti konstante koje smo prethodno deklarirali. Na taj način smo osigurali vrlo jednostavnu promjenu deklarirane veličine polja – dovoljno je samo promijeniti vrijednost konstante koja se vjerojatno koristi u deklaracijama na više mjesta u programu. Dakle, po potrebi možemo veličinu polja definirati za manje ili za više podataka. Primjerice, u slijedećem primjeru broj studenata za koji program radi promijenimo tako da promijenimo vrijednost konstante.

Primjer:

```
const int BROJ_STUDENATA = 50;  
    int bodovi[BROJ_STUDENATA];  
    ...  
    for ( i = 0; i < BROJ_STUDENATA; i++)  
        cout << bodovi[i] << " su manji za " << (max - bodovi[i]) << endl;
```

Većina prevoditelja ne dozvoljava uporabu varijable za deklaraciju veličine polja.

Primjer:

```
cout << "Unesi broj studenata: ";  
    cin >> broj;  
    int bodovi[broj];
```

Ovaj kod je ilegalan za većinu prevoditelja.

Općenita sintaksa deklaracije polja

Da bi deklarirali polje koristimo sintaksu:

```
Ime_tipa Ime_polja[deklarirana_veličina];
```

Ime\_tipa može biti bilo koji tip  
deklarirana\_veličina može biti konstanta

Kada je deklarirano, polje se sastoji od:  
indeksiranih varijabli Ime\_polja[0] do Ime\_polja[deklarirana\_velicina -1]

## POLJA I MEMORIJA RAČUNALA

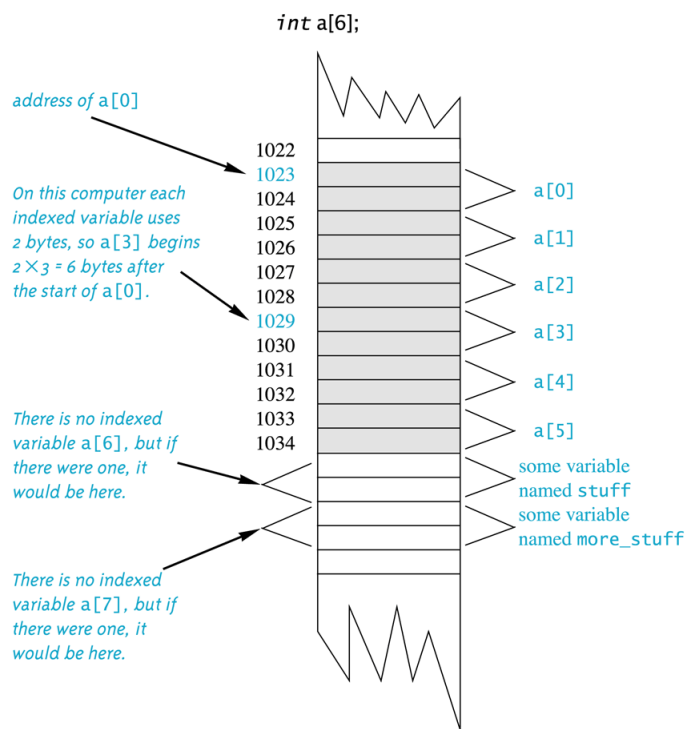
Memorija računala sastoji se od numeriranih lokacija – bajtova. Redni broj bajta je njegova adresa. Varijable jednostavnih tipova su smještene u uzastopnim bajtovima. Ukupan broj bajtova koji predstavlja veličinu memorijske lokacije ovisi o tipu varijable kojoj je ta memorijska lokacija dodijeljena. Adresa varijable je adresa prvog bajta njezine memorijske lokacije.

Promotrimo deklaraciju polja:

```
int a[6]
```

Ovom deklaracijom rezervira se memorija za 6 varijabli tipa int. Varijable se smještaju jedna iza druge. Pri tome se pamti adresa indeksirane varijable a[0]. Adrese ostalih indeksiranih varijabli se ne pamte. Da bi odredili adresu varijable a[3], počnemo od a[0] i pomičemo se za mem. prostor potreban za smještanje 3 cijela broja da bi došli do lokacije a[3].

## An Array in Memory



## INDEKS POLJA IZVAN DOZVOLJENOG RASPONA

Česta greška je uporaba nepostojećeg indeksa. Vrijednost indeksa za `int a[6]` su vrijednosti 0 do 5. Za nedozvoljene vrijednosti indeksa kažemo da su izvan raspona. Uporaba indeksa izvan raspona ne uzrokuje javljanje greške, već dolazi do pada sustava ili do logičke greške!

## Problemi s rasponom indeksa

Ako deklaracija glasi:

```
int a[6];
```

a neka varijabla je deklarirana: `int i = 7;`  
izvođenje naredbe `a[i] = 238;` uzrokuje slijedeće:

Računa se adresa ilegalne varijable `a[7]` (Na toj adresi je možda pohranjena neka druga varijabla)

Vrijednost 238 se sprema na toj adresi

Nema upozorenja!

## Inicijalizacija polja

Da bi inicijalizirali polje prilikom deklaracije u vitičaste zagrade navodimo vrijednosti indeksiranih varijabli odvojene zarezom.

Primjer:

```
int djeca[3] = { 2, 12, 1 };
```

je ekvivalentno:

```
int djeca[3];  
djeca[0] = 2;  
djeca[1] = 12;  
djeca[2] = 1;
```

## Podrazumijevane (default) vrijednosti

Ako je u listi inicijalizacije navedeno premalo vrijednosti inicijaliziraju se prve indeksirane varijable za koje su vrijednosti navedene, a ostale indeksirane varijable se inicijaliziraju na 0 temeljnog tipa.

Primjer:

```
int a[10] = { 5, 5 };
```

inicijalizira `a[0]` i `a[1]` na 5 i `a[2]` do `a[9]` na 0.

## Neinicijalizirana polja

Ako ne inicijaliziramo polje, neki prevoditelji će postaviti vrijednosti polja na 0 temeljnog tipa, ali s time ne možemo sigurno računati!

## Djelomično popunjena polja

Potrebna veličina polja se mijenja i često se razlikuje u uzastopnim pokretanjima programa. Obično nije poznata dok se program piše. Rješenje za ovaj problem je da deklariramo polje veličine koja predstavlja najveću potrebnu veličinu polja, a u programu radimo s djelomično popunjenim poljem.

Primjerice, funkcije koje rade s poljima ne moraju znati deklariranu veličinu polja, već samo koliko elemenata je pohranjeno u polju

## STRUKTURE

U jeziku C++ je uporabu struktura zamijenila uporaba klasa odnosno objekata kao instanci klasa. Budući u našim razmatranjima koristimo proceduralni, a ne objektno orijentirani pristup izradi programa, osnovni način za grupiranje podataka različitih tipova u fizičku cjelinu koji ćemo koristiti su strukture.

### Specificiranje strukture

Struktura nam omogućava da u fizičku cjelinu objedinimo podatke različitih tipova koji primjerice predstavljaju podatke o nekoj osobi, proizvodu ili pojavi. Podaci o primjerice studentu mogli bi se sastojati od imena, prezimena i matičnog broja studenta. U slijedećem primjeru strukturu sa imenom *dio* sačinjavaju komponente *broj\_modela*, *broj\_dijela* i *cijena*.

```
struct dio
{
    int broj_modela;
    broj_dijela;
    float cijena;
}
```

Varijable tipa strukture dio deklariramo na isti način kao i varijable ostalih tipova:

```
dio p1,p2;
```

ili tako da odmah uz definiciju tipa strukture navedemo imena varijabli:

```
struct dio
{
    int broj_modela;
    broj_dijela;
    float cijena;
} p1, p2;
```

### Pristupanje elementima strukture

Pristup elementima strukture izvodimo uporabom točkastog operatora:

```
cout << "Unesi broj modela:"
cin >>p1.broj_modela;          // pristup komponenti strukture broj_modela
```



```
cout << "Unesi broj dijela:"  
cin >> p1.broj_dijela;  
cout << "Unesi cijenu:"  
cin >> p1.cijena;
```

### Inicijalizacija varijable tipa strukture

Varijablu tipa strukture inicijaliziramo na slijedeći način:

```
dio p1={5656, 889, 65.4}
```

U ovom primjeru komponente strukture *broj\_modela*, *broj\_dijela* i *cijena* su postavljene na početne vrijednosti 5656, 889 i 65.4 .

## PRIMJERI PROGRAMA SA POLJIMA

### SEKVENCIJALNO PRETRAŽIVANJE LISTE

Uz uporabu polja upoznat ćemo i dva često korištena algoritma. Prvi je algoritam sekvencijalnog pretraživanja.

U problemima iz stvarnog svijeta često se susrećemo sa problemom pretraživanja. Iako je danas redovita uporaba telefonskog imenika s Interneta, pokušajmo se zamisliti u poslu traženja telefonskog broja u telefonskom imeniku oblika papirnatih knjige. Zamislimo dodatno da podaci u telefonskom imeniku nisu sortirani kako smo navikli. Pretraživanje ovakvih nesortiranih podataka možemo izvesti jedino sekvencijalnim pretraživanjem. Dakle, za zadani ključ, a to je primjerice prezime vlasnika telefona za kojeg tražimo telefonski broj u imeniku, tražimo ključ u imeniku. Ključ tražimo tako da u svakom koraku ovog pretraživanja usporedimo ključ sa elementom na tekućoj poziciji u polju. Kada smo pronašli ključ uz njega ćemo pronaći traženi telefonski broj. Opisani postupak baš i nije sličan onome koji smo navikli izvoditi sa telefonskim imenikom, jer je on obično sortiran pa zapravo ne primjenjujemo sekvencijalno pretraživanje, već primjenjujemo napredniji postupak. No sekvencijalno pretraživanje je jedino koje možemo izvesti za nesortiranu listu podataka.

Da bi podatke učinkovito pretraživali oni moraju biti na određeni način pripremljeni – sortirani. Pogledajmo za sada korake sekvencijalnog pretraživanja na nesortiranoj listi podataka.

```
//Sekvencijalno pretraživanje  
  
#include<iostream.h>  
  
void main()  
{  
  
float polje[12]={4,21,36,14,62,91,8,22,7,81,77,10};  
  
int tekucaPozicija,zadnji=12,nadjenPozicija;
```

```
float trazen_element;
bool nadjen;

cout<<"Unesi trazen element: ";
cin>>trazen_element;           //Unos ključa
                                // za pretraživanje

//Prvi pristup
cout<<endl<<"PRVI PRISTUP"<<endl;
tekucaPozicija=0;

while (tekucaPozicija < zadnji && trazen_element != polje[tekucaPozicija])
// listu pretražujemo dok nismo došli do kraja liste
// i dok nismo našli traženi element
{
    tekucaPozicija++;           // promjena pozicije u polju
                                // koju provjeravamo
}

nadjenPozicija = tekucaPozicija; // pamtimo poziciju kod koje je došlo do izlaza iz
                                // petlje

if (trazen_element == polje[tekucaPozicija])
    nadjen = true;              // ključ je pronađen
else
    nadjen = false;             // ključ nije pronađen

if (nadjen) cout<<"Element je nadjen na poziciji "<<nadjenPozicija<<endl;
else cout<<"Element nije pronađen. "<<endl;

//Drugi pristup
cout<<endl<<"DRUGI PRISTUP"<<endl;
tekucaPozicija=0;

while (tekucaPozicija < zadnji && trazen_element != polje[tekucaPozicija])
{
    tekucaPozicija++;
}

if (trazen_element == polje[tekucaPozicija]) // da li se ključ nalazi na poziciji
    nadjenPozicija=tekucaPozicija;           // kod koje je došlo do izlaza iz petlje?
else
    nadjenPozicija=-1;                       // element nije pronađen

if (nadjenPozicija!=-1) cout<<"Element je nadjen na poziciji "<<nadjenPozicija<<endl;
else cout<<"Element nije pronađen. "<<endl;

}
```

Primjer interakcije sa programom:

Unesi traženi element: 62

PRVI PRISTUP

Element je nadjen na poziciji 4

DRUGI PRISTUP

Element je nadjen na poziciji 4

Press any key to continue

## BINARNO PRETRAŽIVANJE LISTE

Pretraživanje koje je po učinkovitosti najbliže našem pretraživanju sortiranog telefonskog imenika je binarno pretraživanje liste. Binarno pretraživanje u prvom koraku ključ odabire u sredini liste (pozicija se izračunava kao aritmetička sredina broja elemenata početne liste zaokružena na prvi manji cijeli broj). Ako je ključ jednak vrijednosti liste na danoj poziciji zaključak je da je element pronađen. Ako je ključ manji od vrijednosti na danoj poziciji, pretraživanje nastavljamo u donjoj polovici liste, inače u gornjoj. Nastavljamo sa postupkom sve dok se lista u kojoj obavljam pretraživanje ne svede na listu sa jednim elementom. Ako niti to nije traženi element zaključujemo da elementa u listi nema.

Primjer binarnog pretraživanja

```
// *****
// polja_binar.cpp
// BISEKCIJA (BINARNO PRETRAZIVANJE):
// Napisati program koji u rastucem nizu a[1], a[2], ..., a[n] odredjuje
// indeks elementa koji je jednak broju, pomocu binarnog pretrazivanja. Ako
// ne postoji element jednak b ispisuje se 0.
// *****

#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
    typedef float lista[50];
    int gornja,donja,sredina,n,bin_pret,i;
    bool nadjen;
    float b;
    lista a;

    cout<<"Unesi velicinu liste: ";
    cin>>n;
    cout<<endl<<"Unesi elemente liste u rastucem poretku: "<<endl;
    for (i=1;i<=n;i++)
        cin>>a[i];
    cout<<endl<<"Unesi trazeni element: ";
```

```

cin>>b;
donja=1; //inicijalizacija varijabli
gornja=n;
nadjen=false;
while (donja<=gornja && !nadjen)
{
    sredina=(donja+gornja)/2; //sredina intervala
    if (b==a[sredina])
        nadjen=true; //element je nadjen
    else
        if (b<a[sredina])
            gornja=sredina-1; //nova gornja granica
        else donja=sredina+1; //nova donja granica
};
if (!nadjen)
    bin_pret=0; //element nije nadjen
    else bin_pret=sredina; //element je nadjen
cout<<endl<<"Indeks elementa: "<<bin_pret<<endl;
}

//Test podaci:
//1,3,4,6,11,13,17,19,23,29.
//b=19

//donja    gornja
//1.   1      10   (1+10)div2=5   a[5]<b
//2.   6      10   (6+10)div2=8   a[8]=b      }

```

## SORTIRANJE U VALOVIMA – "BUBBLE SORT"

Preduvjet binarnog pretraživanja je da je lista sortirana. Upoznat ćemo jedan postupak sortiranja – sortiranje u valovima (bubble sort). Elementi liste se sortiraju u koracima ili valovima. Sortiranje se izvodi sa ciljem da elementi liste bude u konačnom uzlaznom ili silaznom poretku. Pretpostavimo da želimo da elementi liste budu sortirani uzlazno. U svakom koraku algoritma redom uspoređujemo po dva susjedna elementa liste i ako je potrebno zamijenimo njihove vrijednosti. U svakom koraku algoritma jedan element liste "ispliva" kao mjehurić na svoju konačnu poziciju i više ga ne uspoređujemo u daljnjim koracima algoritma. Koraci algoritma se ponavljaju sve dok se još događaju zamjene vrijednosti elemenata.

Primjer:

Sortiranje u valovima (bubble sort)

|     |    |           |           |           |           |           |
|-----|----|-----------|-----------|-----------|-----------|-----------|
|     | 58 | 35        | 1         | 67        | 12        | 25        |
| (1) | 35 | 1         | <u>58</u> | 12        | 25        | <u>67</u> |
| (2) | 1  | <u>35</u> | 12        | 25        | <u>58</u> | 67        |
| (3) | 1  | 12        | 25        | <u>35</u> | 58        | 67        |
| (4) | 1  | 12        | 25        | 35        | 58        | 67        |

| Korak | Razmjene                                      |
|-------|-----------------------------------------------|
| 1.    | 58 i 35, 58 i 167 i 1267 i 25                 |
| 2.    | 35 i 1, 58 i 12, 58 i 25                      |
| 3.    | 35 i 12, 35 i 25                              |
| 4.    | ne dolazi do zamjena i sortiranje je završeno |

U slijedećem primjeru podaci o studentima se sortiraju uporabom sortiranja u valovima prema broju indeksa.

```

//*****
// student.cpp
// Program koji za unesene podatke o studentima
// ispisuje podatke sortirane prema broju indeksa (bubble sort).
//*****

#include<iostream.h>
#include<iomanip.h>

void main()
{

int i=0,b=0;
char izbor;
bool r;
struct slog1
    {
        char ime[15], prezime[15];int broj_indeksa;
    };

slog1 s[15];

do
    {
        cout<<"Unesi podatke za "<<i+1<<" studenta: ";
        if(i!=0)
            cin.ignore(10,'\n');
        cout<<endl<<"Ime: ";
        cin.get(s[i].ime,15);

        cin.ignore(10,'\n');
        cout<<endl<<"Prezime: ";
        cin.get(s[i].prezime,15);

        cout<<endl<<"broj indeksa: ";
        cin>>s[i].broj_indeksa;

        cout<<endl<<"Jos studenata? (d/n) : ";
    }
}
```

```
        cin>>izbor;
        i++;
    }while(izbor!='n');

    slog1 t;

    do
    {
        r=false;
        b++;
        for (int ii=0;ii<i-b;ii++)
        if (s[ii].broj_indeksa > s[ii+1].broj_indeksa)
        {
            t=s[ii];
            s[ii]=s[ii+1];
            s[ii+1]=t;
            r=true;
        }

        cout<<endl;
        for (int k= 0;k<i;k++)
            cout<<setw(15)<<s[k].broj_indeksa;
    }while (r);

    for(int j=0;j<i;j++)
    {
        cout<<endl<<"Podaci za "<<j+1<<". studenta: "<<endl;
        cout<<setw(10)<<"Ime: "<<setw(15)<<s[j].ime;
        cout<<setw(15)<<"Prezime: "<<setw(15)<<s[j].prezime;
        cout<<setw(20)<<"Broj indeksa: "<<setw(5)<<s[j].broj_indeksa;
    }

    cout<<endl;
}
```

Primjer interakcije s programom:

Unesi podatke za 1. studenta:

Ime: Marko

Prezime: Ilic

broj indeksa: 687

Jos studenata? (d/n) : d

Unesi podatke za 2. studenta:

Ime: Iva

Prezime: Peric

broj indeksa: 115

Jos studenata? (d/n) : d

Unesi podatke za 3. studenta:

Ime: Ivo

Prezime: Bilic

broj indeksa: 444

Jos studenata? (d/n) : n

115 444 687

115 444 687

Podaci za 1. studenta:

Ime: Iva Prezime: Peric Broj indeksa: 115

Podaci za 2. studenta:

Ime: Ivo Prezime: Bilic Broj indeksa: 444

Podaci za 3. studenta:

Ime: Marko Prezime: Ilic Broj indeksa: 687

## ZNAKOVNI NIZ

Znakovni nizovi su jednodimenzionalna znakovna polja koja služe za pohranjivanje tekstova:

```
char padobran[]="Faust Vrancic";
```

Na kraj niza automatski se postavlja nul-znak (engl. *null-character*).

Program daje ispis koji dokazuje postojanje nul-znaka:

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

int main()
{
    char padobran[]="Faust Vrancic";

    for(int i=0;i<=13;i++)
    {
        cout<<padobran[i];
        cout<<" "<<static_cast<int>(padobran[i])<<endl;
    }
    return 0;
}
```

```
/*  
F 70  
a 97  
u 117  
s 115  
t 116  
32  
V 86  
r 114  
a 97  
n 110  
c 99  
i 105  
c 99  
0
```

### Inicijalizacija znakovnog niza

Znakovni niz možemo inicijalizirati na dva načina:

```
char Niz[] = "Faust Vrančić";
```

*ili:*

```
char Niz[] =
```

```
{'F','a','u','s','t',' ','V','r','a','n','ć','i','ć','\0'}; // moramo voditi računa o nul znaku
```

Ispisati znakovni niz možemo na dva načina:

```
cout<<Niz<<endl;
```

*ili:*

```
for(int i=0;i<13;i++)  
    cout<<Niz[i];  
cout<<endl;
```

Učitavanje znakovnog niza možemo izvesti na slijedeći način:

```
char tekst[80];  
cin>>tekst;  
cout<<tekst;
```

Ovaj način učitavanja znakovnog niza koristimo za nizove u kojima nema razmaka. Ako unesemo niz koji sadrži prazninu, ispisati će se samo znakovi do prve praznine.

Za znakovne nizove koji sadrže prazninu koristimo slijedeći unos:

```
#include<iostream.h>  
int main() {  
    char tekst[80];  
    cout<<endl<<"Unesi tekst:"<<endl;  
    cin.get(tekst,sizeof(tekst));
```



```
        cout<<endl<<"Ispis:"<<endl
            <<tekst<<endl;
        return 0;
    }
```

Nul znak se u ovom slučaju dodaje automatski.

Funkcije za rad sa znakovnim nizovima koristimo iz zaglavne datoteke string.h

Na raspolaganju su nam slijedeće funkcije za rad sa znakovnim nozom:

### **Funkcija za određivanje duljine niza znakova**

strlen()

```
char s1[]="Pjesme";
cout<<"Duljina niza s1="
    <<strlen(s1);
```

Ispis:

Duljina niza s1=6

### **Funkcija za kopiranje znakovnih nizova**

```
char izvorni[]="ovo je izvorni niz";
char odredisni[80];
```

```
strcpy(odredisni,izvorni);           //kopira sadrzaj izvornog niza u odredisni
```

### **Funkcija za povezivanje nizova znakova**

Primjer1 (GRESKA!!):

```
char s1[]="Bolje bi bilo "; // rezultat se
char s2[]="da pozuris!!"; // sprema u s1!!
strcat(s1,s2); // Nema dovoljno mjesta!!
```

Primjer2 (ISPRAVNO!!):

```
char s1[80]="Bolje bi bilo "; // rezultat se
char s2[]="da pozuris!!"; // sprema u s1!!
strcat(s1,s2);
```

### **Funkcija za uspoređivanje nizova znakova**

```
char ime[]="Mazda";
n1=strcmp(ime,"Audi"); //vraća 1
n1=strcmp(ime,"Mazda"); // vraća 0
n1=strcmp(ime,"Opel"); // vraća -1
```

Slijedi program u kojem se određuje duljina najkraće riječi u rečenici.

```

//*****
// U danom nizu znakova treba odrediti duljinu najkrace rijeci.
//*****

#include <iostream>
#include <string.h>
using namespace std;

void main()
{
    int min,i,k;
    char s1[50];
    cout<<endl<<"Unesi niz: ";
    cin.get(s1,50);                // unos rečenice
    min=strlen(s1);                // min se postavlja na duljinu rečenice
    k=0;                           // (ukupan broj znakova u rečenici)
    i=0;
    strcat(s1," ");                // dodajemo prazninu na kraj rečenicezbog provjere
    while(s1[i]!='\0')             // provjera kraja niza znakova
    {
        if (s1[i] != ' ')          // provjera kraja riječi
            k++;
        else
        {
            if (k>0)
                if (min>k) min=k;    // da li je riječ kraća?
                k=0;
        }

        i++;
    }
    cout<<"Najkraca rijec ima duljinu: "<<min<<endl;
}

```

Primjer interakcije s programom:  
Unesi niz: Danas je lijepo vrijeme  
Najkraca rijec ima duljinu: 2  
Press any key to continue

## Priprema za kviz

## 11. PROCEDURALNA APSTRAKCIJA I FUNKCIJE KOJE VRAĆAJU VRIJEDNOST

U programu obično možemo uočiti dijelove postupka kao što su ulaz podataka, obrada podataka i prikaz rezultata. Svaki od ovih dijelova možemo kodirati i zapisati u izdvojenu fizičku cjelinu koju u jeziku C++ zovemo **funkcija**. Najprije ćemo upoznati funkcije koje vraćaju samo jednu vrijednost.

### OBLIKOVANJE OD VRHA PREMA DNU ("TOP DOWN")

Dobar način oblikovanja algoritma je raščlanjivanje složenog problema koji rješavamo na manje potprobleme sve dok raščlanivanjem ne dobijemo potprobleme čije je rješenje jednostavno izvesti u jeziku C++. Ovaj postupak zovemo **oblikovanje algoritma "od vrha prema dnu"**. Ovu metodu zovemo i "podijeli i savladaj". Kada smo oblikovali algoritam prevodimo ga u programski jezik.

Za svaki potproblem oblikujemo podpostupak za njegovo rješavanje. Struktura programa "od vrha prema dnu" olakšava razumijevanje programa, pisanje programa, mijenjanje i testiranje programa.

#### Top-down oblikovanje:

Raščlanimo algoritam u potpostupke

Raščlanimo svaki potpostupak u manje potpostupke

Postupak raščlambe nastavljamo sve dok manji potpostupci ne postanu trivijalni za izvedbu u programskom jeziku

### Prednosti top-down oblikovanja

Uz uporabu potpostupaka, ili funkcija u jeziku C++, programi se lakše

- razumiju
- mijenjaju
- pišu
- testiraju
- razvijaju timski
- lakše se ispravljaju greške

### FUNKCIJE KOJE DEFINIRA PROGRAMER

C++ dolazi s bibliotekama standardnih funkcija koje možemo koristiti u programima. Uporabu standardnih funkcija upoznali smo u 2. poglavlju. Ovdje se bavimo funkcijama koje definira sam programer – korisničkim funkcijama.

Pogledajmo primjer programa sa funkcijom. Program učitava podatke o broju proizvoda i jediničnoj cijeni proizvoda, te izračunava i ispisuje ukupnu cijenu proizvoda sa PDV-om.

```
#include <iostream>
```

```
using namespace std;

double ukupna_cijena(int broj_par, double cijena_par);
// Računa ukupnu cijenu, uključujući 22% PDV,
//za broj_par proizvoda po jediničnoj cijeni cijena_par .

int main( )
{
    double cijena, racun;
    int broj;

    cout << "Unesi broj kupljenih proizvoda: ";
    cin >> broj;
    cout << "Unesi cijenu jednog proizvoda: ";
    cin >> cijena;
    racun = ukupna_cijena(broj, cijena);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << broj << " proizvoda po "
        << "cijeni " << cijena << " kn svaki.\n"
        << "Ukupno sa porezom, iznosi " << racun
        << " kuna."<< endl;
    return 0;
}

double ukupna_cijena(int broj_par, double cijena_par)
{
    const double PDV = 0.22; //22% PDV
    double iznos1;

    iznos1 = cijena_par * broj_par;
    return (iznos1 + iznos1*PDV);
}
```

**Deklaracija funkcije**

**Poziv funkcije**

**zaglavlje funkcije**

**Tijelo funkcije**

**Definicija funkcije**

Unesi broj kupljenih proizvoda: 3  
 Unesi cijenu jednog proizvoda: 5  
 3 proizvoda po cijeni 5.00 kn svaki.  
 Ukupno sa porezom, iznosi 18.30 kuna.

Funkcija u gornjem primjeru zove se `ukupna_cijena`. Ima dva parametra – cijenu za jedan proizvod i broj kupljenih proizvoda. Funkcija vraća ukupnu cijenu uključujući PDV za određeni broj proizvoda i jediničnu cijenu. Funkcija se poziva na isti način kao i standardne funkcije iz poglavlja 2. Međutim, sada programer mora brinuti i o definiciji funkcije. Opis funkcije dan je u dva dijela koje zovemo deklaracija funkcije i definicija funkcije. Deklaracija funkcije (ili prototip funkcije) opisuje kako se funkcija poziva. C++ zahtijeva da se prije poziva funkcije u kodu navedu ili definicija funkcije ili deklaracija funkcije. Deklaracija funkcije navedena je prije glavne funkcije:

### Deklaracija funkcije(ili prototip funkcije)

- Pokazuje kako se funkcija poziva
- Mora se pojaviti u kodu prije poziva funkcije
- 

Sintaksa:

```
Vraćeni_tip Ime_Funkcije(Lista_Parametara);  
// Komentar koji opisuje što funkcija radi
```

### Definicija funkcije

- Opisuje **kako** funkcija obavlja svoj zadatak
- Može se pojaviti prije ili nakon poziva funkcije

Sintaksa:

```
Vraćeni_tip Ime_Funkcije(Lista_Parametara)  
{  
    //kod potreban za rad funkcije  
}
```

U deklaraciji se navodi **ime funkcije, tip funkcije, tipovi formalnih parametara i imena formalnih parametara**. Formalni parametri preuzimaju vrijednosti stvarnih parametara pri pozivu funkcije. Imena formalnih parametara mogu biti bilo kakva pravilno odabrana imena. Komentar koji pišemo uz deklaraciju objašnjava što funkcija radi, sadrži precizan opis formalnih parametara i izlaznu vrijednost (značenje i tip).

Primjer deklaracije iz gornjeg programa:

```
double ukupna_cijena(int broj_par, double cijena_par);  
// Računa ukupnu cijenu, uključujući 22% PDV ,  
//za broj_par proizvoda po jediničnoj cijeni cijena_par
```

Definicija funkcije osigurava iste informacije kao i deklaracija. Opisuje kako funkcija izvršava svoj zadatak

Primjer definicije funkcije iz gornjeg programa:

```
double ukupna_cijena(int broj_par, double cijena_par)  
{  
    const double PDV = 0.22; //22% PDV  
    double iznos1;  
    iznos1 = cijena_par * broj_par;  
    return (iznos1 + iznos1 * PDV);  
}
```

**Naredba return** završava izvođenje funkcije i vraća vrijednost koju je funkcija izračunala na mjesto sa kojeg je funkcija bila pozvana.

### Sintaksa naredbe return:

```
return izraz;
```

gdje *izraz* izvodi izračun ili je izraz varijabla koja sadrži izračunatu vrijednost

Primjer:

```
return iznos1 + iznos1 * PDV;
```

**Poziv funkcije** sadržava ime funkcije koja se poziva. U **pozivu** se navode se **stvarni parametri** (argumenti). Poziv se može koristiti na mjestu gdje vraćena vrijednost ima smisla.

Primjer:

```
double račun = ukupna_cijena(broj, cijena);
```

Vrijednosti stvarnih parametara prenose se u formalne parametre (**prosljeđivanje vrijednosti**). Prvi stvarni parametar se koristi za prvi formalni parametar, drugi stvarni parametar za drugi formalni parametar, itd. Vrijednost prenesena u formalni parametar koristi se na mjestu svih pojavljivanja formalnog parametra u tijelu funkcije.

Dozvoljena su dva oblika deklaracije funkcije:

Uz navođenje tipova i imena formalnih parametara

Uz navođenje tipova formalnih parametara, ali ne i imena parametara

U prvom slučaju se u komentaru objašnjava značenje pojedinih parametara pa je zbog toga imena parametara neophodno navesti.

Primjeri:

sa imenim parametara:

```
double ukupna_cijena(int broj_par, double cijena_par);
```

bez imena parametara:

```
double ukupna_cijena(int, double);
```

Zaglavlje definicije funkcije mora uvijek sadržavati listu imena formalnih parametara!

## Redoslijed parametara

Prevoditelj provjerava da li su svi tipovi parametara ispravni i u pravilnom redoslijedu. Prevoditelj ne može provjeriti da li su svi parametri u pravilnom logičkom poretku.

Primjer: Dana je deklaracija funkcije:

```
char ocjena(int postotak_par, int bodovi_par);
```

```
int postotak = 95, min_bodovi = 60;
```

```
cout << ocjena(bodovi1, postotak1);
```

Daje pogrešan rezultat, jer parametri nisu u pravilnom logičkom poretku. **Prevoditelj ne javlja grešku!**

Program sa logičkom greškom:

```
//Nepravilan redoslijed argumenata - logicka greska u programu
//Odredjivanje ocjene:prolaz ili pad.
#include <iostream>
using namespace std;

char ocjena(int dobiveni_par, int min_bodovi_par);
//Vraca 'P' za prolaz, ako je dobiveni_par
//jednak min_bodovi_par ili veci. Inace vraca 'F'.

int main( )
{
    int bodovi, potrebno_za_prolaz;
    char ocjena_slovo;

    cout << "Unesi bodove"
         << " i minimum potreban za prolaz:\n";
    cin >> bodovi >> potrebno_za_prolaz;

    ocjena_slovo = ocjena(potrebno_za_prolaz, bodovi);

    cout << "Broj bodova: " << bodovi << endl
         << "Minimum za prolaz " << potrebno_za_prolaz << endl;

    if (ocjena_slovo == 'T')
        cout << "Prosli ste! Cestitke!\n";
    else
        cout << "Zao mi je. Pali ste.\n";

    cout << ocjena_slovo
         << " ce biti upisano u arhivu.\n";

    return 0;
}

char ocjena(int dobiveni_par, int min_bodovi_par)
{
    if (dobiveni_par >= min_bodovi_par)
        return 'T';
    else
        return 'F';
}

/*
Unesi bodove i minimum potreban za prolaz:
23
```

55

Broj bodova: 23

Minimum za prolaz 55

Prosli ste! Cestitke!

T će biti upisano u arhivu.

\*/

## Sintaksa definicije funkcije

U okviru definicije funkcije lokalne varijable se moraju deklarirati prije nego se koriste. Varijable se obično deklariraju prije navođenja izvršnih naredbi. Funkciju mora završiti najmanje jedna naredba return (govorimo o funkciji koja vraća vrijednost!). Npr. svaka grana if-else naredbe može imati svoju vlastitu return naredbu.

## Smještanje definicije

Pozivu funkcije mora prethoditi jedno od sljedećeg:

Deklaracija funkcije

ili

Definicija funkcije

Ako definicija funkcije prethodi pozivu, deklaracija nije potrebna. Smještanje deklaracije funkcije prije glavne funkcije, a definicije nakon glavne funkcije prirodno vodi do izgradnje vaše vlastite biblioteke funkcija.

## Priprema za kviz

1. Napiši deklaraciju i definiciju funkcije koja ima tri cjelobrojna parametra i vraća zbroj svojih parametara.
2. Napiši deklaraciju i definiciju funkcije koja ima jedan parametar tipa int i jedan parametar tipa double, a vraća vrijednost tipa double – prosjek dvaju parametara.

## PROCEDURALNA APSTRAKCIJA I FUNKCIJE

Pod **crnom kutijom** se misli na nešto što znamo koristiti ali nam je nepoznat način funkcioniranja. Osoba koja koristi program ne mora znati kako je program kodiran. Osoba koja koristi program mora znati što program radi, a ne kako to radi. Programer koji koristi funkciju mora znati što funkcija radi, a ne kako funkcija radi. Programer mora znati što će biti rezultat kada određeni parametri uđu u kutiju.

Oblikovanje funkcija kao crnih kutija je primjer skrivanja informacija. Funkcije se mogu koristiti bez da znamo kako su kodirane. Tijelo funkcije može biti skriveno od pogleda.

Oblikovanje funkcije na principu crne kutije omogućava da mijenjanje ili poboljšavanje definicije funkcije ne traži od programera, koji koristi postojeću funkciju, da promijeni ono



što je prethodno napravljeno (kod iz kojeg se funkcija poziva). Jednostavno čitamo deklaraciju funkcije i njezine komentare, da bi znali kako se funkcija koristi.

**Proceduralna kutija** uključuje pisanje i uporabu funkcija kao da su crne kutije. Općenito, pojam procedura ima značenje skupa instrukcija koji je oblika funkcije. Apstrakcija implicira da uporabom funkcije kao crne kutije, apstrahiramo detalje koda u tijelu funkcije.

Pišemo funkcije tako da su deklaracija i komentari sve što programeru treba da bi koristio funkciju. Komentar funkcije nam kaže sve uvjete tražene za parametre funkcije. Komentar funkcije treba opisati vraćenu vrijednost. Varijable korištene u funkciji (uz formalne parametre) moraju biti deklarirane u tijelu funkcije (opisane su u komentaru funkcije).

Funkcije se oblikuju kao samostalni moduli. Svaku funkciju može pisati drugi programer. Programeri biraju asocijativna imena za formalne parametre. Imena formalnih parametara mogu, ali ne moraju odgovarati imenima varijabli koje se koriste u glavnom dijelu programa. Nije važno ako su imena formalnih parametara ista kao imena nekih drugih varijabli programa.

Važno: samo vrijednost parametra se prenosi u formalni parametar.

### **Promotrimo problem: Naručivanje pizze**

Želimo odrediti veličinu isplativije pizze. kao kriterij odabira uzimamo najnižu cijenu po kvadratnom centimetru. Veličina pizze je opisana promjerom.

#### **Definicija problema**

Ulaz:

Promjer dviju veličina pizze (XXL-80, XL-50)

Cijene navedenih veličina pizze

Izlaz:

Cijena po kvadratnom centimetru za svaku veličinu pizze

Zaključak koja je veličina isplativija. Temelji se na najnižoj cijeni kvadratnog centimetra.

Ako su cijene po kvadratnom centimetru jednake, manja pizza je isplativija.

#### **Analiza problema**

Potpostupak 1

Unos podataka za svaku veličinu pizze

Potpostupak 2

Izračunaj cijenu po cm za manju pizzu

Potpostupak 3

Izračunaj cijenu po cm za veću pizzu

Potpostupak 4

Odredi koju veličinu je bolje kupiti

Potpostupak 5

Izlaz rezultata

Potpostupak 2 i potpostupak 3 trebali bi biti izvedeni kao jedna funkcija jer su to identični zadaci. Izračun za potpostupak 3 je isti kao izračun za potpostupak 2 sa različitim parametrima. Potpostupak 2 i potpostupak 3 oba vraćaju jednu vrijednost.

Odabrali smo odgovarajuće ime za funkciju: **jedinicnacijena**

### Deklaracija funkcije **jedinicnacijena**

```
double jedinicnacijena(int promjer, double cijena);  
// Vraća cijenu kvadratnog cm pizze  
// Formalni parametar promjer je  
// promjer pizze u cm. Formalni  
// parametar cijena je cijena pizze.
```

### Oblikovanje algoritma

Potpostupak 1

Traži ulazne vrijednosti i spremi ih u varijable

```
        promjer_mali        promjer_veliki  
cijena_mali        cijena_veliki
```

Potpostupak 4

Uspoređi cijenu dviju pizza po kvadratnom cm  
uporabom operatora "manje od"

Potpostupak 5

Standardni izlaz rezultata

### Algoritam **jedinicnacijena**

Potpostupci 2 i 3 su izvedeni kao pozivi funkcije **jedinicnacijena**.

Algoritam **jedinicnacijena**:

```
Izračunaj polumjer pizze  
Izračunaj površinu pizze uporabom izraza  
Povratna vrijednost (cijena / površina)
```

Pseudokod funkcije **jedinicna cijena**:

```
polumjer = polovica promjera;  
povrsina =  $\pi$  * polumjer * polumjer  
return (cijena / povrsina)
```

### Pozivi funkcije **jedinicnacijena**

Glavna funkcija **main()** izvodi pozive funkcije:

```
double jedinicna_cijena_mala, jedinicna_cijena_velika;  
...
```

```
jedinicna_cijena_mala = jedinicnacijena(promjer_mala, cijena_mala);  
jedinicna_cijena_velika = jedinicnacijena(promjer_velika, cijena_velika);
```

### Kodiranje funkcije

```
double jedinicnacijena (int promjer, double cijena)  
{  
    const double PI = 3.14159;  
    double polumjer, površina;  
    polumjer = promjer / 2.0;  
    površina = PI * polumjer * polumjer;  
    return (cijena / površina);  
}
```

Evo i potpunog programa:

```
// Program određuje promjer isplativije pizze.  
#include <iostream>  
using namespace std;  
  
double jedinicna_cijena(int promjer, double cijena);  
//Vraca cijenu po kvadratnom cm pizze. Formalni parametar  
//promjer je promjer pizze u cm.  
//Formalni parametar cijena je cijena pizze.  
  
int main( )  
{  
    int promjer_mala, promjer_velika;  
    double cijena_mala, jedinicna_cijena_mala,  
           cijena_velika, jedinicna_cijena_velika;  
  
    cout << "Unesi promjer male pizze (cm): ";  
    cin >> promjer_mala;  
    cout << "Unesi cijenu male pizze: kn ";  
    cin >> cijena_mala;  
    cout << "Unesi promjer velike pizze (cm): ";  
    cin >> promjer_velika;  
    cout << "unesi cijenu velike pizze: kn ";  
    cin >> cijena_velika;  
  
    jedinicna_cijena_mala = jedinicna_cijena(promjer_mala, cijena_mala);  
    jedinicna_cijena_velika = jedinicna_cijena(promjer_velika, cijena_velika);  
  
    cout.setf(ios::fixed);  
    cout.setf(ios::showpoint);  
    cout.precision(2);  
    cout << "mala pizza:\n"  
           << "promjer = " << promjer_mala << " cm\n"  
           << "cijena = kn" << cijena_mala  
           << " Po kv.cm = kn " << jedinicna_cijena_mala << endl
```

```
<< "velika pizza:\n"
<< "promjer = " << promjer_velika << " cm\n"
<< "cijena = kn" << cijena_velika
<< " po kv. cm = kn" << jedinicna_cijena_velika << endl;
if (jedinicna_cijena_velika < jedinicna_cijena_mala)
    cout << "Velika se vise isplati.\n";
else
    cout << "Mala se vise isplati.\n";

    cout << "Dobar tek!\n";

return 0;
}

double jedinicna_cijena(int promjer, double cijena)
{
    const double PI = 3.14159;
    double radius, area;

    radius = promjer/static_cast<double>(2);
    area = PI * radius * radius;
    return (cijena/area);
}
```

```
/*
Unesi promjer male pizze (cm): 30
Unesi cijenu male pizze: kn 25
Unesi promjer velike pizze (cm): 50
unesi cijenu velike pizze: kn 45
mala pizza:
promjer = 30 cm
cijena = kn25.00 Po kv.cm = kn 0.04
velika pizza:
promjer = 50 cm
cijena = kn45.00 po kv. cm = kn0.02
Velika se vise isplati.
Dobar tek!
*/
```

## Testiranje programa

Program koji je preveden i izvršava se može još uvijek sadržavati greške. Testiranje programa povećava pouzdanost u pravilan rad programa. Pokrenite program za podatke za koje znate rezultate izvođenja programa – test podaci. Ove podatke možete pripremiti uz uporabu olovke i papira ili kalkulatora. Pokrenite program za različite skupove podataka. Vaš prvi odabir skupa test podataka može davati dobar rezultat usprkos logičkoj greški u kodu. Sjetimo se problema s dijeljenjem cijelih brojeva. Ako rezultat nema decimalnog dijela, cjelobrojno dijeljenje davati će očito točan rezultat.

## Koristite pseudokod

Pseudokod je mješavina prirodnog i programskog jezika. Pseudokod pojednostavljuje oblikovanje algoritma dozvoljavajući vam da ignorirate specifičnosti sintakse programskog jezika dok razrađujete detalje algoritma. Ako su koraci očiti, koristite C++. Ako je korak teško izraziti u C++, koristite prirodan jezik.

## Priprema za kviz

1. Objasnite namjenu komentara koji su dio deklaracije funkcije.
2. Objasnite što znači kada kažemo da programer mora sagledati funkciju kao crnu kutiju.
3. Objasnite što znači da su dvije funkcije ekvivalentne sa stajališta crne kutije.

## LOKALNE VARIJABLE

Varijable deklarirane u funkciji su lokalne za tu funkciju i ne mogu se koristiti izvan funkcije. One imaju doseg samo unutar funkcije.

Varijable deklarirane u glavnom programu su lokalne u glavnom programu i ne mogu se koristiti izvan glavnog programa. Imaju doseg samo unutar glavnog programa.

## GLOBALNE KONSTANTE

Imenovane globalne konstante su dostupne svim funkcijama kao i glavnom programu. Deklarirane su izvan svih funkcija. Deklarirane su izvan tijela glavnog programa. Deklarirane su prije funkcija koje ih koriste.

Primjer:

```
const double PI = 3.14159;
double volumen(double);
int main()
{...}
```

PI je na raspolaganju glavnom programu i funkciji volumen.

Globalna varijabla - rijetko se koristi kada dvije ili više funkcija moraju koristiti zajedničku varijablu. Deklarira se kao globalna konstanta samo bez riječi const. Njihova uporaba otežava razumijevanje i održavanje programa.

**Formalni parametri** su lokalne varijable. Formalni parametri su zapravo varijable koje su lokalne u definiciji funkcije. Koriste se kao da su deklarirane u tijelu funkcije. **NEMOJTE** ponovno deklarirati formalne parametre u tijelu funkcije, jer oni su deklarirani u deklaraciji funkcije.

Kada je funkcija pozvana formalni parametri se inicijaliziraju na vrijednosti stvarnih parametara iz poziva funkcije.

## Primjer: Faktorijela

$n!$  Predstavlja funkciju faktorijela

Matematička definicija:  $n! = 1 \times 2 \times 3 \times \dots \times n$

C++ verzija funkcije faktorijela

Zahtijeva jedan parametar tipa int, n

Vraća vrijednost tipa int

Koristi lokalnu varijablu za spremanje tekućeg umnoška

Smanjuje n svaki put kada izvrši množenje  
 $n * n-1 * n-2 * \dots * 1$

```
//PROGRAM FAKTORIJELA

#include <iostream>
using namespace std;

//Deklaracija funkcije
int faktorijela(int n);
//Vraca faktorijelu od n.
//Argument n mora biti nenegativan.

int main()
{
    for(int i=1;i<6;i++)
        cout<<"faktorijela("<i>i</i>)<<"<i>endl</i>;

    return 0;
}

// Definicija funkcije
int faktorijela(int n)
{
    int produkt = 1;
    while (n > 0)
    {
        produkt = n * produkt;
        n--;
    }

    return produkt;
}
```

```
/*
Ispis:
1
2
6
24
120
*/
```

## Primjeri sa kviza

1. Kakav izlaz daje program?

```
#include <iostream>

using namespace std;
char m(int prvi, int drugi);

int main()
{
    cout<<m(10,9)<<"OP\n";
    return 0;}

char m(int prvi, int drugi)
{
    if (prvi >= drugi)
        return 'H';
    else
        return 'T';
}
```

2. Napišite definiciju i deklaraciju funkcije *poredak* koja ima tri parametra tipa int. Funkcija vraća true, ako su parametri uređeni uzlazno, inače vraća false.

Npr:

```
poredak(1,2,3) true
poredak(1,2,2) true
poredak (1,3,2) false
```

3. Ako koristite varijablu u definiciji funkcije, gdje trebate deklarirati varijablu?

Slijedeća funkcija pretvara stope i inče u inče. Npr. total\_inches(1,2) daje 14. Da li je funkcija pravilno napisana?

```
double total_inches(int feet,int inches)
{
    inches=12*feet+inches;
    return inches;
}
```

## 12. FUNKCIJE KOJE NEMAJU ODREĐEN TIP: VOID-FUNKCIJE

U top-down oblikovanju potpostupak može proizvesti kao rezultat:

- Ni jednu vrijednost (npr. ima samo ulazne vrijednosti)
- Jednu vrijednost
- Više od jedne vrijednosti

Vidjeli smo kako se izvodi funkcija koja vraća jednu vrijednost. void-funkcija izvodi potpostupak koji ne vraća ni jednu vrijednost ili vraća jednu ili više vrijednosti.

### DEFINICIJA VOID-FUNKCIJE

Dvije su osnovne razlike između definicije void-funkcije i definicija funkcija koje vraćaju jednu vrijednost: Ključna riječ void zamjenjuje tip vraćene vrijednosti. Void znači da funkcija ne vraća vrijednost. Naredba return sada ne sadrži izraz.

Primjer funkcije:

```
void pokazi_rezultate(double f_stupnjevi, double c_stupnjevi)  
{  
    using namespace std;  
    cout << f_stupnjevi  
        << " stupnjeva Fahrenheita iznosi " << endl  
        << c_stupnjevi << " stupnjeva Celzijusa." << endl;  
    return;  
}
```

### Uporaba void-funkcije

Pozivi void-funkcije su izvršne naredbe. Ne mogu biti dio naredbe cout. Završavaju točkazarezom.

Primjer:

```
    pokazi_rezultate(32.5, 0.3);
```

**nedozvoljeno:** cout << pokazi\_rezultate(32.5, 0.3);

### Poziv void-funkcije

Mehanizam je skoro isti kao kod poziva funkcija koje vraćaju vrijednost. Vrijednosti stvarnih parametara se prosljeđuju u formalne parametre. Void-funkcije često nemaju niti jedan parametar. U tom slučaju nema parametara ni u pozivu funkcije.

Izvršavaju se naredbe u tijelu funkcije. Naredba return je opcionalna za završetak funkcije i ne sadrži povratnu vrijednost. Naredba return je skrivena ako je ne navedemo

Da li je naredba return uopće potrebna u void-funkciji budući nema povratne vrijednosti? Da! Može se dogoditi da grana if-else naredbe zahtijeva da funkcija završi s izvođenjem da bi izbjegli nepotrebnii nastavak ispisa ili matematičku pogrešku.



```
Kompletan program:
//Funkcije void
//Program konvertira temperaturu u Fahrenhajtima u temperaturu u
// stupnjeve Celzijusa.
#include <iostream>

void inicijalizacija_ekrana( );
//Odvaja tekuci izlaz od
//izlaza prethodnog pokretanja programa.

double celzijus(double farenhajt);
//Konvertira temperaturu
//u Fahrenhajtima u stupnjeve Celzijusa.

void pokazi_rezultate(double f_stupnjeva, double c_stupnjeva);
//Prikazuje izlaz. Pretpostavlja da je c_stupnjeva
//Celzijusa ekvivalentno f_stupnjeva Fahrenhajta.

int main( )
{
    using namespace std;
    double f_temperature, c_temperature;

    inicijalizacija_ekrana( );
    cout << "Konvertira se temperatura iz stupnjeva Fahrenhajta u"
         << " stupnjeve Celzijusa.\n"
         << "Unesi temperaturu (F): ";
    cin >> f_temperature;

    c_temperature = celzijus(f_temperature);

    pokazi_rezultate(f_temperature, c_temperature);
    return 0;
}

//Definicija koristi iostream:
void inicijalizacija_ekrana( )
{
    using namespace std;
    system("cls"); // operativni sustav brise ekran
    return;
}
double celzijus(double farenhajt)
{
    return ((5.0/9.0)*(farenhajt - 32));
}

//Definicija koristi iostream:
void pokazi_rezultate(double f_stupnjeva, double c_stupnjeva)
{
```

```
using namespace std;
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(1);
cout << f_stupnjeva
    << " stupnjeva Farenhajta iznosi \n"
    << c_stupnjeva << " stupnjeva Celzijusa.\n";
return;
}
```

```
/*
Konvertira se temperatura iz stupnjeva Farenhajta u stupnjeve Celzijusa.
Unesi temperaturu (F): 80
80.0 stupnjeva Farenhajta iznosi
26.7 stupnjeva Celzijusa.
*/
```

## Priprema za kviz

1. Objasni razliku između void-funkcija i funkcija koje vraćaju jednu vrijednost.
2. Što se dogodi ako zaboravimo upotrijebiti naredbu return u void-funkciji?
3. Objasni razliku između funkcija koje se koriste kao izrazi i onih koji se koriste kao naredbe.

## PROSLJEĐIVANJE REFERENCE (REFERENTNI PARAMETRI)

Prosljeđivanje putem vrijednosnih parametara nije prikladno kada je potrebno da potpostupak ima više izlaznih vrijednosti. Prosljeđivanje vrijednosti znači da formalni parametri primaju vrijednosti parametara. Da bi dobili izlazne vrijednosti, moramo promijeniti varijable - stvarne parametre na mjestu poziva funkcije. Sjetimo se da su se vrijednosti formalnih parametara mijenjale u tijelu funkcije, ali nisu se promijenili stvarni parametri u funkcijskom pozivu. **Referentni parametar** omogućava promjenu varijable koja se pojavila u pozivu funkcije kao stvarni parametar. Stvarni parametri iz poziva funkcije zato pri prosljeđivanju referencom moraju biti varijable, a ne brojevi.

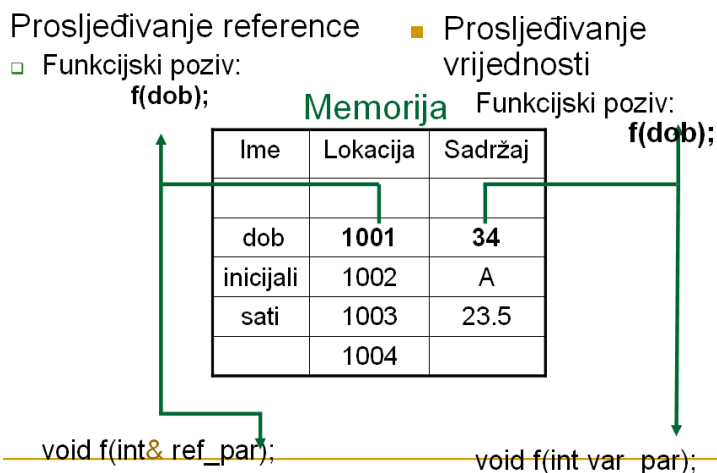
### Primjer prosljeđivanja reference

```
void ulaz(double& f_varijabla)
{
    using namespace std;
    cout << "Temperaturu u Fahrenheitima pretvori"
        << " u Celzijuse.\n"
        << "unesi temperaturu u Fahrenheitima: ";
    cin >> f_varijabla;
}
```

Simbol & (ampersand) označava parametar f\_varijabla kao referentni parametar. Koristi se i u deklaraciji i u definiciji!

## Detalji prosljeđivanja reference

Prosljeđivanje referencom funkcionira skoro kao da je varijabla na mjestu stvarnog parametra zamijenila formalni parametar, a ne vrijednost parametra. Zapravo je adresa memorijske lokacije stvarnog parametra prosljeđena formalnom parametru. Svaka promjena na formalnom parametru u tijelu funkcije se istovremeno događa sa vrijednošću na memorijskoj lokaciji stvarnog parametra u pozivu.



## Primjer: zamjena\_vrijednosti

Da bi se zamijenile vrijednosti stvarnih parametara na mjestu poziva ove funkcije, formalni parametri moraju biti referentni.

```
void zamjena(int& varijabla1, int& varijabla2)
{
    int temp = varijabla1;
    varijabla1 = varijabla2;
    varijabla2 = temp;
}
```

Ako se izvrši poziv funkcije zamjena(prvi\_broj, drugi\_broj); prvi\_broj se supstituira u varijabla1 u listi parametara, drugi\_broj se supstituira u varijabla2 u listi parametara. temp se dodjeljuje vrijednost varijable varijabla1 (prvi\_broj), jer u slijedećoj liniji varijabla1 gubi vrijednost (prvi\_broj), varijabla1 (prvi\_broj) dobiva vrijednost od varijabla2 (drugi\_broj) varijabla2 (drugi\_broj) dobiva originalnu vrijednost od varijabla1 (prvi\_broj) koja je bila spremljena u temp.

U istoj funkciji se mogu pojaviti i reference i vrijednosni parametri.

Primjer:

```
void ff(int& par1, int par2, double& par3);
```

par1 i par3 su referentni formalni parametri. Promjene u par1 i par3 mijenjaju stvarne parametre.

par2 je vrijednosni formalni parametar. Promjene u par2 ne mijenjaju stvarni parametar.

Kompletan program:

```
//Referentni parametri
// Zamjena vrijednosti
#include <iostream>

void unos_brojeva(int& ulaz1, int& ulaz2);
//Cita dva cijela broja sa tipkovnice.

void zamijeni(int& varijabla1, int& varijabla2);
//Zamjenjuje vrijednosti varijable1 i varijable2.

void prikazi_rezultate(int izlaz1, int izlaz2);
//Prikazuje zamjenjene vrijednosti varijabli.

int main( )
{
    int prvi_broj, drugi_broj;

    unos_brojeva(prvi_broj, drugi_broj);
    zamijeni(prvi_broj, drugi_broj);
    prikazi_rezultate(prvi_broj, drugi_broj);
    return 0;
}

//Uses iostream:
void unos_brojeva(int& ulaz1, int& ulaz2)
{
    using namespace std;
    cout << "Unesi dva cijela broja: ";
    cin >> ulaz1
        >> ulaz2;
}

void zamijeni(int& varijabla1, int& varijabla2)
{
    int temp;

    temp = varijabla1;
    varijabla1 = varijabla2;
    varijabla2 = temp;
}

//Uses iostream:
void prikazi_rezultate(int izlaz1, int izlaz2)
```

```
{  
    using namespace std;  
    cout << "Inverzan poredak brojeva: "  
        << izlaz1 << " " << izlaz2 << endl;  
}
```

```
/*  
Unesi dva cijela broja: 3 4  
Inverzan poredak brojeva: 4 3  
*/
```

### Biranje tipa parametara

Kako odlučujemo da li ćemo koristiti referentni ili vrijednosni parametar?

Kriterij koji razmatramo su:

- Da li funkcija treba promijeniti vrijednost varijable koja se pojavljuje kao stvarni parametar?  
Ako je odgovor da, koristite referentni formalni parametar  
Ako je odgovor ne, koristite vrijednosni formalni parametar

### Neželjene lokalne varijable

Ako funkcija treba promijeniti vrijednost varijable, odgovarajući formalni parametar mora biti referentni parametar sa oznakom (&).

Ako zaboravimo oznaku &, kreira se vrijednosni parametar.

Vrijednost stvarnog parametra se neće promijeniti. Formalni parametar je u tom slučaju lokalna varijabla čija promjena nema utjecaj na stvarni parametar. Ovu grešku je teško pronaći ako nismo pažljivi!

### Priprema za kviz

1. Napiši definiciju void funkcije za funkciju s imenom oba\_nula, koja ima dva referentna parametra (oba su varijable tipa int) i postavlja vrijednosti varijabli na 0.
2. Napiši funkciju za izračun umnoška dviju vrijednosti int tipa koje se funkciji proslijeđuju putem vrijednosnih parametara. Rezultat se vraća preko referentnog parametra.

## 13. POLJA U FUNKCIJAMA

Indeksirane varijable mogu biti argumenti funkcija

Primjer: Program sadrži deklaracije:

```
int i, n, a[10];  
void my_function(int n);
```

Varijable a[0] do a[9] su tipa int, pa su ovi pozivi legalni:

```
my_function( a[ 0 ] );  
my_function( a[ 3 ] );  
my_function( a[ i ] );
```

### POLJA KAO ARGUMENTI FUNKCIJA

Formalni parametar (argument) može biti cijelo polje. Takav parametar zovemo parametar tipa polja. To nije prosljeđivanje putem vrijednosti, niti prosljeđivanje reference. Parametri tipa polja se ponašaju slično kao referentni parametri.

#### Deklaracija parametra tipa polja

Parametar tipa polja se označava uporabom praznih uglatih zagrada u listi parametara:

```
void popuni_polje(int a[ ], int velicina);
```

#### Pozivi funkcija sa poljima

Ako je funkcija popuni\_polje deklarirana:

```
void popuni_polje(int a[ ], int velicina);
```

i polje bodovi je deklarirano:

```
int bodovi[5], broj_elemenata;
```

*popuni\_polje* se poziva ovako:

```
popuni_polje(bodovi, broj_elemenata);
```

#### Detalji poziva funkcije

Formalni parametar tipa polja piše se sa uglatim zagradama [ ] bez izraza koji označava veličinu polja:

```
void popuni_polje(int a[ ], int velicina);
```

Stvarni parametar (argument) se piše bez uglatih zagrada

```
popuni_polje(bodovi, broj_bodova);
```

## Polje kao formalni parametar

Polje kao formalni parametar preuzima ulogu argumenta (stvarnog parametra). Kada je polje stvarni parametar u funkcijskom pozivu, postupak koji se izvodi na formalnom parametru zapravo se izvodi na stvarnom parametru. Vrijednosti indeksiranih varijabli funkcija može promijeniti.

## Polje kao stvarni parametar

Što računalo zna o polju?

Temeljni tip, adresu prve indeksirane varijable, broj indeksiranih varijabli.

Što funkcija zna o stvarnom parametru?

Temeljni tip, adresu prve indeksirane varijable.

## Polje kao formalni parametar – razmatranja

Kako funkcija ne zna veličinu polja – stvarnog parametra programer mora uključiti formalni parametar koji specificira veličinu polja. Funkcija može obraditi polja različitih veličina.

Funkcija `popuni_polje` može se koristiti za popunu polja bilo koje veličine:

```
popuni_polje(bodovi, 5);  
popuni_polje(vrijeme, 10);
```

```
void popuni(int a[], int velicina)  
{  
    cout << "Unesi " << velicina << " brojeva:\n";  
    for (int i = 0; i < velicina; i++)  
        cin >> a[i];  
    velicina--;  
    cout << "Posljednji rabljeni indeks polja " << velicina << endl;  
}
```

## Modifikator const

Parametri tipa polja dozvoljavaju funkciji da mijenja vrijednosti pohranjene u polje iz poziva funkcije. Ako funkcija ne smije mijenjati vrijednosti polja iz poziva funkcije, koristimo modifikator `const`

Polje kao parametar označeno sa `const` je konstantni parametar

Primjer:

```
void f1(const int a[ ], int velicina);
```

## Uporaba const sa poljima

Ako je parametar tipa polja konstantan, `const` se koristi i u deklaraciji i u definiciji funkcije. Prevoditelj će javiti grešku ako funkcija pokuša promijeniti vrijednost konstantnog parametra.

## Vraćanje polja

Funkcije mogu vratiti vrijednost tipa int, double, char, ..., ili tipa klase

Funkcije ne mogu vratiti polja. Kasnije ćemo vidjeti da funkcija može vratiti pokazivač na polje.

## Djelomično popunjena polja

Kada koristimo djelomično popunjena polja funkcije koje rade sa poljem ne moraju znati deklariranu veličinu polja, već samo koliko je elemenata pohranjeno u polje. Parametar, broj\_elementa, može biti dovoljan da bi se provjerio legalan raspon vrijednosti indeksa. Funkcija popuni\_polje treba i deklariranu veličinu polja.

## Konstante kao parametri

MAX\_BROJ\_IGRACA se koristi kao parametar. Zar ne možemo MAX\_BROJ\_IGRACA koristiti direktno sa globalne razine? Kada koristimo MAX\_BROJ\_IGRACA kao parametar, jasno je da funkcija popuni\_polje zahtijeva deklariranu veličinu polja. Jasnija je uporaba ove funkcije u drugim programima.

## Pretraživanje polja

Jedan način za pretraživanje polja je sekvencijalno pretraživanje.

Pregledava svaki element od prvog do zadnjeg u postupku traženja zadanog ključa (provjerava se za svaki element polja da li je jednak ključu)

Indeks vrijednosti koja je jednaka ključu predstavlja indikator da je ključ pronađen u polju  
Vrijednost -1 je vraćena ako vrijednost nije pronađena

## Funkcija pretraživanja

Koristi while petlju za uspoređivanje elemenata polja sa ključem

Postavlja varijablu tipa bool na true ako je ključ nađen, a petlja završava

Provjerava logičku varijablu kada petlja završava da bi se provjerilo da li je ključ nađen, inače vraća -1.

Kompletan program:

```
//Pretraživanje polja
//Pretražuje parcijalno popunjeno polje nenegativnih cijelih brojeva.
#include <iostream>
const int DEKLARIRANA_VELICINA = 20;

void popuni_polje(int a[], int velicina, int& broj_elementa);
//Preduvjet: velicina je deklarirana velicina polja a.
//Rezultat izvođenja: broj_elementa je broj elemenata pohranjenih u a.
//a[0] do a[broj_elementa-1] je popunjen sa
//nenegativnim cijelim brojevima unesenim s tipkovnice.

int pretrazivanje(const int a[], int broj_elementa, int kljuc);
```



```
//Preduvjet: broj_elemenata je <= od deklarirane velicine polja a.
//Isto tako, a[0] do a[broj_elemenata -1] imaju vrijednosti.
//Vraca prvi indeks takav da vrijedi a[indeks] == kljuc,
//ako takav indeks postoji; inace, vraca -1.

int main( )
{
    using namespace std;
    int polje[DEKLARIRANA_VELICINA], velicina_liste, kljuc;

    popuni_polje(polje, DEKLARIRANA_VELICINA, velicina_liste);

    char odg;
    int rezultat;
    do
    {
        cout << "Unesi broj koji se trazi u listi: ";
        cin >> kljuc;

        rezultat = pretrazivanje(polje, velicina_liste, kljuc);
        if (rezultat == -1)
            cout << kljuc << " nije u listi.\n";
        else
            cout << kljuc << " je pohranjen na poziciji "
                << rezultat << endl
                << "(Sjeti se: Prva pozicija je 0.)\n";

        cout << "pretrazivanje ponoviti?(d/n): ";
        cin >> odg;
    }while ((odg != 'n') && (odg != 'N'));

    cout << "Kraj programa.\n";
    return 0;
}

//Uses iostream:
void popuni_polje(int a[], int velicina, int& popunjeni_broj)
{
    using namespace std;
    cout << "Unesi ukupno " << velicina << " nenegativnih cijelih brojeva.\n"
        << "ili ako ih uneses manje, oznaci kraj liste sa negativnim brojem.\n";
    int slijedeci, indeks = 0;
    cin >> slijedeci;
    while ((slijedeci >= 0) && (indeks < velicina))
    {
        a[indeks] = slijedeci;
        indeks++;
        cin >> slijedeci;
    }

    popunjeni_broj = indeks;
}
```

```
}  
  
int pretrazivanje(const int a[], int broj_elemenata, int kljuc)  
{  
  
    int indeks = 0;  
    bool nadjen = false;  
    while ((!nadjen) && (indeks < broj_elemenata))  
        if (kljuc == a[indeks])  
            nadjen = true;  
        else  
            indeks++;  
  
    if (nadjen)  
        return indeks;  
    else  
        return -1;  
}
```

```
/*  
Unesi ukupno 20 nenegativnih cijelih brojeva.  
ili ako ih uneses manje, oznaci kraj liste sa negativnim brojem.  
3  
77  
66  
44  
5  
2  
4  
-3  
Unesi broj koji se trazi u listi: 66  
66 je pohranjen na poziciji 2  
(Sjeti se: Prva pozicija je 0.)  
pretrazivanje ponoviti?(d/n): N  
Kraj programa.  
*/
```

## Priprema za kviz

1. Napišite program koji učitava do 10 slova u polje i ispisuje ih u obratnom redosljedu na ekran.

Primjer:

abcd se ispisuje kao dcba

Koristite točku kao oznaku kraja ulaza.

## 14. TESTIRANJE FUNKCIJA

### PREDUVJETI I UVJETI NA REZULTAT IZVRŠAVANJA FUNKCIJE

Preduvjeti opisuju što mora biti zadovoljeno kada je funkcija pozvana. Funkcija ne bi trebala biti pozvana ako nisu zadovoljeni preduvjeti. Uvjeti na rezultate izvršavanja funkcije opisuju učinak poziva funkcije. Navode istinite tvrdnje nakon izvršavanja funkcije (uz zadovoljene preduvjete). Ako funkcija vraća vrijednost, ta se vrijednost opisuje. Opisuju se promjene referentnih parametara.

Uz navođenje preduvjeta i uvjeta na rezultate deklaracija funkcije zamjena izgleda ovako:

```
void zamjena(int& varijabla1, int& varijabla2);  
  //Preduvjet: varijabla n1 i varijabla n2  
  // dobivaju vrijednosti.  
  // Uvjet na rezultate: Vrijednosti varijabla1 i  
  //                   varijabla2 su zamijenjene
```

Preduvjeti i uvjeti na rezultate čine deklaraciju funkcije celzijus:

```
double celzijus(double fahrenheit);  
//Preduvjet: fahrenheit je temperatura  
//          u stupnjevima Fahrenheita  
//Uvjeti na rezultat: Vraća ekvivalentnu temperaturu  
//          u stupnjevima Celzijusa
```

#### Zašto koristimo preduvjete i uvjete na rezultat?

Preduvjeti i uvjeti na rezultat trebaju biti prvi korak u oblikovanju funkcije. Specificiraju što funkcija treba raditi. Uvijek specificirajte što funkcija treba raditi, prije nego počnete oblikovati kako će funkcija to izvršavati.

Minimiziraju greške oblikovanja, minimiziraju vrijeme utrošeno na pisanje koda koji ne odgovara rješavanju zadanog zadatka.

#### Problem: Postavljanje cijena u trgovini

Definicija problema

Odrediti maloprodajnu cijenu proizvoda za odgovarajući ulaz i 5% marže ako se artikl može prodati u roku od 7 dana, a 10% marže ako za prodaju artikla treba više od 7 dana (5% za rok do 7 dana, mijenja se u 10% nakon 7 dana).

Ulaz: Početna cijena proizvoda i procjena broja dana potrebnih za prodaju

Izlaz: Maloprodajna cijena artikla

## Analiza problema

Tri glavna potpostupka:

1. Ulaz podataka
2. Izračun maloprodajne cijene artikla
3. Izlaz rezultata

Svaki postupak može se izvesti u funkciji. Uočimo uporabu vrijednosnih i referentnih parametara u deklaracijama funkcija.

## Funkcija unos\_podataka

```
void unos(double& cijena, int& broj_dana);  
// Preduvjet:  
//      Korisnik je spreman za pravilan  
//      unos podataka.  
// Uvjet na rezultat:  
//      Cijena je postavljena na  
//      početnu cijenu jednog artikla.  
//      Broj očekivanih dana potrebnih  
//      za prodaju artikla – broj_dana
```

## Funkcija m\_cijena

```
double m_cijena(double cijena, int broj_dana);  
//Preduvjet: cijena je početna cijena jednog  
//      artikla. broj_dana je očekivani  
//      broj dana potreban  
//      za prodaju artikla.  
//Uvjet na rezultat: vraća maloprodajnu cijenu  
//      artikla.
```

## Funkcija izlaz\_podataka

```
void izlaz(double cijena, int br_dana, double maloprodajna_cijena);  
//Preduvjet: cijena je početna cijena jednog artikla;  
//      br_dana je očekivano vrijeme do  
//      prodaje jednog artikla;  
//      maloprodajna_cijena je  
//      maloprodajna cijena artikla.  
//Uvjet na rezultat: Vrijednost parametara cijena,  
//      br_dana, i  
//      maloprodajna_cijena ispisuju se na ekran.
```

## Funkcija main

Nakon što deklariramo funkcije možemo pisati glavnu funkciju :

```
int main( )
{
    double ukupna_cijena, maloprodajna_cijena;
    int vrijeme_na_polici;
    uvod( );
    unos(ukupna_cijena, vrijeme_na_polici);
    maloprodajna_cijena = m_cijena(ukupna_cijena, vrijeme_na_polici);
    izlaz(ukupna_cijena, vrijeme_na_polici, maloprodajna_cijena);
    return 0;
}
```

### Oblikovanje algoritma

Izvedba funkcija unos i izlaz je jednostavna, pa ćemo se usredotočiti na funkciju m\_cijena.

Pseudokod za funkciju m\_cijena:

```
Ako je broj dana na polici <= 7 dana tada
    return (cijena + 5% cijene);
inače
    return (cijena + 10% cijene);
```

### Konstante za funkciju m\_cijena

Numeričke vrijednosti u pseudokodu su predstavljene konstantama:

```
const double NISKA_MARZA = 0.05; // 5%
const double VISOKA_MARZA = 0.10; // 10%
const int PRAG = 7; // Nakon 7 dana koristi
                // VISOKA_MARZA
```

### Kodiranje funkcije m\_cijena

Tijelo funkcije

```
{
    if (broj_dana <= PRAG)
        return ( cijena + (NISKA_MARZA * cijena) );
    else
        return ( cijena + (VISOKA_MARZA * cijena) );
}
```

Kompletan program:

```
//Odredjivanje cijena u trgovini
//Odredjivanje maloprodajne cijene artikla ovisno o
```

```
//pravilima odredjivanja cijena lanca supermarketa brze prodaje.
#include <iostream>

const double NISKA_MARZA = 0.05; //5%
const double VISOKA_MARZA = 0.10; //10%
const int PRAG = 7; //Koristi se VISOKA_MARZA ako se proizvod
    //ne moze prodati u roku od 7 dana.

void uvod( );
//Uvjet na rezultat: Opis programa se ispisuje na ekran.

void unos(double& cijena, int& broj_dana);
//Preduvjet: Korisnik je spreman unijeti ispravne vrijednosti.
//Uvjet na rezultat: Vrijednost varijable cijena je postavljena na
//ukupnu cijenu jednog artikla. Vrijednost broj_dana je postavljena
//na ocekivani broj dana do prodaje artikla.

double m_cijena(double cijena, int broj_dana);
//Preduvjet: cijena je ukupna cijena artikla.
//broj_dana ocekivani broj dana do prodaje artikla.
//Vraca maloprodajnu cijenu artikla.

void izlaz(double cijena, int broj_dana, double maloprodajna_cijena);
//Preduvjet: cijena je ukupna cijena artikla; broj_dana je
//ocekivani broj dana do prodaje artikla; cijena je maloprodajna cijena artikla.
//Uvjet na rezultat: Vrijednosti cijena, broj_dana i cijena su
//ispisani na ekran.

int main( )
{
    double pocetna_cijena, maloprodajna_cijena;
    int vrijeme_na_polici;
    uvod( );
    unos(pocetna_cijena, vrijeme_na_polici);
    maloprodajna_cijena = m_cijena(pocetna_cijena, vrijeme_na_polici);
    izlaz(pocetna_cijena, vrijeme_na_polici, maloprodajna_cijena);
    return 0;
}

//Koristi iostream:
void uvod( )
{
    using namespace std;
    cout << "Ovaj program odredjuje maloprodajnu cijenu za\n"
        << "artikl u trgovini brze prodaje.\n";
}

//Koristi iostream:
void unos(double& cijena, int& broj_dana)
{
```

```
using namespace std;
cout << "Unesi cijenu artikla: kn ";
cin >> cijena;
cout << "Unesi očekivani broj dana do prodaje: ";
cin >> broj_dana;
}

//Koristi iostream:
void izlaz(double cijena, int broj_dana, double maloprodajna_cijena)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Ukupna cijena = kn " << cijena << endl
        << "Očekivano vrijeme do prodaje = "
        << broj_dana << " dana" << endl
        << "maloprodajna cijena = kn " << maloprodajna_cijena << endl;
}

//Koristi definirane konstante NISKA_MARZA, VISOKA_MARZA i PRAG:
double m_cijena(double cijena, int broj_dana)
{
    if (broj_dana <= PRAG)
        return ( cijena + (NISKA_MARZA * cijena) );
    else
        return ( cijena + (VISOKA_MARZA * cijena) );
}
```

```
/*
Ovaj program određuje maloprodajnu cijenu za
artikl u trgovini brze prodaje.
Unesi cijenu artikla: kn 29
Unesi očekivani broj dana do prodaje: 3
Ukupna cijena = kn 29.00
Očekivano vrijeme do prodaje = 3 dana
maloprodajna cijena = kn 30.45
*/
```

## STRATEGIJE TESTIRANJA PROGRAMA

Koristimo podatke koji testiraju i slučajeve malog i velikog profita.

Testiramo granične uvjete, za koje se očekuje da program mijenja ponašanje ili izvrši odabir.

U funkciji `m_cijena`, 7 dana je granični uvjet. Testiraj program za točno 7 dana kao i za jedan dan manje i jedan dan više.

Svaka funkcija treba se testirati kao odvojena jedinica. Testiranje individualnih funkcija olakšava nalaženje grešaka. Program za pokretanje (driver program) omogućava testiranje individualnih funkcija. Kada je funkcija testirana može se koristiti u programu za pokretanje u testiranju drugih funkcija. Funkcija unos se testira u programu za pokretanje:

```
//Program za pokretanje (Driver Program)
//Program za pokretanje funkcije unos.
#include <iostream>

void unos(double& cijena, int& broj_dana);
//Preduvjet: Korisnik je spreman unijeti ispravne vrijednosti.
//Uvjet na rezultat: Vrijednost varijable cijena je postavljena na
//ukupnu cijenu jednog artikla. Vrijednost broj_dana je postavljena
//na očekivani broj dana do prodaje artikla.

int main( )
{
    using namespace std;
    double ukupna_cijena;
    int vrijeme_na_polici;
    char odg;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    do
    {
        unos(ukupna_cijena, vrijeme_na_polici);

        cout << "ukupna cijena je sada kn "
             << ukupna_cijena << endl;
        cout << "broj dana "
             << vrijeme_na_polici << endl;

        cout << "Ponovno testirati?"
             << " (Unesi d za da ili n za ne): ";
        cin >> odg;
        cout << endl;
    } while (odg == 'd' || odg == 'D');

    return 0;
}
//Uses iostream:
void unos(double& cijena, int& broj_dana)
{
    using namespace std;
    cout << "Unesi pocetnu cijenu artikla: kn";
    cin >> cijena;
    cout << "Unesi očekivani broj dana do prodaje: ";
    cin >> broj_dana;
}
```



```
/*  
Unesi pocetnu cijenu artikla: kn 44  
Unesi ocekivani broj dana do prodaje: 8  
ukupna cijena je sada kn 44.00  
broj dana 8  
Ponovno testirati? (Unesi d za da ili n za ne): d  
  
Unesi pocetnu cijenu artikla: kn22  
Unesi ocekivani broj dana do prodaje: 2  
ukupna cijena je sada kn 22.00  
broj dana 2  
Ponovno testirati? (Unesi d za da ili n za ne): n  
*/
```

### Zamjenski programski element (stub)

Kada funkcija koju testiramo zove druge funkcije koje još nisu testirane, koristimo zamjenski programski element (stub). “Stub” je pojednostavljena verzija funkcije. “Stub” je obično privremena pojednostavljena verzija funkcije (za testiranje) umjesto funkcije koja sadrži detaljnu izvedbu traženog izračuna. “Stub” mora biti jednostavan i mora raditi pouzdano te davati točne rezultate. Funkcija `m_cijena` se koristi kao stub za testiranje ostatka programa:

```
//Program koji sadrzi "stub"  
//Odredjivanje maloprodajne cijene artikla ovisno o  
//pravilima odredjivanja cijena lanca supermarketa brze prodaje.  
#include <iostream>  
  
const double NISKA_MARZA = 0.05; //5%  
const double VISOKA_MARZA = 0.10; //10%  
const int PRAG = 7; //Koristi se VISOKA_MARZA ako se proizvod  
//ne moze prodati u roku od 7 dana.  
  
void uvod( );  
//Uvjet na rezultat: Opis programa se ispisuje na ekran.  
  
void unos(double& cijena, int& broj_dana);  
//Preduvjet: Korisnik je spreman unijeti ispravne vrijednosti.  
//Uvjet na rezultat: Vrijednost varijable cijena je postavljena na  
//ukupnu cijenu jednog artikla. Vrijednost broj_dana je postavljena  
//na ocekivani broj dana do prodaje artikla.  
  
double m_cijena(double cijena, int broj_dana);  
//Preduvjet: cijena je ukupna cijena artikla.  
//broj_dana ocekivani broj dana do prodaje artikla.  
//Vraća maloprodajnu cijenu artikla.
```

```
void izlaz(double cijena, int broj_dana, double maloprodajna_cijena);
//Preduvjet: cijena je ukupna cijena artikla; broj_dana je
//ocekivani broj dana do prodaje artikla; cijena je maloprodajna cijena artikla.
//Uvjet na rezultat: Vrijednosti cijena, broj_dana i cijena su
//ispisani na ekran.

int main( )
{
    double ukupna_cijena, maloprodajna_cijena;
    int vrijeme_na_polici;
    uvod( );
    unos(ukupna_cijena, vrijeme_na_polici);
    maloprodajna_cijena = m_cijena(ukupna_cijena, vrijeme_na_polici);
    izlaz(ukupna_cijena, vrijeme_na_polici, maloprodajna_cijena);
    return 0;
}

//Koristi iostream:
void uvod( )
{
    using namespace std;
    cout << "Ovaj program odredjuje maloprodajnu cijenu za\n"
         << "artikl u trgovini brze prodaje.\n";
}

//Koristi iostream:
void unos(double& cijena, int& broj_dana)
{
    using namespace std;
    cout << "Unesi cijenu artikla: kn ";
    cin >> cijena;
    cout << "Unesi ocekivani broj dana do prodaje: ";
    cin >> broj_dana;
}

//Koristi iostream:
void izlaz(double cijena, int broj_dana, double maloprodajna_cijena)
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "Ukupna cijena = kn " << cijena << endl
         << "Ocekivano vrijeme do prodaje = "
         << broj_dana << " dana" << endl
         << "maloprodajna cijena = kn " << maloprodajna_cijena << endl;
}

//Ovo je samo "stub":
double m_cijena(double cijena, int broj_dana)
```

```
{  
    return 9.99; //Nije točno, ali dovoljno je dobro za test.  
}
```

```
/*  
Ovaj program određuje maloprodajnu cijenu za  
artikl u trgovini brze prodaje.  
Unesi cijenu artikla: kn 33  
Unesi očekivani broj dana do prodaje: 4  
Ukupna cijena = kn 33.00  
Očekivano vrijeme do prodaje = 4 dana  
maloprodajna cijena = kn 9.99  
*/
```

## 15. NADJAČAVANJE IMENA FUNKCIJA

C++ omogućava više od jedne definicije za isto ime funkcije. To je vrlo prikladno za situacije u kojima je “ista” funkcija potrebna za različiti broj ili tip parametara.

Nadjačavanje imena funkcije znači osiguravanje više od jedne deklaracije i definicije za isto ime funkcije.

### Primjer nadjačavanja:

```
double prosjek(double n1, double n2)  
{  
    return ((n1 + n2) / 2);  
}  
  
double prosjek(double n1, double n2, double n3)  
{  
    return (( n1 + n2 + n3) / 3);  
}
```

Prevoditelj provjerava broj i tip parametara u funkcijskom pozivu da bi odlučio koja funkcija će se koristiti. Sljedeći poziv funkcije koristi drugu definiciju.

```
cout << prosjek( 10, 20, 30);
```

Program:

```
// Nadjacavanje funkcija  
//Ilustrira nadajacavanje funkcije prosjek.  
#include <iostream>  
  
double prosjek(double n1, double n2);  
//vraca prosjek broeva n1 i n2.  
  
double prosjek(double n1, double n2, double n3);
```

```
//Vraca prosjek triju brojeva n1, n2 i n3.

int main( )
{
    using namespace std;
    cout << "Prosjeak brojeva 2.0, 2.5 i 3.0 je "
        << prosjek(2.0, 2.5, 3.0) << endl;

    cout << "Prosjeak brojeva 4.5 i 5.5 je "
        << prosjek(4.5, 5.5) << endl;

    return 0;
}

double prosjek(double n1, double n2)
{
    return ((n1 + n2)/2.0);
}

double prosjek(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3)/3.0);
}
```

```
/*
Prosjeak brojeva 2.0, 2.5 i 3.0 je 2.5
Prosjeak brojeva 4.5 i 5.5 je 5
*/
```

## Detalji nadjačavanja

Nadjačane funkcije moraju imati različit broj formalnih parametara I / ILI moraju imati najmanje jedan različit tip parametra.

Primjer nadjačavanja:

Ponovno se bavimo narudžbom pizze:

Nude se i pravokutne pizze! Promijeni ulaz i dodaj funkciju za izračun jedinične cijene pravokutne pizze.

Nova funkcija može se nazvati `jedinicnacijena_pravokutne`, ili nova funkcija može biti nova (nadjačana) verzija funkcije `jedinicnacijena` koju već koristimo.

Primjer:

```
double jedinicnacijena(int duljina, int sirina, double cijena)
{
    double površina = duljina * sirina;
    return (cijena / površina);
}
```

```
//Nadjacavanje funkcije
//Odredjuje da li se vise isplati okrugla ili pravokutna pizza.
#include <iostream>

double jedinica_cijena(int promjer, double cijena);
// Vraca cijenu kvadratnog cm okrugle pize.
// Formalni parametar promjer je promjer pize u cm.
// Formalni parametar cijena je cijena pize.

double jedinica_cijena(int duljina, int sirina, double cijena);
//Vraca cijenu kvadratnog cm pravokutne pize.
//sa dimezijama duljina i sirina u cm.
//Formalni parametra cijena je cijena pize.

int main( )
{
    using namespace std;
    int promjer, duljina, sirina;
    double cijena_okrugle, jedinica_cijena_okrugle,
           cijena_pravokutne, jedinica_cijena_pravokutne;

    cout << "Unesi promjer (cm) za okruglu pizzu: ";
    cin >> promjer;
    cout << "Unesi cijenu okrugle pize: kn ";
    cin >> cijena_okrugle;
    cout << "Unesi duljinu i sirinu (cm)\n"
         << "za pravokutnu pizzu: ";
    cin >> duljina >> sirina;
    cout << "unesi cijenu pravokutne pize: kn ";
    cin >> cijena_pravokutne;

    jedinica_cijena_pravokutne =
        jedinica_cijena(duljina, sirina, cijena_pravokutne);
    jedinica_cijena_okrugle = jedinica_cijena(promjer, cijena_okrugle);

    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << endl
         << "Okrugla pizza: promjer = "
         << promjer << " cm\n"
         << "cijena = kn " << cijena_okrugle
         << " \nPo kvadratnom cm = kn " << jedinica_cijena_okrugle
         << endl
         << "\nPravokutna pizza: duljina = "
         << duljina << " cm\n"
         << "sirina = "
         << sirina << " cm\n"
         << "cijena = kn " << cijena_pravokutne
```

```
<< " \nP o kvadratnom cm = kn " << jedinica_cijena_pravokutne
<< endl;

if (jedinica_cijena_okrugle < jedinica_cijena_pravokutne)
    cout << "\n\nOkrugla se vise isplati.\n";
else
    cout << "\n\nPravokutna se vise isplati.\n";

return 0;
}

double jedinica_cijena(int promjer, double cijena)
{
    const double PI = 3.14159;
    double radius, area;

    radius = promjer/static_cast<double>(2);
    area = PI * radius * radius;
    return (cijena/area);
}

double jedinica_cijena(int duljina, int sirina, double cijena)
{
    double area = duljina * sirina;
    return (cijena/area);
}
```

```
/*
Unesi promjer (cm) za okruglu pizzu: 40
Unesi cijenu okrugle pizze: kn 30
Unesi duljinu i sirinu (cm)
za pravokutnu pizzu: 23
20
unesi cijenu pravokutne pizze: kn 60

Okrugla pizza: promjer = 40 cm
cijena = kn 30.00
Po kvadratnom cm = kn 0.02

Pravokutna pizza: duljina = 23 cm
sirina = 20 cm
cijena = kn 60.00
Po kvadratnom cm = kn 0.13

Okrugla se vise isplati.
*/
```

### Automatska pretvorba tipa

Dana je definicija funkcije:

```
double kpl(double km, double litara)
{
    return (km / litara);
}
```

Što će se dogoditi ako kpl pozovemo kako slijedi?

```
cout << kpl(45, 2) << " km po litri";
```

Vrijednost parametara će se automatski pretvoriti u tip double (45.0 i 2.0).

### Problem pretvorbe tipa

Neka se uz prethodnu definicije funkcije kpl u programu definira još jedna definicija iste funkcije.

```
int kpl(int pokusaja, int promasaja)
// vraća broj pogodaka
{
    return (pokusaja – promasaja);
}
```

Što se događa ako se kpl poziva kako slijedi?

```
cout << kpl(45, 2) << " km po litri";
```

Prevoditelj bira funkciju koja odgovara parametrima po tipu i izračunava se broj savršenih pogodaka.

### PODRAZUMIJEVANI PARAMETRI FUNKCIJE

Neki formalni parametri mogu imati podrazumijevane (default) vrijednosti, a drugi ih nemaju. Svi formalni parametri sa podrazumijevanim vrijednostima moraju biti na kraju liste parametara. Stvarni parametri se prosljeđuju na mjestu ovih formalnih parametara redom kao što smo vidjeli u dosadašnjim primjerima.

Poziv funkcije mora osiguravati najmanje toliko parametara koliko ima formalnih parametara bez podrazumijevane vrijednosti.

Primjer:

```
void default_param(int arg1, int arg2 = -3)
{
    cout << arg1 << ' ' << arg2 << endl;
}
```

default\_param može se pozvati sa jednim ili sa dva parametra

```
default_param(5); //izlaz je 5 -3
```

```
default_param(5, 6); //izlaz je 5 6
```

## 16. TEHNIKE PROGRAMIRANJA: REKURZIJA

Koncept **rekurzije** je temeljni koncept u matematici i računarstvu. **Rekurzivna funkcija** u programu napisanom na nekom programskom jeziku je ona funkcija koja poziva samu sebe (kao što je u matematici rekurzivna funkcija ona koja je definirana pomoću same sebe). Rekurzivna funkcija ne može samu sebe «vječno» pozivati, jer funkcija nikada ne bi završila sa izvođenjem (teoretski, ali zbog ograničenog prostora na **stogu** u jednom trenutku dolazi do **preljeva stoga** i prekida se izvođenja programa). Zbog toga je drugi važan element kod definiranja rekurzivne funkcije **uvjet završetka** njezinog **izvođenja**.

Osnovni nam je cilj upoznati rekurzivne algoritme i neke rekurzivne strukture podataka kroz praktičnu primjenu.

U okviru ovog poglavlja se bavimo vezom rekurzivne definicije funkcije u matematici i jednostavnih rekurzivnih funkcija, te ćemo promotriti nekoliko primjera rekurzivnih funkcija. Slijedeće teme obuhvatiti će metodu «**podijeli i vladaj**» i **dinamičko programiranje** (temeljni pristup na kojem se često temelji rekurzivni postupak sa ciljem izvedbe učinkovitih rješenja problema). Baviti ćemo se i **nerekurzivnim rješenjima** istih problema kao alternativnim postupcima.

### Rekurzivni algoritmi

**Rekurzivni algoritam** je algoritam koji rješava problem tako da rješava jedan ili više jednostavnijih potproblema istog problema. Kad izvodimo rekurzivne algoritme u C++-u, koristimo rekurzivne funkcije. Rekurzivna funkcija u C++-u odgovara rekurzivnoj definiciji matematičke funkcije.

Pogledajmo program koji izračunava faktoriјelu prirodnog broja  $n$ :  $n!$ .

Funkcija je definirana rekurzivno:

$n! = n \cdot (n-1)!$ , za  $n \geq 1$ ,  $0! = 1$

Ova definicija odgovara funkciji za izračunavanje faktoriјele prirodnog broja (rekurzivna izvedba, pogledati detaljnije objašnjenje u [Šribar]):

```
int faktoriјela(int n) // Rekurzivna funkcija računa funkciju  $n!$  uporabom standardne
{
    // rekurzivne definicije. Funkcija vraća točnu vrijednost kada se
    if(n==0) return 1; // poziva za nenegativan i dovoljno mali  $n$ , tako da  $n!$  može biti
    return n*faktoriјela(n-1); // predstavljeno sa int.
}
```

Iterativno rješenje istog problema:

```
for (t=1, i=1; i<=N; i++) t*=i;
```



Uvijek je moguće **rekurzivno rješenje** transformirati u **iterativno rješenje** istog problema i obratno. Rekurziju obično koristimo jer nam omogućava da izrazimo složene algoritme u **kompaktnom obliku**, kada nema izrazitog gubitka na učinkovitosti. Npr. rekurzivna izvedba funkcije za računanje faktorijele ne zahtijeva uporabu lokalnih varijabli. Iako većina programerskih okruženja za razvoj programa omogućava rekurzivnu izvedbu uz uporabu sistemskog stoga, nije teško napisati jednostavnu rekurzivnu funkciju koja je ekstremno neučinkovita. Moramo biti pažljivi i **izbjegavati** takve **neučinkovite rekurzivne izvedbe**.

Gore navedena funkcija za izračunavanje faktorijele ima osnovne značajke rekurzivne funkcije: **poziva samu sebe (za manju vrijednost parametra) i sadrži uvjet za kraj rekurzije**, pri čemu se izračunava rezultat za taj granični slučaj (za  $n==0$ ,  $n!=1$ ).

Možemo uporabom **matematičke indukcije** provjeriti da funkcija pravilno radi:

- Računa  $0!$  (osnovni slučaj – jednostavan problem za koji je rješenje poznato)
- Pod pretpostavkom da funkcija izračunava  $k!$  za  $k < n$  (induktivna hipoteza), funkcija računa i  $n!$ .

Ovakvo razmišljanje pomaže u rješavanju vrlo složenih problema. Mogućnost dokazivanja pravilnog rada rekurzivne funkcije uz kompaktnost rekurzivnog rješenja čini rekurziju snažnom tehnikom programiranja.

Rekurzivna funkcija mora dakle zadovoljavati dva osnovna dvojstva:

- Mora eksplicitno rješavati osnovni slučaj problema.
- Svaki rekurzivni poziv mora se dogoditi za manje vrijednosti parametara funkcije.

Primjer: **Rekurzivna funkcija koja naglašava potrebu za induktivnim parametrom funkcije** (upitna pravilnost definicije funkcije).

Primjer: Za neparan  $N$  funkcija poziva samu sebe sa parametrom  $3N+1$ . Ako je  $N$  paran, funkcija poziva samu sebe sa parametrom  $N/2$ . Pravilan izračun funkcije ne može se dokazati matematičkom indukcijom, tj ne možemo dokazati da se rekurzija zaustavlja, jer se parametar na smanjuje u svakom slijedećem rekurzivnom pozivu.

```
int enigma (in N)
{
if(N==1) return 1;
if(N%2==0)
    return enigma(N/2);
else return enigma(3*N+1);
}
```

Ako ne postavimo ograničenje na veličinu broja  $N$ , ne možemo znati da li će se rekurzija zaustaviti ili ne. U lancu rekurzivnih poziva koji slijedi rekurzija se zaustavlja:  $\text{enigma}(3)$ ,  $\text{enigma}(10)$ ,  $\text{enigma}(5)$ ,  $\text{enigma}(16)$ ,  $\text{enigma}(8)$ ,  $\text{enigma}(4)$ ,  $\text{enigma}(2)$ ,  $\text{enigma}(1)$  – zadnji poziv

Primjer: **Euklidov algoritam (izračunavanje najvećeg zajedničkog djelitelja za dva cijela broja)**

Ovo je jedan od najstarijih poznatih algoritama (star oko 2000 godina). Algoritam koristi rekurzivan postupak.

```
int nzd(int m,int n)
{
if(n==0)return m;
return nzd(n,m%n);
}
```

Dakle,  $\text{nzd}(x,y)$  se izračunava prema definiciji:

za  $y=0$              $\text{nzd}(x,y)=x$   
inače             $\text{nzd}(x,y)=\text{nzd}(y, x \bmod y)$

Algoritam se temelji na zapažanju da je najveći zajednički djelitelj dvaju cijelih brojeva  $x$  i  $y$ , gdje je  $x > y$ , isti kao najveći zajednički djelitelj od  $y$  i  $x \bmod y$  (ostatak dijeljenja  $x$  sa  $y$ ). Primjeri nizova rekurzivnih poziva:

$\text{nzd}(314159,271828)$ ,  $\text{nzd}(271828,42331)$ ,  $\text{nzd}(42331,17842)$ ,  $\text{nzd}(17842,6647)$ ,  
 $\text{nzd}(6647,4458)$ ,  $\text{nzd}(4458,2099)$ ,  $\text{nzd}(2099,350)$ ,  $\text{nzd}(350,349)$ ,  $\text{nzd}(349,1)$ ,  $\text{nzd}(1,0)$   
-> brojevi su relativno prosti

$\text{nzd}(12,18)$ ,  $\text{nzd}(18,12)$ ,  $\text{nzd}(12,6)$ ,  $\text{nzd}(6,0)$  ->  $\text{nzd}(12,18)=6$

Za Euklidov algoritam **dubina rekurzije** (broj rekurzivnih poziva u lancu) ovisi o aritmetičkim svojstvima parametara.

### Prednosti i nedostaci rekurzije u odnosu na iteraciju

Kao što je već rečeno **svako rekurzivno rješenje problema možemo zamijeniti iterativnim rješenjem i obratno**. Često je **rekurzivno rješenje prirodniji način** za opis izračuna od petlje. Ako programska okolina podržava rekurziju, možemo iskoristiti prednosti toga, ali moramo znati da postoji skrivena cijena takve odluke. **Za vrijeme rekurzivnih poziva se naime funkcije gnijezde jedna unutar druge sve dok ne dođemo do točke kada više nema rekurzivnih poziva već dolazi do vraćanja. Lokalni podaci funkcija i parametri pohranjuju se u sistemski stog. Dubina rekurzije je najveći stupanj gnježđenja funkcijskih poziva za vrijeme izračuna.** Općenito, dubina rekurzije ovisi o ulaznom parametru prvog poziva funkcije. **Veličina stoga koji se koristi za vrijeme rekurzije proporcionalna je dubini rekurzije.** Za opsežnije probleme to može biti **prepreka za izvedbu rekurzivnog rješenja**.

### **Temelj rekurzije: «Podijeli i savladaj»**

**(«Divide and conquer», «podijeli pa savladaj», «podijeli i vladaj»)**

Mnogi rekurzivni programi koriste **dva rekurzivna poziva** od kojih se svaki poziva za jednu polovicu ulaznih podataka. Rekurzija je jedan od najvažnijih pristupa paradigme «podijeli i savladaj» u oblikovanju algoritma i temelj je mnogih vrlo važnih algoritama.

Kao primjer promotrimo problem nalaženja maksimuma između N podataka pohranjenih u polje  $a[0], \dots, a[N-1]$ . Ovaj problem možemo jednostavno riješiti iterativno jednim prolaskom kroz polje:

```
for(t = a[0], i=1; i<N; i++)  
    if (a[i] > t) t=a[i];
```

Rekurzivno rješenje koje se temelji na metodi »podijeli i savladaj« je isto jednostavan algoritam (iako potpuno drukčiji) koji rješava dani problem. Na njemu ćemo ilustrirati koncept »podijeli i savladaj«.

### **Primjer 1: «Podijeli i savladaj» u traženju maksimuma polja**

Funkcija dijeli polje  $a[l], \dots, a[r]$  na  $a[l], \dots, a[m]$  i  $a[m+1], \dots, a[r]$ , nalazi najveći element u oba dijela (rekurzivno) i vraća veći od ta dva elementa kao maksimum cijelog polja.

Podrazumijeva se da je za elemente polja definirana relacija «>». Ako je broj elemenata u polju paran, oba dijela su jednake veličine, Ako je veličina polja neparan broj, tada je prva polovica polja za 1 veća od druge polovice.

```
#include <iostream.h>  
#include <iomanip.h>
```

```
int max(int a[], int l, int r);
```

```
void main()  
{  
    int a[]={ 1,4,300,6,90};  
    cout<<"\n Maksimum: "<<max(a,0,4);  
}
```

```
int max(int a[], int l, int r)  
{  
    int u,v;  
    int m=(l+r)/2;  
    if(l==r)return a[l];  
    u=max(a,l,m);  
    v=max(a, m+1,r);  
    if(u>v)return u;  
    else return v;  
}
```

Često metodu «podijeli i savladaj» koristimo jer osigurava rješenja brže nego jednostavni iterativni algoritmi, a istovremeno je to i način razumijevanja i sagledavanja temeljnih izračuna.

Slijedi prikaz rekurzivnih poziva koji se aktiviraju kada se rekurzivno poziva funkcija max za slijedeće polje (tip elemenata polja je char).

|   |   |   |   |   |   |   |   |   |   |    |
|---|---|---|---|---|---|---|---|---|---|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| T | I | N | Y | E | X | A | M | P | L | E  |

```
Y max(0,10)
  Y max(0,5)
    T max(0,2)
      T max(0,1)
        T max(0,0)
          I max(1,1)
            N max (2,2)
              Y max(3,5)
                Y max(3,4)
                  E max(4,4)
                    X max(5,5)
                      P max(6,10)
                        P max(6,8)
                          M max(6,7)
                            A max(6,6)
                              M max(7,7)
                                P max(8,8)
                                  L max(9,10)
                                    L max(9,9)
                                      E max(10,10)
```

Struktura rekurzivnih poziva djeluje složeno, ali možemo se osloniti na provjeru pravilnosti izračuna indukcijom:

- Funkcija max nalazi maksimum za polje veličine 1 eksplicitno i trenutno (granični slučaj).
- Za  $N > 1$  funkcija dijeli polje na dva polja veličine manje od  $N$ , nalazi maksimum za oba dijela, na temelju induktivne hipoteze (pogledaj o matematičkoj indukciji Sedgwick, pogl.5 – materijali uz prethodno predavanje). Funkcija vraća veću od dviju vrijednosti i ta vrijednost predstavlja maksimum cijelog polja.

Vrijedi svojstvo: **Rekurzivna funkcija koja dijeli problem veličine  $N$  na dva nezavisna (neprazna dijela) koji se rješavaju rekurzivno, poziva samu sebe manje od  $N$  puta.** (Svojstvo se dokazuje indukcijom.)

Kako ovaj program za vrijeme svakog rekurzivnog poziva obavlja isti opseg posla, njegova vremenska zahtjevnost je linearna. Neki algoritmi tipa »podijeli i savladaj« mogu obavljati i više posla za vrijeme svakog funkcijskog poziva, pa određivanje ukupnog vremena potrebnog za izvođenje funkcije zahtijeva složeniju analizu. Vrijeme izvođenja takvih algoritama ovisi o preciznom dijeljenju problema na dijelove. U prethodno promatranoj funkciji max ukupni

zbroj dijelova problema daje upravo kompletan problem. Neki drugi algoritam može dijeliti problem na manje dijelove tako da zbroj ne čini cijeli problem, ili zbroj može biti i veći od početnog problema u slučaju kada dolazi do preklapanja dijelova. Ti algoritmi su također rekurzivni jer je svaki dio manji od početnog problema, ali njihova je analiza puno složenija od analize naše funkcije max.

Npr. algoritam binarnog pretraživanja koji smo izveli iterativno, može se izvesti i rekurzivno kao algoritam tipa «podijeli i savladaj». Ovaj algoritam problem u svakom koraku dijeli na pola i zatim savladava točno jednu polovicu problema.

### Primjer: **Rekurzivna izvedba binarnog pretraživanja**

Postupak: Uređeno polje elemenata dijeli se na dva dijela, a nakon toga se usredotočimo na onaj dio u kojem se nalazi ključ pretraživanja. Indekse polja koristimo za praćenje donje i gornje granice dijela polja koje se trenutno pretražuje.

Program s rekurzivnom funkcijom:

```
#include <iostream.h>

int binarnoPretrazivanje(int s[],int l, int r, int kljuc);

void main()
{
    int st[]={1,4,5,6,90};
    int kljuc=90,l=0,r=4,pozicija;
    pozicija=binarnoPretrazivanje(st,l,r,kljuc);
    if(pozicija!=-1)cout<<"\n Pozicija elementa: "<<pozicija<<endl;
        else cout<<"\nElement nije nadjen."<<endl;
}

int binarnoPretrazivanje(int s[],int l, int r, int kljuc)
{
    int m=(l+r)/2; // nadji sredinu
    if(l>r)return -1; // pretraživanje nije uspjelo
    if (kljuc==s[m]) return m; // usporedi vrijednost na sredini sa ključem
    if(l==r) return -1;
    // prepolovi listu
    if (kljuc<s[m])
        return binarnoPretrazivanje(s,l,m-1,kljuc);
    else return binarnoPretrazivanje(s,m+1,r,kljuc);
}
```

U postupku traženja zadanog ključa v u uređenom polju, najprije se uspoređuje element v sa elementom na srednjoj poziciji polja. Ako je v manji tada mora biti u prvoj polovici polja; ako je veći, tada se nalazi u drugoj polovici polja.

Polje mora biti uređeno. Ako polje nije uređeno moramo prethodno primjenom nekog postupka sortiranja urediti (sortirati) polje.

Primjer binarnog pretraživanja polja sa znakovnim elementima (za ključ L):

```
A A A C E E E G H I L M N P R
                H I L M N P R
                H I L
```

## L

Npr. za polje sa 200 elemenata binarno pretraživanje se izvrši u 7 koraka. Svaki slijedeći dio polja u kojem se vrši pretraživanje je manje od prethodnog dijela.

Svojstvo binarnog pretraživanja: **Binarno pretraživanje nikada ne koristi više od  $\lfloor \log N \rfloor + 1$  uspoređivanja prilikom pretraživanja (bez obzira da li je ključ pronađen ili nije i bez obzira da li ga izvodimo iterativno ili rekurzivno).**

U slučaju da polje nije sortirano isplati se primijeniti i neučinkovito sortiranje kao što je sortiranje umetanjem, ako pretraživanje vršimo više puta, jer učinkovitost binarnog pretraživanja to nadoknađuje.

Što se tiče primjene ovog algoritma na povezanoj listi, bez obzira da li izvodimo binarno pretraživanje iterativno ili rekurzivno, još uvijek imamo problem pristupanja središnjem elementu povezane liste. Dakle, povezana lista nije pogodna struktura za primjenu binarnog pretraživanja, jer pristup središnjem elementu liste nije direktan (moramo krenuti od glave liste slijedeći veze). Da bi mogli kombinirati učinkovitost binarnog pretraživanja sa fleksibilnošću povezanih struktura trebaju nam puno složenije strukture kao što su **stabla**. Procesiranje stabla je uz metodu «podijeli i savladaj»

Primjeri algoritama koji koriste tehniku «posijeli i savladaj»: Sortiranje spajanjem i brzo sortiranje.

## **Dinamičko programiranje**

Temeljna značajka algoritama tipa «podijeli i savladaj» koje smo razmatrali je da problem dijele na nezavisne potprobleme. Kada potproblemi nisu nezavisni, situacija postaje složenija, u prvom redu zato jer rekurzivna izvedba i za najjednostavnije algoritme može zahtijevati nezamislivo puno vremena. U ovom poglavlju se bavimo sustavnim tehnikama za izbjegavanje ovog problema za važnu klasu problema.

### **Primjer: Rekurzivna izvedba izračunavanja Fibonaccijevih brojeva**

Ovaj algoritam je vrlo neučinkovit i njegova primjena se ne preporučuje. Vremensku zahtjevnost ovog algoritma za jednostavani izračun opisuje eksponencijalna funkcija! Vrijeme potrebno za izračunavanje  $F_{N+1}$  je 1.6 puta veće od vremena potrebnog za izračunavanje  $F_N$ . Ako našem računalu treba oko 1 sekunda da izračuna  $F_N$ , tada će trebati više od minute da izračuna  $F_{N+9}$  i više od 1 sata za izračunavanje  $F_{N+18}$ .

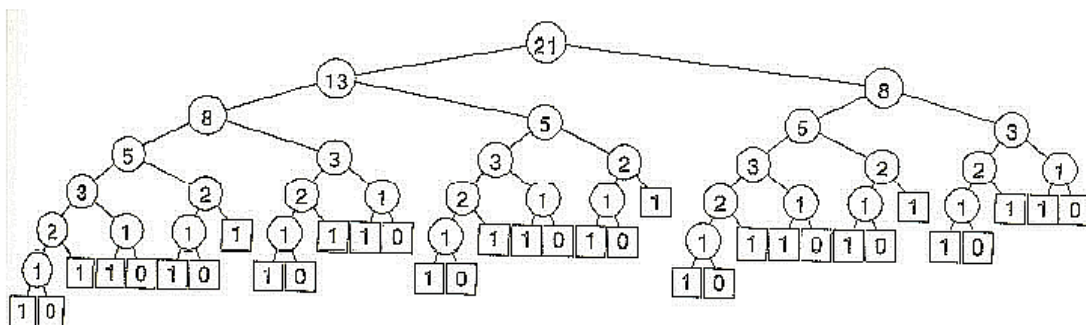
Kod:

```
int F(int i)
{
    if (i<1) return 0;
    if (i==1) return 1;
    return F(i-1)+F(i-2);
}
```

$$\}$$

### Struktura rekurzivnog algoritma:

Prikazan je niz rekurzivnih poziva koji se koriste u izračunu F8. Potproblemi se preklapaju što uzrokuje eksponencijalnu cijenu algoritma. Npr. drugi rekurzivni poziv zanemaruje izračune iz prvog poziva, što dovodi do masovne pojave ponavljanja istih izračuna.

$$\begin{array}{r}
8 \text{ F}(6) \\
5 \text{ F}(5) \\
3 \text{ F}(4) \\
2 \text{ F}(3) \\
1 \text{ F}(2) \\
1 \text{ F}(1) \\
0 \text{ F}(0) \\
1 \text{ F}(1) \\
1 \text{ F}(2) \\
1 \text{ F}(1) \\
0 \text{ F}(0) \\
2 \text{ F}(3) \\
1 \text{ F}(2) \\
1 \text{ F}(1) \\
0 \text{ F}(0) \\
1 \text{ F}(1) \\
3 \text{ F}(4) \\
2 \text{ F}(3) \\
1 \text{ F}(2) \\
1 \text{ F}(1) \\
0 \text{ F}(0) \\
1 \text{ F}(1) \\
1 \text{ F}(2) \\
1 \text{ F}(1) \\
0 \text{ F}(0)
\end{array}$$


Fibonaccijevi brojevi se mogu računati i u linearnom vremenu (proporcionalnom sa  $N$ ) tako da se najprije izračuna prvih  $N$  brojeva koji se pohrane u polje:

```
F[0]=0; F[1]=1;
for(i=2;i<=N;i++)
```

$$F[i]=F[i-1]+F[i-2];$$

Brojevi rastu eksponencijalno pa je polje malo, npr.  $F_{46}=1836311903$  je najveći Fibonaccijev broj koji je još u rasponu 32-bitnog tipa int. Dakle, dovoljno je polje sa 46 elemenata. U ovom slučaju koriste se samo dvije prethodno izračunate vrijednosti. U mnogim drugim rekurzivnim algoritmima koriste se i ostale vrijednosti iz polja.

U prethodnom primjeru rekurzivni algoritam počinje izračun sa najmanjim vrijednostima, a svaka slijedeća vrijednost se računa na temelju prethodno izračunatih vrijednosti. Ova tehnika je poznata kao **«bottom-up» dinamičko programiranje (s dna prema vrhu)**. Primjenjiva je u svim rekurzivnim algoritmima ako se može osigurati smještaj za sve prethodno izračunate vrijednosti. To je tehnika oblikovanja algoritma koja se koristi za rješavanje širokog raspona problema. **Ova jednostavna tehnika skraćuje vrijeme izvođenja algoritma od eksponencijalnog na linearno.**

Još jednostavnija tehnika koja osigurava rekurzivnim funkcijama istu ili manju cijenu izvođenja automatski, je **«top-down» dinamičko programiranje (od vrha prema dnu)**. Opremimo rekurzivnu funkciju mehanizmom koji sprema svaku vrijednost koju funkcija izračuna (konačna akcija funkcije) i provjerava spremljene vrijednosti da bi se izbjeglo ponovno računanje neke od tih vrijednosti (prva akcija funkcije).

#### Primjer: Izračunavanje Fibonaccijevih brojeva (dinamičko programiranje)

Spremanjem vrijednosti koje se računaju u polje izvan rekurzivne funkcije, eksplicitno izbjegavamo ponovni izračun. Slijedeća funkcija računa  $F_N$  u vremenu proporcionalnom sa  $N$ , u suprotnosti sa vremenom  $O(2^N)$  prethodne funkcije. Koristi se «top-down» dinamičko programiranje.

```
#include <iostream.h>
#include <iomanip.h>

const int max=50;
int spremljeniF[max];
int F(int i);

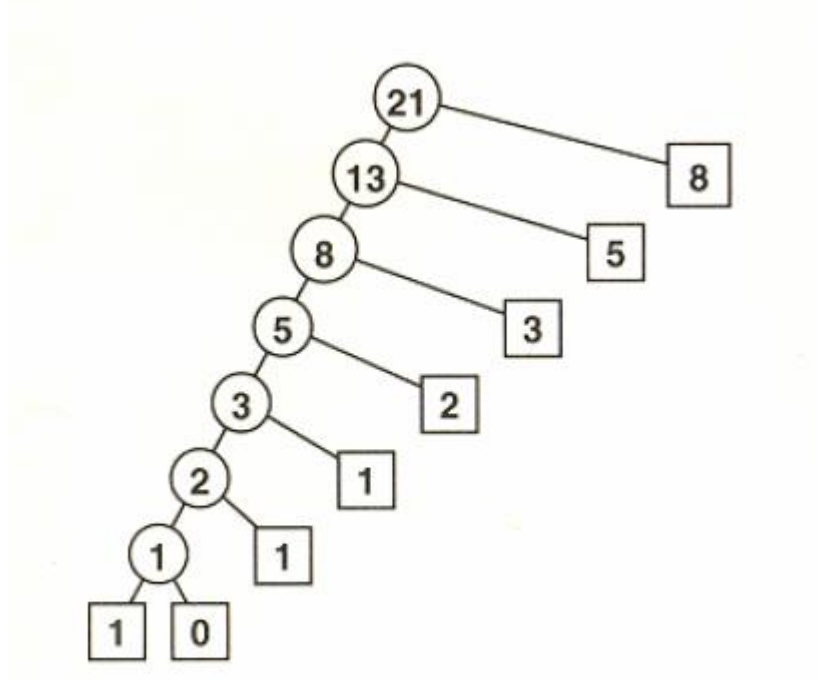
void main()
{
    for(int i=0;i<max;i++)
    {
        spremljeniF[i]=0;
    }
    for(i=0;i<=46;i++)
    {
        cout<<i<<" "<<F(i)<<endl;
    }
}

int F(int i)
{
```



```
if(spremljeniF[i] > 0) return spremljeniF[i];
if(i==0) return 0;
if(i==1) return 1;
spremljeniF[i]=F(i-1) + F(i-2);
return spremljeniF[i];
}
```

Slika prikazuje rekurzivne pozive pri računanju Fibonaccijevih brojeva uz uštedu vremena jer se u svakom slijedećem izračunu koriste spremljene vrijednosti. «Top-down» dinamičko programiranje se ponekad zove i **memoizacija**.



Mnogo složeniji problem koji se rješava istom tehnikom je problem «knapsack».

#### Primjer: «Knapsack» problem (problem naprtnjače)

Lopov pljačka sef i nalazi unutra N tipova predmeta različite veličine i vrijednosti, ali ima samo malu naprtnjaču kapaciteta M za spremanje predmeta. Problem naprtnjače je naći kombinaciju predmeta koje će lopov ponijeti sa ciljem da ukupna vrijednost predmeta bude čim veća.

|            |   |   |    |    |    |
|------------|---|---|----|----|----|
|            | 0 | 1 | 2  | 3  | 4  |
| predmet    | A | B | C  | D  | E  |
| veličina   | 3 | 4 | 7  | 8  | 9  |
| vrijednost | 4 | 5 | 10 | 11 | 13 |

Za gore opisane predmete lopov sa naprtnjačom veličine 17 može ponijeti 5 predmeta A (ali ne i 6) ukupne vrijednosti 20, ili po jedan D i E ukupne vrijednosti 24 ili neku drugu

kombinaciju predmeta. Cilj je naći algoritam koji određuje maksimum između svih mogućnosti za bilo koji kapacitet naprtnjače i bilo koji skup predmeta.

Postoje mnoge aplikacije za koje je problem naprtnjače važan. Primjerice, ovaj problem se javlja pri ukrcaju tereta na brod, kamion ili avion gdje je prostor je ograničen. Pojavljuju se varijante problema: ograničen je broj nekih od predmeta, na raspolaganju možemo imati više kamiona i sl.

U rekurzivnom rješenju ovog problema, svaki put kad odaberemo predmet, pretpostavljamo da možemo rekurzivno pronaći optimalni način za punjenje preostalog dijela naprtnjače. Za naprtnjaču veličine *cap* određujemo za svaki predmet *i* određenog tipa koji imamo na raspolaganju, koliku ukupnu vrijednost možemo nositi ako se u naprtnjači nalazi predmet *i* uz optimalno odabrane ostale predmete. Optimalni odabir predmeta je onaj koji trazimo za manju naprtnjaču veličine *cap*-predmeti[i].veličina. Kada jednom pronađemo optimalne skupove predmeta, ne moramo ponovno ispitati problem, bez obzira koji su ostali predmeti.

Rekurzivno rješenje temeljeno na gornjoj diskusiji je neuporabivo (eksponencijalno vrijeme!); algoritam se lako poboljšava top-down dinamičkim programiranjem:

Pretpostavljamo da su predmeti opisani strukturom:

```
typedef struct{int size;int val;} Predmet;
```

U polju se nalazi N predmeta tipa Predmet.

Za svaki mogući predmet, računamo (rekurzivno) najveću vrijednost koju možemo dobiti ako uključimo taj predmet. Zatim određujemo maksimum od svih tako dobivenih vrijednosti.

```
int knap (int cap)
{
    int i, space, max, t;
    for(i=0,max=0;i<N;i++)
        if((space=cap-predmeti[i].size)>=0)
            if((t=knap(space)+ipredmeti[i].val)>max)
                max=t;
    return max;
}
```

Varijanta sa dinamičkim programiranjem:

```
int knap(int M) {
int i, space, max, maxi, t;
    if (maxKnown[M] != unknown) return maxKnown[M];
    for (i = 0, max = 0; i < N; i++)
        if ((space = M-items[i].size) >= 0)
            if ((t = knap(space) + items[i].val) > max)
                { max = t; maxi = i; }
    maxKnown[M] = max; itemKnown[M] = items[maxi];
return max;
}
```

Sve vrijednosti koje se izračunaju se spremaju, a kada su potrebne pristupa im se.

**Svojstvo: Dinamičko programiranje reducira vrijeme izvođenja rekurzivne funkcije najviše na vrijeme potrebno za vrednovanje funkcije za sve parametre manje ili jednake danom parametru, smatrajući cijenu rekurzivnog poziva konstantom.**

U problemu naprtnjače ovo svojstvo implicira da je vrijeme izvođenja proporcionalno umnošku  $M \cdot N$ . To znači da problem naprtnjače možemo jednostavno riješiti za malu naprtnjaču, ali za veći kapacitet naprtnjače vremenski i prostorni zahtjevi algoritma mogu naglo narasti.

Dinamičko programiranje «sa dna prema vrhu» isto se može primijeniti u problemu naprtnjače. Za razliku od pristupa «sa vrha prema dnu» treba samo paziti da se vrijednosti funkcije izračunavaju pravim redoslijedom, tako da se svaka potrebna vrijednost izračuna na vrijeme. Za funkcije sa jednim cjelobrojnim parametrom, jednostavno nastavljamo u rastućem poretku parametra. Za složenije rekurzivne funkcije, određivanje pravog redoslijeda može biti izazov.

Ne moramo se ograničiti na rekurzivne funkcije sa jednim cjelobrojnim parametrom. Kada imamo funkciju sa više cjelobrojnih parametara, možemo spremati rješenja manjih problema u višedimenzionalna polja, jedno za svaki parametar.

Dakle, u pristupu «s vrha prema dnu» dinamičkog programiranja spremamo poznate vrijednosti, a u pristupu «sa dna prema vrhu» ih prethodno računamo. **Obično dajemo prednost pristupu «sa vrha prema dnu» jer:**

- **To je mehanička transformacija prirodnog rješenja problema.**
- **Ne moramo brinuti o poretku izračuna potproblema.**
- **Ne moramo izračunavati vrijednosti za sve potprobleme.**

Aplikacije temeljene na dinamičkom programiranju razlikuju se po prirodi potproblema i po količini informacija koje moramo pohraniti za potprobleme.

U dinamičkom programiranju ne možemo previdjeti činjenicu da postaje neupotrebivo kada je broj mogućih vrijednosti funkcije koje su nam potrebne toliko velik da nema dovoljno prostora za njihovo spremanje (s vrha prema dnu) ili prethodno računanje (s dna prema vrhu).

Dinamičko programiranje je tehnika oblikovanja algoritama prvenstveno prikladna za napredne probleme kao što su:

- uzastopno množenje više matrica (<http://www.cs.sunysb.edu/~algorith/lectures-good/node12.html>)
- izračunavanje sličnosti dvaju nizova znakova (Levenshtein-ova udaljenost: <http://www.levenshtein.net/index.html>)
- općenito raspoznavanje uzoraka (<http://www.stat.washington.edu/wxs/Stat534-s06/Slides/dynamic-programming-3-22-06.pdf>)

|                    |
|--------------------|
| <b>Literatura:</b> |
|--------------------|

- Walter Savitch: Problem Solving in C++, Pearson Publishing, 2006.
- J.Šribar, B.Motik: Demistificirani C++, Dobro upoznajte protivnika da biste njime ovladali, Element, Zagreb, 2001. (prvih 100 stranica i primjeri iz knjige dostupno je na web-u na adresi <http://free-zg.htnet.hr/jsribar/download.html> )
- Nina Lipljin: Programiranje/1, TIVA Tiskara Varaždin, 2004.
- Gvozdanović, Ikica, Kos, Lipljin, Milijaš, Srnec, Zvonarek: Informatika / Računalstvo 1 i 2, udžbenik za 1. i 2. razred srednjih škola, PRO-MIL, Varaždin, 2005.
- Vulin, R.: Zbirka riješenih zadataka iz C-a, Školska knjiga, Zgb, 2003.
- Robert Sedgewick: Algorithms in C, Addison-Wesley, 1998.
- Brođanac