

C++

# Uvod u programiranje

*01010101011101100110111101100100 01110101*

*011100000111001001101111011001110111001001100001011011010110*

*10010111001001100001011011100110101001100101*

Saša Fajković

2014



## Sadržaj :

|   |    |
|---|----|
| 1) Što je to programiranje?.....  | 6  |
| 2) Razvojna sučelja i programski jezici.....                                    | 7  |
| a. Microsoft Visual Studio .....  | 7  |
| 3) Instalacija i konfiguracija Microsoft Visual C++ 2010 Express .....          | 8  |
| 4) Prva aplikacija .....  | 13 |
| 5) Osnovni pojmovi i savjeti .....  | 15 |
| a. Prevoditelj .....  | 15 |
| b. Program za uređivanje teksta.....  | 15 |
| c. Program za otkrivanje pogrešaka .....  | 15 |
| d. Sintaksa.....  | 15 |
| e. Sintaksna pogreška .....   | 15 |
| f. Logička pogreška .....   | 16 |
| g. Redoslijed izvršavanja programa.....   | 16 |
| h. Numeracija kôdnih linija.....  | 16 |
| i. Intelli-Sense .....  | 16 |
| j. .cpp ekstenzija .....  | 16 |
| k. „#include <iostream>“, „using namespace std“, „int main()“, „return 0“ ..... | 17 |
| l. Naredbe „cin“ i „cout“ (učitavanje iz konzole i ispis u konzolu) .....       | 18 |
| m. Ispis teksta i uloga dvostrukih navodnika.....                               | 19 |
| n. Simbol točka-zarez -> „ ; „ .....  | 19 |
| o. Naredba „endl“ → End Line (prelazak u novi red).....                         | 19 |
| p. Važna napomena.....  | 19 |
| 6) Identifikatori.....  | 20 |
| a. Što su identifikatori i koja su dopušтана imena .....                        | 20 |
| b. Popis zabranjenih riječi.....  | 21 |
| 7) Varijable.....   | 22 |
| a. Što su varijable .....   | 22 |
| b. Deklaracija varijabli .....  | 22 |
| c. Inicijalizacija varijable .....  | 22 |
| d. Konstantne varijable.....  | 22 |
| 8) Tipovi podataka .....  | 23 |
| a. <i>integer</i> .....   | 23 |
| b. <i>float</i> .....   | 23 |
| c. <i>double</i> .....  | 23 |

|     |  |    |
|-----|--|----|
| d.  | <i>character</i> .....   | 23 |
| e.  | <i>boolean</i> .....   | 24 |
| f.  | <i>string</i> .....  | 24 |
| g.  | Popis tipova podataka i njihova svojstva.....  | 24 |
| 9)  | Operatori .....  | 25 |
| a.  | Unarni operatori.....  | 25 |
| b.  | Inkrementacija.....  | 25 |
| c.  | Dekrementacija .....   | 26 |
| d.  | Binarni operatori .....  | 26 |
| e.  | Operatori pridruživanja .....  | 27 |
| f.  | Operatori usporedbe.....   | 28 |
| g.  | Logički operatori.....   | 28 |
| 10) | Komentari.....   | 29 |
| 11) | Uvodni zadatci (tipovi podataka, upis i čitanje iz konzole, <i>sintaksa</i> , operatori) ..... | 30 |
| 12) | Dijagram toka i pseudo kôd.....  | 34 |
| a.  | Pseudo kôd .....   | 34 |
| b.  | Dijagram toka .....  | 35 |
| c.  | Primjer pseudko kôda i dijagrama toka.....   | 36 |
| 13) | Petlje i kontrole toka .....   | 37 |
| 14) | Osnovne kontrole toka → if, if-else, ugniježđeni if.....                                       | 38 |
| a.  | Kontrola toka pomoću „if“-a .....  | 38 |
| b.  | if-else .....  | 40 |
| c.  | Ugniježđeni if.....  | 41 |
| d.  | Zadatci za vježbu kontrole toka .....  | 42 |
| 15) | Petlje for, while, do-while .....  | 43 |
| a.  | for petlja .....   | 43 |
| b.  | <i>Sintaksa</i> i primjeri „for“ petlje : .....  | 44 |
| c.  | while petlja .....   | 48 |
| d.  | <i>Sintaksa</i> i primjeri while petlje .....  | 48 |
| e.  | Beskonačna petlja.....   | 50 |
| f.  | do-while petlja.....   | 51 |
| g.  | Zadatci .....  | 55 |
| 16) | Kontrole toka 2 .....  | 56 |
| a.  | switch .....   | 56 |
| 17) | Formatiranje ispisa - osnove .....   | 60 |
| a.  | <<endl;.....   | 60 |

|     |   |    |
|-----|---|----|
| b.  | \n .....  | 61 |
| c.  | Zadatci .....   | 62 |
| 18) | Polja (nizovi) .....  | 63 |
| a.  | uvod.....   | 63 |
| b.  | Čitanje i pridruživanje vrijednosti elementu polja .....                            | 64 |
| c.  | Korištenje konstantnih vrijednosti varijabli za deklaraciju velične polja .....     | 65 |
| d.  | for petlja za brže čitanje elemenata polja .....                                    | 66 |
| e.  | for petlja za brže upisivanje u polje.....  | 68 |
| f.  | Objedinjen unos i ispis vrijednosti elemenata polja pomoću „for“ petlje .....       | 69 |
| g.  | Zadatci .....   | 70 |
| 19) | Stringovi – uvod.....   | 71 |
| a.  | Uvod – što je <i>string</i> tip podatka .....                                       | 71 |
| b.  | Stringovi NISU primitivni tipovi podataka .....                                     | 71 |
| c.  | Ispis i učitavanje <i>stringova</i> .....   | 71 |
| d.  | Zadatci .....   | 73 |
| 20) | Funkcije – uvod.....  | 74 |
| a.  | Što su funkcije i koja je njihova uloga.....  | 74 |
| b.  | Korištenje prototipa funkcije.....  | 75 |
| c.  | Dekompozicija problema korištenjem funkcija .....                                   | 76 |
| d.  | Funkcije i argumenti .....  | 76 |
| e.  | Stvaranje i korištenje privremenih varijabli unutar funkcije .....                  | 76 |
| f.  | Primjeri korištenja funkcija za uređivanje ispisa .....                             | 76 |
| g.  | Primjeri korištenja funkcija za matematičke izračune.....                           | 76 |
| h.  | Primjeri korištenja funkcija za ispisivanje isto teksta određeni broj puta.....     | 77 |
| 21) | Funkcije s „return“ naredbom bez parametara .....                                   | 78 |
| a.  | Što su tipovi funkcija, <i>sintaksa</i> funkcija i što je to „return“ naredba ..... | 78 |
| b.  | Jednostavne funkcije koje vraćaju „ <i>integer</i> “ tip podatka .....              | 78 |
| c.  | Jednostavne funkcije koje vraćaju „ <i>float</i> “ tip podatka .....                | 79 |
| d.  | Jednostavne funkcije koje vraćaju „ <i>boolean</i> “ tip podatka .....              | 80 |
| e.  | Jednostavne funkcije koje vraćaju „ <i>string</i> “ tip podatka.....                | 81 |
| f.  | Jednostavne funkcije koje ne vraćaju ništa – VOID.....                              | 82 |
| g.  | Zadatci .....   | 84 |
| 22) | Funkcije koje primaju argumente istog tipa.....                                     | 85 |
| a.  | Uvod .....  | 85 |
| b.  | Funkcije koje primaju jedan argument s „return“ naredbom .....                      | 85 |
| c.  | Funkcije koje primaju jedan argument bez „return“ naredbe .....                     | 87 |

|     |   |     |
|-----|---|-----|
| d.  | Funkcije koje primaju više argumenata s „return“ naredbom .....                           | 88  |
| e.  | Zadatci .....   | 90  |
| 23) | Formatiranje ispisa 2 – naprednije formatiranje.....                                      | 91  |
| a.  | biblioteka <iomanip> - uvjet za bolje formatiranje .....                                  | 91  |
| b.  | setprecision() .....  | 91  |
| c.  | width() .....   | 93  |
| d.  | Zadatci .....   | 94  |
| 24) | Stringovi 2 – korištenje funkcija nad stringovima .....                                   | 95  |
| a.  | Uvod .....  | 95  |
| b.  | getline() funkcija.....   | 95  |
| c.  | length() funkcija.....  | 98  |
| d.  | prolaz slovo po slovo kroz <i>string</i> pomoću „for“ i „while“ petlje .....              | 99  |
| e.  | toupper() i tolower() funkcije .....  | 102 |
| f.  | atoi funkcija .....   | 105 |
| g.  | Slične funkcije za pretvorbu iz <i>string</i> tipa u numerički tip – atof(), atol() ..... | 106 |
| h.  | Funkcija find() .....   | 106 |
| i.  | Funkcija insert() .....   | 107 |
| j.  | Funkcija replace() .....  | 107 |
| k.  | Funkcija clear() .....  | 107 |
| l.  | Funkcija empty().....   | 107 |
| m.  | Zadatci .....   | 108 |
| 25) | Izrada vlastitih funkcija.....  | 109 |
| a.  | Rekurzija .....   | 109 |
| b.  | Prva vlastita funkcija.....   | 109 |
| c.  | O izradi vlastitih funkcija .....   | 110 |
| d.  | Preopterećenje funkcija .....   | 110 |
| 26) | Objektno orijentirano programiranje.....  | 112 |
| 27) | Popis slika .....   | 113 |
| 28) | Popis tablica: .....  | 115 |

# 1) Što je to programiranje?

Opće je poznato kako je programerski posao zapravo posao snova kojeg karakterizira velika plaća, rad od kuće, fantastični radni uvjeti i da su takvi ljudi izuzetno inteligentni. Za takav posao ljudi se školuju desetljećima a oni istinski programeri nemaju socijalni život kao niti razvijene socijalne vještine, ali zato imaju pogrbljena leđa, lice puno akni, masnu kosu i nemaju djevojku, ženu ili zaručnicu. Od svih navedenih pretpostavki niti jedna nije točna osim (na apstraktnoj razini) možda one o školovanju. Sagledamo li pojam školovanja kao edukaciju na nekoj visokoobrazovnoj ustanovi onda je teško naći nekoga tko je proveo desetljeće studirajući isti studij, a da istovremeno spada u vrsne programere. Sagledamo li ipak pojam učenja malo šire, onda se zasigurno može reći kako se programeri educiraju desetljećima, no to znači suočavanje s novim problemima, tehnologijama i načinima poslovanja. Slična je situacija i u svakom drugom zanimanju. Promjeni li se računovodstvena aplikacija u nekom poduzeću koje se bavi vođenjem knjiga za druga poduzeća, zaposlenici će se morati naučiti koristiti novim softverom (*engl. software*) što je također učenje odnosno edukacija. Ipak, za programere možemo reći kako je količina i učestalost usvajanja novih znanja češća jer se radi o poslu koji nerijetko zahtijeva jedinstvena rješenja (iako možda slična, ali rijetko kada potpuno identična) a time je i svaki takav zadatak novo učenje.

Važno je napomenuti kako programiranje može biti izuzetno zabavan posao no način usvajanja gradiva je „*bottom-up*“, proces i čini se kao da ništa nema smisla i da nikada nećete stići do cilja da postanete programer. Za buduće programere to znači da će se trebati prvo naučiti određena količina znanja i tek na kraju će se sva ta znanja spojiti u veliku cjelinu i sve će imati smisla. Programiranje ipak, suprotno vjerovanju mnogih, **NIJE** bazirano na jednostavnom grafičkom sučelju. Povlačenje gotovih trodimenzionalnih modela koji nalikuju na ljude, civilne aute, cestu, zgrade i policijske aute nije način kako nastaju igre. Jako puno „dosadnih“ linija teksta i još teksta i još teksta leži iza svega onoga što krajnjim korisnicima omogućuje da pritiskom na tipku „A“ ili „lijevu strelicu“ auto zaista skrene u lijevo i spektakularnim manevrom izbjegnute policijski auto koji Vas je pratio posljednjih dvadeset minuta.

*Summa summarum*, programiranje je jako puno pisanja kôda (*engl. code*) i smišljanja algoritama, ispijanja kave u velikim količina (stereotip koji čak i je često istinit), traženja i velikih, ali i jako malih grešaka te frustriranja svaki put kada klijent zatraži promjenu u programu jer nije znao opisati što točno želi i sada misli kako je dodavanje njegove „male želje“ još samo par minuta posla. Programiranje nije „*drag-n-drop*“ sistem povlačenja grafičkih modela i pritisak tipke „Start“ nakon čega sve nekako magično proradi samo od sebe.

**Programiranje se ne uči štrebanjem.** Sintaksu jezika je potrebno naučiti napamet i prihvatiti je „zdravo za gotovo“, ali dobar programer se postaje iskustvom. Što više kôda napišete postajete sve bolji programer.

**Zapamtite**, računalo radi ono što mu kažemo, ne ono što mi želimo. Jako često u programerskom svijetu, te su dvije stvari potpuno različite.

## 2) Razvojna sučelja i programski jezici

Razvijanje aplikacija (drugi često korišteni nazivi su programiranje, pisanje kôda, kôdiranje i sl.) moguće je čak i u najjednostavnijoj aplikaciji za uređivanje teksta. Primjer je popularni Notepad<sup>1</sup> koji dolazi pred instaliran u sklopu Windows operativnog sustava. Naravno, takva aplikacija nije namijenjena razvijanju aplikacija već pisanju jednostavnog teksta i takav pristup kodiranju nema smisla. Uz danas već zaista veliku paletu aplikacija namijenjenih kodiranju često možete čuti kako se koristi Notepad++, Code::Blocks, Eclipse, Microsoft Visual Studio, NetBeans i slične aplikacije. Njih zovemo **razvojna sučelja** odnosno **integrirana razvojna sučelja** (*engl. integrated development environment*). U sklopu ovakvih specijaliziranih aplikacija nalazimo veliki broj gotovih rješenja koje nam omogućuju više fokusiranja na programiranje i smišljanje algoritama a smanjuju količinu vremena potrebnu za pisanje ponavljajućeg kôda. Naravno, nisu sve aplikacije specijalizirane za sve programske jezike kojih također postoji veliki broj a svaki od njih donosi neke prednosti ali i mane naspram drugih. Uz programski jezik C++, iznimnu popularnost dijele i Java (izgovara se ili java ili đava), C (izgovara se „ce“ ili često zvan „čisti C“), C# (izgovara se „ce šarp“ ili „si šarp“), Python (izgovara se „pajton“) te mnogi drugi. Važno je napomenuti kako **HTML**, **CSS**, **JavaScript** i sl. ne spadaju u programske jezike već u opisne odnosno skriptne jezike.

### a. Microsoft Visual Studio

Kôdiranje u bilo kojem razvojnom sučelju je jako slično. Sve važne stvari koje se tiču samog jezika su uvijek iste, neovisno o razvojnom okruženju. Razlika se očituje u dizajnu sučelja, dodatnim opcijama, pozicioniranjem dijelova aplikacije na ekranu no isti kôd će svuda izvršavati identičan proces. Kroz ove primjere koristiti će se **Microsoft Visual C++ 2010 Express**. Ovo razvojno sučelje je besplatno i moguće ga je preuzeti s ove stranice : <http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express>. Ukoliko u trenutku čitanja ovog teksta postoji novije razvojno sučelje, slobodno preuzmite najnoviju verziju.

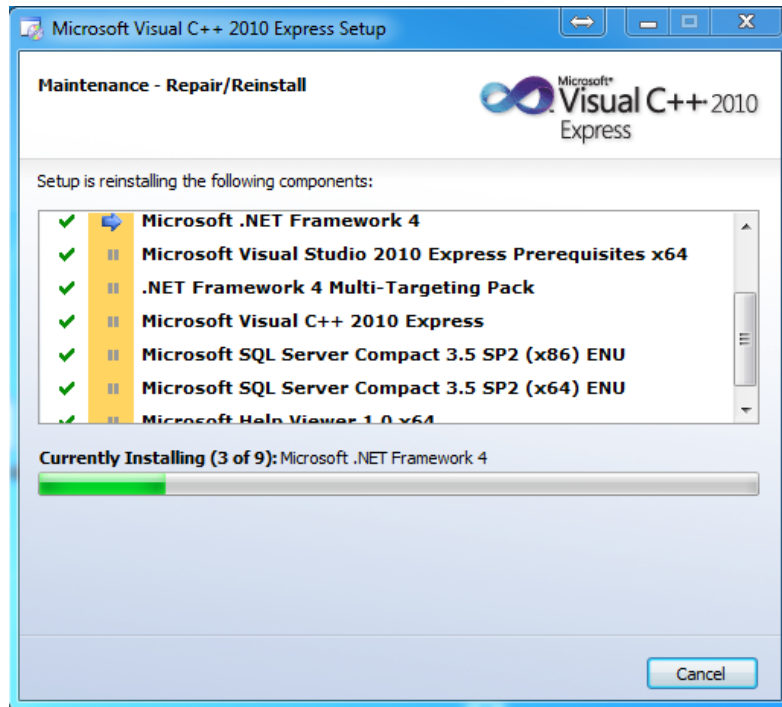
---

<sup>1</sup> Notepad je program za osnovno uređivanje i stvaranje teksta a dolazi u sklopu Microsoft Windows operativnog sustava



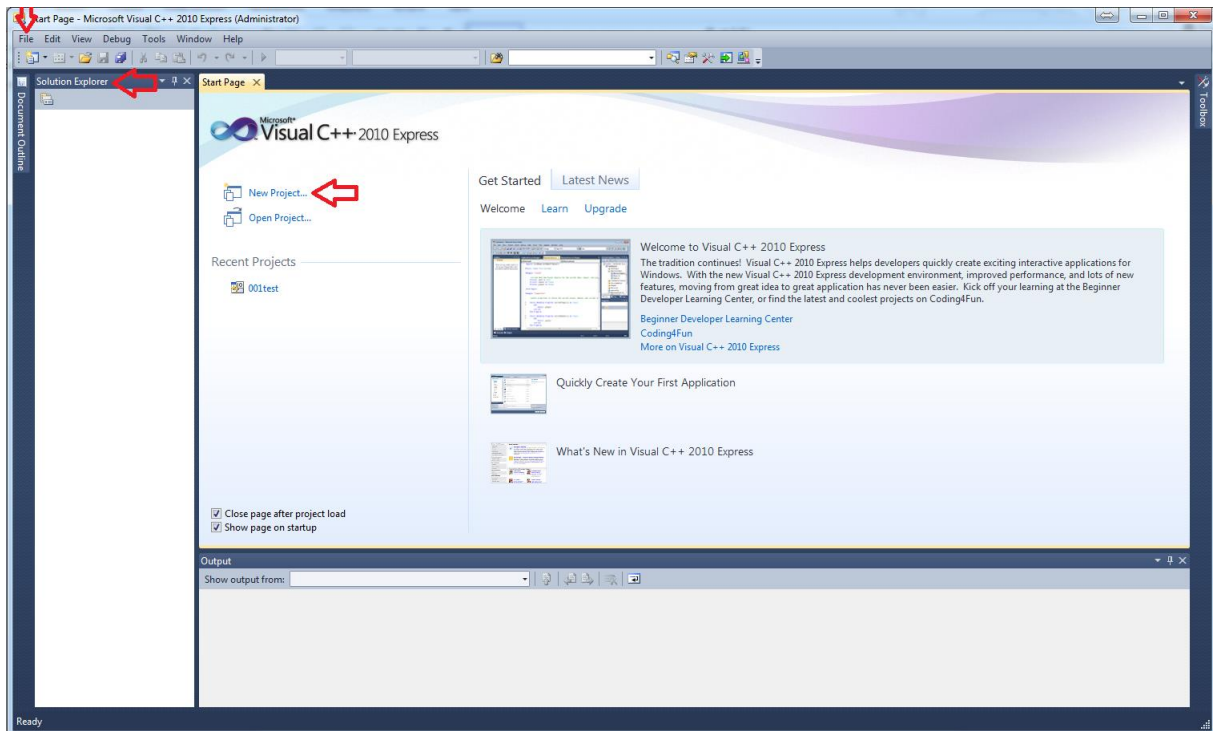
### 3) Instalacija i konfiguracija Microsoft Visual C++ 2010 Express

Potrebno je preuzeti instalacijsku datoteku s prije spomenute web stranice te pokrenuti instalaciju. Prilikom instalacije potrebno je odabrati željeno mjesto instalacije i pratiti čarobnjak za instalaciju. Sama instalacija nije ništa drugačija od bilo koje druge instalacije za Windows operativni sustav.



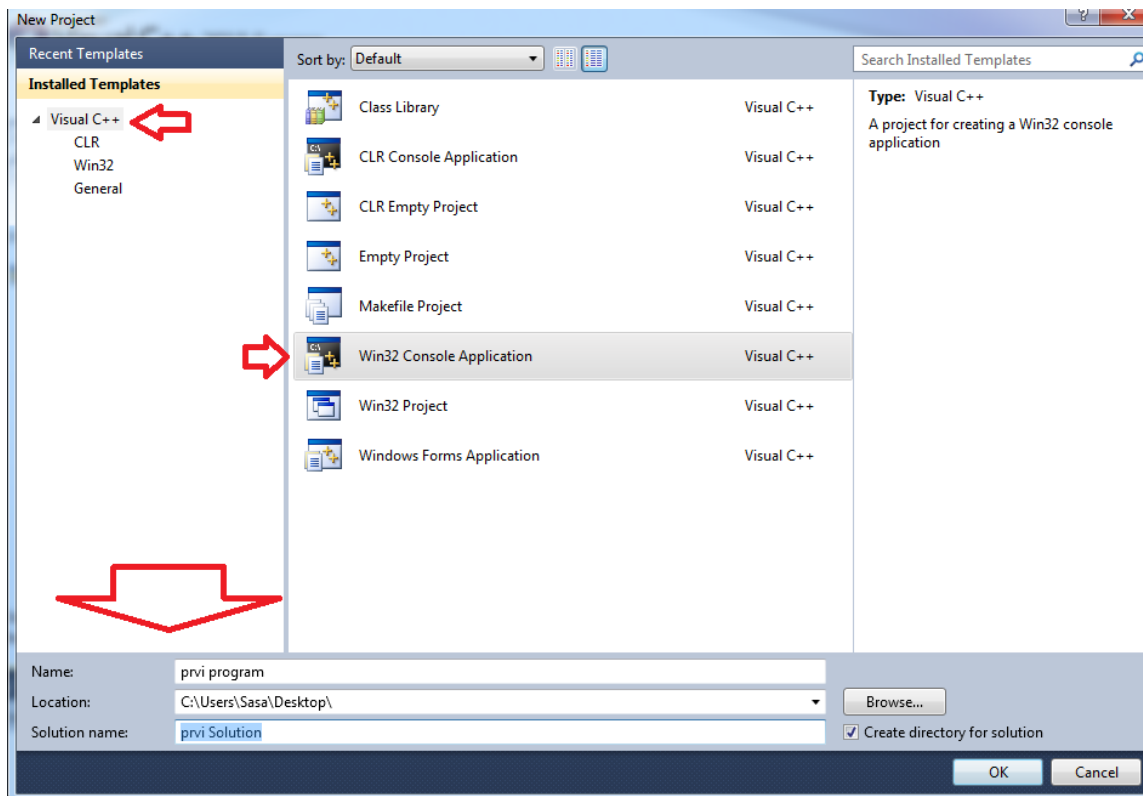
Slika 1 - Instalacija Microsoft Visual C++ 2010 Express razvojnog sučelja

Po završetku instalacije otvorite Microsoft Visual C++ 2010 Express (u daljnjem tekstu „MS VC++“) nakon čega će aplikacija izvršiti početno podešavanje i napokon ste spremni za stvaranje aplikacija.



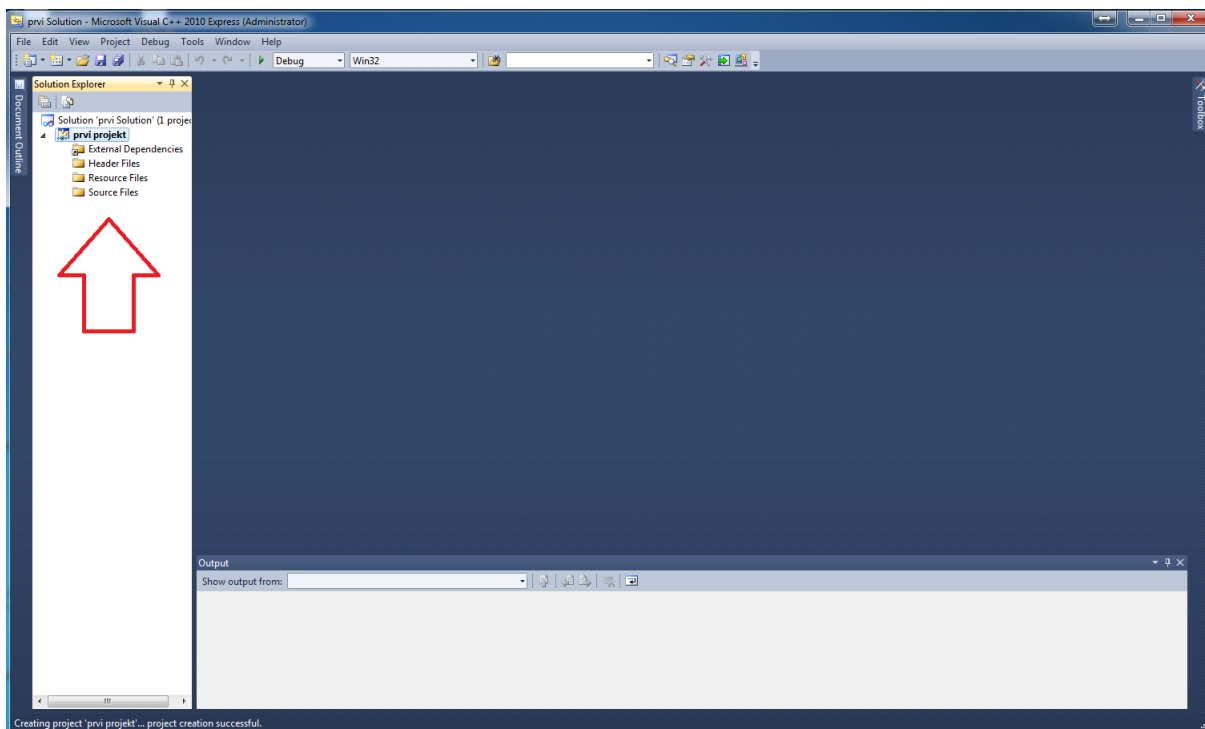
Slika 2 - Početni izbornik nakon otvaranja Microsoft Visual C++ 2010 Express razvojnog sučelja

Odaberite u File – New – Project (kratica je Ctrl+Shtift+N) i na sljedećem izborniku odaberite „**Win32 Console Application**“. U donjem dijelu zadajte ime projekta (primjerice „prvi projekt“), lokaciju na tvrdom disku (*engl. hard drive*) za spremanje projekta te „**Solution name**“ koje je po početnim postavkama jednako imenu projekta. Za potrebe objašnjavanja u polje „Solution name“ upišite „prvi solution“.



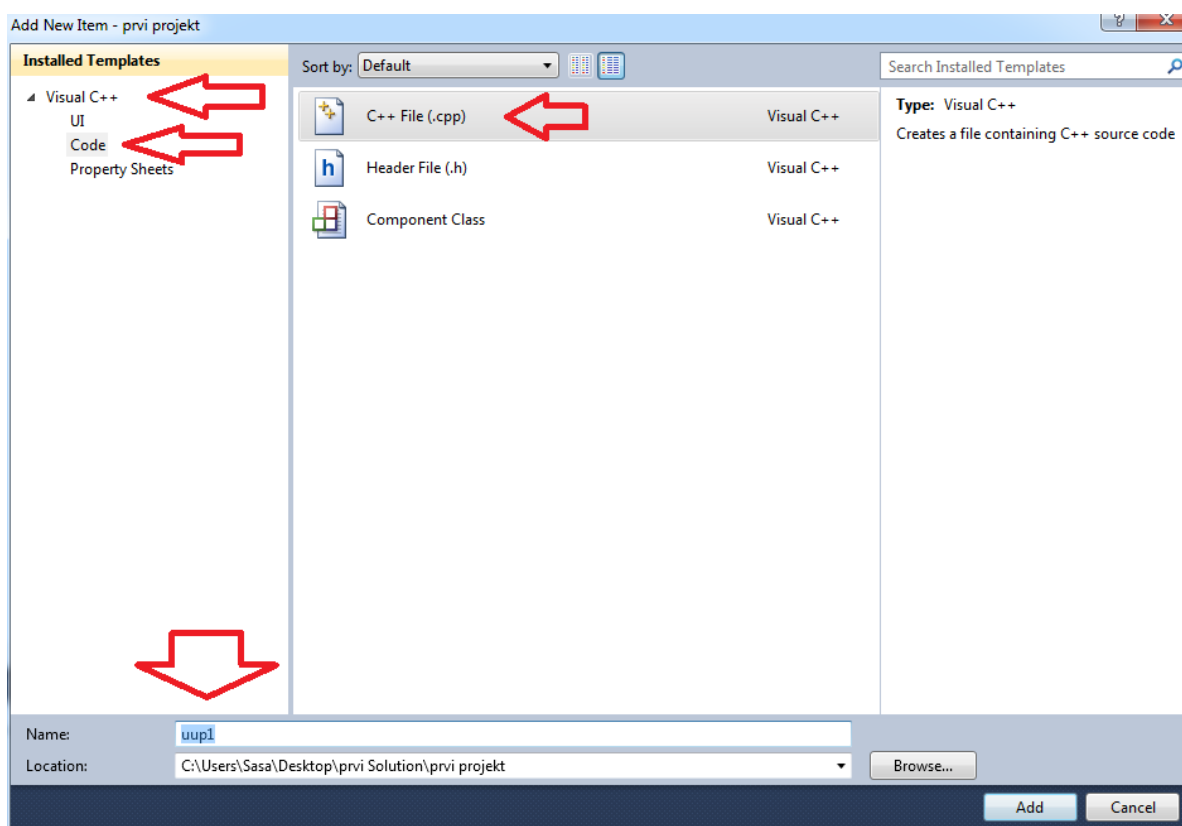
Slika 3 - Stvaranje novog praznog projekta

Pritisnite „OK“ nakon čega se pojavljuje čarobnjak za postavljanje projekta. Odaberite „Next“. Sada Vam se nude opcije poput odabira tipa aplikacije, dodatnih opcija i dodavanja zaglavlja. Ovdje je potrebno odabrati „**Console application**“ i obilježiti u izborniku dodatnih opcija „**Empty project**“ te pritisnuti „Finish“.



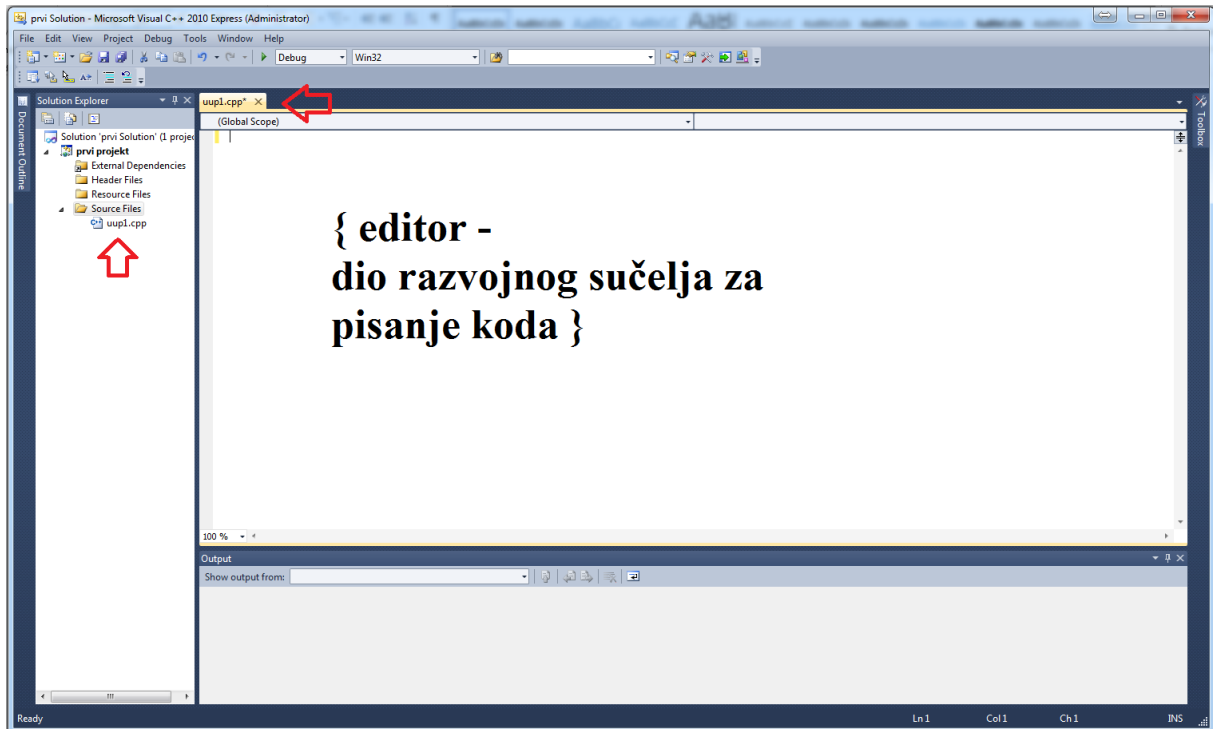
Slika 4 - Izgled nakon stvaranja novog praznog projekta

Otvorilo nam se razvojno sučelje. S lijeve strane trebate vidjeti „**Solution Explorer**“. Ukoliko ga ne vidite potrebno ga je upaliti odabirom na “View – Other Windows – Solution Explorer“ ili kraticom za brzi pristup „Ctrl+Alt+L“. Sada vidimo kako se unutar našeg „*solutiona*“ nalazi naš prvi projekt koji u sebi sadrži četiri mape (*engl. "folder"*). Sada je potrebno dodati našu prvu C++ datoteku. Najbrži način je da desnim klikom miša na folder „Source Files“ odaberemo opciju „Add“ pa „New Item“ (kratica je Ctrl+Shift+A). Iz ponuđenog izbornika potrebno je odabrati „C++ File (.scpp)“. Ova opcija se nalazi u pod-izborniku „Code“ ali je vidljiva i ukoliko nam je odabran glavni izbornik „Visual C++“. Nakon odabira C++ datoteke potrebno je zadati ime i lokaciju na tvrdom disku za spremanje te datoteke. U polje „Name:“ upisujemo željeno ime (nazovimo je „uup1“). Iza riječi „uup1“ moguće je upisati i ekstenziju (*engl. extension*) „.cpp“ no ukoliko to ne učinimo, razvojno sučelje će to automatski učiniti za nas. Lokaciju ostavite kako je zadana (po početnim postavkama, ova C++ datoteka će biti spremljena u mapu našeg „*solutiona*“). Za kraj odaberite gumb „Add“.



Slika 5 - Dodavanje prazne C++ datoteke (ekstenzija je .cpp)

Uočite kako se unutar „Source Files“ mape sada nalazi naša C++ datoteka te se automatski naša C++ datoteka otvorila za uređivanje. Napokon smo spremni za kôdiranje.



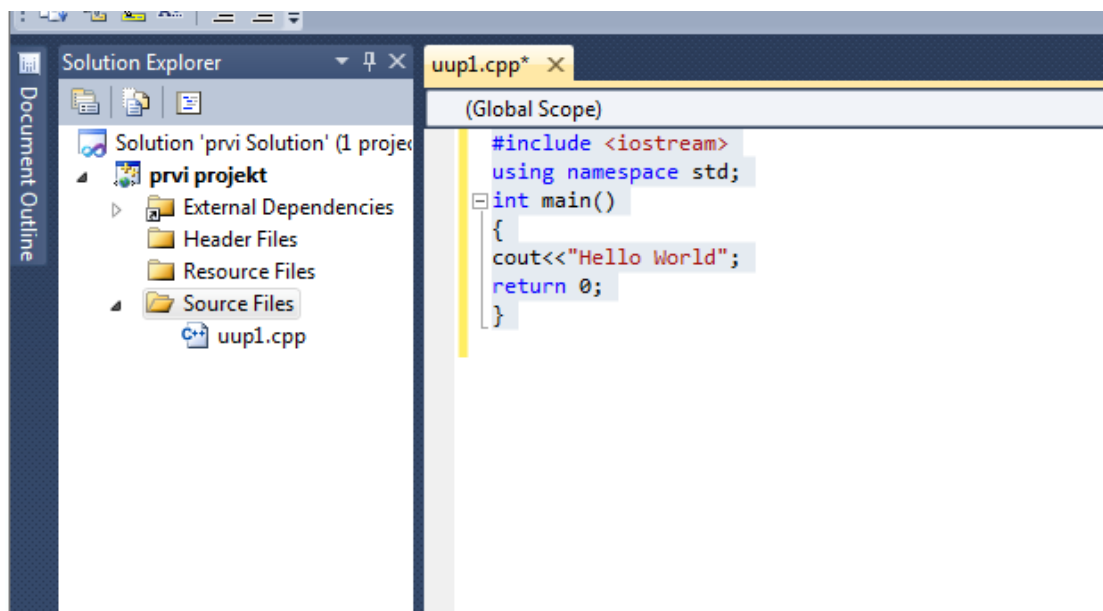
Slika 6 - Razvojno sučelje nakon što je dodana prazna C++ datoteka

## 4) Prva aplikacija

Tradicionalno kako to biva u svijetu programera, sada ćemo napraviti našu prvu „Hello World“ aplikaciju. Unutar našeg *editora* (dio u razvojnom sučelju za pisanje koda) upisat ćemo ove naredbe :

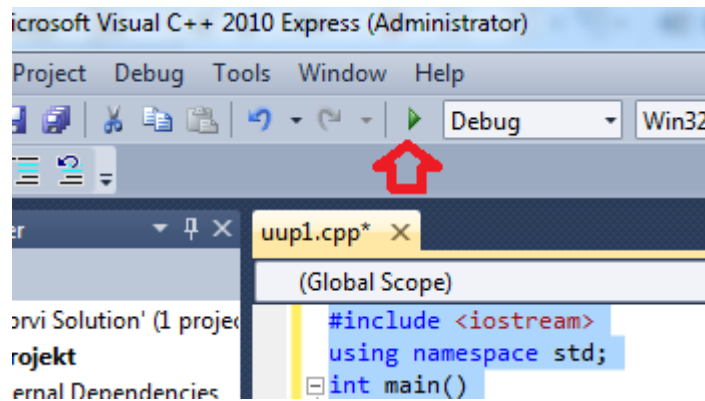
```
#include <iostream>
using namespace std;
int main()
{
cout<<"Hello World";
return 0;
}
```

Ukoliko se ispod niti jednog dijela teksta nije ništa *zacrvenilo* znači da ste točno upisali potrebne naredbe. Ako se ipak nešto *crveni* odnosno da je podvučeno crvenom bojom, provjerite točnost upisa. Velik i mala slova također čine razliku.



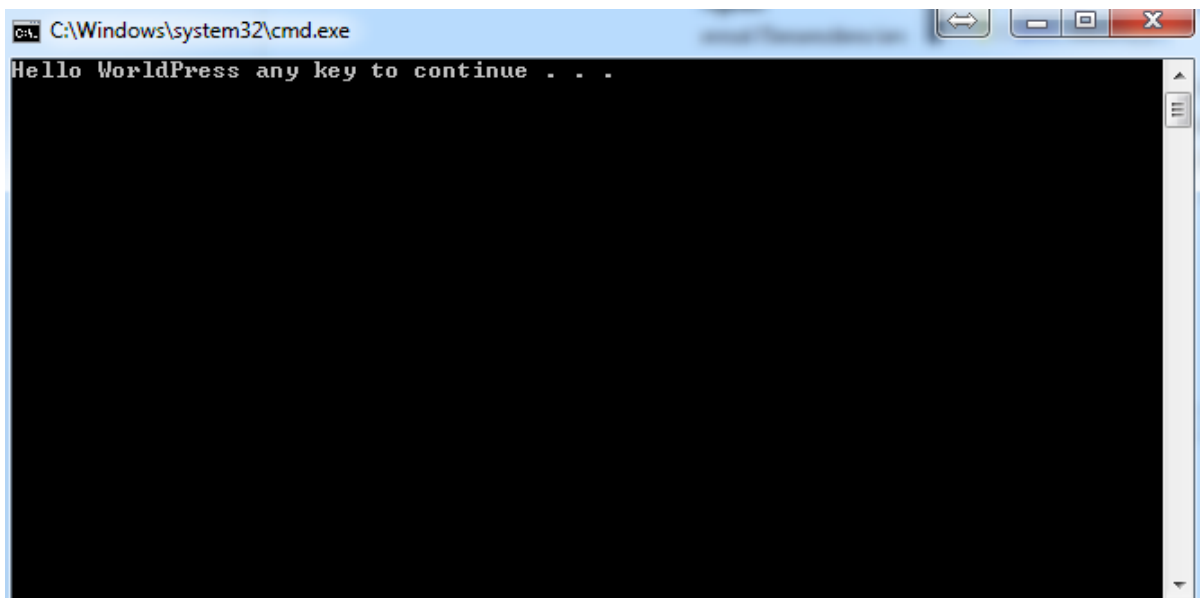
Slika 7 - Prva aplikacija u C++ programskom jeziku

Sada pritisnite tipku „F5“ ili zeleni gumb koji nalikuje na trokut i potom odaberite opciju „Yes“ kako biste pokrenuli postupak debugiranja (*engl. debugging*). Što je debugiranje objasniti ćemo kasnije.



Slika 8 - prikaz gumba za pokretanje procesa debugiranja

Pokretanjem postupka debugiranja Vaša će se aplikacija izvršiti no odmah će se i zatvoriti. Ovakav rezultat je normalan no nema previše smisla pokretati aplikaciju koja odmah nestaje. Rješavanju ovakvog problema doskočit ćemo pomoću dva trika. Prvi način je dodavanje još jedne linije koda koja glasi : `system("pause");` Ovu liniju je potrebno dodati iza linije `cout<<"Hello World";` a ispred linije `return 0;`. Drugi način je bez dodavanja ikakvih dodatnih linija. Umjesto pritiska na tipku „F5“ potrebno je pritisnuti kombinaciju „Ctrl+F5“. Na ovaj način smo dobili našu aplikaciju na ekranu no nismo pokrenuli proces debugiranja. Oba načina su za naše potrebe dobra iako bih preporučio korištenje opcije **bez debugiranja** (Ctrl+F5) radi bržeg pokretanja aplikacije. Napokon, naša aplikacija je uspješno prikazana na monitoru i nije nestala odmah po izvršenju. Aplikaciju gasimo pritiskom na gumb „X“ za izlaz iz aplikacije (kao i u svim ostalim aplikacijama u Windows okruženju).



Slika 9 - Konzolna aplikacija - Hello World

Sada se možete pohvaliti kako ste izradili Vašu prvu aplikaciju u C++ programskom jeziku. U ovom poglavlju ste se upoznali s osnovnim značajkama i radom razvojnog sučelja „MS VC++“. **U daljnjem tekstu neće se upućivati na cijeli postupak pokretanja Vaše aplikacije već će se podrazumijevati da ste s tim dijelom upoznati.** Isto pravilo vrijedi i za stvaranje novog projekta i C++ datoteka.

Sada će te započeti s učenjem programiranja a ukoliko vas brine što su značile prijašnje naredbe; slobodno se prestanite brinuti. Sve potrebno će biti spomenuto, objašnjeno i sami će te isprobati. Naravno, samoinicijativno istraživanje je uvijek preporučljivo i uvijek se pokaže izuzetno korisnim.

## 5) Osnovni pojmovi i savjeti

U nadolazećim odlomcima ukratko ćete saznati često korištene riječi vezane uz samo razvojno sučelje i njihovo značenje.

### a. Prevoditelj

Prevoditelj (*engl. compiler*) nam omogućuje prevađanje našeg napisanog kôda u izvedbeni kôd. Računala razumiju binarni jezik koji se sastoji od samo dva simbola – znamenke jedan (1) i znamenke nula (0). Ljudima je ovakav zapis nepraktičan pa za komunikaciju između čovjeka i računala koristimo prevoditelja. Ovaj program dolazi *integriran* u MS VC++ tako da se o njemu ne morate brinuti.

### b. Program za uređivanje teksta

Kao što mu i samo ime kaže, ovaj dio razvojnog sučelja služi za pisanje teksta odnosno kôda. Mnogi današnji programi za uređivanje teksta (*engl. editor*) dolaze s podrškom za isticanje ključnih riječi (*engl. syntax highlighting*). Ovo ste već vidjeli upisom naredbe `#include` u našem editoru. Tekst je postao plave boje što sugerira da smo upisali jednu od ključnih riječi.

### c. Program za otkrivanje pogrešaka

Ovaj program, koji također dolazi *integriran* unutar VS C++ okruženja će Vam pomoći pri otkrivanju grešaka (*engl. debugger*). Kroz pisanje kôda susret će te se s dvije vrste pogrešaka; *sintaksne* (*engl. syntax error*) i logičke (*engl. logic error*). Debugger može pomoći pri otkrivanju *sintaksnih* pogrešaka ali ne i logičkih.

### d. Sintaksa

*Sintaksa* je skup pravila kojih se moramo pridržavati pri korištenju nekog jezika. Doslovno, to su striktno predodređena pravila pisanja kôda koja je potrebno naučiti i zapamtiti. Zamislite ih kao definicije no znatno manje duljine.

### e. Sintaksna pogreška

Na ovakvu vrstu pogreške će Vas sam *debugger* često upozoriti i neće dopustiti pokretanje aplikacije. Izostavljanje simbola „;“ (točka-zarez) na kraju linije, pogrešna uporaba velikog ili malog slova i slične pogreške se smatraju *sintaksnim* pogreškama.



## f. Logička pogreška

Ovo su pogreške koje je znatno teže otkriti. U ovom slučaju Vam *debugger* ne može pomoći. Ukoliko napišete : „X = 3“ i „Y = 4“ i kažete da je „Z = X+Y“ (zbroj dvije vrijednosti) a pri ispisu napišete da slovo Z predstavlja razliku umjesto zbroja tih brojeva, *debugger* neće prijaviti pogrešku i uredno će izvršiti aplikaciju. Naravno, rezultat u ovom slučaju neće biti u skladu s našim očekivanjem. Ispitivanje na način da *ručno* provjeravamo dobivene vrijednosti je jedini način kontrole.

## g. Redoslijed izvršavanja programa

Program prevoditelj (*compiler*) uvijek izvršava naredbe od prve linije prema zadnjoj gledano u vertikalnom smjeru, odnosno kôd se uvijek izvršava od vrha prema dnu. Ukoliko imamo kôd poput

```
cout<<111<<endl;
```

```
cout<<222<<endl;
```

prvo će se izvršiti ispis broja 111 u konzolu te će program prijeći u novi red nakon čega će se izvršiti ispis broja 222 i program će ponovno prijeći u novi red te ako nema više kôda, aplikacija će završiti s radom.

## h. Numeracija kôdnih linija

Kako bi se lakše snalazili u *text editoru* preporuča se upaliti numerički prikaz linija. Ova opcija će samo dodati numeraciju svake linije u *editoru*. Opciju uključujemo tako da odaberemo Tools-Options-Text Editor-All Languages i upalimo opciju „Line numbers“.

## i. Intelli-Sense

U razvojnom okruženju MS VC++ učit ćete kako će razvojno okruženje ponuditi neke opcije kako pišete kôd. Na ovaj način razvojno sučelje pokušava olakšati posao programeru tako da automatski nadopuni riječ koju programer želi napisati. Ovim načinom se mogu smanjiti i *sintaksne* pogreške. Željenu nadopunu odabirete dvostrukim lijevim klikom miša na željenu riječ ili odabirom željene riječi pomoću strelica te pritiskom na tipku „Tabulator“ (iznad „Caps lock“ tipke).

## j. .cpp ekstenzija

„.cpp“ je naziv ekstenzije (*engl. extension*) svake C++ datoteke. Ekstenzije s kojima ste se već susreli su zasigurno „.txt“, „.pdf“, „.doc“, „.docX“ i slično. Primjerice „Programiranje.cpp“ će označavati da se radi o C++ datoteci čije je ime „Programiranje“.

Mape datoteka (*engl. folder*) nam omogućuju jednostavnije i smislenije grupiranje određenih datoteka (*engl. file*). Unutar našeg *solutiona* nalaze se četiri unaprijed određene mape.

„**External Dependencies**“ – u njoj su pohranjene datoteke koje su potrebne za korištenje funkcija unutar C++ jezika. Ovisno o potrebama programera mogu se dodavati i uklanjati datoteke. Kako i samo ime mape kaže, datoteke smještene u ovoj mapi su one o kojima naša aplikacija ovisi.

„**Header Files**“ – u ovu mapu postavljamo vrste datoteka koje imaju ekstenziju „.h“ a koriste se pri imalo zahtjevnijem pisanju kôda kako bi se kompleksne aplikacije s stotinama ili tisućama linija kôda rastavile u manje, logične cjeline ili primjerice ukoliko više ljudi istovremeno radi na istoj aplikaciji i svaka osoba radi na određenom dijelu aplikacije. U tom slučaju ne bi imalo smisla koristiti samo jednu „.cpp“ datoteku već je logičnije rastaviti problem u više cjelina kako bi svaka osoba mogla raditi svoj dio posla i na kraju se sve uklopi u veliku cjelinu.

„**Resource Files**“ – mapa koja sadrži resurse bitne za našu aplikaciju. Često u ovu mapu stavljamo datoteke bitne za našu aplikaciju ali koje nisu dio našeg trenutnog projekta. Primjerice, datoteke koje nisu napisane u C++ jeziku i zahtijevaju prevođenje u C++ jezik se često mogu naći u ovoj mapi.

„**Source Files**“ – jedina mapa koja će nama trebati. U njoj će se nalaziti naše „.cpp“ datoteke. Kako i samo ime kaže, to je mapa namijenjena za izvorne datoteke. Naravno, u naš projekt je moguće dodavanje i vlastitih datotečnih mapa no potreba za takvim postupkom izlazi iz okvira ove skripte.

#### k. „#include <iostream>“, „using namespace std“, „int main()“, „return 0“

Naredbe „`#include<iostream>`“ i „`using namespace std;`“ su jedne od mnogih predefinih naredbi unutar C++ jezika koje smanjuju vrijeme potrebno da programer obavi neke osnovne funkcije. Sama (predprocesorska) naredba „`#include`“ govori *programu prevoditelju* da ćemo uključiti neku datoteku a „`<iostream>`“ specificira što točno uključujemo. U ovom slučaju konkretno radi se o datoteci (točni naziv za ovakvu vrstu datoteke je **biblioteka** (*engl. library*) koja u sebi sadrži kratice poput „`cin`“ (čita se : „ce in“) i „`cout`“ (čita se: „ce aut“). Značenje ove dvije kratice predstavlja ulazni i izlazni tok konzole (učitavanje iz konzole (`cin`) i pisanje u konzolu (`cout`)). Naredba „`using namespace std;`“ nam omogućuje korištenje unaprijed definiranih riječi, funkcija, klasa, objekata a time ujedno skraćuje duljinu kôda. Oznaka „`std`“ je kratica za englesku riječ „*standard library*“ i govori programu prevoditelju da ćemo koristiti standardne (predefiniране) izraze za neke operacije. Ukoliko ne bismo napisali „`std`“ naš program prevoditelj ne bi znao što znače neke od ključnih riječi koje ćete upoznati uskoro i morali bismo mu eksplicitno navesti na što točno mislimo. Primjerice, tada bi naša aplikacija umjesto ovakvog kôda :

```
#include <iostream>
using namespace std;
int main()
{
cout<<"Hello World";
return 0;
}
```

morala biti zapisana na ovakav način :

```
#include <iostream>
int main()
{
std::cout<<"Hello World";
return 0;
}
```

Uklonimo li „`std::`“ dio iz drugog primjera pojavit će se *sintaksna* pogreška jer prevoditelj ne zna na što točno se odnosi naredba „`cout`“ (iako smo mu mi rekli da koristi biblioteku „`iostream`“ u kojoj je naredba „`cout`“ sadržana).

Naredba „`return 0;`“ govori da će naša glavna (*main*) funkcija po završetku izvođenja vratiti znamenku „0“ (nula). Više o ovome ćete naučiti u dijelu u kojem se obrađuju funkcije. Važno je napomenuti kako nije striktno definirano pravilo da se vrati vrijednost nula (0), ali je običaj kojeg se valja pridržavati. Ova vrijednost ukazuje da nije došlo do pogreške u izvođenju programa odnosno vrati li naša *main* funkcija neku drugu vrijednost znat ćemo da je došlo do pogreške.

„*int main()*“ je glavna funkcija i izvođenje programa počinje s njom. Sav kôd ćemo pisati unutar ove funkcije. Program može imati samo jednu *main* funkciju. Što su funkcije i što predstavlja oznaka *int* ćemo obrazložiti u nastavku ove skripte.

## I. Naredbe „cin“ i „cout“ (učitavanje iz konzole i ispis u konzolu)

Ove dvije naredbe ćemo često koristiti. Naredba „*cin*“ predstavlja učitavanje iz konzole (**C**onsole **I**N) a naredba „*cout*“ predstavlja pisanje u konzolu (**C**onsole **O**UT). *Sintaksa* korištenja između ove dvije naredbe se razlikuje u simbolu „<<“ odnosno „>>“. Želimo li učitati neku vrijednost iz konzole (ono što je korisnik unio) i spremiti to u varijablu imena „*upis*“ koristit ćemo ovu naredbu `cin>>upis;`

Želimo li nešto ispisati u konzolu (primjerice vrijednost varijable „*upis*“) potrebno je upisati naredbu `cout<<upis.`

### m. Ispis teksta i uloga dvostrukih navodnika

Želimo li **ispisati neki tekst** na ekran (primjerice „Dobar dan“) tada koristimo dvostruke navodnike. `cout<<“Pozdrav, nadam se da shvacate C++“<<endl;` **Važno** – pokušamo li ispisati na ovaj način `cout<<“2+3“;` rezultat u konzoli će biti „2+3“ odnosno neće se zbrojiti te dvije vrijednosti već će bit prikazane kao tekst. Razlog tome je što smo prevoditelju rekli da želimo ispisati tekst koji se sastoji od znamenke, simbola i još jedne znamenke. **Zapamtite – sve što se nalazi ispod dvostrukih navodnika se tretira kao tekst**, neovisno jesmo li upisali slova, znamenke ili simbole. Ukratko rečeno, ono što se nalazi između dvostrukih navodnika će se upravo tako i ispisati.

Želimo li **saznati neku vrijednost** (primjerice prezime neke osobe) i **kasnije ispisati** tu vrijednost, potrebno je prvo pomoći „`cin>>`“ naredbe upisati vrijednost prezimena u neku varijablu tipa *string* (recimo da se varijabla zove „*prez*“). Nakon što je vrijednost spremljena možemo tu vrijednost i ispisati u kombinaciji s nekim tekstom poput `cout<< „Dobar dan „<<prez<<endl;`

### n. Simbol točka-zarez -> „;“

U programskom jeziku C++ prilikom završavanja naredbe koristi se simbol „;“ (točka-zarez). Ovim načinom govorimo programu prevoditelju (*compiler*) gdje je kraj naredbe. U jednoj komandnoj liniji moguće je imati i više naredbi poput `cout<<111<<endl;` `cout<<222<<endl;` `cout<<333<<endl;`. Iako se ovaj kôd sastoji od tri naredbe, zahvaljujući razdjelniku *točka-zarez*, moguće ih je upisati u istoj komandnoj liniji što se često koristi kako bi sam kôd programa bio čitljiviji.

### o. Naredba „endl“ → End Line (prelazak u novi red)

Pokušamo li ispisati nešto u konzolu poput recimo `cout<<“Test“;` vjerojatno će Vam smetati rečenica „Press any key to continue . . . “. Ovdje nam pomaže naredba „**End Line**“ odnosno njena skraćunica koja se koristi u programiranju „**endl**“. Implementacija ove naredbe u naš kôd je jednostavna. Na mjestu gdje želimo prijeći u novi red (prekinuti trenutnu liniju u kojoj se nalazimo) potrebno je dodati „`<<endl;`“. **Primjer :**

```
cout << 111 << endl;
cout << 222 << endl;
```

### p. Važna napomena

U svim programima ćemo uvijek koristiti naredbe `#include <iostream>`, „`using namespace std;`“ i „`return 0;`“. Korištenje dodatnih biblioteka će biti posebno napomenuto a spomenute tri naredbe uzmite **zdravo za gotovo** jer detaljno i kvalitetno objašnjavanje prelazi okvire ove skripte.

## 6) Identifikatori

### a. Što su identifikatori i koja su dopušтана imena

Identifikatori su zapravo imena naših dijelova programa poput varijabli ili funkcija. Svaka varijabla ili funkcija moraju imati jedinstveno ime kako bi se mogli referencirati na njih. Imenovanja su proizvoljna no ipak postoje neka ograničenja.

- 1) Svaki identifikator može sadržavati velika i mala slova engleske abecede, znamenke i simbol podcrte (`_`) no uz uvjet da :
- 2) prvi znak imena odnosno identifikatora mora biti ili slovo ili podcrta.
- 3) Identifikator ne smije biti jednak nekoj od ključnih riječi jezika ali je može sadržavati. To znači da naš identifikator ne može biti riječ „*int*“ jer je ta riječ predodređena u C++ jeziku za cjelobrojni tip podataka (uskoro ćemo ih obraditi) ali smije se zvati primjerice „*pinta*“. Dobra je praksa (nije striktno zabranjeno ali se ne preporuča) izbjegavati stavljanje dvije ili više podcrta jednu do druge )primjerice : „`moje__program`“) jer lako može doći do krivog imenovanja pri pozivanju takvih imena u programu. Također se ne preporuča započinjati ime s dvije podcrte i velikim slovom jer su neke riječi koje su striktno rezervirane za C++ pisane po takvom predlošku (primjerice : „`__FILE__`“).

#### **Napomena !!!**

Sugestija je da ne štedite na broju znakova u identifikatorima jer je generalno pravilo da treba koristiti deskriptivna imena varijabli i funkcija. Primjerice ako nazovemo varijablu „*brojPi*“ i dodijelim joj vrijednost „3.14“ i nakon par mjeseci ću znati na što se odnosi ta varijabla no nazovem li je samo „*x*“ postoji izuzetno velika šansa kako se svrhe te varijable neću odmah moći prisjetiti.

## b. Popis zabranjenih riječi

|            |              |                  |              |
|------------|--------------|------------------|--------------|
| and        | decltype     | new              |              |
| and_eq     | default      | noexcept         | switch       |
|            | delete       | not              | template     |
| alignas    | double       | not_eq           | this         |
|            | dynamic_cast |                  | thread_local |
| alignof    | else         | nullptr          | throw        |
| asm        | enum         | operator         | true         |
| auto       |              | or               | try          |
| bitand     | explicit     | or_eq            | typedef      |
| bitor      | export       | private          | typeid       |
| bool       |              | protected        | typename     |
| break      | extern       | public           | union        |
| case       | false        | register         | unsigned     |
| catch      | float        | reinterpret_cast | using        |
| char       | for          | return           | virtual      |
| char16_t   | friend       | short            | void         |
| char32_t   | goto         | signed           | volatile     |
| class      | if           | sizeof           | wchar_t      |
| compl      | inline       | static           | while        |
| const      | int          | static_assert    | xor          |
| constexpr  | long         | static_cast      | xor_eq       |
| const_cast | mutable      | struct           |              |
| continue   | namespace    |                  |              |

Slika 10 - Popis ključnih riječi u jeziku C++<sup>2</sup>

---

<sup>2</sup> izvor : <http://4.bp.blogspot.com/-rGivgUD2QZc/TrMeK7sMjNI/AAAAAAAAAGc/FgjP2KB7xvw/s640/kew.PNG>

## 7) Varijable

### a. Što su varijable

Varijable ste zasigurno već susreli. Osnovni zadatci poput :

`X=5, Y=5, Z=X+Y`, Pronađi „Z“.

u sebi sadrže tri različite varijable. U ovom primjeru to su varijable X, Y i Z. Varijable neće uvijek poprimiti unaprijed definiranu vrijednost i tu će se vidjeti moć varijabli. Svaku varijablu je potrebno definirati a logično je da ćemo je htjeti i inicijalizirati

### b. Deklaracija varijabli

Deklaracija varijabli je zapravo govorenje našem programu da ćemo koristiti varijablu nekog imena. Primjerice naredba `int x;` će našem programu reći da stvaramo varijablu koja je tipa *integer* (skraćena je *int* a predstavlja cijele brojeve) i koja će se zvati „x“. U ovom se trenutku ne brinite oko značenja riječi *integer* i ostalih tipova podataka. Njih ćemo ubrzo obraditi.

### c. Inicijalizacija varijable

Svaku varijablu, nakon što je stvorimo (u gornjem primjeru je to bilo `int x;`) poželjno je i postaviti na neku vrijednost odnosno pridružiti joj neku vrijednost. Postupak dodjeljivanja neke vrijednosti nekoj varijabli zovemo inicijalizacija. Primjer inicijalizacije bi bio da smo varijabli „x“ dodijelili neku cjelobrojnu vrijednost, poput recimo 5 (pet). Naš kod bi tada izgleda ovako : `int x = 5;` ili smo mogli kroz dva zasebna koraka obaviti postupak, prvo stvoriti varijablu a zatim joj dodijeliti vrijednost. Tada bi kôd ovako izgledao :

```
int x;  
x=5;
```

**Varijabla** je dakle simbolično ime za neku vrijednost kako bismo se lakše snašli u našem programu. Iskoristivost varijabli će te uskoro upoznati.

### d. Konstantne varijable

Poseban tip varijabli su konstante. Ovu vrstu varijabli možemo samo jednom postaviti na neku vrijednost (inicijalizirati). Koriste se kada želimo zaštititi vrijednost neke varijable u našem programu da ju drugi programeri ne mogu naknadno mijenjati u svom kôdu. *Sintaksa* je jednostavna, samo se doda ključna riječ „const“ ispred postavljanja tipa podatka te varijable, odnosno ovako :

```
const int x = 5;
```

## 8) Tipovi podataka

Slično poput ljudi, i računala razlikuju tipove podatka. Ljudi razlikuju znamenke od slova ili simbola, brojeve od riječi i unutar tih osnovnih podjela možemo napraviti još manje podjele. Slova možemo podijeliti na velika i mala a brojeve primjerice na cjelobrojne (1, 2, 3, ...) i na realne odnosno s pomičnim zarezom (3.141592653589793238, 6.543, 1.61803, ...). U programerskom svijetu, osnovne (**primitivne**) tipove podataka dijelimo na : *integer*, *float*, *double*, *boolean*, *character*. Uz njih postoje i tipovi podataka poput *long* i *string*, ali i programer može sam definirati svoje tipove podataka. Točnu veličinu i raspone možete pogledati tablici na kraju ovog poglavlja a deklaraciju varijable svakog tipa možete vidjeti ispod svakog podnaslova u ovom poglavlju.

### a. integer

*integer* (skraćenica je *int*) označava cijele brojeve. Tipu *integer* možemo pridodati i „podtipove“ poput „short“, „long“ i „unsigned“. Tip podatka **short int** će nam omogućiti manji raspon brojeva, a **long int** veći raspon brojeva ali sukladno tome i manje odnosno veće zauzeće memorije. Ukoliko se ukaže potreba za korištenjem striktno pozitivnih brojeva tada možemo koristiti **unsigned int**. Za naše potrebe, koristit ćemo samo *int* tipove.

```
int x = 5;
```

### b. float

Ovaj tip podatka (nema skraćenice) nam omogućuje veći raspon brojeva odnosno radi se o cijelim brojevima s pomičnim zarezom (tzv. decimalni brojevi). Ovakav tip podatka nam naravno daje puno veću preciznost. Decimalni zarez se **ne zapisuje** simbolom zareza ( , ) već simbolom točka ( . ).

```
float x = 3.1415926;
```

### c. double

Double tip podatka (nema skraćenice) daje okvirno duplo veću preciznost od *float* tipa ali zauzima i duplo više memorije. Iako se u našim aplikacijama ne morate brinuti da ćete zauzeti previše memorije dobro je od početka koristiti tipove sukladno potrebama.

```
double x = 3.141592653589793;
```

### d. character

Za razliku od prethodna tri tipa, ovaj tip podatka (skraćenica je *char*) ne predstavlja brojeve niti znamenke već simbole (karaktere). Može sadržavati isključivo jedan znak (znamenku, slovo ili simbol). Prilikom inicijalizacije ovog tipa podatka koriste se jednostruki navodnici.

```
char x = 'a'; ili char x = '1'; ili char x = '#';
```



## e. boolean

Ovaj tip podataka (skraćenica je *bool*) ima predodređene samo dvije vrijednosti. To su vrijednosti *true* (istina) i *false* (laž). Sinonim za te vrijednosti su ujedno i znamenke 1 i 0 gdje znamenka 0 (nula) predstavlja laž (*false*) a istina se predstavlja znamenkom 1 (jedan). Važno je napomenuti da varijabli koja je tipa *bool* možemo pridijeliti i vrijednost 2, 3, 5, itd. ali će ta vrijednost biti zapisana kao 1 (jedan) odnosno kao istina. Ovakav pristup nemojte koristiti već koristiti *true* i *false* vrijednosti.

```
bool x = true; ili bool x=1; ili bool x = false; ili bool x = 0;
```

## f. string

Od svih predstavljenih tipova podataka, ovaj tip (nema skraćenice) jedini ne spada u primitivne tipove i njih ćemo posebno obraditi naknadno. Važno je znati da je *string* tip podatka zapravo skup znakova (simboli, slova i brojke). Nečije ime bi bilo tipa *string* ali isto tako možemo i brojeve zapisati kao *stringove* no tada se nad njima neće moći vršiti matematičke operacije. Ovaj tip podatka ćemo koristiti ali se neće detaljno obrazložiti jer je potrebno ulaziti u područje objektno orijentiranog programiranja što nije tematike ove skripte. Važno je zapamtiti da se *stringovi* inicijaliziraju tako da se varijabli pridružuje neka vrijednost koja se nalazi pod dvostrukim navodnicima i da sve što je *string* tretiramo kao tekst neovisno što se nalazi između dvostrukih navodnika.

```
string ime = „program“; ili string broj = „3“; ili string broj = „3.14“;
```

*String* tip podatka je potrebno posebno uključiti u našu aplikaciju dodavanjem naredbe „`#include <string>`“.

## g. Popis tipova podataka i njihova svojstva

| Name                        | Description   | Size            | Range   |
|-----------------------------|---|-----------------|---|
| <i>char</i>                 | Character or small <i>integer</i> .                             | 1byte           | signed: -128 to 127<br>unsigned: 0 to 255                         |
| short<br><i>int</i> (short) | Short <i>integer</i> .  | 2bytes          | signed: -32768 to 32767<br>unsigned: 0 to 65535                   |
| <i>int</i>                  | <i>integer</i> .  | 4bytes          | signed: -2147483648 to<br>2147483647<br>unsigned: 0 to 4294967295 |
| long<br><i>int</i> (long)   | Long <i>integer</i> .   | 4bytes          | signed: -2147483648 to<br>2147483647<br>unsigned: 0 to 4294967295 |
| bool                        | Boolean value. It can take one of two values:<br>true or false. | 1byte           | true or false   |
| <i>float</i>                | Floating point number.  | 4bytes          | +/- 3.4e +/- 38 (~7 digits)                                       |
| double                      | Double precision floating point number.                         | 8bytes          | +/- 1.7e +/- 308 (~15 digits)                                     |
| long double                 | Long double precision floating point number.                    | 8bytes          | +/- 1.7e +/- 308 (~15 digits)                                     |
| <i>wchar_t</i>              | Wide character.   | 2 or 4<br>bytes | 1 wide character  |

Slika 11 - Tipovi podataka u C++ programskom jeziku i njihova svojstva <sup>3</sup>

<sup>3</sup> Izvor : <http://www.cplusplus.com/doc/tutorial/variables/>

## 9) Operatori

Najpoznatije i najvažnije vrste operatora, zvanim „aritmetički operatori“ dijelimo u dvije skupine; unarne i binarne. Unarni operatori se koriste ukoliko vršimo neku operaciju nad samo jednom varijablom dok se binarni koriste kada vršimo operacije nad dvije varijable.

### a. Unarni operatori

Postoji šest aritmetičkih unarnih operatora.

|            |              |
|------------|--------------|
| <b>+X</b>  | Unarni plus  |
| <b>-X</b>  | Unarni minus |
| <b>X++</b> | Uvećaj nakon |
| <b>++X</b> | Uvećaj prije |
| <b>X--</b> | Umanji nakon |
| <b>--X</b> | Umanji prije |

Tablica 1 - Prikaz Unarnih operatora i njihovih naziva

### b. Inkrementacija

Od navedenih unarnih operatora, prva dva nećemo koristiti. Drugi i treći unarni operatori se koriste u procesu **inkrementacije**. Naredba „**X++;**“ označava povećanje vrijednosti varijable „X“ za jedan. Ako je varijabla „X“ imala vrijednost 2 i nad njom izvršimo inkrementaciju, varijabla „X“ će tada poprimiti vrijednost 3 (tri).

```
int x = 2;  
x++;
```

Sada varijabla „X“ ima vrijednost 3.

### c. Dekrementacija

Suprotno inkrementaciji, postoji i **dekrementacija**, a u gornjoj tablici to su peti i šesti unarni operatori. Očekivano, postupkom dekrementacije će se vrijednost varijable umanjiti za jedan.

```
int x = 2;  
x--;
```

Sada varijabla „X“ ima vrijednost 1.

Važno je uočiti razliku između „X++“ i „++X“ odnosno „X--“ i „--X“. Najlakše će se razlika uočiti prilikom ispisa vrijednosti varijable. Postavimo li vrijednost varijable „x“ na 2 i izvršimo naredbu ispisa `cout<<x++;` dobit ćemo rezultat 2 (dva) jer će se prvo ispisati vrijednost varijable a zatim će se izvršiti uvećanje njene vrijednosti. Suprotno tome, izvršimo li naredbu `cout<<++x;` tada će rezultat biti broj 3 (tri), odnosno prvo će se izvršiti uvećanje vrijednosti varijable za jedan a zatim će se ispisati njena vrijednost. Inkrementacija i dekrementacija su u većini slučajeva koriste kod petlji koje ćemo obraditi u kasnijem dijelu skripte.

### d. Binarni operatori

Poznatija vrsta operatora su vjerojatno binarni. Korištenje neke od nižih matematičkih funkcija u kombinaciji s dvije varijable je svima poznato.

|              |            |
|--------------|------------|
| <b>X + Y</b> | Zbrajanje  |
| <b>X – Y</b> | Oduzimanje |
| <b>X * Y</b> | Množenje   |
| <b>X / Y</b> | Dijeljenje |
| <b>X % Y</b> | Modulo     |

Tablica 2 - Popis binarnih operatora i njihovi nazivi

Vjerujem kako ste sa prve četiri matematičke operacije i njihovim svojstvima jako dobro upoznati, no nepoznanica može biti operacija *modulo*. Ova operacija nam vraća kao rezultat ostatak pri cjelobrojnom dijeljenju dva broja.

```
int x = 10;  
int y = 3;  
int z = x % y;
```

U ovom slučaju će vrijednost varijable „z“ iznositi 1 (jedan). Kako smo došli do toga? Podijelimo li broj 10 s brojem 3 na kalkulatoru, dobit ćemo beskonačnu vrijednost od 3.3333...). Pogledajmo znamenku ispred decimalnog zareza; znamenku 3. Pomnožimo li taj broj (samo gledamo znamenke ispred decimalnog zareza) s brojem kolika je vrijednost varijable „y“ (djelitelj) odnosno isto brojem 3 u ovom slučaju dobit ćemo da je rezultat broj 9. Sada nam preostaje samo napraviti razliku između varijable „x“ (djeljenik) i broja 9, pa oduzmemo li od broja 10 broj 9 dobit ćemo broj 1 kao rezultat.

| DEKLARACIJA I INICIJALIZACIJA     | REZULTAT NAKON OPERACIJE MODULO |
|-----------------------------------|---------------------------------|
| <code>INT X = 10%7;</code>        | x = 3                           |
| <code>INT X = 5%2;</code>         | x = 1                           |
| <code>INT X = 8 % 2;</code>       | x = 0;                          |
| <code>INT X = 654321 % 17;</code> | x = 8;                          |

Tablica 3 - Korištenje binarnih operatora i prikaz rezultata

### e. Operatori pridruživanja

Operator pridruživanja je svima dobro poznat. Često se pozivamo ne njega pod nazivom „jednako“ a njegov simbol je „=“. Uz ovaj operator postoje i izvedenice koje omogućuju manje pisanja kôda a najčešće korištene operatore, njihove oznake i objašnjenja se nalaze u tablici, a primjeri ispod tablice.

| OPERATOR | OBJAŠNENJE  |
|----------|---|
| =        | pridruži vrijednost                               |
| +=       | uvećaj za   |
| -=       | umanji za   |
| *=       | pomnoži sa  |
| /=       | podijeli sa                                       |
| %=       | dodaj vrijednost nakon izvršenja modulo operacije |

Tablica 4 – Popis najčešće korištenih operatora pridruživanja i njihovi nazivi

| NAREDBA                                       | REZULTAT |
|---|----------|
| <code>int x = 3;</code>                       | x = 3    |
| <code>int x = 6;</code><br><code>x+=2;</code> | x = 8    |
| <code>int x = 6;</code><br><code>x-=2;</code> | x = 4    |
| <code>int x = 6;</code><br><code>x*=2;</code> | x = 12   |
| <code>int x = 6;</code><br><code>x/=2;</code> | x = 3;   |
| <code>int x = 6;</code><br><code>x%=2;</code> | x=0;     |

Tablica 5 - Korištenje operatora pridruživanja i prikaz rezultata

## f. Operatori usporedbe

Posljednja vrsta operatora koje ćemo obraditi su operatori usporedbe. Slično kako ste već i navikli u matematici postoje mogućnosti da su prilikom usporedbe dvije vrijednosti, prva bude veća od druge, manja od druge, da budu iste ili da budu različite. Naravno, tu su još i operatori „veće ili jednako“ i „manje ili jednako“.

| OPERATOR | OBJAŠNENJE        |
|----------|-------------------|
| >        | striktno veće     |
| <        | striktno manje    |
| >=       | veće ili jednako  |
| <=       | manje ili jednako |
| ==       | potpuno identično |
| !=       | potpuno različito |

Tablica 6 - Popis operatora uspoređivanja i njihovi nazivi

Važno je primijetiti posljednja dva operatora, „==“ i „!=“. Operator koji provjeravamo jesu li dvije vrijednosti identične se sastoji od dva znaka jednakosti. Velika je razlika stavimo li dva ili jedan znak jednakosti jer stavljanjem jednog znaka jednakosti koristimo operator pridruživanja odnosno nekoj varijabli pridjeljujemo neku vrijednost. Suprotno očekivanom, operator usporedbe jesu li dvije vrijednosti potpuno različite nije „<>“ već je to „!=“.

Operatore usporedbi ćemo koristiti prilikom kontroliranja toka našeg programa odnosno kod petlji i grananja.

## g. Logički operatori

U programiranju se najčešće koriste dva operatora a zovu se „logički i“ i „logički ili“.

| Simbol operatora | Naziv operatora |
|------------------|-----------------|
| &&               | logički i       |
| //               | logički ili     |

Tablica 7 - Simboli logičkih operatora i njihova značenja

Operator „logički i“ se koristi ukoliko želimo da se nešto dogodi odnosno da se neki uvjet izvrši ukoliko su barem dva uvjeta točna. Primjerice, želimo izračunati opseg i površinu pravokutnika samo ako su vrijednosti stranice „a“ i stranice „b“ obavezno veće od nule. Dovoljno je da jedan uvjet nije zadovoljen i proces se neće izvršiti. Simbol je „&&“ (& se dobije pritiskom na Shift+6)

Operator „logički ili“ koristimo kada nam je dovoljno da je barem jedan od uvjeta zadovoljen. Primjerice, pitamo korisnika na koji mu je pogonski ovjes na osobnom automobilu. Mogući odgovori su : „pogon na prednje kotače, pogon na stražnje kotače ili pogon na sva četiri kotača“. Na osnovu tih vrijednosti upisat će mu se informacija o pogonskom ovjesu u prometnu knjižicu vozila. Kako bi računalo poslalo naredbu pisaču za ispis korisnik mora unijeti ili opciju 1 ili opciju 2 ili opciju 3. Sve izvan predviđenih vrijednosti će se smatrati greškom i možemo u tom slučaju zahtijevati od korisnika ponovni upis. Simbol za „logički ili“ je „||“ (simbol | se dobije pritiskom na Alt Gr+W).

## 10) Komentari

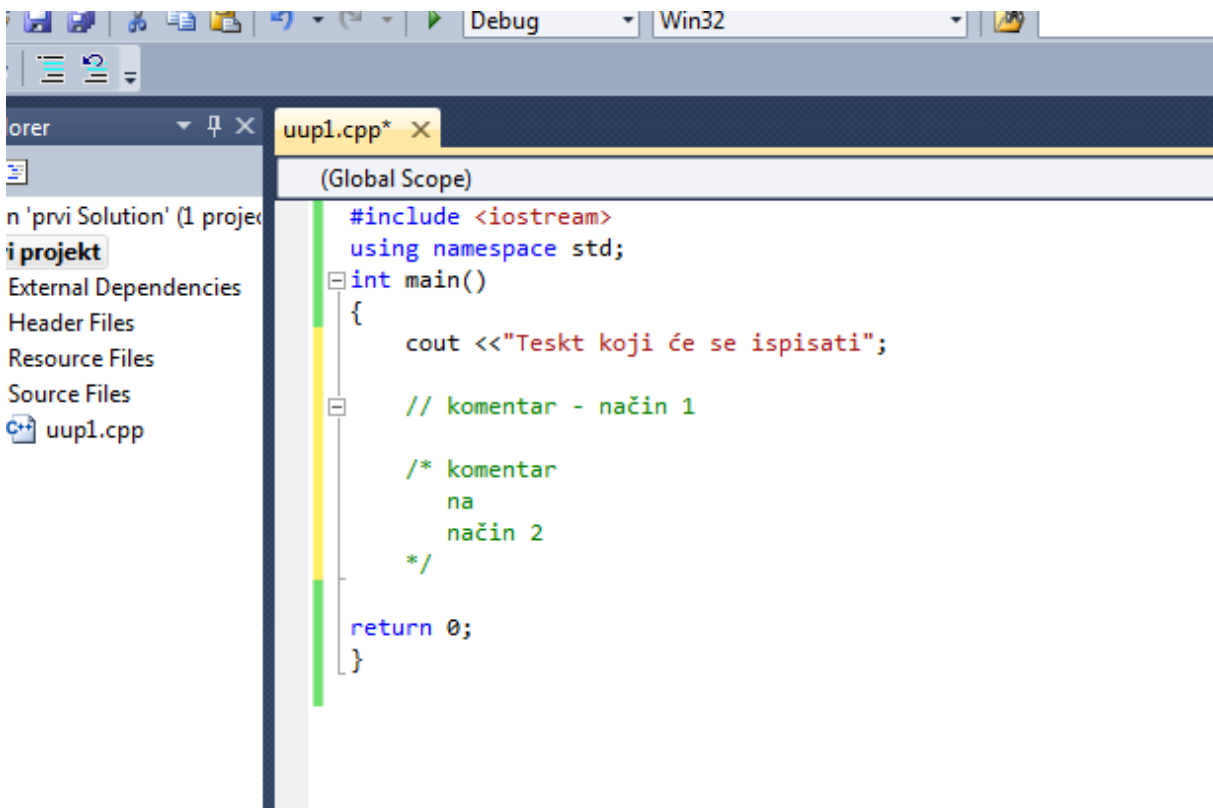
Jedna od najvažnijih stvari prilikom pisanja kôda je komentiranje. Iako je mnogima u početku teško se priviknuti na ovu praksu, s vremenom programeri se nauče obavezno komentirati svoj kod kako bi se i oni sami ali i drugi programeri koji će proučavati njihov kod mogli snaći i lakše shvatiti „što je pjesnik htio reći“.

Druga svrha komentara je kada programeru određeni dio kôda ne treba a ne želi ga izbrisati i kao brzo rješenje se nudi opcija da se kôd zakomentira. Postoje dva načina komentiranja.

Ukoliko želimo zakomentirati samo jednu liniju koristit ćemo „//“. Dva operatora dijeljenja odnosno dva *slash* znaka. Sve što se nalazi u toj jednoj liniji koda iza dva *slash* znaka će se smatrati komentarom (MS VC++ će komentare obilježiti zelenom bojom).

Ukoliko imamo veći dio kôda za zakomentirati (zamislite 50 linija kôda) tada nebi imalo smisla ići liniju po liniju i svuda stavljati po dva *slash* simbola i kanije svaki od njih ručno i uklanjati. Naravno da postoji lakši način a to je korištenjem „/\*“ i „\*/“ simbola. Početak (od kuda želimo početi komentiranje) se obilježava sa *slash* simbolom i *zvjezdicom* a kraj (do kuda da se kôd zakomentira) obilježavamo prvo simbolom *zvjezdice* a zatim *slash* simbolom.

U oba slučaja će zakomentirani kod postati zelen i sve što je zakomentirano se neće izvoditi prilikom prevođenja programa. Ukoliko imamo zakomentirano više linija, moguće je taj dio koda minimizirati pritiskom na simbol „-“ (*minus* odnosno *povlaka*) koji se nalazi uz početnu liniju komentara. Na isti način, samo pritiskom na simbol „+“ (*plus*) će se sav zakomentirani tekst prikazati.



```
#include <iostream>
using namespace std;
int main()
{
    cout <<"Teskt koji će se ispisati";

    // komentar - način 1

    /* komentar
    na
    način 2
    */

    return 0;
}
```

Slika 12 - Prikaz komentara u editoru

## 11) Uvodni zadatci (tipovi podataka, upis i čitanje iz konzole, sintaksa, operatori)

- 1) Deklarirati i inicijalizirati dvije varijable imena „prviBroj“ i „drugiBroj“, te varijablu imena „zbroj“ čija će vrijednost biti jednaka zbroju vrijednosti varijabli „prviBroj“ i „drugiBroj“. Vrijednost varijable je potrebno i ispisati u konzolu pomoću `cout` naredbe. Potrebno je napraviti varijacije tipova podataka kako je prikazano u tablici i pogledati rezultat :

| prviBroj     | drugiBroj    | zbroj        |
|--------------|--------------|--------------|
| <i>int</i>   | <i>int</i>   | <i>int</i>   |
| <i>int</i>   | <i>float</i> | <i>int</i>   |
| <i>float</i> | <i>int</i>   | <i>int</i>   |
| <i>float</i> | <i>float</i> | <i>int</i>   |
| <i>int</i>   | <i>int</i>   | <i>float</i> |
| <i>int</i>   | <i>float</i> | <i>float</i> |
| <i>float</i> | <i>float</i> | <i>float</i> |

### Primjer

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

int main()
{
    int prviBroj = 3;
    float drugiBroj=3.456;
    int zbroj = prviBroj + drugiBroj;

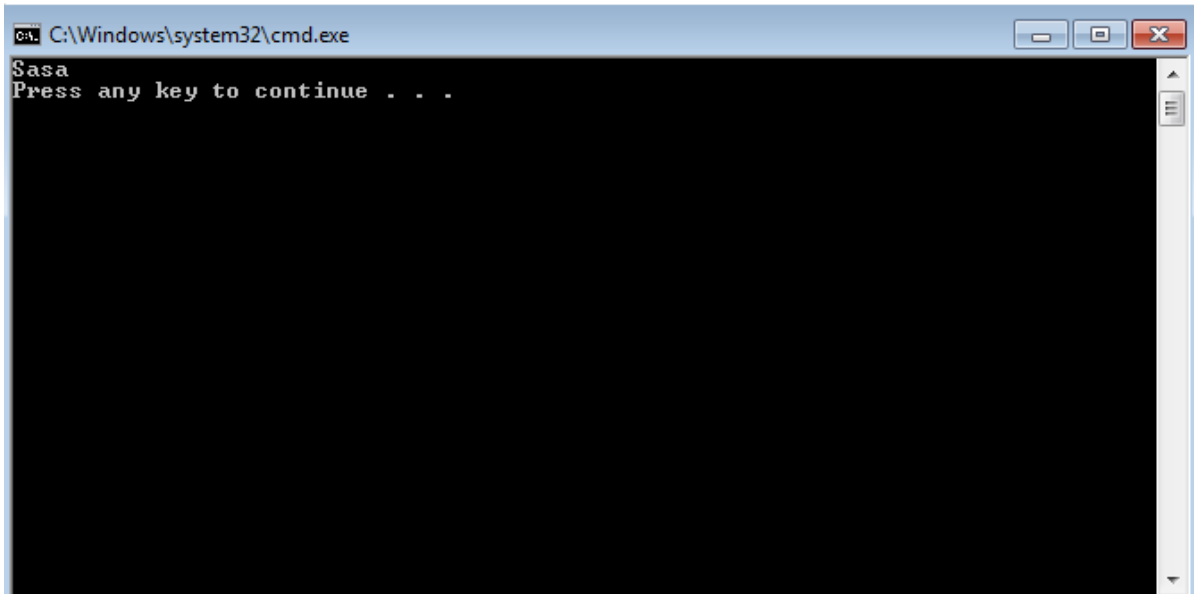
    cout <<zbroj<<endl;

    return 0;
}
```

Uočite što se događa ako je varijabla „zbroj“ tipa „*integer*“ a varijabla „prviBroj“ i/ili „drugiBroj“ su tipa „*float*“ i sadrže znamenke različite od 0 (nula) iza decimalnog zareza.

- 2) Deklarirajte, inicijalizirajte i ispišite vrijednost varijable mijenjajući tip podatka varijable (*int*, *float*, *string*, *char*, *double*, *bool*). Isprobajte sve kombinacije.

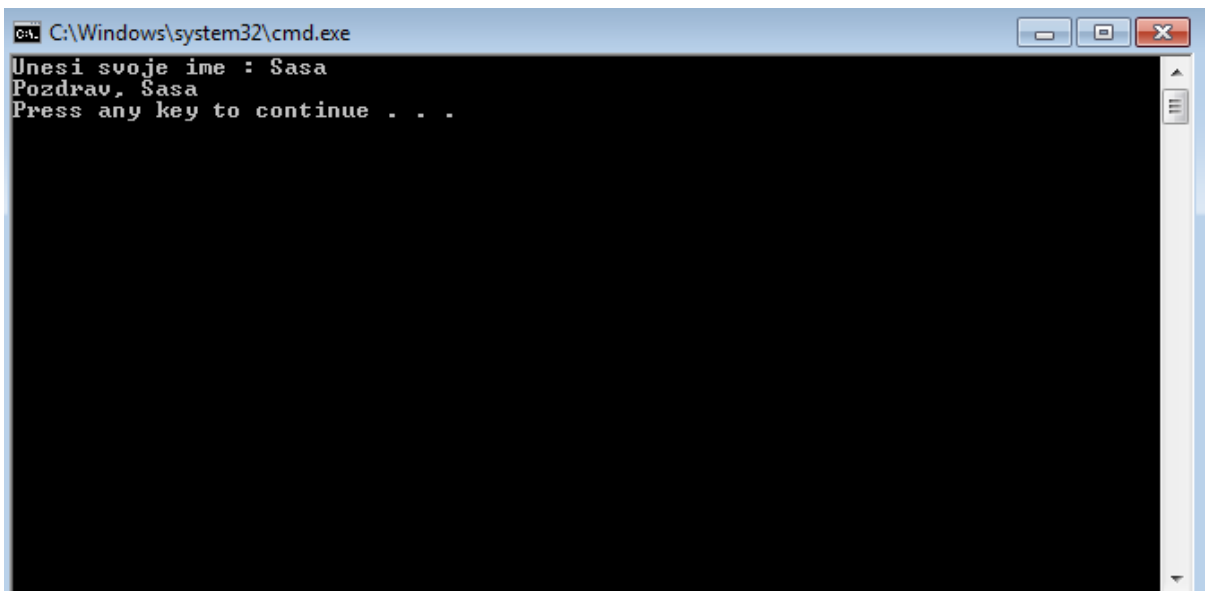
- 3) Uključite biblioteku „*string*“, deklarirajte varijablu „mojelme“ te inicijalizirajte vrijednost te varijable na vaše ime (podsjetnik – biblioteka *string* se uključuje pomoću `#include<string>` naredbe i *string* tip podataka nije osnovni tip). Za kraj, ispišite vrijednost varijable „mojelme“ u konzolu.



```
C:\Windows\system32\cmd.exe
Sasa
Press any key to continue . . .
```

Slika 13 - Rješenje zadatka 03 u dijelu Uvodni zadatci

- 4) Deklarirajte varijablu „ime“ tipa *string*. U konzolu ispišite tekst „Unesi svoje ime : “. Nakon toga korisnik treba unijeti svoje ime i ta se vrijednost sprema u varijablu „ime“. Za kraj ispišite poruku u konzoli „Pozdrav, <ime>“ gdje je <ime> potrebno zamijeniti stvarnim imenom korisnika. (Podsjetnik – tekst za ispis u konzolu se piše pod dvostrukim navodnicima dakle `cout<<“Unesi svoje ime : „`).

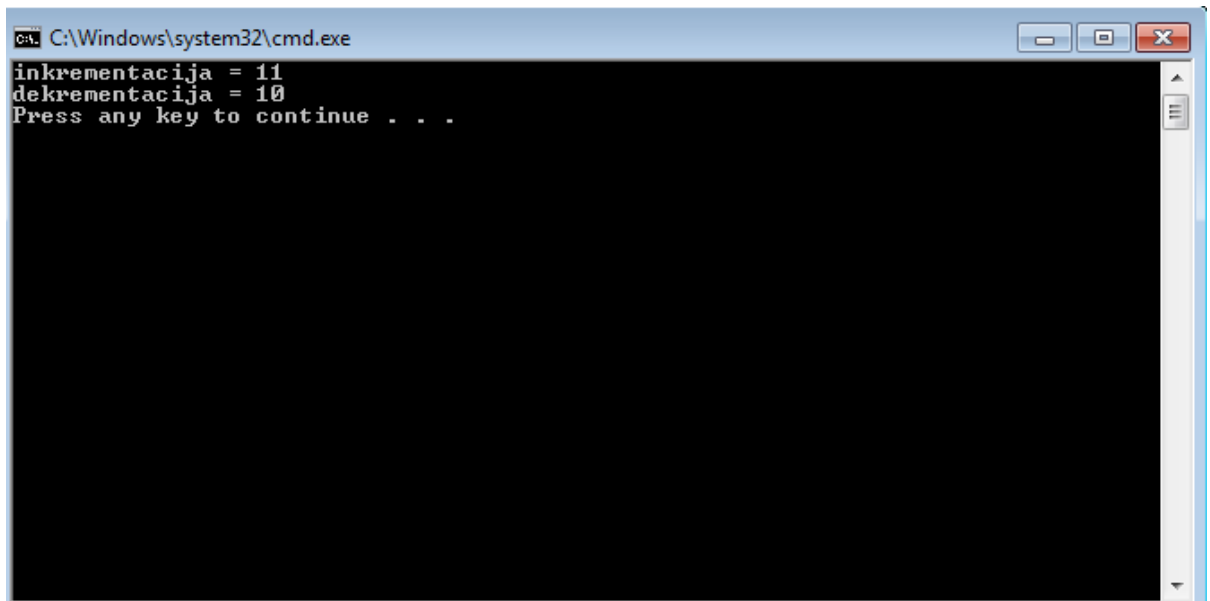


```
C:\Windows\system32\cmd.exe
Unesi svoje ime : Sasa
Pozdrav, Sasa
Press any key to continue . . .
```

Slika 14 - Rješenje zadatka 04 u dijelu Uvodni zadatci



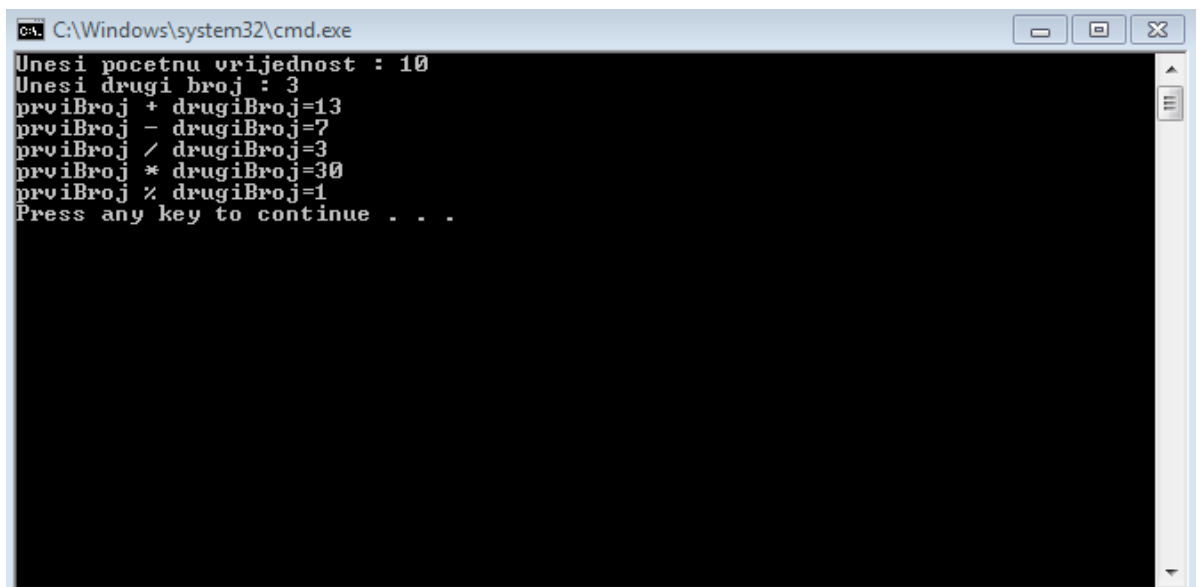
- 5) Deklarirajte varijablu „broj“ tipa *integer* i postavite joj vrijednost na broj 10. U konzolu ispišite njenu vrijednost nakon inkrementacije i tu novu vrijednost zatim dekrementirajte i ispišite tu vrijednost u konzolu. Dopušteno je korištenje samo unarnih operatora koje smo obradili.



```
C:\Windows\system32\cmd.exe
inkrementacija = 11
dekrementacija = 10
Press any key to continue . . .
```

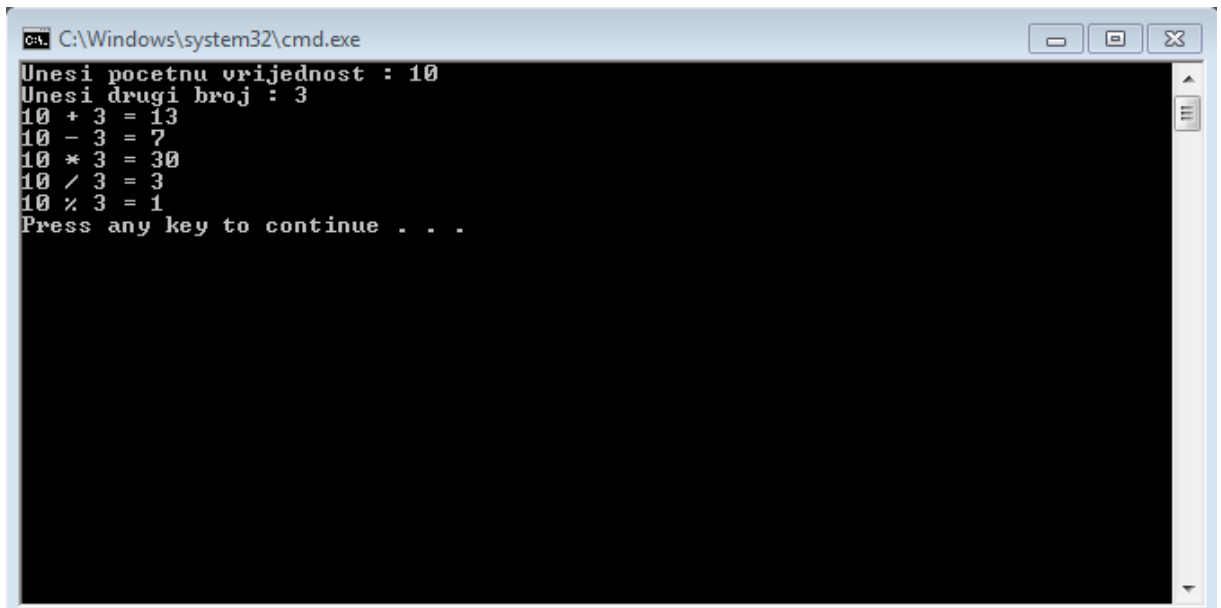
Slika 15 - Rješenje zadatka 05 u dijelu Uvodni zadatci

- 6) Deklarirajte varijable „prviBroj“ i „drugiBroj“, sve tipa *integer*. Od korisnika zatražite da unese vrijednosti varijabli i te vrijednosti spremite u pripadajuće varijable. Za kraj, u konzolu ispišite rezultate matematičkih operacija korištenjem te dvije varijable i svih spomenutih binarnih operatora. Napomena – prikazana su dva načina rješavanja.



```
C:\Windows\system32\cmd.exe
Unesi početnu vrijednost : 10
Unesi drugi broj : 3
prviBroj + drugiBroj=13
prviBroj - drugiBroj=7
prviBroj / drugiBroj=3
prviBroj * drugiBroj=30
prviBroj % drugiBroj=1
Press any key to continue . . .
```

Slika 16 - Rješenje zadatka 06 (prvi način) u dijelu Uvodni zadatci



```
C:\Windows\system32\cmd.exe
Unesi pocetnu vrijednost : 10
Unesi drugi broj : 3
10 + 3 = 13
10 - 3 = 7
10 * 3 = 30
10 / 3 = 3
10 % 3 = 1
Press any key to continue . . .
```

Slika 17 - Rješenje zadatka 06 (drugi način) u dijelu Uvodni zadatci

Objašnjenje (usporedba) prvog i drugog načina :

Rezultat i način kako smo došli do njega je u oba primjera isti ali je razlika u ispisu. U **prvom načinu** smo koristili puno naziv varijabli pa smo prilikom ispisa stavljali ime naše varijable pod dvostruke navodnike (isto vrijedi i za prikaz simbola operatora pridruživanja i binarnih operatora). Tek na kraju (rezultat) smo koristili vrijednosti koje se nalaze pohranjene u varijablama. U **drugom načinu** smo odmah koristili vrijednosti upisane u varijable a prikaz simbola koju operaciju obavljamo smo naravno riješili korištenjem dvostrukih navodnika.

## 12) Dijagram toka i pseudo kôd

Iako se na prvu čini kako se programiranje svodi samo na neprestano pisanje kôda, istina je ipak nešto drugačija. Prilikom izrade aplikacija, programeri najčešće znaju krajnju točku odnosno čemu aplikacija koju izrađuju treba služiti (i početak naravno a to je prazan *editor*). Kako bi si mogli olakšati ali i predočiti probleme s kojima će se susresti, programeri problem razbijaju u manje probleme a te probleme u još manje i sve do određenih dijelova koji su im znatno jednostavniji za riješiti. Po završetku jednog dijela pomiču se na drugi. Ovakav pristup se također zove **DEKOMPOZICIJA**. Papir i olovka su dobri prijatelji svakom programeru s kojima se često i druže. Dva su osnovna načina prikazivanja problema. Za prvi primjer uzet ćemo da je problem izgradnja drvene kuće a u drugom primjeru ćemo izračunavati površinu naše kuće.

### a. Pseudo kôd

Iako stručan i zanimljiv izraz, pseudo kôd se zaista odnosi na tekstualno opisivanje problema. Napravimo li malu dekompoziciju problema gradnje kuće možemo doći do više manjih i jasnijih dijelova:

- 1) izgraditi temelje
- 2) izgraditi drvenu konstrukciju tijela kuće
- 3) izgraditi drvenu konstrukciju krova
- 4) Dodati daske za tijelo kuće
- 5) Dodati krov kuće

Svaki od tih problema možemo rastaviti na još manje cjeline pa tako izgradnju temelja možemo podijeliti prvo na obradu zemlje a zatim na betoniranje. Svaki od tih problema možemo rastaviti na još manje problemske jedinice i tako više-manje u nedogled. **Napomena** – treba znati granicu kada prestati razdvajati probleme u manje cjeline kako se ne bi previše vremena utrošilo na pisanje pseudo kôda za koji postoji dobra šansa da na kraju neće biti ni uporabljiv.

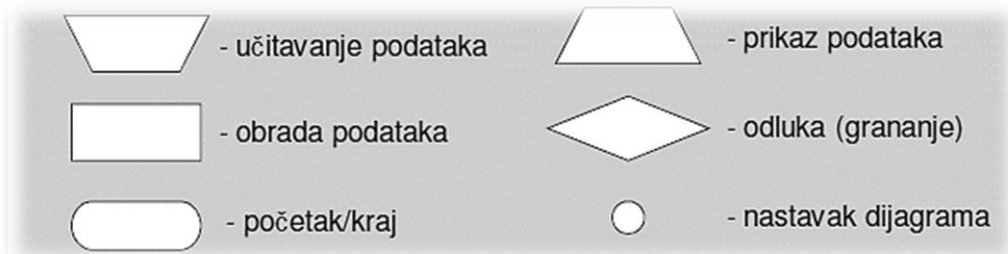
Naš pseudo kod bi za prvih pet koraka izgledao :

```
izgradi temelje
napravi konstrukciju
napravi krov
dodaj daske
dodaj krov
```

Kao što vidite, opis problema kuće je sada razumljiviji i smisleniji a nismo napisali niti jednu *sintaksno* ispravnu rečenicu odnosno liniju kôda.

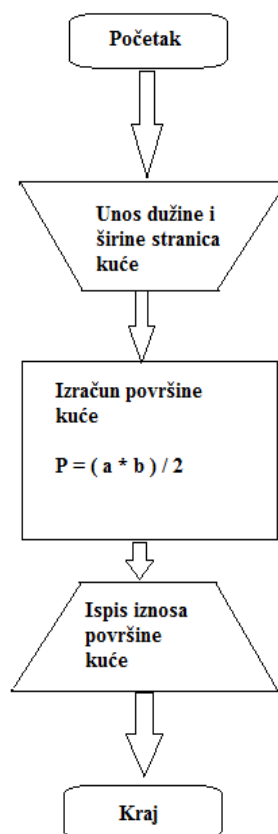
## b. Dijagram toka

Pristupiti dijagramu toka možemo kao i pseudo kôdu. Osnovni način razmišljanje je isti a razlika je samo u prikazu. Za razliku od pseudo kôda, dijagram toka karakterizira grafičko skiciranje problema. Za potrebe dijagrama toka postoji šest najčešće korištenih simbola čijih se značenja kao niti simbola nije potrebno pridržavati ali radi konvencije je poželjno.



Slika 18 - Simboli koji se koriste u dijagramu toka <sup>4</sup>

- Radi lakšeg pojašnjavanja, zamislite da je naša kuća pravokutnog oblika (ravan krov, jednaka lijeva i desna strana kuće te prednja i stražnja strana)

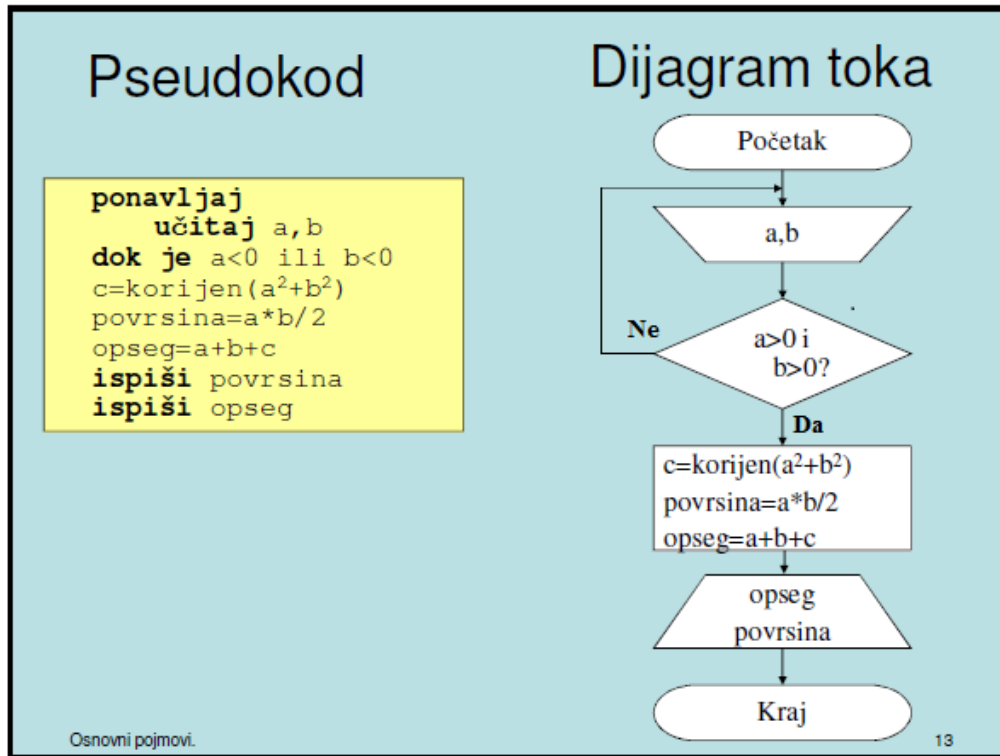


Slika 19 - Dijagram toka pri izračunu površine kuće

<sup>4</sup> izvor : Ivo Beroš, VERN' pdf – POINT – Uvod u programiranje – 2010. – stranica 2

### c. Primjer pseudokôda i dijagrama toka

U ovom primjeru, zadatak je izračunati površinu i opseg pravokutnog trokuta na osnovu veličina stranica koje upisuje korisnik te u konzolu ispisati vrijednosti površine i opsega.



Slika 20 - Primjer pseudo kôda i dijagrama toka <sup>5</sup>

<sup>5</sup> izvor : Ivo Beroš pdf materijali – POINT – Uvod u programiranje – 2010. – stranica 2

## 13) Petlje i kontrole toka

U programiranju često koristimo kontrole toka i petlje koje nam omogućuju kontroliranje izvedbe našeg programa ili ponavljanje neke operacije određeni broj puta. Petlje i kontrole toka su izuzetno važan dio programiranja, no kao i kod svega u programiranju, neke petlje su češće u uporabi a neke rjeđe. Loša vijest je što ćete morati znati *sintaksu* svih navedenih petlji i kontroli toka. Dobra vijest je što ih je zaista malo. ☺

**Kontrole toka** se koriste želimo li neku operaciju izvršiti isključivo ako je neki uvjet zadovoljen. Primjerice, površinu pravokutnog trokuta ćemo računati samo ako je korisnik ispravno unio sve tražene parametre. Ako nije zahtijevat ćemo od njega ponovni upis.

**Petlje** nam omogućuju ponavljanje neke linije ili skupa linija određeni broj puta. Najčešće se točan broj koliko se puta nešto treba izvršiti u petlji ne zna unaprijed. Možda zvuči malo čudno, ali je istiniti. Primjerice, izradite li aplikaciju koja će na ekran ispisati slovo „A“ onoliko puta koliko puta korisnik kaže nećete moći unaprijed znati točno koliko će se slova ispisati.

Unese li korisnik da želi slovo „A“ ispisati 3 puta vi u svom programu možete sljedeće opcije :

- opcija 1 (korisnik je unio da se slovo A ispiše nula puta odnosno niti jednom
  - vaš pseudo kôd bi glasio : nemoj ništa ispisati
- opcija 2 (korisnik je unio da se slovo A ispiše jednom)
  - vaš pseudo kôd bi glasio : ispiši slovo „A“
- opcija 3 (korisnik je unio da se slovo A ispiše dva puta)
  - vaš pseudo kôd bi glasio : ispiši slovo „A“ i opet ispiši slovo „A“
- opcija 4 (korisnik je unio da se slovo A ispiše tri puta)
  - vaš pseudo kôd bi glasio : ispiši slovo „A“ i opet ispiši slovo „A“ i opet ispiši slovo „A“

**Ovakav pristup naravno nema smisla** posebice razmotrimo li mogućnost da korisnik može poželjeti slovo „A“ ispisati 10.000 puta. To znači da bi morali imati 10.001 opciju i da bi u posljednjoj opciji morali deset tisuća puta ponoviti naredbu „ispisati slovo A“.

**Ispravan pristup** je da napravimo petlju koja će ispisivati slovo „A“ određeni broj puta. To znači da pitamo korisnika koliko puta želi ispisati slovo „A“ i petlji damo uvjet „ispisuj na ekran slovo „A“ dok god ne ispišeš onoliko puta koliko je korisnik zatražio“.

## 14) Osnovne kontrole toka → if, if-else, ugniježđeni if

### a. Kontrola toka pomoću „if“-a

Jedan od najčešćih načina kontroliranja toka programa je korištenje „**If izjava**“ (*engl. If statement*). Prevedeno na hrvatski jezik, radi se o „**ako izjavi**“ no u programiranju se koristi naravno engleski jezik pa je jedini ispravan naziv – „if“. Sve kontrole toka (kao i petlje i većina toga u programiranju) se pišu malim slovima. „if“ nam omogućuje izvršavanje nekog dijela našeg kôda samo ako je neki uvjet zadovoljen.

#### **Sintaksa :**

```
if (uvjet koji treba biti zadovoljen)
{ kôd će se izvršiti ako je uvjet zadovoljen }
```

Za primjer ćemo uzeti da želimo izračunati opseg kružnice. Formula  $O = 2 * r * \pi$ . Mi trebamo pitati korisnika za iznos polumjera kružnice i tu vrijednost spremiti u varijablu „r“. Ako je korisnik unio vrijednost veću od nule onda ćemo izvršiti dio kôda u kojem se izvršava izračun i ispisati na ekran vrijednost opsega. Ako korisnik nije unio broj veći od nule, korisniku ćemo ispisati poruku o pogrešnom unosu i završiti izvođenje aplikacije.

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

int main()
{
    float r=0;
    cout << "Unesi polumjer kruznice : "; cin>>r;

    if ( r>0 )
    {
        cout << "Opseg kruznice iznosi " << 2*r*3.14 << " cm"<<endl;
    }

    if ( r<=0 )
    {
        cout << "Kriva vrijednost. Gasim aplikaciju! "<<endl;
    }
    return 0;
}
```

Zamislamo sada da smo željeli ispisati posebnu obavijest ako korisnik unese nulu kao vrijednost polumjera a posebnu vrijednost ako unese vrijednost manje od nula. Naš kôd bi bio još veći.

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

int main()
{
    float r=0;
    cout << "Unesi polumjer kruznice : "; cin>>r;

    if ( r>0 )
    {
        cout << "Opseg kruznice iznosi " << 2*r*3.14 << " cm"<<endl;
    }

    if ( r==0 )
    {
        cout << "Ne moze biti 0 jer to znaci da niti nema kruznice.
Gasim aplikaciju! "<<endl;
    }

    if ( r<0 )
    {
        cout << "Ne moze biti manje od 0 nikako. Pozdrav od C++"<<endl;
    }
    return 0;
}
```

Sada zamislite da smo još morali paziti i na najveću dozvoljenu vrijednost. Recimo da je maksimalna vrijednost 10. To znači da sve što je između 0 i 10 (ne uključujući broj nula ali uključujući broj deset) dozvoljeno i tada ćemo izračunati površinu a ostalo nije dozvoljeno. Morali bismo imati još jedan „if“ čiji bi uvjet bio „ako je polumjer veći od 10 izbaciti pogrešku“. Sagledamo li malo bolje vidimo kako imamo samo jedan uvjet koji nam odgovara i ukoliko je samo taj jedan uvjet zadovoljen izvršit će se izračun površine. Za sve ostale mogućnosti trebamo baciti grešku. U ovakvim slučajevima koristimo dodatak „if“-u a zove se (čak i logično), „else“.

Važno je napomenuti kako se unutar jednog „if“-a može nalaziti još jedan „if“ i unutar njega još jedan i tako možemo ići u dubinu koliko god poželimo. Naravno, pisanje desetak „if“-ova jednog unutar drugog je krajnje nečitko pa time niti nema smisla. Postavljanje jednog „if“-a unutar drugog se zove **ugniježđeni if**.



## b. if-else

Else nam omogućuje izvršavanje koda ukoliko jedan ili više uvjeta unutar „if“-a nisu zadovoljeni kao što je to slučaj u gornjem primjeru.

### Sintaksa :

```
else
{
kôd koji će se izvršiti ukoliko uvjet u if-u nije zadovoljen
}
```

Korištenjem „else“-a naš kôd će sada ovako izgledati :

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

int main()
{
    float r=0;
    cout << "Unesi polumjer kruznice : "; cin>>r;

    if ( r>0 )
    {
        cout << "Opseg kruznice iznosi " << 2*r*3.14 << " cm"<<endl;
    }

    else
    {
        cout << "Kriva vrijednost. Gasim aplikaciju!"<<endl;
    }
    return 0;
}
```

Sada je lako uočiti kako nam „else“ omogućava puno jednostavnije i sažetije pisanje kôda. Važno je napomenuti kako se unutar svakog „else“-a može nalaziti novi „if“ i novi „else“ i unutar u dubinu možemo ići koliko god poželimo.

Pokušamo li upisati „else“ bez „if“-a dobit ćemo *sintaksnu* pogrešku jer se moramo reći *compileru* koji je primarni uvjet i ukoliko on nije izvršen što da se izvrši. Suprotno „else“-u, „if“ može biti zapisan bez „else“-a.

### c. Ugniježđeni if

Ugniježđivanje jednog „if“-a unutar drugog može biti izuzetno korisno. Pogledajmo naš zadatak kako bi izgledao bez korištenja „else“-a a da smo željeli ispisati dvije različite poruke o pogrešci ovisno da li je korisnik unio vrijednost nula za polumjer ili vrijednost manju od nule.

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

int main()
{
    float r=0;
    cout << "Unesi polumjer kruznice : "; cin>>r;

    if ( r<=0 )
    {
        if( r == 0 )
        {
            cout<<"Ne moze biti nula"<<endl;
        }
        if ( r<0 )
        {
            cout<<"Ne moze niti manje od nule"<<endl;
        }
    }
    return 0;
}
```

Sada vidimo primjer ugniježđenog „if“-a čija prava primjena se zaista počinje uočavati pri malo zahtjevnijim zadacima. Korištenje tri „if“-a ili jednog „if“-a unutar kojeg se nalaze još dva nam u ovom primjeru daje iste rezultate pa je sasvim svejedno koji ćemo pristup za rješavanje ovog problema odabrati. S vremenom i kroz rješavanje kompleksnijih problema ćete sami početi uočavati kada je koji pristup kvalitetniji.

#### d. Zadaci za vježbu kontrole toka

- 1) Napraviti aplikaciju koja pita korisnika da unese neki broj. Ukoliko je broj negativan treba ispisati poruku da je broj negativan odnosno ako je broj pozitivan treba ispisati poruku da se radi o pozitivnom broju. Ako je unesen broj 0 (nula) treba ispisati da je unesen broj 0 (nula).
- 2) Napraviti aplikaciju koja pita korisnika da unese neki cijeli broj. Ako je broj manji ili jednak 0 (nula) treba provjeriti da li se radi o broju koji je negativan ili se radi o nuli. Ako se radi o nuli aplikaciju ispisuje poruku da je unesena vrijednost nula. Ako je broj negativan treba ga kvadrirati i izbaciti vrijednost kvadrata unesenog broja. Ako je broj pozitivan i u rasponu između 1 i 10 treba ga također kvadrirati, a ako je broj u rasponu od 11 od 100 treba ispisati razliku tog unesenog broja i broja 25. Dobije li se kao rezultat negativan broj treba još ispisati i kvadriranu vrijednost negativnog broja. Ako je rezultat pozitivan treba samo ispisati njegovu vrijednost. Ako je uneseni broj veći od 100 treba samo ispisati da se radi o pozitivnom broju.
- 3) Tražite od korisnika da unese dva broja. Ako je prvi broj veći od drugog, na ekran ispišite razliku prvog i drugog broja, njihov umnožak te razliku njihovih kvadrata. Ako je drugi broj veći od prvog, na ekranu ispišite zbroj oba broja, njihov količnik i zbroj njihovih kvadrata.

## 15) Petlje for, while, do-while

Logika koja stoji iza petlji je da omogućimo izvršavanje nekog kôda dok god je neki uvjet zadovoljen. Ovisno od petlje do petlje, mijenja se njihova svrha, *sintaksa* i svojstva.

### a. for petlja

For petlja svoju vrlo čestu primjenu nalazi kako i prilikom izvršavanja istog kôda više puta tako i prilikom prolaska kroz polja. Što su polja i koja su njihova svojstva će te saznati u narednim poglavljima no spomenuta su čisto kako bi bili upućeni u njihovo postojanje i često korištenje „for“ petlji u kombinaciji s poljima. *Sintaksa* je nešto kompliciranija za shvatiti naspram ostalih petlji no nakon nekoliko korištenja nećete imati problema.

#### Sintaksa :

```
for ( inicijalizacija, izvođenje, uvećanje )
{
kôd koji se izvršava dok god je uvjet zadovoljen
}
```

**inicijalizacija** – postavljamo vrijednost neke varijable na neku određenu vrijednost. Često se koriste slovo „i“ i slovo „j“. Korištenje ovih slova nije striktno definirano ali se radi pridržavanja konvencije najčešće koriste ta dva slova. Važno – varijablu koju inicijaliziramo nije potrebno definirati prije korištenja „for“ petlje već se ona najčešće deklarira prilikom stvaranja „for“ petlje i to unutar „for“ petlje na mjestu „inicijalizacija“. Primjer inicijalizacije : `int i = 1;`

**izvođenje** – ovaj parametar nam govori do kada će se for petlja izvršavati. Ako smo u prvom parametru (inicijalizacija) postavili vrijednost varijable „i“ na 1 (jedan) i sada kao parametar „izvođenje“ upišemo „i < 5“ mi smo rekli da se petlja izvrši 4 (četiri) puta.

Parametar „izvođenje“ možemo gledati kao uvjet kod „if“-a dok ćemo parametar „inicijalizacija“ gledati kao vrijednost koju uspoređujemo s onom postavljenom u uvjetu. Ne brinite, sve će imati više smisla u prikazu primjera kojih će za potrebe „for“ petlje biti nekoliko.

**uvećanje** – srećom posljednji parametar u ovoj kompliciranoj petlji i vjerojatno najjednostavniji za shvatiti. Ovdje upisujemo za koliko da se uveća varijabla koju smo inicijalizirali u parametru 1 (inicijalizacija) nakon što se izvrši kod u tijelu petlje. Često ćete koristiti inkrementaciju odnosno `i++;`

## b. Sintaksa i primjeri „for“ petlje :

### primjer 1 :

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

int main()
{
    for ( int i = 0; i<5;i++)
    {
        cout << "Pokusavam razumjeti for petlju" <<endl;
    }
}
```

Tijelo for petlje sačinjava jedna naredba i to ispisa : `cout << "Pokusavam shvaitit for petlju" <<endl;`. kod ove naredbe nemamo ništa novo. Problem u shvaćanju se javlja kod čudnog teksta unutar obliha zagrada.


**Prvi parametar** (do prvog točka-zarez simbola) je inicijalizacija varijable čije je ime „i“ a ona je tipa *integer*. Vrijednost varijable „i“ smo postavili na 0 (nula).

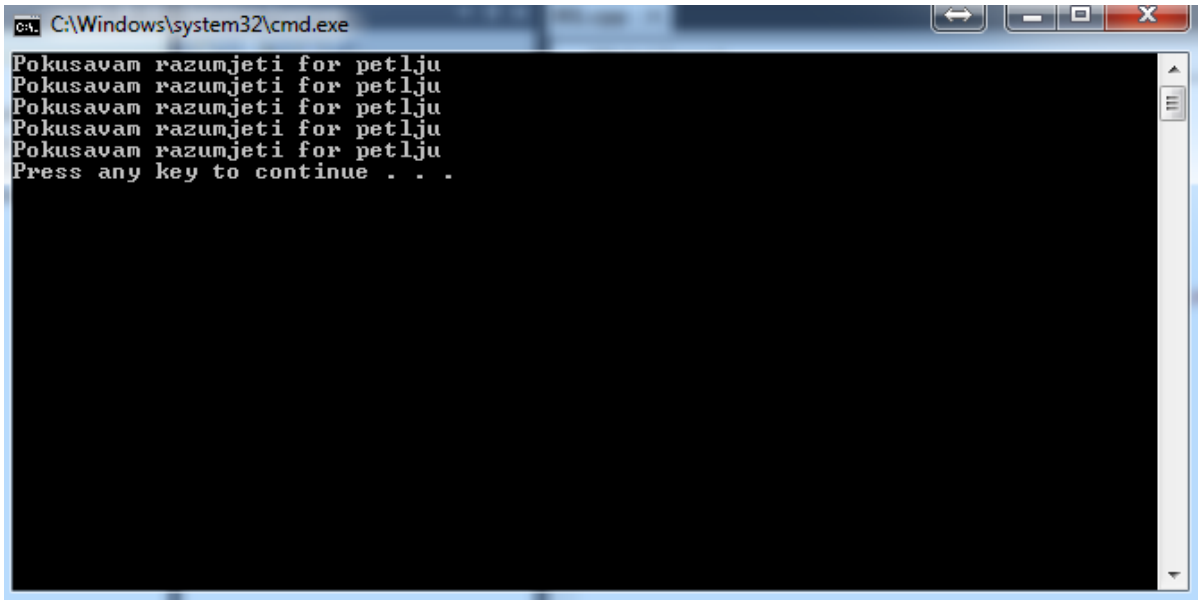
**Dalje** smo rekli programu : „sve dok je vrijednost varijable „i“ manja od broja 5 ti izvršavaj kôd koji ti piše u tijelu petlje“. To znači da će program izvršiti naredbu ispisa i ako je vrijednost varijable „i“ još uvijek manja od 5 ponovno će se izvršiti isti kôd i tako sve dok je vrijednost varijable „i“ manja od 5 odnosno, u ovom konkretnom slučaju dok varijabla „i“ ne poprimi vrijednost 5.

**Promjenu vrijednosti varijable „i“** izvršavamo pomoću trećeg parametra. U ovom slučaju mi smo programu rekli „svaki put kada izvršiš kôd koji ti piše u tijelu petlje, ti povećaj vrijednost varijable za jedan.

Detaljno objašnjenje što se i kako događa :

- 1) vrijednost varijable „i“ je 0 (nula)
- 2) program provjerava ako je „i“ manji od 5. (**0 < 5**) 😊
- 3) program uvidi da je „i“ manji od 5
- 4) ispisuje se naredba (**prvo ispisivanje**)
- 5) uvećava se vrijednost varijable za jedan i sada „i“ ima vrijednost 1
- 6) provjera da li je „i“ manji od 5 (**1 < 5**) 😊
- 7) program utvrdi da je „i“ manji od 5
- 8) ispisuje se naredba (**drugo ispisivanje**)
- 9) uvećava se vrijednost varijable za jedan i sada „i“ ima vrijednost 2
- 10) provjera da li je „i“ manji od 5 (**2 < 5**) 😊
- 11) program utvrdi da je „i“ manji od 5
- 12) ispisuje se naredba (**treće ispisivanje**)
- 13) uvećava se vrijednost varijable za jedan i sada „i“ ima vrijednost 3
- 14) provjera da li je „i“ manji od 5 (**3 < 5**) 😊
- 15) program utvrdi da je „i“ manji od 5
- 16) ispisuje se naredba (**četvrto ispisivanje**)
- 17) uvećava se vrijednost varijable za jedan i sada „i“ ima vrijednost 4
- 18) provjera da li je „i“ manji od 5 (**4 < 5**) 😊

- 19) program utvrdi da je „i“ manji od 5
- 20) ispisuje se naredba **(peto ispisivanje)**
- 21) uvećava se vrijednost varijable za jedan i sada „i“ ima vrijednost 5
- 22) provjera da li je „i“ manji od 5 (**5 < 5**) 
- 23) program utvrdi da sada „i“ više nije manji od 5
- 24) Petlja se prekida
- 25) Program nastavlja s izvođenjem kôda koji se nalazi nakon „for“ petlje



```
C:\Windows\system32\cmd.exe
Pokusavam razumjeti for petlju
Pokusavam razumjeti for petlju
Pokusavam razumjeti for petlju
Pokusavam razumjeti for petlju
Pokusavam razumjeti for petlju
Press any key to continue . . .
```

Slika 21 - Rezultat nakon izvršenja kôda u primjeru 1

#### Savjet

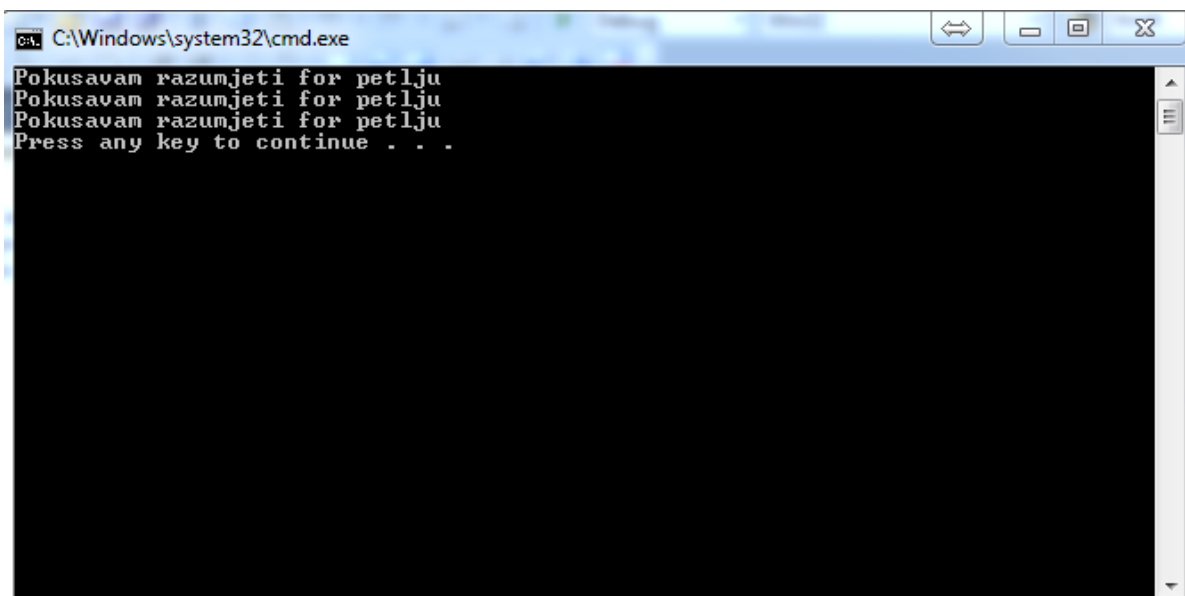
Uzmite prazan papir i zapišite 2-3 varijacije for petlji (mijenjajte samo drugi parametar) i pišite si svaki korak i izgovarajte ga na glas. Nakon 10-ak minuta će sve postati jednostavnije.

### Primjer 2:

```
for ( int i = 0; i<5;i+=2)
{
    cout << "Pokusavam razumjeti for petlju" <<endl;
}
```

Detaljno objašnjenje :

- 1) vrijednost varijable „i“ je 0 (nula)
- 2) program provjerava ako je „i“ manji od 5. ( $0 < 5$ ) 😊
- 3) program uvidi da je „i“ manji od 5
- 4) ispisuje se naredba (**prvo ispisivanje**)
- 5) uvećava se vrijednost varijable za dva i sada „i“ ima vrijednost 2
- 6) provjera da li je „i“ manji od 5 ( $2 < 5$ ) 😊
- 7) program utvrdi da je „i“ manji od 5
- 8) ispisuje se naredba (**drugo ispisivanje**)
- 9) uvećava se vrijednost varijable za dva i sada „i“ ima vrijednost 4
- 10) provjera da li je „i“ manji od 5 ( $4 < 5$ ) 😊
- 11) program utvrdi da je „i“ manji od 5
- 12) ispisuje se naredba (**treće ispisivanje**)
- 13) uvećava se vrijednost varijable za dva i sada „i“ ima vrijednost 6
- 14) provjera da li je „i“ manji od 5 ( $6 < 5$ ) ❌
- 15) program utvrdi da sada „i“ više nije manji od 5
- 16) Petlja se prekida
- 17) Program nastavlja s izvođenjem kôda koji se nalazi nakon „for“ petlje



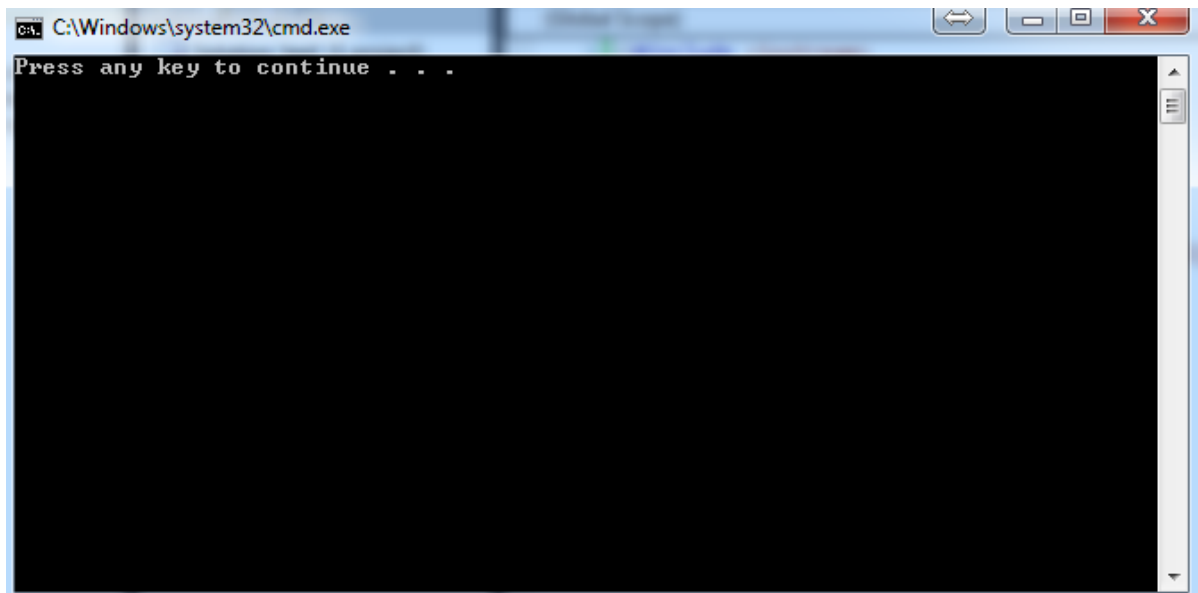
```
C:\Windows\system32\cmd.exe
Pokusavam razumjeti for petlju
Pokusavam razumjeti for petlju
Pokusavam razumjeti for petlju
Press any key to continue . . .
```

Slika 22 - Rezultat nakon izvršenja kôda u primjeru 2

### Primjer 3:

```
for ( int i = 2; i<0;i--)
{
    cout << "Pokusavam razumjeti for petlju" <<endl;
}
```


Pogledajte rješenje! Zašto se nije ništa ispisalo?



Slika 23 - Rezultat nakon izvršenja kôda u primjeru 3

Pogledajte opet što je upisano unutar uglatih zagrada „for“ petlje. Postavljena je varijabla „i“ na vrijednost 2 a provjera glasi „ako je  $i < 0$  izvrši kôd u tijelu petlje“. Kako je od samog početka vrijednost varijable „i“ postavljena na 2 (dva), odmah u startu nije uvjet zadovoljen i petlja se prekida.

Detaljno objašnjenje :

- 1) vrijednost varijable „i“ je 2 (dva)
- 2) program provjerava ako je „i“ manji od 5. ( $2 < 0$ ) 
- 3) program utvrdi da „i“ nije manji od 0
- 4) Petlja se prekida
- 5) Program nastavlja s izvođenjem kôda koji se nalazi nakon „for“ petlje



### c. while petlja

Ako ste uspjeli svladati „for“ petlju onda nećete imati nikakvih problema sa slijedeće dvije petlje. Prvo ćemo predstaviti „while“ petlju na koju ćemo nadograditi „do-while“ petlju. Ako bismo preveli na hrvatski jezik, ovu petlju bismo nazvali „dok petljom“. Kako već i samo ime sugerira, kôd koji se nalazi unutar tijela ove petlje će se izvršavati dok god je neki uvjet ispunjen. Kao što ste možda zamijetili, ideologija je slična kao „for“ petlji. Obje će se izvršavati dok je neki uvjet zadovoljen ali razlika je što će se „for“ petlja izvršavati točno definiran broj puta (imamo početnu vrijednost i krajnju vrijednost koju može varijabla poprimiti) dok u slučaju „while“ petlje ne znamo koliko će se puta neki dio kôda izvršiti.

### d. Sintaksa i primjeri while petlje

#### Sintaksa :

```
while ( uvjet )
{
kôd koji će se izvršavati dok god je uvjet ispunjen
}
```

**uvjet** – opisujemo što mora biti zadovoljeno da bi se naš kôd izvršio

#### **Primjer 1 :**

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

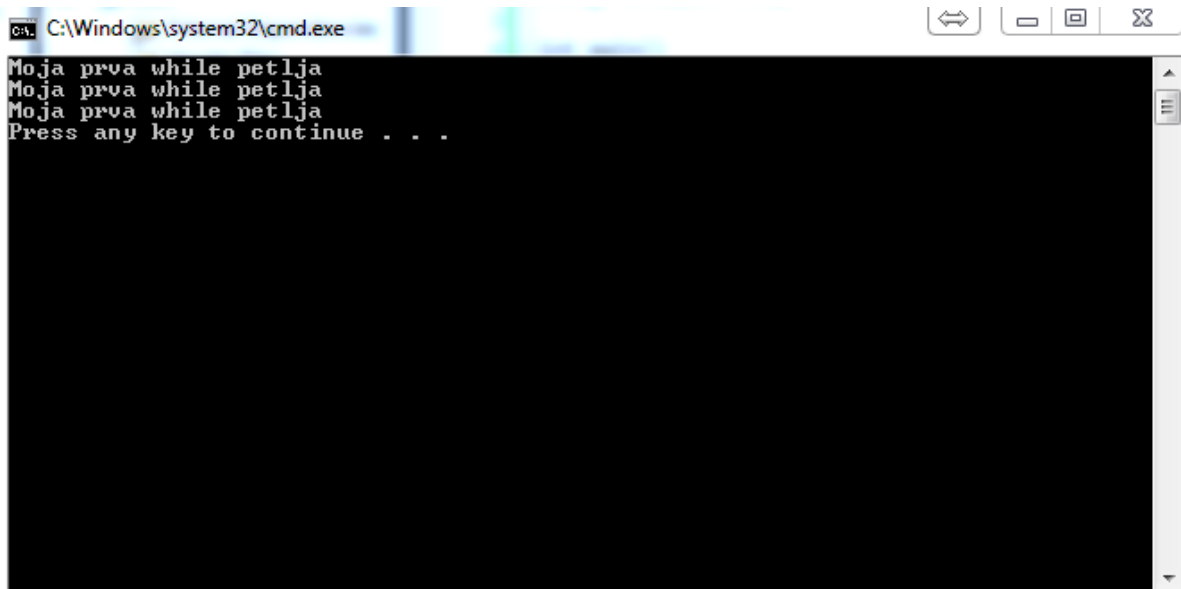
int main()
{
    int i = 0;
    while (i<3)
    {
        cout<<"Moja prva while petlja"<<endl;
        i++;
    }
}
```

**Uočite** – varijabla koja se nalazi unutar uglatih zagrada odnosno dio je našeg uvjeta morala se deklarirati i inicijalizirati prije provjere uvjeta.

#### Detaljno objašnjenje :

- 1) provjera uvjeta da li je „i“ manje od 3 ( **0 < 3** ) 🟢
- 2) uvjet je zadovoljen i ulazimo u petlju
- 3) izvršava se ispis
- 4) uvećavamo vrijednost varijable za jedan i sada „i“ ima vrijednost 1
- 5) provjera uvjeta da li je „i“ manje od 3 ( **1 < 3** ) 🟢
- 6) uvjet je zadovoljen i ulazimo u petlju
- 7) izvršava se ispis
- 8) uvećavamo vrijednost varijable za jedan i sada „i“ ima vrijednost 2

- 9) provjera uvjeta da li je „i“ manje od 3 ( $2 < 3$ ) 😊
- 10) uvjet je zadovoljen i ulazimo u petlju
- 11) izvršava se ispis
- 12) uvećavamo vrijednost varijable za jedan i sada „i“ ima vrijednost 3
- 13) provjera uvjeta da li je „i“ manje od 3 ( $3 < 3$ ) ❌
- 14) uvjet nije zadovoljen i ne ulazimo u petlju
- 15) izvršava se kôd iza while petlje



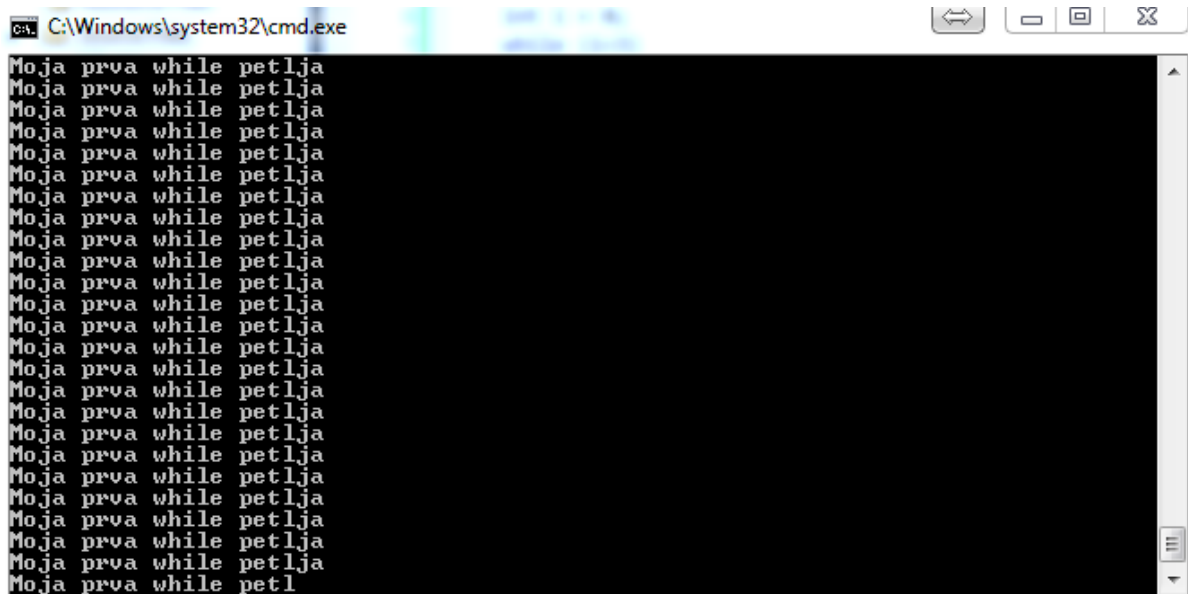
```
ca. C:\Windows\system32\cmd.exe
Moja prva while petlja
Moja prva while petlja
Moja prva while petlja
Press any key to continue . . .
```

Slika 24 - Rezultat nakon izvršenja kôda u primjeru 1

**Uočite** – na kraju kôda unutar tijela petlje se nalazi `i++`. Razmislite što bi se dogodilo da nismo postavili uvećanje varijable za 1 odnosno da nam je kôd ovako izgleda :

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

int main()
{
    int i = 0;
    while (i<3)
    {
        cout<<"Moja prva while petlja"<<endl;
    }
}
```

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\Windows\system32\cmd.exe'. The window contains a list of 25 lines of text, each line reading 'Moja prva while petlja'. The text is white on a black background. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Slika 25 - Rezultat nakon izvršenja kôda u primjeru 2 - beskonačna petlja

U konzoli se ista rečenica počela nekontrolirano ispisivati. Ovakav problem se zove **BESKONAČNA PETLJA**.

### e. Beskonačna petlja

Beskonačnu petlju je relativno lagano za uočiti i pri nešto složenijim aplikacijama. Kako joj i samo ime kaže, naredbe unutar tijela petlje će se izvršavati skoro beskonačno puta mnogo. Razlog ovome „skoro“ je zato što se teoretski neće beskonačno puta odvititi jer će u jednom trenutku se morati zaustaviti pošto svako računalo ima ograničene resurse koji bi se kad-tad popunili. Naravno, vrijeme potrebno da ovako malo kôda popuni svu vašu memoriju je ovisno o Vašem računalu ali još važnije, skoro je beskonačno.

**Koji je razlog ovome?** Odgovor je zapravo jednostavan. Pokrenuli smo petlju i rekli joj „dok god je vrijednost varijable „i“ manja od 3, ti izvršavaj kôd koji ti piše u tijelu petlje“. Svaki put kada se sve linije u tijelu izvrše ponovno se radi provjera da li je uvjet još uvijek zadovoljen. Kako mi ni na jedan način ne utječemo na vrijednost varijable „i“ ona ostaje nepromijenjena cijelo vrijeme odnosno njena vrijednost je zauvijek 0 (nula).

## f. do-while petlja

Već iz samog imena možete zaključiti kako je ova vrsta petlje slična prethodno objašnjenom „while“ petlji. *Sintaksa* im je poprilično slična uz jedan dodatak bloka kôda. U „do-while“ petlji prvo navodimo kôd koji želimo da se izvrši a u drugom dijelu navodimo uvjet koji treba biti zadovoljen. Kako znamo da se naš program izvodi od najgornje linije prema dolje možemo uočiti kako u „do-while“ petlji to znači da će prvo doći do dijela u kojem se opisuje koji kôd se treba izvesti tek nakon toga do uvjeta. Upravo to je svojstvo „do-while“ petlje i to je jedina petlja za koju kažemo da će se **sigurno izvršiti barem jednom**.

### Sintaksa :

```
do {  
kôd koji će se izvršiti  
}  
while ( uvjet );
```

**Primjetite** simbol točka-zarez na kraju retka u kojem se nalazi uvjet i pripazite da ne stvorite beskonačnu petlju.

### **Primjer 1 :**

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int i = 0;  
    do {  
        cout << "Moja prva do-while petlja"<<endl;  
        i++;  
    }  
    while ( i < 3 );  
  
    return 0;  
}
```

U ovom primjeru izradili smo aplikaciju koja će ispisivati rečenicu „Moja prva do-while petlja“ dok god je vrijednost varijable „i“ (koju smo prethodno deklarirali i postavili na vrijednost 0 (nula)) manja od 3 (tri). Svaki put kada je uvjet zadovoljen ispisat će se naša rečenica i vrijednost varijable „i“ će se uvećati za 1 (jedan).

### Detaljno objašnjenje :

- 1) Stvori varijablu imena „i“ tipa *integer* i postavi joj vrijednost na 0
- 2) izvrši dio koda koji se sastoji od :
  - a. ispis rečenice u konzolu
  - b. uvećaj varijablu „i“ za jedan (**i = 1**)
- 3) provjeri da li je vrijednost varijable „i“ manja od 3 (**1 < 3**)
- 4) izvrši dio koda koji se sastoji od :
  - a. ispis rečenice u konzolu
  - b. uvećaj varijablu „i“ za jedan (**i = 2**)

- 5) provjeri da li je vrijednost varijable „i“ manja od 3 ( $2 < 3$ )
- 6) izvrši dio koda koji se sastoji od :
  - a. ispis rečenice u konzolu
  - b. uvećaj varijablu „i“ za jedan ( $i = 3$ )
- 7) provjeri da li je vrijednost varijable „i“ manja od 3 ( $3 < 3$ )
- 8) prekini izvođenje kôda u ovoj petlji i nastavi dalje s izvođenjem programa

```

C:\Windows\system32\cmd.exe
Moja prva do-while petlja
Moja prva do-while petlja
Moja prva do-while petlja
Press any key to continue . . .
  
```

Slika 26 - Rezultat nakon izvršenja kôda u primjeru 1

### Primjer 2 :

```

int main()
{
    int i = 5;
    do {
        cout << "Moja prva do-while petlja "<<endl;
        i++;
    }
    while ( i < 3 );

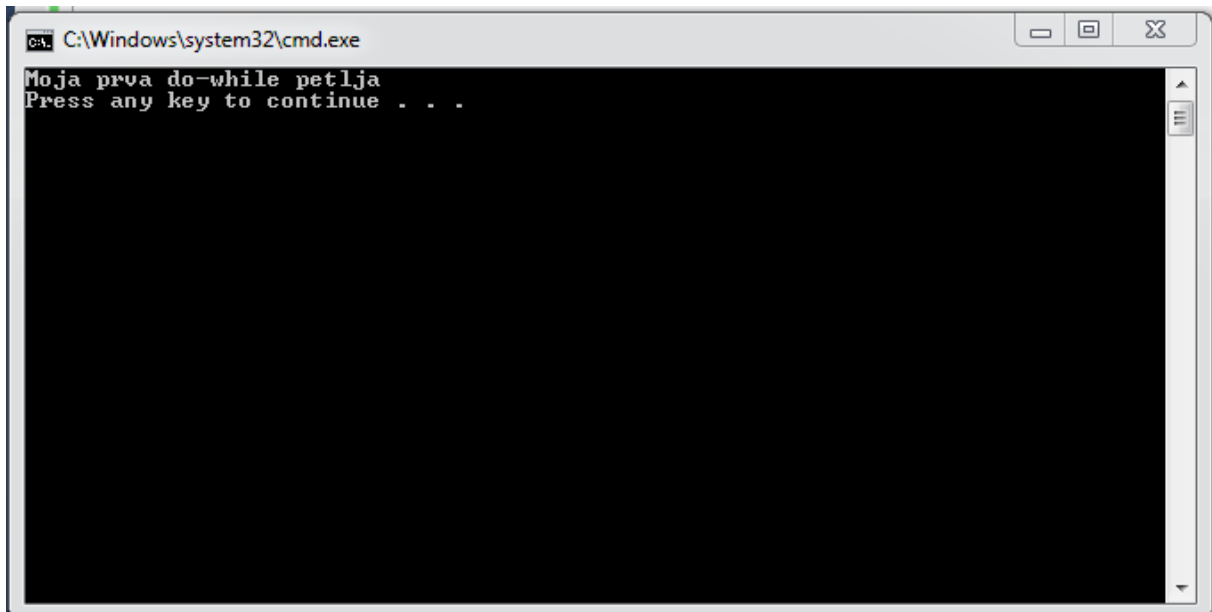
    return 0;
}
  
```

**Uočite** kako je vrijednost varijable „i“ u ovom primjeru iznosi 5 (pet) a da nam je uvjet za izvršavanje petlje da je vrijednost varijable „i“ manja od 3.

### Detaljno objašnjenje :

- 1) stvori varijablu tipa *integer* imena „i“ i dodijeli joj vrijednost 5
- 2) izvrši dio kôda koji se sastoji od :
  - a. ispiši rečenicu
  - b. uvećaj vrijednost varijable „i“ ( $i = 6$ )
- 3) provjeri da li je vrijednost varijable „i“ manja od 3 ( $6 < 3$ )
- 4) prekini izvršavanje petlje i nastavi s daljnjim izvođenjem programa

**Uočite** da iako odmah u početku uvjet petlje nije bio zadovoljen, svejedno se kod u tijelu petlje izvršio. Ovdje se lijepo može uočiti slijed kojim se „do-while“ petlja izvršava odnosno da se prvo izvršava kôd u tijelu petlje a zatim se radi provjera da li je uvjet zadovoljen.



Slika 27 - Rezultat izvršavanja kôda u primjeru 2

### Primjer 3 :

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

int main()
{
    int i = 3;
    do {
        cout << "Unesi cijeli broj veci od 0 a manji od 5"<<endl;
        cin>>i;
    }
    while ( i < 1 || i > 4 );

    return 0;
}
```

Uvjet smo postavili pomoću logičkog operatora „logički ili“. U ovom programu mi tražimo od korisnika da unese broj koji se nalazi između broja 0 i broja 5 (ne uključujući krajnje granice). Naš uvjet koji nam u ovom slučaju služi kao provjera da li je vrijednost varijable *i* manja od 1 ili je veća od 4. Morali smo postaviti brojeve 1 i 4 kako ne bismo napravili **logičku pogrešku**.

Da smo postavili uvjet da provjerava ako je uneseni broj manji od 0 tada bi to bila logička pogreška jer bi 0 (nula) ulazila kao jedno od mogućih zadovoljavajućih slučajeva. Kako u ovom slučaju radimo samo s cjelobrojnim vrijednostima, dovoljno je postaviti uvjet da uneseni broj mora biti manji od 1 (jedan) jer prvi manji broj je upravo 0 (nula) koja u ovom slučaju nije povoljna vrijednost. Isti princip vrijedi i za obrazloženje zašto smo upotrijebili usporedbu da unesena vrijednost mora biti veća od 4 (četiri) a ne od broja 5 (pet).



## g. Zadaci

- 1) Napraviti aplikaciju u koju korisnik upisuje brojeve sve dok ne unese malo slovo „k“ ili veliko slovo „K“. Sve brojeve koje korisnik unese treba zbrojiti i ispisati konačni rezultat.
- 2) Napraviti aplikaciju u koju korisnik upisuje cijele brojeve sve dok ne unese broj 0 (nula). Aplikacija treba ispisati koliko je uneseno parnih a koliko neparnih brojeva, zbroj parnih i zbroj neparnih brojeva te ukupan zbroj brojeva.
- 3) Izraditi aplikaciju koja ispisuje tablicu množenja za brojeve od 1 do 10 ali tako da u jednom redu bude zapisano najviše 5 brojeva.
- 4) Napraviti aplikaciju koja ispisuje tablicu množenja od broja koji korisnik navede do broja koji korisnik navede. Uz to aplikacija treba ispisati koliko se parnih brojeva nalazi u tablici za množenje a koliko neparnih
- 5) Napraviti aplikaciju koja ispisuje sve cijele brojeve od 1 do 999 i ispisuje koliko je prostih brojeva u tom rasponu.



## 16) Kontrole toka 2

Uz već spomenute načine kontroliranja toka našeg programa, postoji još jedna izuzetno korisna i često korištena naredba. Korištenje ove naredbe je primjenjivo ukoliko postoji nekoliko mogućih odgovora od strane korisnika. Primjerice, izradit ćemo aplikaciju za ocjenjivanje učenika. Ocjene koje se mogu pojaviti su 1,2,3,4,5 i još ćemo dodati ocjenu 0 (nula) koja će predstavljati oznaku da učenik nije pisao ispit ili je uhvaćen u prepisivanju pa se poništava ispit. Ovo bi zahtijevalo da izradimo 6 „if“-ova i za svaki od njih navodimo da varijabla mora biti točno neki broj te ovisno o tome da se izvrši neka operacija. Postoji naravno i lakši način a on se zove **switch**.

### a. switch

#### Sintaksa :

```
switch ( test )
{
    Slučaj 1 :
    {
        kôd koji će se izvršiti ako je ovaj uvjet broj 1 zadovoljen;
        break;
    }
    Slučaj 2 :
    {
        kôd koji će se izvršiti ako je ovaj uvjet broj 2 zadovoljen;
        break;
    }
    Slučaj 3 :
    {
        kôd koji će se izvršiti ako je ovaj uvjet broj 3 zadovoljen;
        break;
    }
    default :
    {
        kôd koji će se izvršiti ako niti jedan od gornjih uvjeta nije
        zadovoljen;
    }
}
```

### Primjer : 1

Korisnik treba unijeti ocjenu koju je dobio iz ispita. Uzet ćemo da je prosjek razreda na ispitu bio 3.22. Ako korisnik unese 0 (nula) prikazat ćemo mu obavijest da ispit nije uopće niti pisao. Ako upiše 1 onda ćemo mu ispisati poruku da nije položio. Ako unese 2 ispisat ćemo mu da je položio ali jedva. Ako upiše 3 dobit će obavijest da je prosječno napisao ispit. Ako upiše 4 ispisat ćemo mu da je napisao bolje od većine razreda a ako upiše 5 ispisat ćemo mu da je jedan od bistrijih u razredu. Ako unese broj manji od 0 (nula) ili veći od 5 (pet) znači da je pogrešno unio i ispisat ćemo mu poruku o pogrešnom unosu.

```
int main()
{
    int ocjena;
    cout << "Unesi ocjenu : ";
    cin >> ocjena;

    switch (ocjena)
    {
        case 0 : {
            cout << "Nisi niti pisao ispit ili si uhvacen u
varanju."<<endl;
            break; }

        case 1: {
            cout << "Bitno je sudjelovati."<<endl;
            break; }

        case 2: {
            cout << "Ajde nekako si položio ali jedva."<<endl;
            break; }

        case 3: {
            cout << "Prosjecno si rijesio ispit."<<endl;
            break; }

        case 4: {
            cout << "Nije lose, bolji si od vecine korisnika."<<endl;
            break; }

        case 5: {
            cout << "Svaka cast. Ti si jedan od bistrijih u
razredu."<<endl;
            break; }

        default: {
            cout << "Pa zar ni ovako jednostavan zadatak ne znas
napraviti kako treba???"<<endl;
            break; }

    }
    return 0;
}
```

### Detaljno objašnjenje :

Stvaramo varijablu imena „ocjena“ koja je tipa „*integer*“ te pitamo korisnika da unese ocjenu koju je dobio na ispitu nakon čega tu vrijednost spremamo u varijablu „ocjena“. Program dolazi do „switch“ naredbe kojoj smo kao uvjet za testiranje postavili vrijednost varijable „ocjena“. Ovisno koliko iznosi vrijednost varijable „ocjena“ izvršit će se taj slučaj odnosno ako je vrijednost manja od 0 (nula) ili veća od 5(pet) ispisat će se poruka o pogrešnom unosu.

**Uočite** kako u svakom od ponuđenih slučajeva iza naredbe za ispis poruke piše „**break**“. „Break“ naredba govori našem programu da izađe iz petlje odnosno u našem slučaju da prekine sa izvođenjem „switch“ naredbe. Da nismo upotrijebili naredbu „break“ ako bi korisnik unio broj 1 (jedan) ispisala bi se naredba navedena pod brojem jedan, ali i naredbe napisane u svim ostalim slučajevima. Ovu naredbu ćemo obraditi detaljnije uskoro pa će i njena uporaba imati više smisla.

### **Primjer 2:**

Zamislimo da vodimo kafić koji ima tri zaposlena konobara (jedan u svakoj smjeni i jedan u među-smjenama). Kako bi mogli voditi njihovu evidenciju o radu napravili smo svoju aplikaciju gdje se konobari upisuju kada dođu i odu na posao ali i koja sadrži njihove kontakt informacije kako bi uvijek mogli znati na koji broj mobitela ih možemo kontaktirati. U ovom primjeru obradit ćemo ispis informacija o našem konobaru. Gazda treba upisati broj pod kojim je upisan konobar nakon čega se ispisuju informacije o tom konobaru.

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

int main()
{
    int sifra;
    cout << "Unesi sifru konobara : "<<endl;
    cout << "  1 - Pero"<<endl;
    cout << "  2 - Marta"<<endl;
    cout << "  3 - Ivana"<<endl;
    cin>>sifra;
    switch(sifra)
    {
        case 1:
        {
            cout << "Ime : Pero"<<endl;
            cout << "Prezime : Peric"<<endl;
            cout << "Adresa : Ulica Vlatka Maceka 123b"<<endl;
            cout << "Kontakt : 091/234-5678"<<endl;
            break;
        }
        case 2:
        {
            cout << "Ime : Marta"<<endl;
            cout << "Prezime : Martic"<<endl;
            cout << "Adresa : Senjska 1"<<endl;
            cout << "Kontakt : 098/765-4321"<<endl;
            break;
        }
    }
}
```

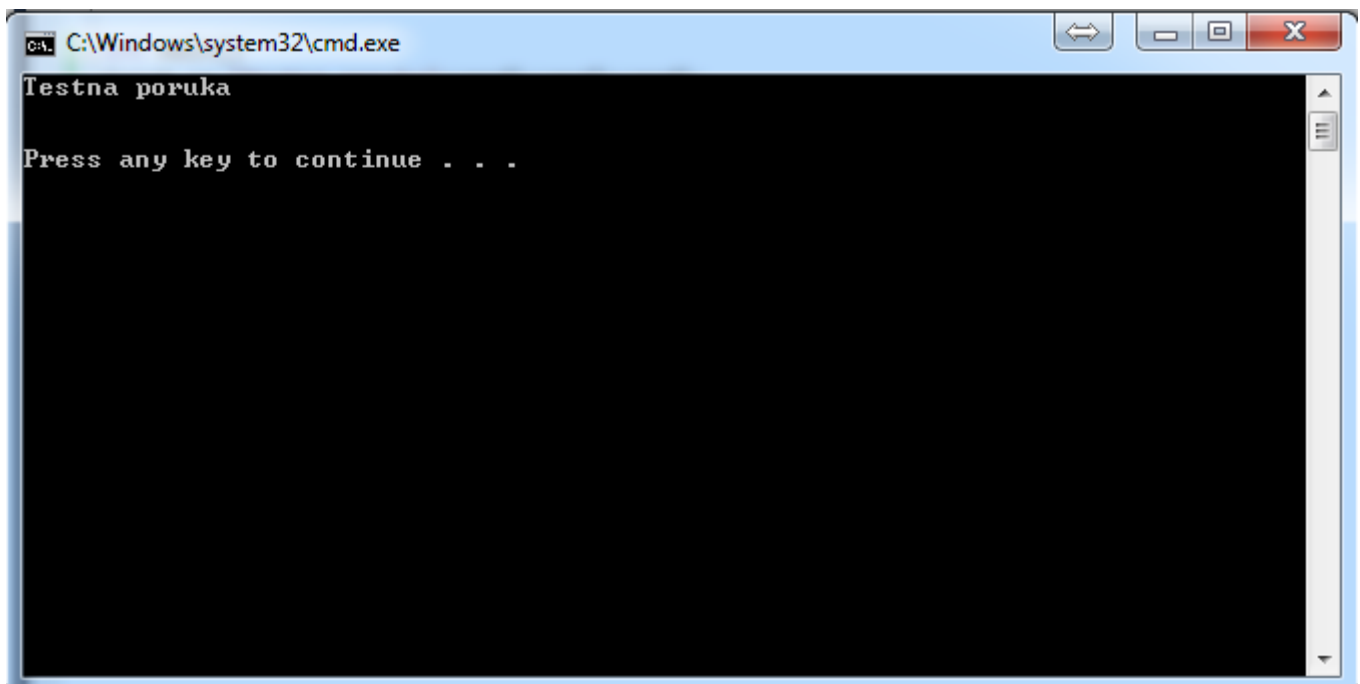
```
case 3:
{
    cout << "Ime : Ivana"<<endl;
    cout << "Prezime : Ivanic"<<endl;
    cout << "Adresa : Ulica Vladimira Nazora 14c"<<endl;
    cout << "Kontakt : 095/678-910"<<endl;
    break;
}
default :
{
    cout << "Ne postoji konobar pod tom sifrom"<<endl;
    break;
}
}
return 0;
}
```

## 17) Formatiranje ispisa - osnove

Do sada smo ispisivali neke informacije no nismo obraćali pažnju na uređivanje tog ispisa (koliko je moguće u konzolnoj aplikaciji. U ovom poglavlju ćemo obraditi neke osnovne načine formatiranja ispisa kako bi naše hvale vrijedne poruke korisniku izgledale koliko-toliko oku ugodno. Ukupno ćemo obraditi dva poglavlja formatiranja ispisa no radi potrebnih dodatnih znanja kod ostalih načina formatiranja, ne možemo ih trenutno obraditi. Poznavanje funkcija je neophodno kako bi se ta formatiranja obradila.

### a. <<endl;

Naredba „endl“ koju smo već koristili ali i spomenuli nam omogućuje prelazak u novi red. Moguće je dodavati više ovakvih naredbi želimo li preskočiti više redova. Primjerice `cout << "Testna poruka"<<endl<<endl<<endl;` će rezultirati ispisom „Testna poruka“ i preskakanjem u prvi novi red pa u drugi pa u treći novi red.



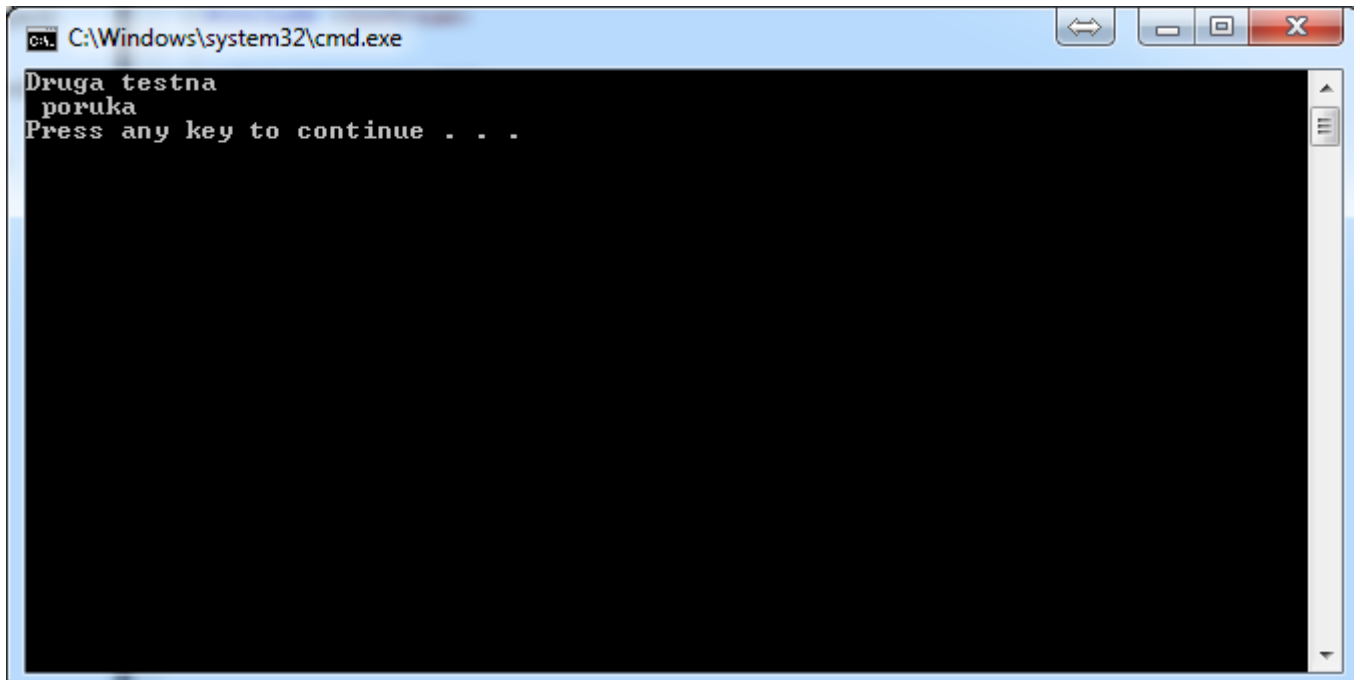
```
C:\Windows\system32\cmd.exe
Testna poruka

Press any key to continue . . .
```

Slika 28 - Formatiranje ispisa pomoću višestruke "endl" naredbe

## b. `\n`

Ova naredba je skoro pa identična „endl“ naredbi odnosno također nam omogućuje prelazak u novi red. Glavna razlika je što se „endl“ koristi na kraju linije kôda dok nam „\n“ omogućuje prelazak u novi red usred neke rečenice. Koristi se unutar samog teksta koji želimo ispisati. Primjerice `cout << "Druga testna \n poruka"<<endl;` će rezultirati ovakvim ispisom :



```
ca. C:\Windows\system32\cmd.exe
Druga testna
poruka
Press any key to continue . . .
```

Slika 29 - Formatiranje ispisa pomoću „\n“

### Primjer 1 :

Ovdje ćemo iskoristiti dio kôda koji smo već obradili kada smo tražili omogućili gazdi kafića prikaz informacija o zaposlenicima u njegovom kafiću.

Original : (poglavlje 10, primjer 1)

```
int sifra;
    cout << "Unesi sifru konobara : "<<endl;
    cout << "    1 - Pero"<<endl;
    cout << "    2 - Marta"<<endl;
    cout << "    3 - Ivana"<<endl;
    cin>>sifra;
```

Kako bi se izbjeglo nepotrebno višestruko korištenje „cout“ naredbe koristit ćemo jednostavnije rješenje korištenjem „\n“ simbola kako bi razlomili tekst na više dijelova a naredbu „cout“ iskoristili samo jednom.

Korištenje „\n“

```
int sifra;
    cout << "Unesi sifru konobara : ";
    cout << "\n    1 - Pero \n    2 - Marta \n    3 - Ivana"<<endl;
    cin>>sifra;
```

### c. Zadaci

- 1) Napraviti aplikaciju koja ispisuje u konzolu oblik kvadrata. Duljine stranica ćemo mjeriti u simbolu „\*“ (može i neki drugi simbol). Korisnik treba unijeti koliko da stranice budu dugačke
- 2) Napraviti aplikaciju koja u konzolu ispisuje simbole u obliku pravokutnika. Duljinu stranica mjerimo u simbolima („\*“) a korisnik unese vrijednost duljine i visine stranica.
- 3) Napravite aplikaciju koja prikazuje oblik jednakokravnog trokuta gdje korisnik određuje visinu trokuta.
- 4) Napravite oblik simbola „kara“. Korisnik treba unijeti dimenzije, tako da unese koliko se radova treba iscrtati iznad središnje linije.

## 18) Polja (nizovi)

### a. uvod

U programiranju će te se često sresti s uporabom polja odnosno nizova. Korištenjem polja možemo na smislen način grupirati više podataka istog tipa. Primjerice, želimo li izračunati ali i pohraniti ocjene iz ispita učenika nekog razreda jedan od načina je korištenjem polja. Polja možete zamisliti kao jednu vreću u kojoj se nalaze čokolade ali samo čokolade iste vrste. Tako možemo imati dvije vreće, jednu za crnu čokoladu a drugu za bijelu čokoladu, a unutar svake vreće ćemo staviti podvrste čokolada. U vreću s crnom čokoladom možemo staviti čokoladu s lješnjacima i s orasima dok u onu s bijelim čokoladama možemo staviti čisto mliječnu čokoladu i čokoladu s kokosom. Svaka od tih čokolada je element svog polja odnosno element svoje vreće i svaka ima svoj redni broj odnosno svoj indeks.

Postoje jednodimenzionalna polja i dvodimenzionalna polja a mi ćemo obraditi jednodimenzionalna polja koja ulaze u domenu uvoda u programiranje.

Polja se sastoje

#### Sintaksa :

```
tip_podatka ime_polja [ veličina_niza ];
```

#### **primjer polja i pojašnjenje :**

```
int moguće_ocjene_iz_ispita [ 5 ];
```

Sada smo deklarirali polje realnih brojeva čije je ime „moguće\_ocjene\_iz\_ispita“ i koje u sebi može sadržavati 5 elemenata. To polje će nam služiti za upis mogućih ocjena iz ispita kako i samo ime kaže (vrijednosti od 1 do 5 uključujući granične brojeve). Sada se postavlja pitanje kako se pridjeljuju vrijednosti polju. Svako polje, ovisno o veličini koju mu unaprijed odredimo ima određeni broj elemenata. U našem slučaju taj broj je 5. Svakom elementu polja se pristupa (čitanje vrijednosti nekog elementa ili upis vrijednost u neki element) pomoću indeksa tog elementa. Želimo li vizualno si predočiti naše polje to će izgledati ovako (napomena – brojke upisane unutar našeg polja predstavljaju indekse svakog pojedinog elementa).

|   |   |   |   |   |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|

Kao što vidite, naš **prvi element** nema indeks polja 1 (jedan), već ima **indeks polja 0 (nula)**. U programerskom odnosno računalnom svijetu, brojevi počinju od broja 0 (nula) a ne od broja 1 (jedan) kako su ljudi naviknuli brojati. Vjerojatno će ovakav način brojanja biti pomalo zbunjujući na početku no nakon par vježbi će zasigurno ući pod normalno shvaćanje.



## b. Čitanje i pridruživanje vrijednosti elementu polja

Pogledajmo *sintaksu* za upisivanje vrijednosti odnosno čitanje vrijednosti iz elemenata polja :

### Pridruživanje vrijednosti elementu polja :

**Sintaksa :**

```
ime_polja [ indeks_pojednog_elementa ] = vrijednost_koju_pridružujemo;
```

U našem slučaju će to izgledati ovako :

```
moguće_ocjene_iz_ispita [ 0 ] = 1;  
moguće_ocjene_iz_ispita [ 1 ] = 2;  
moguće_ocjene_iz_ispita [ 2 ] = 3;  
moguće_ocjene_iz_ispita [ 3 ] = 4;  
moguće_ocjene_iz_ispita [ 4 ] = 5;
```

**Pojašnjenje :**

U polju realnih brojeva koje smo nazvali „moguće\_ocjene\_iz\_ispita“ naš prvi element koji ima indeks 0 (nula) sada ima vrijednost 1 (jedan). Drugi element s indeksom 1 (jedan) ima vrijednost 2 (dva). Proces se nastavlja do zadnjeg elementa polja.

### Čitanje vrijednosti elementa polja :

**Sintaksa :**

```
ime_polja [ indeks_pojednog_elementa ];
```

Želimo li ispisati vrijednost nekog elementa, u našem primjeru će to ovako izgledati :

```
cout << moguće_ocjene_iz_ispita [ 0 ] << endl;  
cout << moguće_ocjene_iz_ispita [ 1 ] << endl;  
cout << moguće_ocjene_iz_ispita [ 2 ] << endl;  
cout << moguće_ocjene_iz_ispita [ 3 ] << endl;  
cout << moguće_ocjene_iz_ispita [ 4 ] << endl;
```

### c. Korištenje konstantnih vrijednosti varijabli za deklaraciju velične polja

U prijašnjem primjeru koristili smo fiksno postavljenu vrijednost polja. Ovakav pristup može biti dobar ukoliko smo u potpunosti sigurni da se veličina našeg polja neće s vremenom trebati mijenjati ali čak i tada je bolja solucija koristiti neku *integer* varijablu koja će sadržavati vrijednost koliko želimo da naše polje bude veliko odnosno koliko elemenata da sadržava. Osim bolje fleksibilnosti ukoliko dođe do promjene veličine polja, posebno je korisno imati takvu varijablu za brže čitanje iz cijelog polja koristeći „for“ petlju koju ćemo ubrzo obraditi.

*Sintaksa* za stvaranje polja gdje umjesto broja koji deklarira veličinu polja koristimo varijablu je skoro pa identična „običnoj“ deklaraciji polja. Razlika je jedino što umjesto upisivanja broja upisujemo varijablu kojoj smo prethodno dodijelili neku vrijednost. Gledano na prijašnji primjer, to bi izveli ovako :

```
const int velicina = 5;  
int moguće_ocjene_iz_ispita [ velicina ];
```

**Uočite** korištenje ključne riječi „*const*“. Konstantne tipove varijabli smo spomenuli na početku no prisjetimo se – to su varijable koje imaju sve osobine „običnih“ varijabli no razlikuju se što se njihova vrijednost može samo jednom postaviti. Promjene te vrijednosti u daljnjem dijelu programa nisu moguće o čemu ćemo dobiti i poruku da je počinjena *sintaksna* pogreška. Kada želimo varijablu postaviti kao pokazatelj koliko će elemenata naše polje sadržavati, obavezno je koristiti ključno riječ „*const*“ jer i sama veličina polja je konstantna. Polje koje jednom stvorimo s određenim brojem elemenata nije moguće naknadno proširivati niti smanjivati. U memoriji će program zauzeti točno određenu veličinu potrebnu za to polje. Kako dakle niti polja nemaju promjenjive vrijednosti broja elemenata, obavezno je korištenje konstantnih varijabli koje moraju biti cjelobrojnog tipa.

Ukoliko se u programu koristi više polja koja će uvijek imati međusobno isti broj elemenata (ako imamo 3 polja i svako ima 5 elemenata i obavezno je da sva tri polja uvijek imaju isti broj elemenata) tada je povoljnija opcija koristiti varijablu koja opisuje veličinu polja jer će se promjena u veličini polja morati izvršiti samo na jednoj lokaciji (u usporedbi da ne koristimo ovaj pristup, morali bismo na tri mjesta mijenjati vrijednosti čime si ujedno povećavamo mogućnost pogreške).

#### d. for petlja za brže čitanje elemenata polja


Zamislamo li polje koje ima 10 ili čak više elemenata i da svaki element moramo „ručno“ čitati lako je za shvatiti da takav pristup nema smisla. Ovdje nam pomaže korištenje „for“ petlje. Kako for petlja kreće brojati od nekog broja do nekog broja mi možemo napraviti petlju koja će brojati od 0 (nula) do 5 (pet) kako bi nam ispisala sve vrijednosti nekog polja. Važno je zapamtiti da indeksi elemenata u polju kreću s indeksom 0 (nula) a ne od broja 1 (jedan).

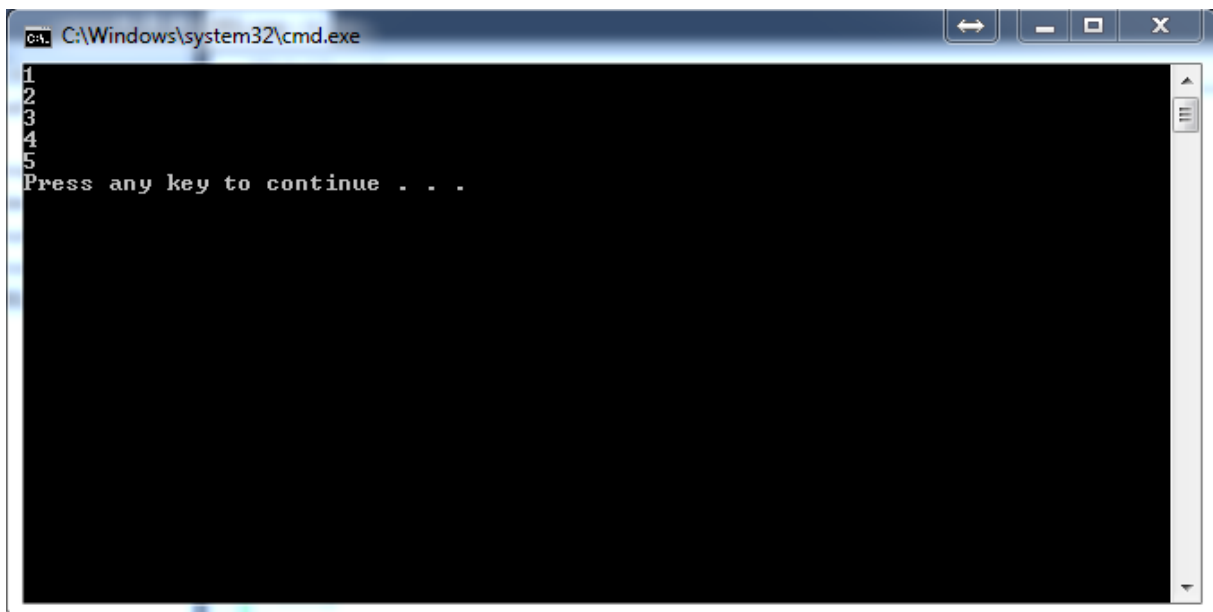
##### Primjer za čitanje iz polja s 5 elemenata :

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

##### Detaljno objašnjenje primjera :

- 1) Stvaramo „for“ petlju s početkom od broja 0 (nula) do broja 5 (ne uključujući jer smo rekli da mora biti striktno manje od 5. Mogli smo napisati i „<=4“ što bi bilo potpuno identično), uz povećanje nakon svakog koraka (jedan korak se smatra kada petlje prođe kroz sav kôd napisan u njenom tijelu).
- 2) Provjera da li je vrijednost varijable „i“ manja od 5 **( 0 < 5 )** 😊
- 3) Ispiši na ekran vrijednost elementa koji se nalazi u polju imena „moguće\_ocjene\_iz\_ispita“ na indeksu čiju vrijednost predstavlja vrijednost varijable „i“ što je u ovom slučaju **0**. Nakon ispisa poruke preskoči u novi red.
- 4) Uvećaj vrijednost varijable „i“ za jedan **( i = 1 )**
- 5) Provjera da li je vrijednost varijable „i“ manja od 5 **( 1 < 5 )** 😊
- 6) Ispiši na ekran vrijednost elementa koji se nalazi u polju imena „moguće\_ocjene\_iz\_ispita“ na indeksu čiju vrijednost predstavlja vrijednost varijable „i“ što je u ovom slučaju **1**. Nakon ispisa poruke preskoči u novi red.
- 7) Uvećaj vrijednost varijable „i“ za jedan **( i = 2 )**
- 8) Provjera da li je vrijednost varijable „i“ manja od 5 **( 2 < 5 )** 😊
- 9) Ispiši na ekran vrijednost elementa koji se nalazi u polju imena „moguće\_ocjene\_iz\_ispita“ na indeksu čiju vrijednost predstavlja vrijednost varijable „i“ što je u ovom slučaju **2**. Nakon ispisa poruke preskoči u novi red.
- 10) Uvećaj vrijednost varijable „i“ za jedan **( i = 3 )**
- 11) Provjera da li je vrijednost varijable „i“ manja od 5 **( 3 < 5 )** 😊
- 12) Ispiši na ekran vrijednost elementa koji se nalazi u polju imena „moguće\_ocjene\_iz\_ispita“ na indeksu čiju vrijednost predstavlja vrijednost varijable „i“ što je u ovom slučaju **3**. Nakon ispisa poruke preskoči u novi red.
- 13) Uvećaj vrijednost varijable „i“ za jedan **( i = 4 )**
- 14) Provjera da li je vrijednost varijable „i“ manja od 5 **( 4 < 5 )** 😊

- 15) Ispiši na ekran vrijednost elementa koji se nalazi u polju imena „moguće\_ocjene\_iz\_ispita“ na indeksu čiju vrijednost predstavlja vrijednost varijable „i“ što je u ovom slučaju 4. Nakon ispisa poruke preskoči u novi red.
- 16) Uvećaj vrijednost varijable „i“ za jedan (  $i = 5$  )
- 17) Provjera da li je vrijednost varijable „i“ manja od 5 (  $5 < 5$  ) 
- 18) Izadi iz „for“ petlje (prekini njeno izvršavanje)
- 19) Nastavi s daljnjim izvođenjem iz programa



```

C:\Windows\system32\cmd.exe
1
2
3
4
5
Press any key to continue . . .

```

Slika 30 - Rezultat nakon izvršavanja kôda u primjeru broj 1

### **VAŽNO – PRIMJER 2**

Još ispravniji način gornjeg kôda bi bio da smo unutar „for“ petlje drugačije postavili vrijednost do koje će se petlja izvršavati. Kako smo prije stvaranja polja imali konstantnu varijablu imena „velicina“ u koju smo pohranili vrijednost o veličini našeg polja, mogli smo (i trebali) ovako napisati uvjet „for“ petlje :

```

for ( int i = 0; i<velicina;i++)
{
    cout << moguće_ocjene_iz_ispita [ i ] << endl;
}

```

Rezultat i opis koraka je identičan ali je ovo puno kvalitetniji pristup. Dođe li do promjene veličine polja sada smo se osigurali da će se i „for“ petlja pravilno izvršiti odnosno da će ispisati sve vrijednosti našeg polja. Primjerice, proširimo li polje na 6 elemenata tako da element s indeksom 0 (nula) ima vrijednost 0 (nula) što simbolizira da učenik nije pisao ispit, a ostavimo staru „for“ petlju, tada bi nam se ispisale vrijednosti mogućih ocjena 0,1,2,3,4 ali ocjena 5 se ne bi ispisala jer se ona sada nalazi na indeksu broj 5 do kojeg „for“ petlja neće doći.

## e. for petlja za brže upisivanje u polje

Opisivanje kako nam „for“ petlja može pomoći i pri upisu podataka ćemo nastaviti na gornji primjer o ocjenama na ispitu. Želimo li unijeti moguće ocjene u naše polje koje se sastoji od 5 elemenata koristit ćemo „for“ petlju. Ovaj put ćemo odmah krenuti s ispravnim načinom korištenja pa ćemo unutar uvjeta „for“ petlje umjesto upisivanja broja do koliko da „for“ petlja broji koristiti varijablu „velicina“ koja je u našem slučaju postavljena na vrijednost 5.

Dođe li do naknadne promjene u programu, ovakvim korištenjem smo si osigurali točnost prilikom stvaranja polja s ispravnim brojem elemenata, ispravnost prilikom upisa podataka i ispravnost prilikom ispisa podataka kao i daljnjom obradom tih podataka.

### Primjer :

```
for (int i=0; i<velicina; i++)
{
    cout << "Unesi ocjenu :";
    cin>>moguće_ocjene_iz_ispita [i];
}
```

### Detaljno objašnjenje :

- 1) stvaramo „for“ petlju s brojanjem od **0** (nula) do vrijednosti varijable „**velicina**“ koja je u ovom slučaju postavljena na broj 5 uz povećanje za 1 za svaki korak.
- 2) Provjera vrijednosti cjelobrojne varijable „i“ da li je manja od vrijednosti cjelobrojne varijable „**velicina**“ (**0 < 5**) 😊
- 3) Ispisujemo poruku „Unesi ocjenu :“
- 4) Broj koji je korisnik unio spremamo u element polja „moguće\_ocjene\_iz\_ispita“ koji se nalazi na indeksu „i“ odnosno sada je to indeks **0** (nula).
- 5) Uvećaj vrijednost varijable „i“ za jedan pa sada iznosi **1** (**i = 1**)
- 6) Provjera vrijednosti cjelobrojne varijable „i“ da li je manja od vrijednosti cjelobrojne varijable „**velicina**“ (**1 < 5**) 😊
- 7) Ispisujemo poruku „Unesi ocjenu :“
- 8) Broj koji je korisnik unio spremamo u element polja „moguće\_ocjene\_iz\_ispita“ koji se nalazi na indeksu „i“ odnosno sada je to indeks **1** (jedan).
- 9) Uvećaj vrijednost varijable „i“ za jedan pa sada iznosi **2** (**i = 2**)
- 10) Provjera vrijednosti cjelobrojne varijable „i“ da li je manja od vrijednosti cjelobrojne varijable „**velicina**“ (**2 < 5**) 😊
- 11) Ispisujemo poruku „Unesi ocjenu :“
- 12) Broj koji je korisnik unio spremamo u element polja „moguće\_ocjene\_iz\_ispita“ koji se nalazi na indeksu „i“ odnosno sada je to indeks **2** (dva).
- 13) Uvećaj vrijednost varijable „i“ za jedan pa sada iznosi **3** (**i = 3**)
- 14) Provjera vrijednosti cjelobrojne varijable „i“ da li je manja od vrijednosti cjelobrojne varijable „**velicina**“ (**3 < 5**) 😊

- 15) Ispisujemo poruku „Unesi ocjenu :“
- 16) Broj koji je korisnik unio spremamo u element polja „moguće\_ocjene\_iz\_ispita koji se nalazi na indeksu „i“ odnosno sada je to indeks **3** (tri).
- 17) Uvećaj vrijednost varijable „i“ za jedan pa sada iznosi **4 ( i = 4 )**
- 18) Provjera vrijednosti cjelobrojne varijable „i“ da li je manja od vrijednosti cjelobrojne varijable „velicina“ **( 4 < 5 )** 😊
- 19) Ispisujemo poruku „Unesi ocjenu :“
- 20) Broj koji je korisnik unio spremamo u element polja „moguće\_ocjene\_iz\_ispita koji se nalazi na indeksu „i“ odnosno sada je to indeks **4** (četiri).
- 21) Uvećaj vrijednost varijable „i“ za jedan pa sada iznosi **5 ( i = 5 )**
- 22) Provjera vrijednosti cjelobrojne varijable „i“ da li je manja od vrijednosti cjelobrojne varijable „velicina“ **( 5 < 5 )** ❌
- 23) Prekini izvršavanje petlja
- 24) Nastavi s izvršavanjem ostalog kôda programa

#### f. Objedinjen unos i ispis vrijednosti elemenata polja pomoću „for“ petlje

Iskoristimo li gore napisane kôdove za upis i čitanje vrijednosti elemenata polja možemo dobiti automatiziranu aplikaciju za upis brojeva u polje i ispis tih vrijednosti.

```
int main()
{
    const int velicina = 5;
    int moguće_ocjene_iz_ispita [ velicina ];

    for ( int i = 0; i < velicina; i++ )
    {
        cout << "Unesi ocjenu :";
        cin>>moguće_ocjene_iz_ispita [ i ];
    }

    for ( int i = 0; i<velicina;i++)
    {
        cout << moguće_ocjene_iz_ispita[ i ] << endl;
    }

    return 0;
}
```

## g. Zadaci

- 1) Kreirati polje *integers* koje ima 10 elemenata. U svaki element treba upisati broj koji korisnik unese ali samo ako je broj pozitivan. Nakon unosa svih elemenata aplikacija treba provjeriti ima li, i ako ima koliko se nalazi prostih brojeva u tom polju te treba provjeriti da li je zbroj svih brojeva paran ili neparan. Ako je zbroj paran treba ispitati da li je zbroj prost broj.
- 2) Napraviti polje realnih brojeva od 10 elemenata. Korisnik treba unijeti vrijednosti za sve elemente. Aplikacija po završetku unosa korisnika ispisuje koji ne najmanji broj unesen a koji najveći i eventualno koliko je puta taj broj unesen u polje.

## 19) Stringovi – uvod

### a. Uvod – što je *string* tip podatka

Sa „*string*“ tipom podatka smo se susreli već iako toga niste bili niti svjesni. Sve što se smatra tekстом je zapravo *string* a mi smo koristili *string*ove kako bi ispisali neku poruku korisniku. Vrijednost varijable koja je tipa *string* piše se unutar dvostrukih navodnika (kako smo do sada to i radili).

**Sintaksa :**

```
string ime_varijable = „neka_vrijednost“;
```

### b. Stringovi NISU primitivni tipovi podataka

Da bi mogli uopće koristiti *string*ove (osim na način na koji smo ih sada koristili tj. samo za ispis unaprijed definiranih poruka) potrebno je uključiti zasebnu biblioteku. Razlog tome je što *string* ne ulazi u kategoriju osnovnih odnosno primitivnih tipova podataka. *String* je sastavljen od simbola (karaktera) koje smo već upoznali (engl. *character*) i koji jesu primarni tip podatka. Njihova opsežnija izvedba su *string*ovi.

Zamislite to ovako; slova poput „a,b,c,d,e i sl.“ su primarni tipovi podataka. Korištenjem primarnih slova možemo složiti slova poput „nj, lj, dž“ koja su izvedenice nastale korištenjem dva slova. Korištenjem slova općenito dolazimo do stvaranja riječi koje su sastavljene od slova odnosno riječ je složeni tip podatka koji je sastavljen od više vrijednosti koje su primarni tipovi podataka.

*String*ovi ujedno spadaju pod objektno orijentirano programiranje ali područje objektno orijentiranog programiranja je daleko izvan domene koju ova skripta obrađuje. **Važno** je jedino zapamtiti kako se nad *string*ovima mogu vršiti razne operacije koje ćemo upoznati s vremenom i da ih je potrebno posebno uključiti u naš projekt.

**Sintaksa za uključivanje rada sa *string*ovima :**

```
#include <string>
```

### c. Ispis i učitavanje *string*ova

Upoznavanjem *string*ova, mogućnosti naših aplikacija se višestruko povećavaju. Više ne moramo korisniku nuditi točno određene mogućnosti već mu možemo pustiti slobodu odabira. Prisjetimo se primjera gdje je gazda kafića koristio aplikaciju kako bi vidio osnovne informacije o svojim konobarima. U prijašnjem slučaju, morali smo ručno upisivati svakog konobara i registrirati ga pod rednim brojem. Zatim je gazda trebao upisati broj pod kojim se nalazi neki konobar i onda bi mu se izbacile informacije o tom konobaru.

Korištenjem *string*ova možemo napraviti aplikaciju koja će prikazivati te iste informacije ali na drugačiji način. Ponudit ćemo gazdi samo da upiše ime konobara, a naša će aplikacija prepoznajući



ime konobara ispisivati vrijednosti o tom konobaru. **Napomena** – kako bi se lakše nadovezali na prijašnju verziju naša aplikacija će ovisno o upisanom imenu pridijeliti neku vrijednost odnosno šifru u varijablu „sifra“ koja je tipa *integer*.

**Primjer :**

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

int main()
{
    string ime;
    cout << "Unesi ime konobara : ";
    cin>> ime;

    int sifra;
    if ( ime == "pero" )
    { sifra = 1; }
    else if ( ime == "marta" )
    { sifra = 2; }
    else if ( ime == "ivana" )
    { sifra = 3; }
    else
    { sifra = 0; }

    switch(sifra) {
    case 1:
        {   cout << "Ime : Pero"<<endl;
            cout << "Prezime : Peric"<<endl;
            cout << "Adresa : Ulica Vlatka Maceka 123b"<<endl;
            cout << "Kontakt : 091/234-5678"<<endl;
            break;
        }
    case 2:
        {   cout << "Ime : Marta"<<endl;
            cout << "Prezime : Martic"<<endl;
            cout << "Adresa : Senjska 1"<<endl;
            cout << "Kontakt : 098/765-4321"<<endl;
            break;
        }
    case 3:
        {   cout << "Ime : Ivana"<<endl;
            cout << "Prezime : Ivanic"<<endl;
            cout << "Adresa : Ulica Vladimira Nazora 14c"<<endl;
            cout << "Kontakt : 095/678-910"<<endl;
            break;
        }
    default :
        {   cout << "Ne postoji konobar pod tom sifrom"<<endl;
            break;
        }
    }
    return 0; }
```

### Objašenje primjera :

Stvorili smo varijablu tipa „string“ i nazvali smo je „ime“. Korisnika (gazdu) pitamo da unese ime konobara čije informacije želi vidjeti i tu vrijednost spremimo u varijablu „ime“. Ukoliko gazda upiše vrijednost „pero“ to, je kao da smo mi napisali `string ime = „pero“;`. Nakon što je vrijednost uspješno spremljena dolazimo do „if“ odnosno „if-else“ dijela u kojem ovisno o imenu konobara koje je gazda unio, varijabli „sifra“ pridjeljujemo određenu vrijednost a ako nije unio niti jedno od pretpostavljenih imena, varijabla „sifra“ će poprimiti vrijednost 0 (nula). Ako je gazda za ime konobara unio ime „pero“, tada će varijabla „sifra“ poprimiti vrijednost 1 (jedan). Napokon, dolazimo do „switch“ naredbe koja je identična kao i zadnji put kada smo radili ovakvu aplikaciju pa tu nema ništa novo za objasniti.

### Napomena :

Umjesto korištenja „switch“ naredbe mogli smo koristiti i „if“ uvjete te ovisno o imenu koje unese gazda, u konzolu ispisati odgovarajuće podatke.

## d. Zadatci

- 1) Izraditi aplikaciju koja će pitati korisnika da unese imena svojih pet dobrih prijatelja. Te vrijednosti treba pohraniti u polje i zatim ispisati poruku korisniku pod kojim indeksom je spremljena koja osoba.
- 2) Korisnik treba definirati koliko riječi želi unijeti (da je manje od 20). Svaku zasebnu riječ spremamo u pojedini element polja i nakon toga korisniku ispisujemo kako bi izgledala pjesma da je koristio odabrane riječi.
- 3) Zahtijevamo od korisnika da unese podatke o imenu i spolu za 10 osoba. Aplikacija treba ispisati koliko ima muških a koliko ženskih osoba (za muški spol treba unijeti oznaku „m“ a za ženski „f“ i sortirati ih po spolovima. Imena ćemo spremati u jedno polje a spolove u drugo polje. Kada se unesu sve vrijednosti trebat ćemo proći kroz polja i prvo ispisati recimo samo imena ženskih osoba a zatim samo muških osoba.

## 20) Funkcije – uvod

### a. Što su funkcije i koja je njihova uloga

**Napomena** – u uvodnom poglavlju funkcija je za očekivati da će većina navedenog biti nejasna no to je normalno. Ovaj uvod predstavlja samo teorijsko i površinsko upoznavanje s funkcijama a u kasnijim poglavljima se detaljnije obrađuju i tada će se nejasnoće razrješavati.

Funkcije su jedno veliko poglavlje i njihova primjena je od sasvim jednostavnih aplikacija do krajnje kompleksnih aplikacija i računalnih igara. Funkcija u programiranju predstavlja najčešće skup radnji. Uzmemo li za primjer da imamo robota koji može hodati samo u onom smjeru u kojem je okrenut možemo reći da je svaki njegov korak funkcija. Taj korak odnosno funkcija se sastoji od niza drugih naredbi poput „pomakni lijevu nogu naprijed“ i zatim „pomakni desnu nogu naprijed“. Uzmemo li da naš robot ima mogućnost samo okretanja u lijevu stranu možemo napraviti posebnu funkciju da se naš robot može okrenuti i u desnu stranu. Umjesto da tri puta kažemo robotu da se okrene u lijevo, napraviti ćemo funkciju koja će u sebi sadržavati tri naredbe okretanja u lijevo. Na ovaj način ćemo samo koristiti našu funkciju u kôdu kada god to bude potrebno i morat ćemo napisati samo jednu liniju kôda naspram tri linije koliko bi zauzimalo da smo robotu morali reći da se tri puta okrene u lijevu stranu.

Ako ćemo trebati 10 puta u našem kôdu raditi zbroj dvije varijable čije će se vrijednosti mijenjati možemo svaki put pisati „zbroj = varijabla\_A + varijabla\_B“ ali možemo i napraviti gotovu funkciju koja će to raditi pa će i sam kod izgledati preglednije.

Funkcije dakle možemo definirati kao skup naredbi koje će kroz nekoliko operacija nama vratiti određeni rezultat.

#### **Sintaksa :**

```
tip_podatka_koji_funkcija_vraća ime_funkcij (parametri_ako_ih_ima)
{
  Kôd koji se izvršava
  return naredba (osim u slučaju ako je tip podatka „void“)
}
```

Funkcije se stvaraju izvan naše glavne funkcije odnosno „main“ funkcije. Unutar „main“ funkcije se naše funkcije pozivaju. Ovim načinom je osigurana bolja čitljivost kôda. Funkcije je moguće smjestiti ispred „main“ funkcije i iza „main“ funkcije. Ukoliko funkciju deklariramo ispred „main“ funkcije ona će odmah biti spremna za korištenje. Ukoliko funkciju deklariramo iza „main“ funkcije tada ćemo morati koristiti **prototip funkcije**. Prototip funkcije je prvi red funkcije (tip podatka koji vraća funkcija, ime funkcije, parametri) koji se stavlja prije „main“ funkcije. Razlog tome je što prilikom izvođenja programa, kada *compiler* dođe do dijela gdje pozivamo neku funkciju koju smo napisali iza „main“ funkcije, *compiler* neće znati što želim od njega jer se programi izvode odozgora prema dolje i u trenutku kada naiđe na našu funkciju on se još nije susreo s njom pa time niti ne zna što mu pokušavamo reći. Korištenjem prototipa funkcije mi *complieru* govorimo da će se negdje u našem kôdu pojaviti funkcija određenog imena i da smo je deklarirali negdje iza „man“ funkcije. Kada *complier* u našem kôdu naiđe na ime funkcije koju smo deklarirali iza „main“ funkcije i postavili joj prototip, *complier* će znati da je ta funkcija negdje i zapisana te će ići potražiti i naravno upotrijebiti.

## b. Korištenje prototipa funkcije

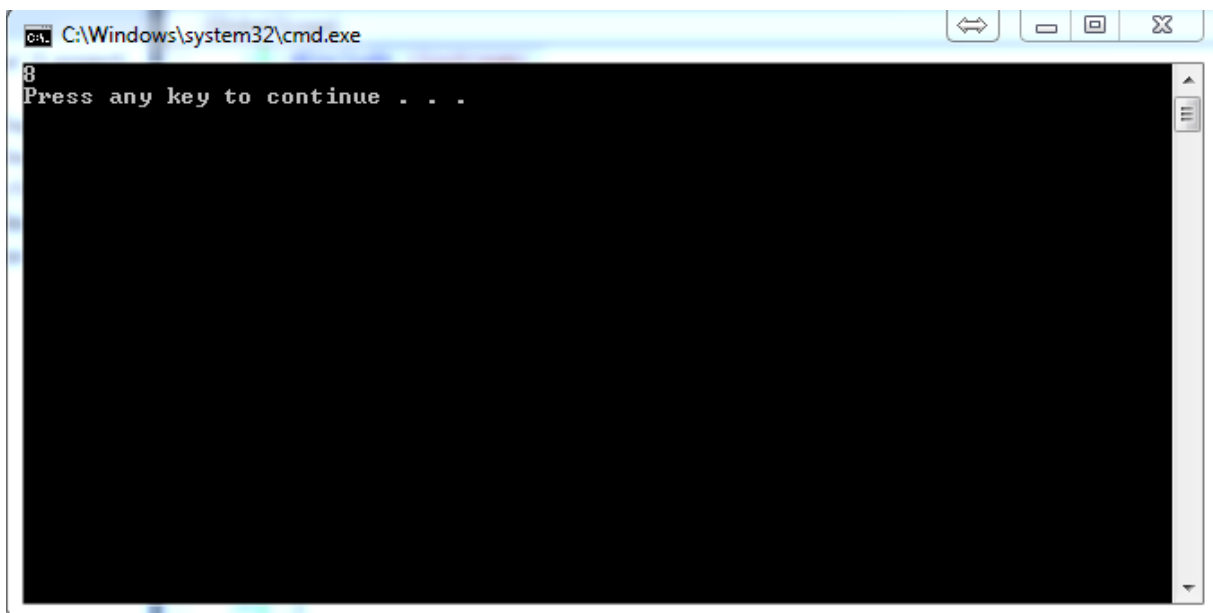
```
#include <iostream>
#include <string>
using namespace std;

int prototip();

int main()
{
    cout << prototip()<<endl;
    return 0;
}

int prototip()
{
    int a = 10;
    int rezultat;
    rezultat = a/2+3;

    return rezultat;
}
```



Slika 31 - Rezultat izvršavanja kôda u primjeru korištenja prototipa funkcije

### c. Dekompozicija problema korištenjem funkcija

Kako smo do sada već spominjali **dekompoziciju**, sada ćemo joj se još malo posvetiti. Razbijanje problema na manje dijelove posebno će biti uočljivo prilikom korištenja funkcija. Zamislite da morate napraviti funkciju za izračun kvadratne jednadžbe čija je formula :  $ax^2 + bx + c = 0$ . U ovoj funkciji možemo imati koristiti još dodatnih funkcija. Prva će izračunavati kvadrat broja ( $x^2$ ) i pomnožiti tu vrijednost s vrijednosti varijable „a“. Druga će vrijednost varijable „x“ pomnožiti s vrijednosti varijable „b“ i na kraju će treća zbrojiti sve te vrijednosti (moći će se zbrojiti prvi i drugi član ako je primjerice  $x=1$ ). Sada bismo još mogli posebno raspisati glavne korake za daljnji izračun i tek kada bi ispisali sve glavne korake, onda bi krenuli u rješavanje problema.

### d. Funkcije i argumenti

Upoznat ćemo funkcije koje primaju jedan ili više argumenata istih i/ili različitih tipova i one koje ne primaju nikakve argumente. Argument je zapravo neka vrijednost koju ćemo mi dati našoj funkciji da obavi neku radnju nad njom ili će tu vrijednost uključiti u neku radnju. Primjerice ako napravimo funkciju za kvadriranje tada bi naš argument bio broj koji želimo kvadrirati. U tijelu funkcije će se taj broj pomnožiti sam sa sobom i na kraju će nam funkcija vratiti kvadriranu vrijednost.

### e. Stvaranje i korištenje privremenih varijabli unutar funkcije

Tokom korištenja funkcija susrest ćemo se s potrebom stvaranja varijabli koje nam trebaju samo za operacije te funkcije ali nam ne trebaju u ostatku našeg programa. Varijable koje se deklariraju unutar neke funkcije su vidljive samo unutar te funkcije odnosno, deklariramo li varijablu „zbroj“ unutar funkcije tada toj varijabli nećemo moći pristupiti iz ostalih dijelova našeg programa kao niti iz drugih funkcija.

### f. Primjeri korištenja funkcija za uređivanje ispisa

Želimo li u našoj aplikaciji ispisati recimo simbol „\*“ u jednom retku 50 puta naravno da to nećemo ručno obavljati. Jedna od opcija je korištenje „for“ ili „while“ petlje no što ako ćemo trebati jednom ispisati simbol „\*“ a drugi put simbol „+“. Tada bismo ponovno morali raditi novu petlju a jedina razlika bi bila u promjeni simbola koji se ispisuje. U ovakvim primjerima možemo napraviti našu funkciju koja će primiti dva argumenta – simbol koji da ispiše i koliko puta da ga ispiše. Nakon što predamo našoj funkciji ta dva argumenta ona će određeni simbol ispisati željeni broj puta.

### g. Primjeri korištenja funkcija za matematičke izračune

Primjena funkcija je velika i u matematičkim izračunima. Vrlo jednostavno možemo izraditi funkciju koja će ispisivati fibonačijev niz (*engl. Fibonacci Sequence*). Možemo funkciju i malo proširiti tako da prima jedan argument u kojem navodimo koliko brojeva da ispiše.

## h. Primjeri korištenja funkcija za ispisivanje isto teksta određeni broj puta

Uz već prikazani način ispisivanja istog teksta više puta korištenjem „for“ petlje, možemo naravno napraviti i funkciju za takvu operaciju. Jednostavno bi funkciji rekli koji tekst da ispiše i kroz drugi parametar koliko puta da ga ispiše.

## 21) Funkcije s „return“ naredbom bez parametara

### a. Što su tipovi funkcija, sintaksa funkcija i što je to „return“ naredba

Kao što postoje različiti tipovi podataka za varijable, tako razlikujemo i tipove podataka za funkcije. Same vrste tipova su iste a sama vrsta tipa koju navedemo prije naziva funkcije govori našem programu kakvu će vrijednost vratiti naša funkcija odnosno što će funkcija vratiti kao rezultat preko naredbe „return“. *Sintaksa funkcija* je :

```
tip_podatka_koji_funkcija_vraća ime_funkcij (parametri_ako_ih_ima)
{
  Kôd koji se izvršava
  return naredba
}
```

Ime funkcije zadajemo sami a parametre ćemo nešto kasnije obraditi pa se oko tog dijela još ne morate brinuti. Ono što je važno za uočiti i zapamtiti je naredba **return** koja se nalazi na kraju funkcije. S ovom naredbom ste se već susreli jer se nalazi na kraju naše „main“ funkcije u kojoj smo pisali sav naš kôd. Naredba „return“ simbolizira kraj funkcije i tu navodimo što želimo da nam funkcija vrati odnosno navodimo rezultat nakon što je funkcija obavila svoj dio kôda koji ćemo koristiti u daljnjem dijelu aplikacije. Tip podatka koji funkcija vraća u „return“ naredbi mora odgovarati tipu podatka koji smo naveli prilikom deklaracije funkcije.

### b. Jednostavne funkcije koje vraćaju „integer“ tip podatka

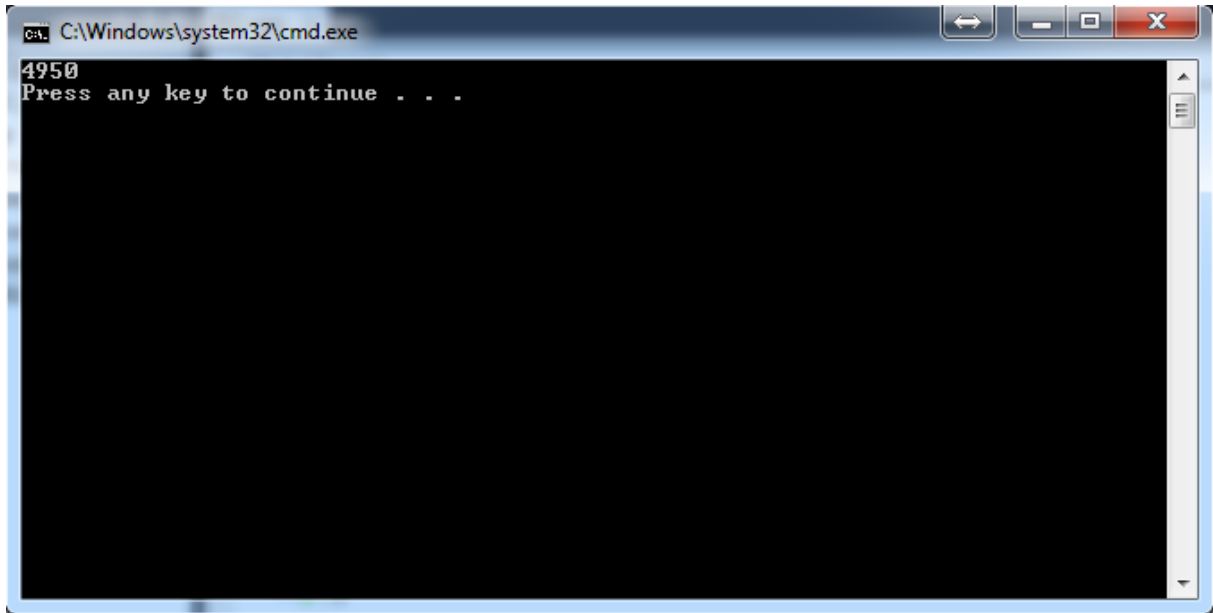
S funkcijama koje vraćaju *integer* tip podatka smo se susreli već puno puta. To je naša „main“ funkcija u koju smo do sada upisivale kôd. Na kraju se nalazila naredba „return 0“. Prilikom deklaracije funkcije naveli smo da će funkcija vraćati *integer* tip podatka a takvu vrijednost smo vratili pošto broj 0 (nula) je *integer*.

Izuzev „main“ funkcije možemo napraviti i vlastite *integer* funkcije pa ćemo za ovaj primjer napraviti funkciju koja će vratiti zbroj prvih 100 cjelobrojnih brojeva (od 0 do 99). Za ovu potrebnu morat ćemo stvoriti zasebnu varijablu tipa *integer* unutar naše funkcije u koju će se pohraniti naša vrijednost. Postupak je moguće izvesti i bez deklaracije zasebne varijable ali nema smisla komplicirati, posebice u uvodnom dijelu programiranja.

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

int zbroj()
{
    int suma = 0;
    for ( int i = 0; i<100;i++)
    { suma+=i; }
    return suma;
}
```

```
int main()
{
    cout << zbroj()<<endl;
    return 0;
}
```



Slika 32 - Rezultat izvršenja kôda jednostavne funkcije koja vraća integer tip podatka

### c. Jednostavne funkcije koje vraćaju „float“ tip podatka

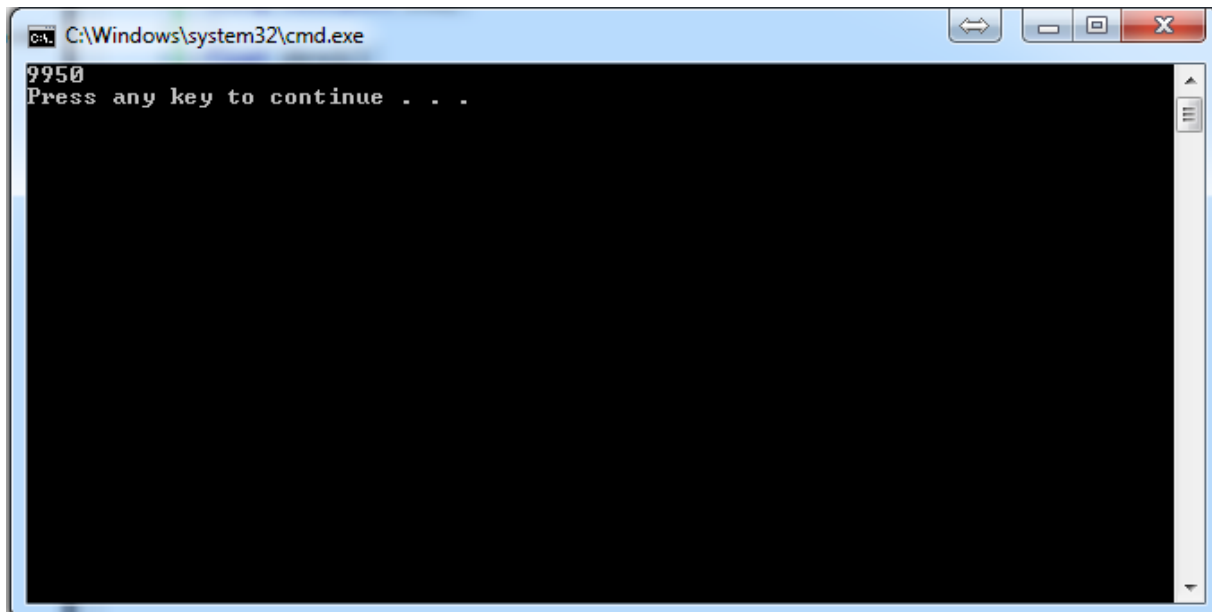
Slično kao i prijašnji primjer, napraviti ćemo funkciju koja će zbrojiti prvih 100 brojeva (0-99) no sada nećemo imati pomak za cijeli broj već za 0.5. Bitno je uočiti da se sada radi o funkciji koja vraća *float* tip podatka.

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

float zbroj()
{
    float suma = 0;
    for ( float i = 0; i<100;i+=0.5)
    { suma+=i; }
    return suma;
}

int main()
{
    cout << zbroj()<<endl;
    return 0;
}
```





Slika 33 - Rezultat izvršavanja kôda jednostavne funkcije koja vraća float tip podatka

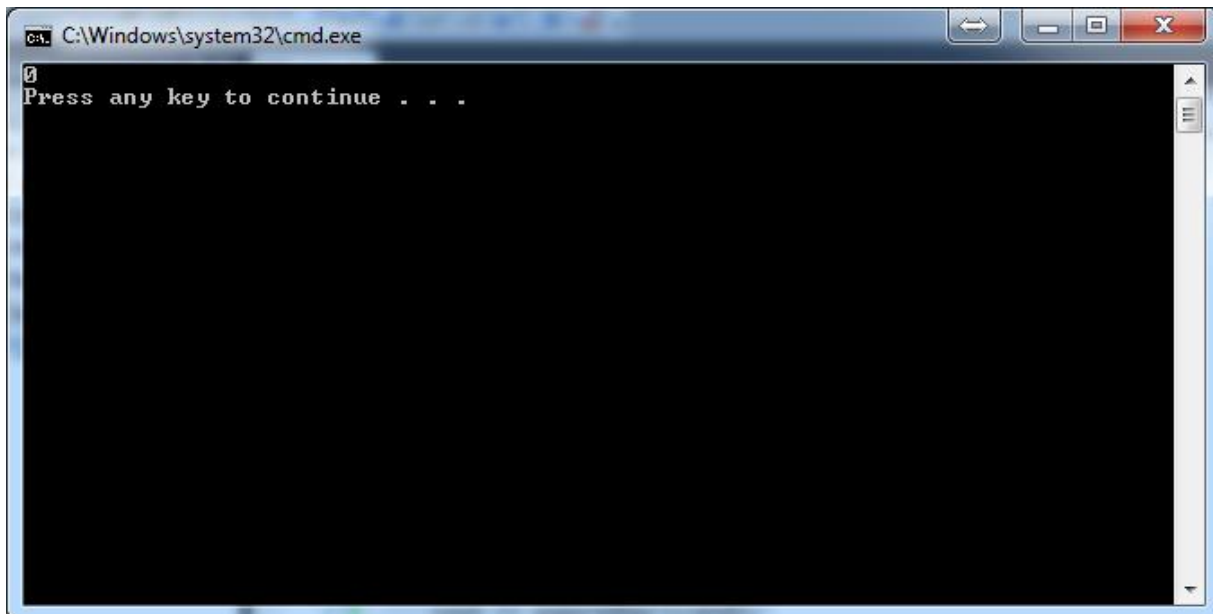
#### d. Jednostave funkcije koje vraćaju „boolean“ tip podatka

U slučaju funkcije za koju želimo da vrati boolean tip podatka moguća su naravno samo dva ishoda; *true* ili *false*. Ovakve funkcije možemo koristiti za usporedbu dva broja ili nekih drugih vrijednosti koje korisnik unese no takve usporedbe ćemo obraditi uskoro kada dođemo do funkcija koje primaju parametre. Za jednostavni primjer funkcije bez parametara izadit ćemo funkciju koja će uspoređivati dva broja i ovisno o uvjetu vratiti *true* ili *false* vrijednost.

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

bool usporedba()
{
    bool rjesenje;
    int broj1 = 10;  int broj2 = 20;
    if ( broj1 > broj2)
        { rjesenje = true; }
    else
        { rjesenje = false; }
    return rjesenje;
}

int main()
{
    cout << usporedba()<<endl;
    return 0;
}
```



Slika 34 - Rezultat izvođenja jednostavne funkcije tipa boolean

Obrazloženje :

Unutar funkcije uspoređujemo vrijednosti varijable „broj1“ i „broj2“. Ako je vrijednost varijable „broj1“ veća od vrijednosti varijable „broj2“ tada će varijabla „rjesenje“ imati vrijednost *true*. U slučaju da je vrijednost varijable „broj1“ manja od vrijednosti varijable „broj2“, tada će varijabla „rjesenje“ imati vrijednost *false*. Kako je u ovom slučaju vrijednost varijable „broj1“ manja od vrijednosti varijable „broj2“, varijabla „rjesenje“ je poprimila vrijednost *false* što se i ispisalo u konzolnoj aplikaciji kao broj 0 (nula) što simbolizira vrijednost *false*.

### e. Jednostavne funkcije koje vraćaju „string“ tip podatka

Funkcije koje vraćaju *string* zahtijevaju uključenje „string“ biblioteke. Sam rad s ovakvim funkcijama je identičan kao i sa svim do sada spomenutim funkcijama. Pogledajmo primjer funkcije koja vraća skup rečenica kao rezultat. Korištenje ispisa pomoću funkcije može biti korisno ukoliko u našoj aplikaciji imamo više mjesta na kojima je potrebno ispisati isti tekst nekoliko puta a radi se o tekstu koji ima nekoliko rečenica.

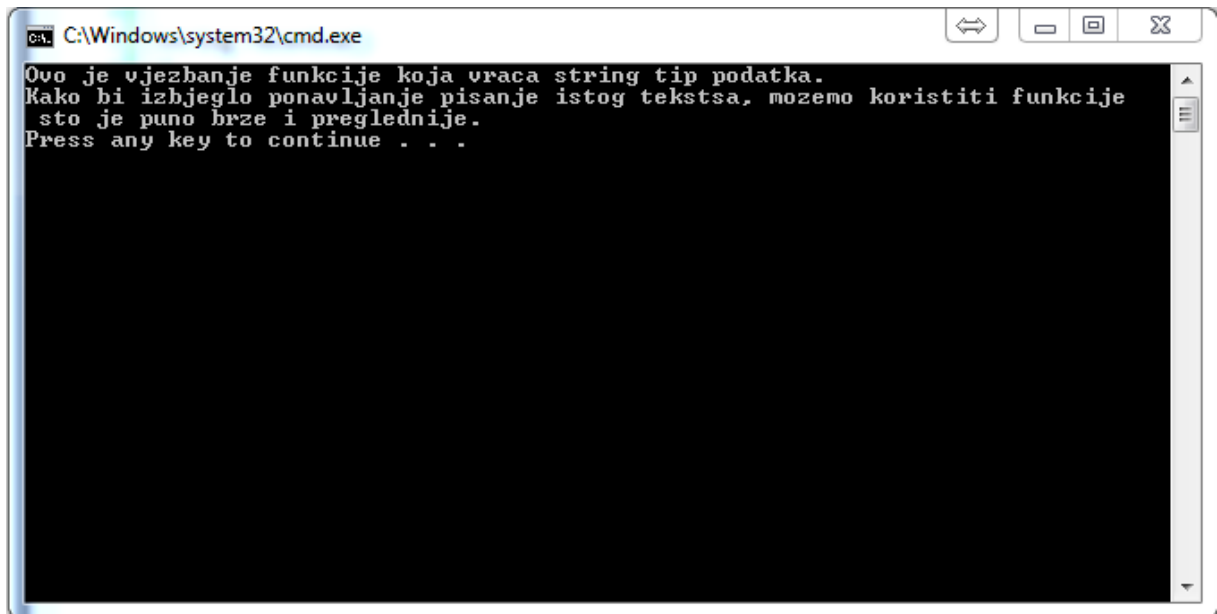
```
#include <iostream>
#include <string>
using namespace std;

string recenica()
{
    string izjava = "Ovo je vježbanje funkcije koja vraca string tip
podatka.\n\
Kako bi izbjeglo ponavljanje pisanje istog teksta, \
mozemo koristiti funkcije \n sto je puno brze i preglednije.";

    return izjava;
}
```

```
int main()
{
    cout << recenica()<<endl;
    return 0;
}
```

**Uočite** simbol „\“ koji nam omogućuje pisanje kôda kroz više linija u našem *editoru*. Stavljanjem simbola „\“ *editoru* govorimo da iako više ništa ne piše u toj liniji da se tekst u slijedećoj liniji također isti dio kôda. Ovim načinom možemo na oku ugodniji način napisati neki duži tekst.



```
C:\Windows\system32\cmd.exe
Ovo je vjezbanje funkcije koja vraca string tip podatka.
Kako bi izbjeglo ponavljanje pisanje istog teksta, mozemo koristiti funkcije
sto je puno brze i preglednije.
Press any key to continue . . .
```

Što se tiče samo *return* naredbe nema ništa novo za reći što već nije obrazloženo u prijašnjim primjerima.

## f. Jednostavne funkcije koje ne vraćaju ništa – VOID

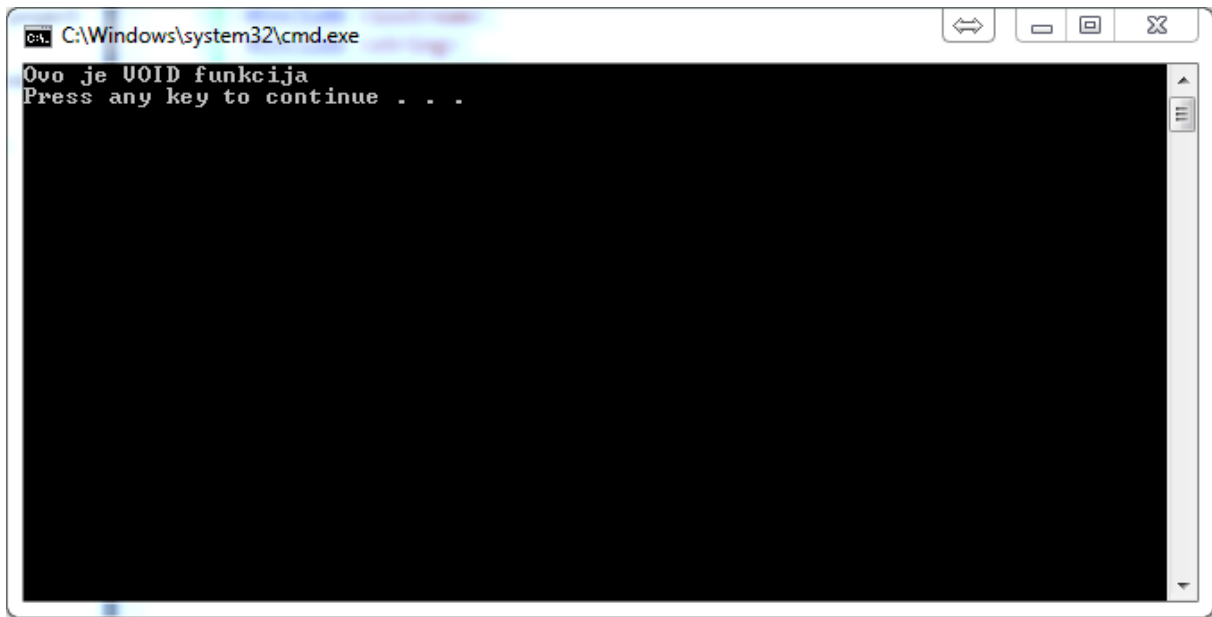
Posebna vrsta funkciju su one koje ne vraćaju nikakvu vrijednost. Njihova svrha je kada trebamo samo neke podatke obraditi ali nam ne treba nikakva izlazna vrijednost. Imamo li u programu neku varijablu kojoj pomoću funkcije želimo naknadno promijeniti vrijednost ili želimo ispisati tekst kao što je to bio slučaj sa „*string*“ tipom funkcija koristit ćemo „*void*“ funkcije. Ove funkcije nemaju „*return*“ naredbu dok je ostatak *sintakse* identičan svim ostalim funkcijama.

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

void ispis()
{
    cout << "Ovo je VOID funkcija" << endl;
}
```

```
int main()
{
    ispis();
    return 0;
}
```

*Slika 35 - Rezultat izvođenja kôda u void tipu funkcije*



*Slika 36 - Rezultat izvođenja kôda uz void funkciji*

## g. Zadaci

- 1) Napravite aplikaciju koja sadrži jednu funkciju u kojoj će se izračunati zbroj prvih 10 znamenki *fibonaccievog niza*.
- 2) Napravite aplikaciju koja sadrži funkciju u kojoj se maloprodajna cijena proizvoda. Veleprodajne cijene proizvoda su 10 kn, 20 kn, 50 kn i 100 kn. Maloprodajnu cijenu dobijemo tako da veleprodajnu cijenu uvećamo za iznos PDV-a.
- 3) Napravite polje *integers* koje sadrži 10 elemenata. Prvi element sadrži brojku 100, drugi 200, ... a deseti 1000. Napravite funkciju smještenu unutar for petlje (koja prolazi kroz cijelo polje) i zbraja sve brojeve u jednu varijablu i po završetku čitanja elemenata, funkcija vraća sumu svih brojeva te ispisuje tu vrijednost u konzolu.

## 22) Funkcije koje primaju argumente istog tipa

### a. Uvod

Do sada smo obradili funkcije koje ne primaju nikakve argumente. Takve funkcije nemaju veliku funkcionalnost jer nemaju *interakciju* s varijablama odnosno podacima unutar našeg ostatka programa odnosno „main“ funkcije. Sada ćemo upoznati funkcije koje primaju parametre te na taj način omogućiti *interakciju* funkcija s ostatkom dijela programa. U ovom poglavlju ćemo obraditi funkcije koje primaju argumente istog tipa a i sljedećem poglavlju ćemo obraditi funkcije koje primaju funkcije argumente različitog tipa.

### b. Funkcije koje primaju jedan argument s „return“ naredbom

Svaka funkcija može primiti onoliko argumenata koliko joj navedemo prilikom deklaracije. Broj argumenata se mora poštivati pa tako nije moguće predati više ali niti manje argumenata kao niti argument drugačijeg tipa od onog koji smo prvotno naveli te je bitan i poredak. Tip podatka koji funkcija vraća može biti kao i tip podatka argumenta koji predajemo funkciji ali i ne mora, sve ovisi o potrebama naše aplikacije.

Argument koji predajemo navodimo unutar obliha zagrada navodeći tip podatka tog argumenta i njegovo ime. **Važno** je znati da ime argumenta može i ne mora odgovarati imenu varijable koju prenosimo u funkciji.

#### **Sintaksa :**

```
tip_podatka_koji_funkcija_vraća  ime_funkcije  (  tip_podatka_argumenta
ime_argumenta)
{
Kôd koji se izvodi u tijelu funkcije;
return naredba;
}
```

#### **Primjer 1:**

Napravit ćemo aplikaciju koja pita korisnika da unese neki cijeli broj. Ta broj ćemo proslijediti našoj funkciji koja će provjeriti radi li se o pozitivnom ili negativnom broju. Ako se radi o negativnom broju, funkcija će vratiti vrijednost da je broj negativan odnosno ako je uneseni broj pozitivan tada će funkcija vratiti vrijednost da je unesen pozitivan broj. Na kraju ćemo ispisati rezultat koji vraća funkcija.

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

string provjera(int broj)
{
    string rjesenje;
    if ( broj <0)
    { rjesenje = "Broj je negativan."; }
}
```

```

    else
    { rjesenje = "Broj je pozitivan."; }

    return rjesenje;
}

int main()
{
    int x;
    cout << "Unesi neki cijeli broj : ";
    cin >> x;

    cout << provjera(x)<<endl;
    return 0;
}

```

```

C:\Windows\system32\cmd.exe
Unesi neki cijeli broj : 5
Broj je pozitivan.
Press any key to continue . . .

```

Slika 37 - Rješenje primjera 1

### Objašnjenje :

Aplikacija stvara varijablu „x“ tipa *integer* i zatim pita korisnika da unese neki cijeli broj. Nakon što korisnik unese broj ta se vrijednost sprema u varijablu „x“. Pozivamo naredbu „cout“ za ispis u konzolu a ispisat ćemo ono što kao rezultat daje funkcija „**provjera**“. Funkciju smo pozvali predajući joj jedan argument a to je varijabla „x“. Funkcija očekuje da primi argument tipa *integer* što je ovdje i napravljeno i funkcija nastavlja s izvršavanjem kôda u tijelu. Ovisno koji broj se preda funkcija stvara drugačije rješenje. U našem slučaju korisnik je upisao broj 5 koji je pozitivan i varijabla „rjesenje“ koja je tipa „*string*“ a deklarirana je u tijeku funkcije poprima vrijednost „Broj je pozitivan.“. To je vrijednost koju funkcija vraća a time će se i ta vrijednost ispisati u konzolu.

### c. Funkcije koje primaju jedan argument bez „return“ naredbe

Slično kao i funkcije koje imaju „return“ naredbu, prenijet ćemo jedan argument no ovdje nećemo vratiti nikakav rezultat jer se radi o „void“ funkciji. To znači da ćemo moći vrijednost argumenta iskoristiti za naše potrebe ali nećemo moći vratiti nikakvu vrijednost kao rezultat izvršavanja funkcije.

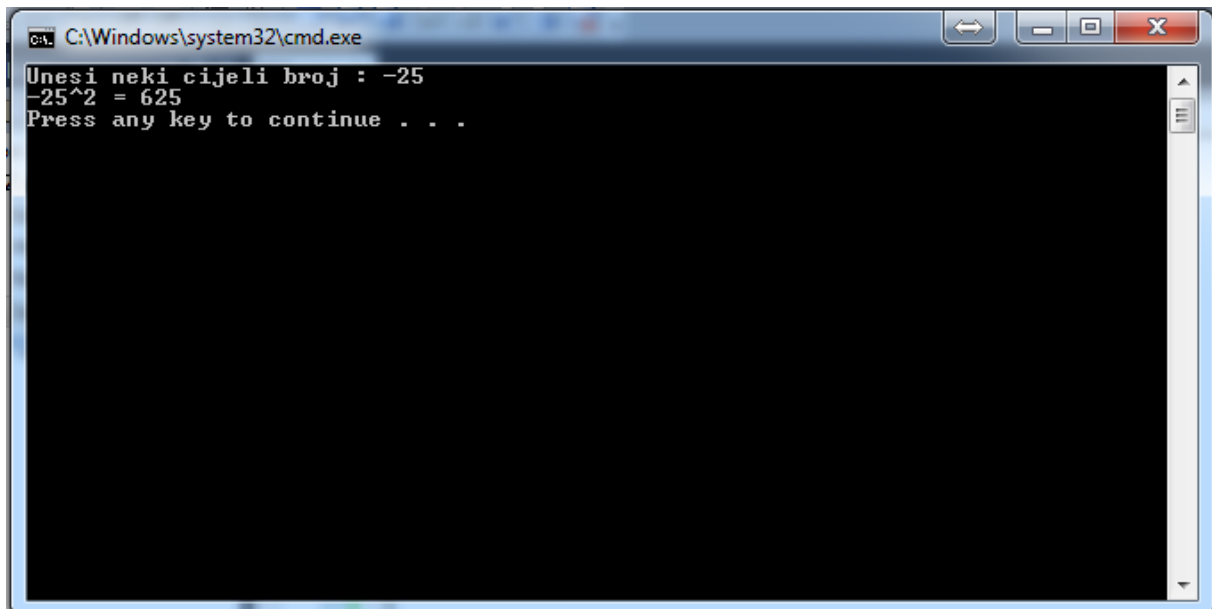
#### Primjer 1 :

Napravit ćemo aplikaciju koja sadrži jednu jednostavnu funkciju. Aplikacija će pitati korisnika da unese neki cijeli broj te ćemo taj broj predati kao argument funkciji koja će izvršiti potenciranje i ispis vrijednosti u konzolu.

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

void potenciranje(int broj)
{
    cout << broj << "^2 = " << broj*broj << endl;
}

int main()
{
    int x;
    cout << "Unesi neki cijeli broj : ";
    cin >> x;
    potenciranje(x);
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Unesi neki cijeli broj : -25
-25^2 = 625
Press any key to continue . . .
```

Slika 38 - Rezultat izvršavanja kôda u primjeru 1



#### d. Funkcije koje primaju više argumenata s „return“ naredbom

Na isti način kako smo predavali jedan argument nekoj funkciji isto tako je moguće predati i više argumenata. Konačan broj argumenata ovisi o našoj deklaraciji funkcije odnosno našim potrebama. **Važno** je zapamtiti da je redoslijed kojim predajemo vrijednosti važan i ne smije se zamijeniti jer može doći do logičkih i/ili *sintaksnih* pogrešaka.

##### Primjer 1 :

Napravit ćemo aplikaciju koja pita za dva broja koja ćemo potom prenijeti kao argumente u funkciju. Funkcija će kao rezultat vratiti njihov zbroj.

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

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

int main()
{
    int prvi, drugi;
    cout << "Unesi prvi broj : ";cin>>prvi;
    cout << "Unesi drugi broj : ";cin>>drugi;
    cout << "Rezultat je : "<<zbroj(prvi, drugi)<<endl;
    return 0;
}
```

A screenshot of a Windows command prompt window. The title bar shows the path "C:\Windows\system32\cmd.exe". The window content displays the following text:  
Unesi prvi broj : 3  
Unesi drugi broj : 6  
Rezultat je : 9  
Press any key to continue . . .  
The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Slika 39 - Rezultat nakon izvršavanja kôda u primjeru 1

Uočite da u ovom slučaju poredak predavanja argumenata nije važan, no što bi bilo da smo napravili funkciju za oduzimanje. Tada bismo dobili pogrešan rezultat odnosno počinili bismo logičku pogrešku. Da smo imali funkciju koja prima dva parametra različitog tipa i da pokušamo pogrešnim

redosljedom predati parametre počinut ćemo *sintaksnu* pogrešku no o tome ćemo dobiti obavijest prije pokretanja aplikacije.

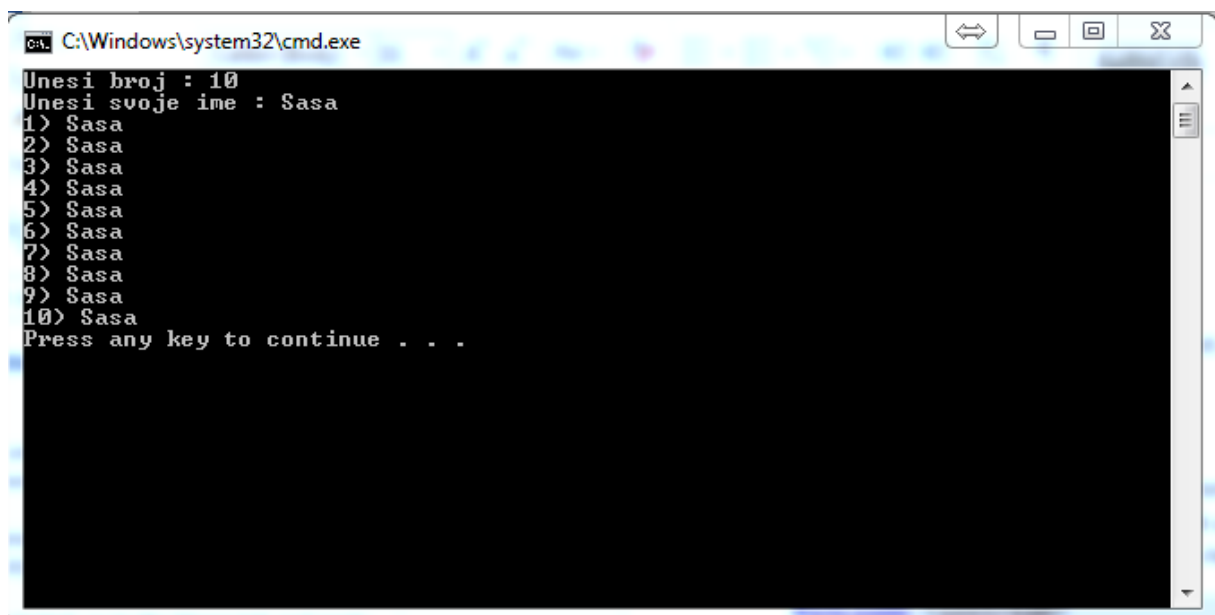
### Primjer 2:

Napravit ćemo aplikaciju koja pita korisnika da unese svoje ime i neki cijeli broj. Aplikacija će zatim mu konzolu ispisati ime koje je korisnik unio onoliko puta koliki je broj korisnik naveo. Broj i ime ćemo predati kao parametre funkciji u kojoj će se izvršavati ispisivanje u konzolu. **Uočite** da je sada poredak kojim predajemo parametre od ključne važnosti.

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

void zbroj( int a, string b)
{
    for ( int i = 1; i<=a; i++)
    {
        cout <<i<<" ) „<< b << endl;
    }
}

int main()
{
    int prvi;
    string rijec;
    cout << "Unesi broj : ";cin>>prvi;
    cout << "Unesi svoje ime : ";
    cin >> rijec;
    zbroj(prvi,rijec);
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Unesi broj : 10
Unesi svoje ime : Sasa
1) Sasa
2) Sasa
3) Sasa
4) Sasa
5) Sasa
6) Sasa
7) Sasa
8) Sasa
9) Sasa
10) Sasa
Press any key to continue . . .
```

Slika 40 - Rezultat nakon izvođenja kôda u primjeru 2

## e. Zadaci

- 1) Napraviti aplikaciju koja će korisnika pitati za unos dva broja. Unutar aplikacije napraviti funkciju koja će u konzolu ispisati rezultate i prikaz operacije koja se obavila za zbrajanje, oduzimanje, dijeljenje, množenje, modulo.
- 2) Napraviti aplikaciju koja pita korisnika za unos neke riječi i koliko puta da se ta riječ ispiše na ekran. Unutar aplikacije potrebno je kreirati funkciju koja prima ta dva podatka kao argumente i na ekran ispisuje tu riječ onoliko puta koliko je koliko je korisnik naveo.

## 23) Formatiranje ispisa 2 – naprednije formatiranje

### a. biblioteka <iomanip> - uvjet za bolje formatiranje

Za korištenje dalje navedenih opcija formatiranja ispisa morat ćemo uključiti još jednu biblioteku koja se zove „*iomanip*“. Podsjetimo se, do sada smo koristili biblioteku „*iostream*“ a biblioteke uključujemo koristeći naredbu „*#include*“ nakon čega slijedi naziv biblioteke koju uključujemo, okružena izlomljenim (špičaste) zagrada. Za primjer „*iomanip*“ biblioteke cijela linija kôda za uključivanje te biblioteke će izgledati ovako : `#include <iomanip>`.

Vaš projekt bi sada trebao ovako izgledati :

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

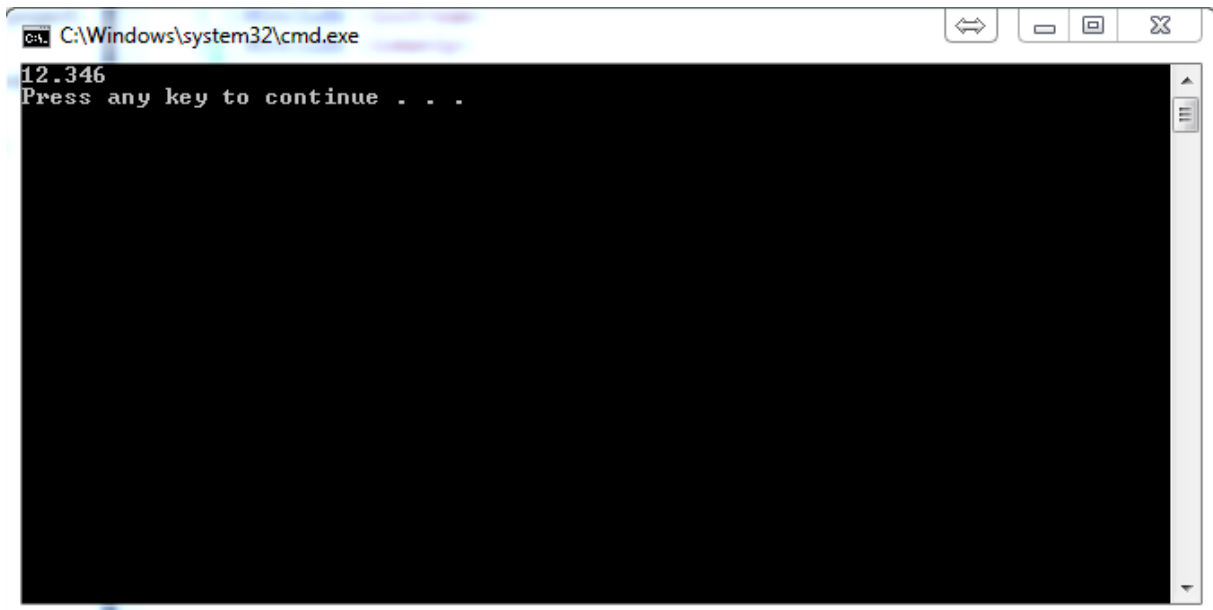
int main()
{
    return 0;
}
```

### b. setprecision()

Funkcija „*setprecision*“ omogućuje nam formatiranje brojeva. Funkcija prima jedan argument tipa *integer*. Pomoću ove funkcije možemo odrediti koliko znamenki da se prikaže u konzolnoj aplikaciji neovisno koji tip podatka ispisujemo dok god je on neki od brojevnih tipova. Funkciju koristimo tako da napišemo „*cout*“ naredbu za ispis u konzolu, zatim „*<<*“ iza čega slijedi funkcija i argument, `setprecision(5)`, pa opet „*<<*“ i zatim `broj ili varijabla` koja ima brojevanu vrijednost koju želimo ispisati.

#### **Primjer 1 :**

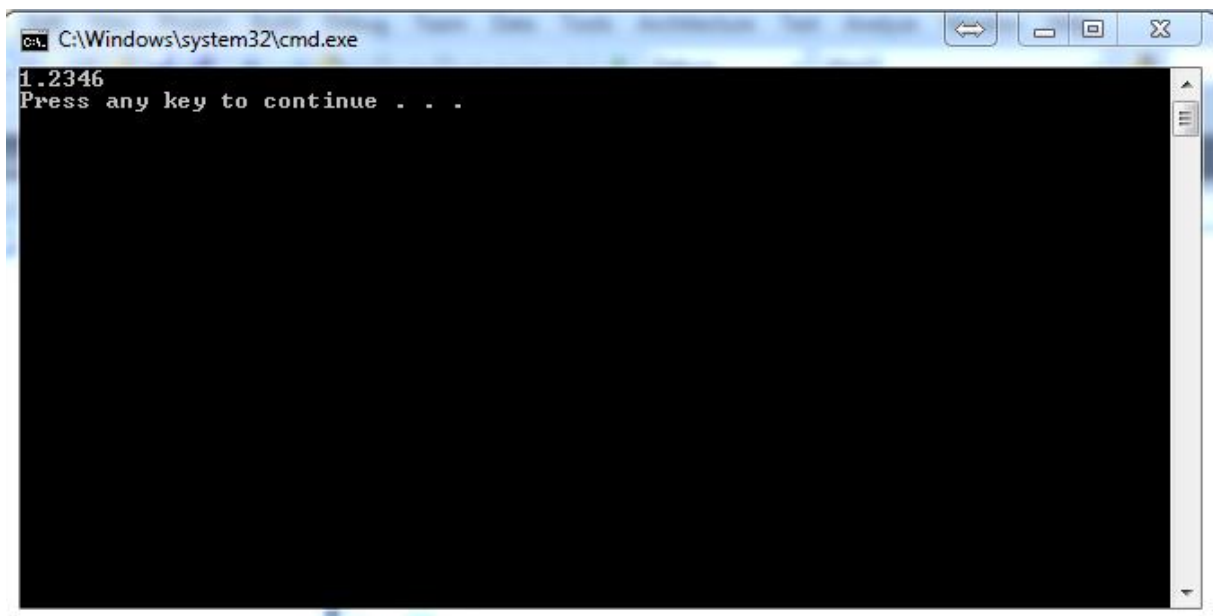
```
int main()
{
    cout << setprecision(5) << 12.34567<<endl;
    return 0;
}
```



Slika 41 - Rezultat nakon izvođenja kôda u primjeru 1

**Primjer 2 :**

```
int main()
{
    float broj = 1.234567890;
    cout << setprecision(5) << broj << endl;
    return 0;
}
```



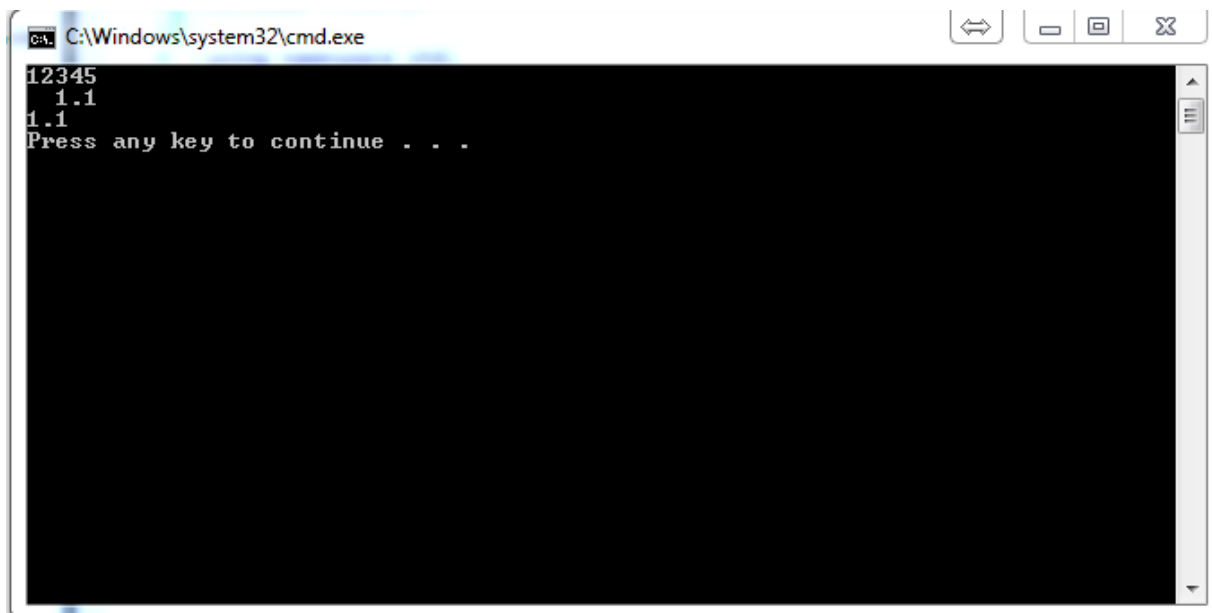
Slika 42 - Rezultat izvođenja kôda u primjeru 2

### c. width()

Funkcija „*width*“ nam omogućuje da odredimo minimalnu širinu prilikom ispisa. Ukoliko naš ispis ne sadrži dužinu koju smo naveli kao minimalnu ova funkcija će u ispisu razliku nadopuniti simbolom za nadopunjavanje koji je po početnim vrijednostima prazno mjesto (*engl. space*). Funkcija prima jedan argument tipa *integer* koji označava našu minimalnu potrebnu širinu prilikom ispisa. Prilikom korištenja funkcije potrebno je prvo napisati `cout.width(neki_broj_tipa_integer);` i onda opet `cout<<ono_što_ispisujemo;` Primjena širine odnosi se samo na prvu *cout* naredbu koja slijedi nakon specificiranja širine.

#### Primjer 1:

```
int main()
{
    float broj = 1.1;
    cout << "12345"<<endl;
    cout.width(5); cout << broj<<endl;
    cout<<broj<<endl;
    return 0;
}
```



```
C:\Windows\system32\cmd.exe
12345
 1.1
1.1
Press any key to continue . . .
```

Slika 43 - Rezultat izvođenja kôda u primjeru 1

**Uočite** popunjavanje prazninom u drugom retku ispisa i zatim uočite kako se svojstvo širine nije primijenilo i za treći redak. Ako bismo željeli ispisati i treći redak tako da je minimalna širina 5 mjesta, tada bismo opet morali navesti `cout.width(5);`.

#### d. Zadaci

- 1) Napraviti aplikaciju koja ispisuje koliko američkih dolara (\$), britanskih funti (£) i eura (€) možemo dobiti za 10 kn, 20 kn, 50 kn i 100kn a da maksimalan broj znamenki bude 5
- 2) Napraviti tablicu množenja za brojeve od 1 do 10 (uključujući granične brojeve) tako da svaki svi stupci i redovi budu poravnati (izgled tablice).
- 3) Ispisati sve cijele brojeve od 1 do 100 (uključujući granične brojeve) tako da redovi i stupci budu poravnati (izgled tablice) a da se red prekida nakon ispisanih „n“ brojeva gdje slovo „n“ predstavlja broj koji korisnik unese. Unese li primjerice korisnik broj 3 to znači da će se u jednom redu nalaziti po tri broja.

## 24) Stringovi 2 – korištenje funkcija nad stringovima

### a. Uvod

Kako smo već spomenuli, *string* tip podatka nije primitivan odnosno osnovni tip podatka. *Stringovi* su objekti o čemu u ovoj skripti neće detaljnog spomena jer spada u znatno zahtjevniju i kompleksniju domenu. Bitno je znatno da se nad objektima odnosno u našem slučaju *stringovima* mogu vršiti razne operacije odnosno funkcije. Zapamtite dakle, *string* je nakupina slova, brojeva i simbola i s takvom nakupinom možemo raditi svašta. Pretvaranje u brojčane vrijednosti, čitanje samo jednog znaka, prebacivanje slova u nekoj riječi i slično samo su neke od operacija koje možemo raditi sa *stringovima*. Sada ćete upoznati osnovne operacije koje je moguće raditi nad *stringovima*.

### b. getline() funkcija

Do sada smo prilikom korištenja *string* tipova podataka prilikom unosa radili primjere samo sa jednom riječi. Kako znamo da su *stringovi* sva slova, brojke i znakovi zar nije logično da možemo raditi unos i za dvije, tri ili više riječi? Pokušajmo korisnika pitati za njegovo ime i prezime.

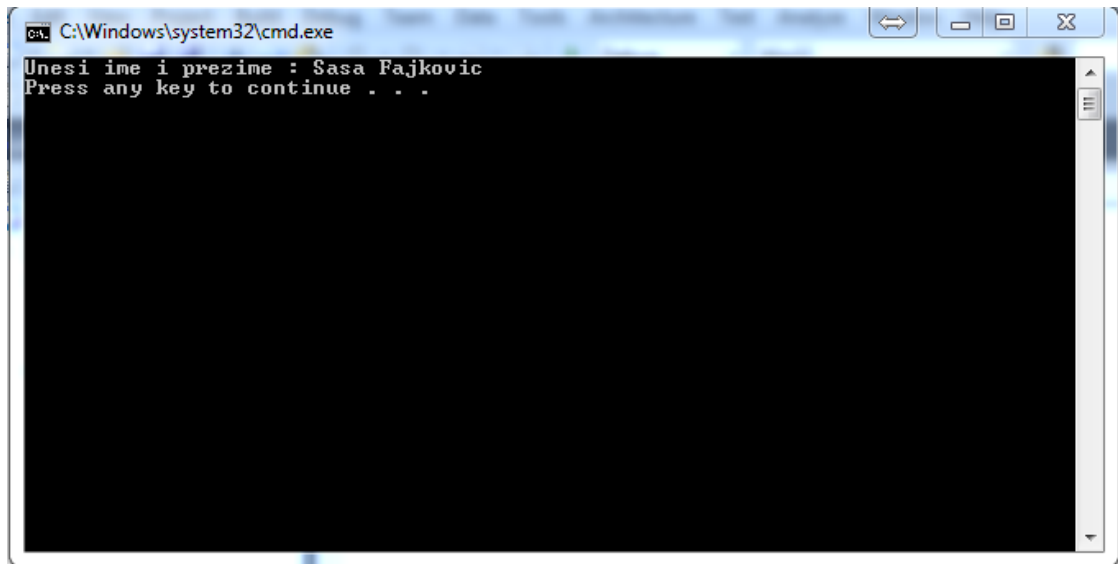
```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string podatci;
    cout << "Unesi ime i prezime : ";
    cin >> podatci;

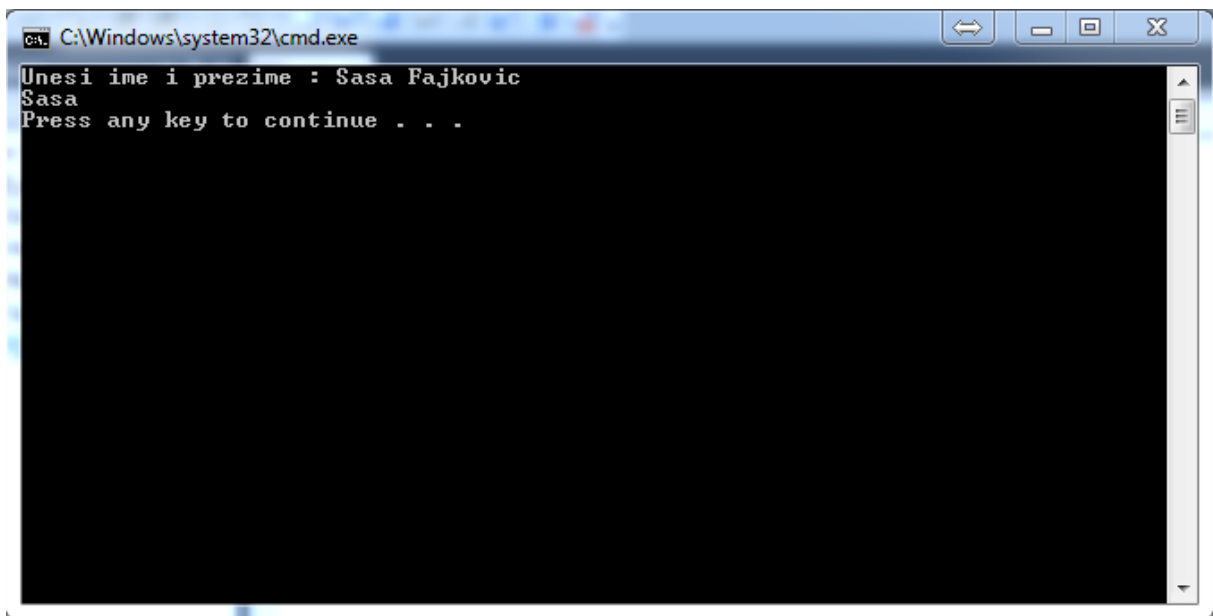
    return 0;
}
```





Slika 44 - rezultat kôda za unos imena i prezimena u string varijablu

Čini se kako je cijeli proces uspješno izvršen. No pokušajmo sada ispisati vrijednost spremljenu u varijablu „*podatci*“ koristeći naredbu `cout << podatci<<endl;`.



Slika 45 - Rezultat ispisa vrijednosti iz varijable tipa string

Kao što vidite, ispisalo se samo ime ali ne i prezime. Razlog tome je što se korištenjem naredbe „*cin*“ pohranjuje vrijednost unesena DO prvog razmaka (razmak se također ne sprema u varijablu). U varijablu „*podatci*“ je sada spremljena samo vrijednost „Sasa“. Kako bismo mogli učitati sve što je upisano odnosno cijelu liniju teksta koristimo funkciju ***getline***. Funkcija će učitavati sve dok ne naiđe na simbol „*\n*“ koji obilježava prelazak u novi red.

#### **Sintaksa :**

```
getline(od_kuda_da_čitam_podatke, gdje_da_spremim_podatke);
```

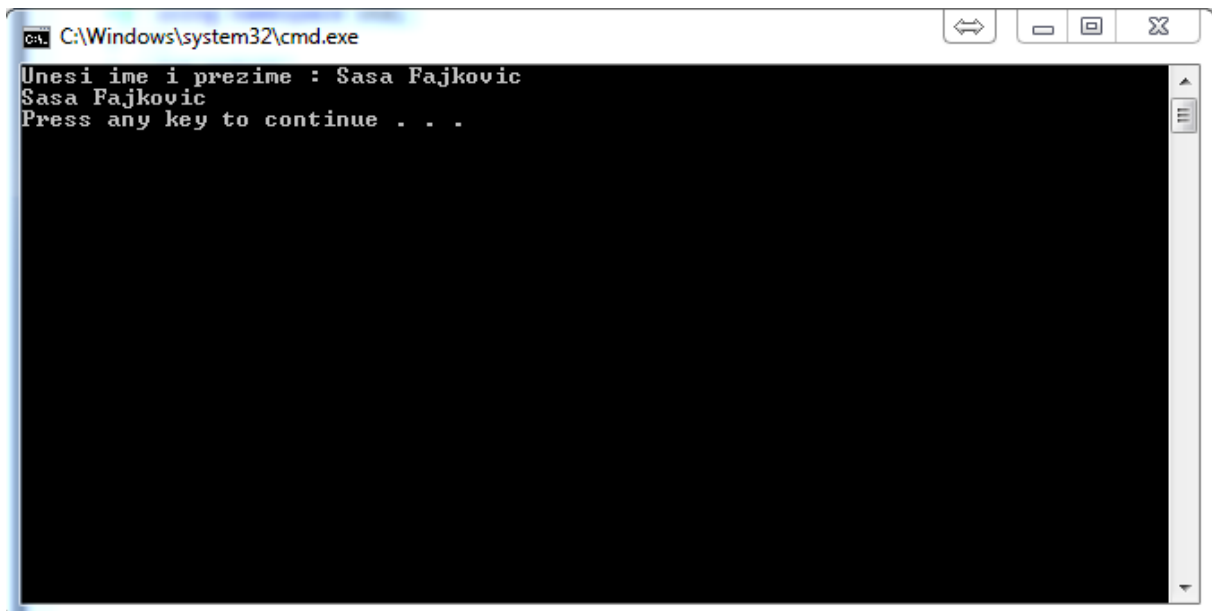
### Primjer :

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string podatci;
    cout << "Unesi ime i prezime : ";
    getline(cin,podatci);
    cout << podatci<<endl;

    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Unesi ime i prezime : Sasa Fajkovic
Sasa Fajkovic
Press any key to continue . . .
```

Slika 46 - Rezultat korištenja kôda s *getline* funkcijom

### Objašnjenje :

Deklarirali smo varijablu imena „*podatci*“ tipa *string* i pitali korisnika da unese svoje ime i prezime; Nakon što korisnik unese tražene podatke i pritisne tipku „*Enter*“ koristi se funkcija *getline* kojoj smo kao ulazni parametar (od kuda da čita podatke) naveli da se radi o konzoli a kao izlazni parametar (gdje da spremi podatke) smo naveli varijablu „*podatci*“. Ispis pomoću *cout* naredbe je standardan.

### c. length() funkcija

Korištenjem funkcije *length* možemo saznati duljinu odnosno veličinu našeg *stringa*. Duljina se mjeri u simbolima i ova funkcija vraća kao rezultat brojčani podatak tipa *integer*. Funkcija ne prima nikakve argumente. U duljinu *stringa* su uključeni i razmaci.

#### Sintaksa :

```
ime_varijable.length();
```

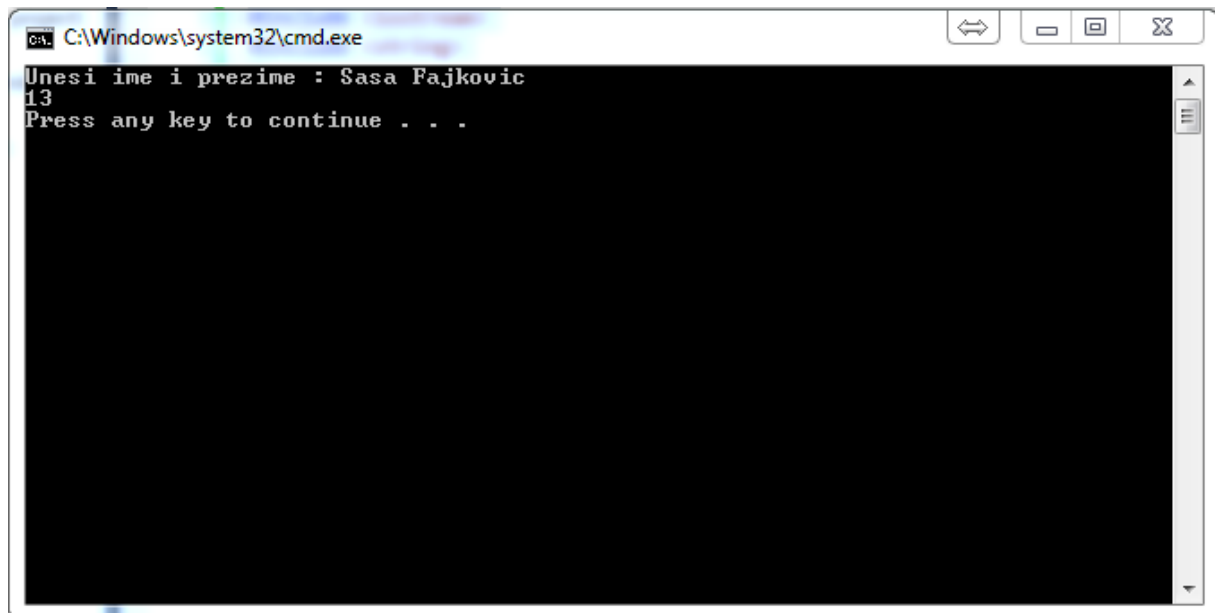
#### Primjer :

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string podatci;
    cout << "Unesi ime i prezime : ";
    getline(cin,podatci);
    cout << podatci.length()<<endl;

    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Unesi ime i prezime : Sasa Fajkovic
13
Press any key to continue . . .
```

Slika 47 - Rezultat korištenja kôda s *length* funkcijom

#### d. prolaz slovo po slovo kroz *string* pomoću „for“ i „while“ petlje

Svaki *string* se sastoji od slova, brojeva i/ili simbola i svaki od njih moguće zasebno izdvojiti. *String* tip podatka možete promatrati kao polje elemenata koje također počinje s indeksom 0 (nula) a svaki simbol, broj ili slovo u *stringu* se nalazi pod određenim indeksom. U riječi „auto“ slovo „a“ se nalazi na indeksu 0 (nula), slovo „u“ se nalazi na indeksu 1 (jedan), slovo „t“ na indeksu 2 (dva) a slovo „o“ na indeksu 3 (tri). Pristupamo im slično kao i polju elemenata. Pogledajmo primjer kada želimo ispisati svako slovo zasebno :

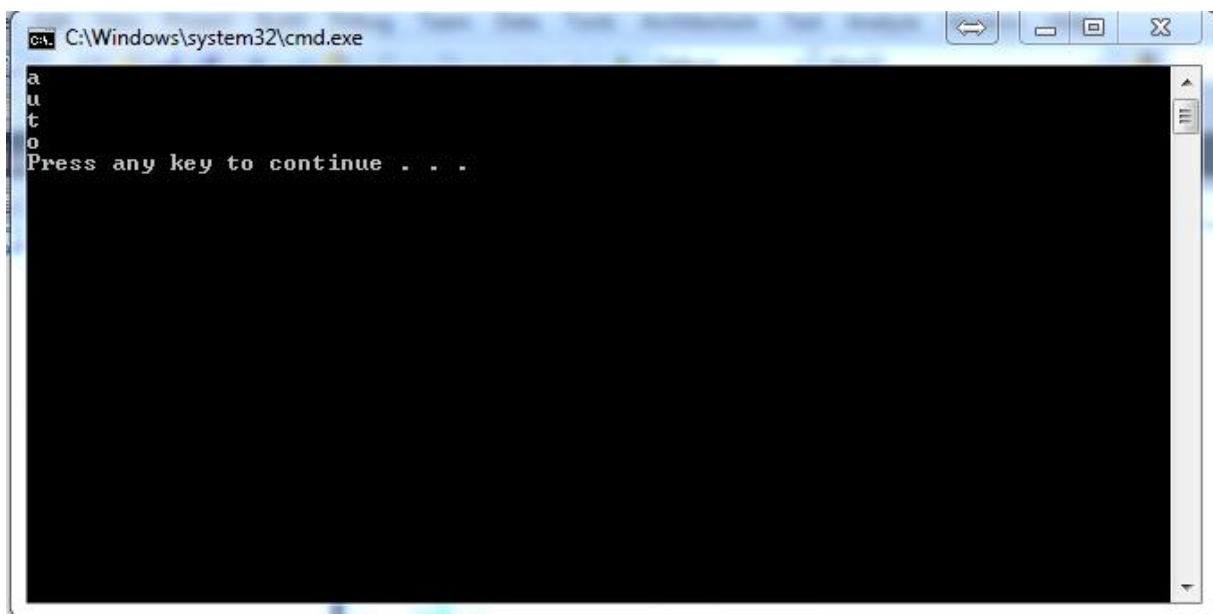
##### **Primjer 1 :**

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string rijec;
    rijec = "auto";
    cout << rijec[0]<<endl;
    cout << rijec[1]<<endl;
    cout << rijec[2]<<endl;
    cout << rijec[3]<<endl;

    return 0;
}
```



The screenshot shows a Windows command prompt window titled "cmd.exe" with the path "C:\Windows\system32\cmd.exe". The window displays the output of the C++ program: the characters 'a', 'u', 't', and 'o' are printed on four separate lines. Below the output, the prompt "Press any key to continue . . ." is visible. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Slika 48 - Ispis svakog slova u riječi "auto"

Naravno da ovakav pristup nije ispravan jer nećemo znati koju riječ će korisnik unijeti pa tako nećemo niti znati od koliko znakova će se *string* sastojati. U kombinaciji s funkcijom *length()* i *for* petljom možemo ispisati sve znakove nekog *stringa* bez obzira na njegovu dužinu. Pogledajmo kako :

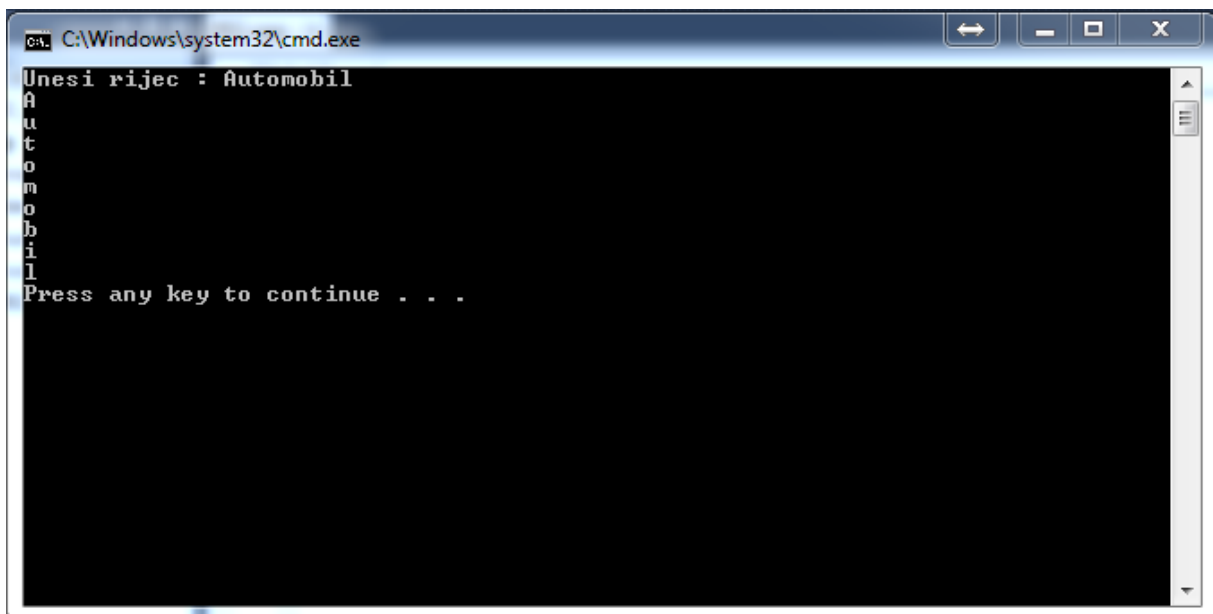
### Primjer 2 :

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string rijec;
    cout <<"Unesi rijec : ";
    cin >> rijec;
    int duljina = rijec.length();
    for ( int i=0; i<duljina;i++ )
    {
        cout << rijec[i]<<endl;
    }

    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Unesi rijec : Automobil
A
u
t
o
m
o
b
i
l
Press any key to continue . . .
```

Slika 49 - Ispis slova u jednoj riječi pomoću for petlje, lenght funkcije i pristupa znakovima stringa preko indeksa

### Objašnjenje :

U varijablu „rijec“ smo spremili riječ koju unese korisnik i pomoću funkcije „*length*“ smo našli koliko znakova se nalazi u toj riječi. U „for“ petlji postavljamo početak brojanja da krene od broja 0 (nula) i neka broji do onog broja koji je spremljen u varijablu „duljina“. Za svaki korak u „for“ petlji ispisuje se onaj znak koji se nalazi na „i-tom“ indeksu *stringa* spremljenog u varijablu „rijec“. Nakon što se svaki korak završi vrijednost varijable „i“ se uvećava za jedan.

### Primjer 3 :

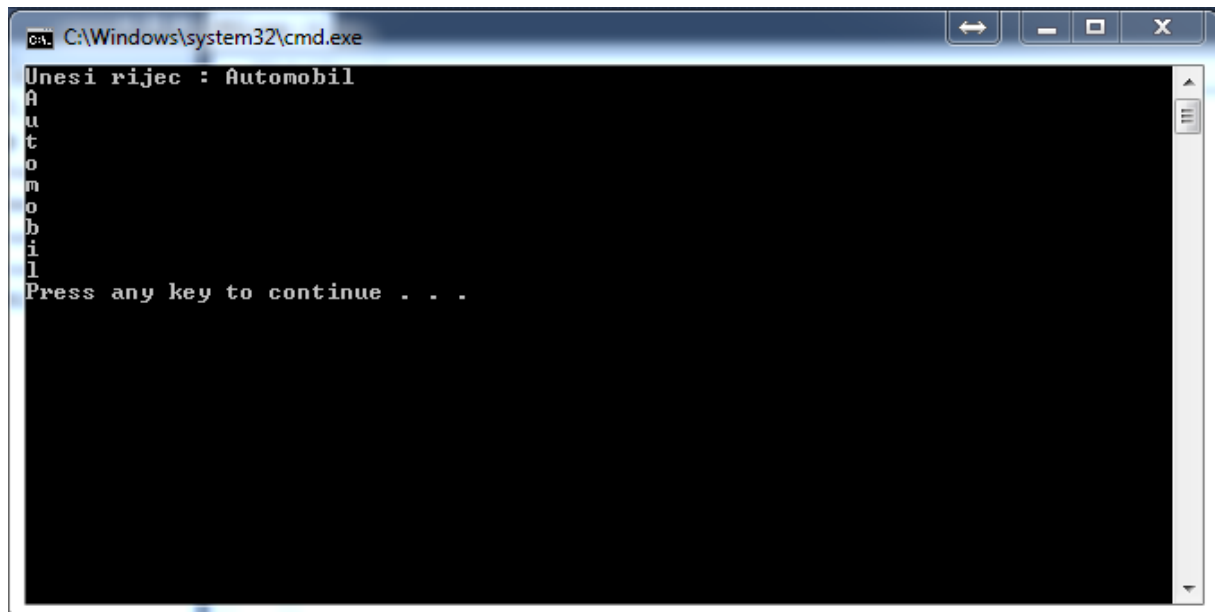
Još brži i korektniji način bi bio da nismo uopće deklarirali varijablu „duljina“ već da smo unutar same „for“ petlje naveli krajnju vrijednost do kuda da „for“ petlja broji tako da krajnja vrijednost bude jednaka vrijednosti koliko se nalazi znakova u *stringu* spremljenom u varijabli „rijec“.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string rijec;
    cout << "Unesi rijec : ";
    cin >> rijec;
    for ( int i=0; i<rijec.length();i++ )
    {
        cout << rijec[i]<<endl;
    }

    return 0;
}
```



```
C:\Windows\system32\cmd.exe
Unesi rijec : Automobil
A
u
t
o
m
o
b
i
l
Press any key to continue . . .
```

Slika 50 - Rezultat nakon izvršenja kôda u primjeru 3

Rezultat u primjeru 2 i primjeru 3 je na očigled identičan no razlika je u kôdu koji smo napisali. U primjeru broj 3 nismo stvarali zasebnu varijablu čime smo štedjeli resurse jer nije bilo potrebe za dodatnim zauzimanjem memorije. Naravno, prilikom korištenja ovakvih jednostavnih aplikacija zauzeće memorije nije primjetno no svakako treba obratiti pažnju na ovakve stvari kako bi ovakav „pametniji“ način pisanja kôda „ušao pod prste“.

## e. toupper() i tolower() funkcije

Zamislite da imate opet aplikaciju gdje je gazda kafića tražio podatke o svojim konobarima i da je samo upisivao njihova imena nakon čega bi se izbacili podatci o njima. Do sada smo koristili uvjet `if(ime=="pero") { izvrši neki kôd; }`. Morali smo se uzdati u točnost korisnika (gazde kafića) da će ime unijeti zaista malim slovom inače naša aplikacija ne bi mogla pronaći podatke o traženom konobaru. Svako vjerovanje da će korisnik unijeto podatke točno kako je predviđeno je suludo i vjerojatno nikada u povijesti programiranja se nije pokazalo da korisnik zaista svaki put unese podatke kako je predviđeno. U ovom slučaju će nam pomoći funkcije „toupper“ i „tolower“. Ove funkcije trenutno pretvaraju vrijednost nekog teksta u velika odnosno mala slova. Ako mi u našoj aplikaciji imamo navedeno da će za konobara imenom „Pero“ uvjet za ispis biti zadovoljen samo ako su sva slova u imenu napisana malim slovima tada pomoću funkcije „tolower“ možemo sva slova pretvoriti u mala i tako nadmudriti korisnika koji može ime unijeti kao „Pero“ ili „PERO“ ili „pERO“ te sve ostale varijacije.

**Pažnja** – ove funkcije ne pretvaraju riječ po riječ već slovo po slovo. To znači da ćemo morati proći kroz svako slovo unutar neke riječi i promijeniti ga u malo. Ako je slovo već bilo „malo“ tada će i ostati „malo“.

**Pažnja 2** – Svaki simbol, brojka ili slovo u svim mogućim varijacijama su zapisani u čuvenom **ASCII kodu** odnosno svako slovo ima svoju šifru pod kojim je spremljeno u tablici. Pokušamo li ispisati vrijednost funkcije „tolower()“ ili „toupper()“ dobit ćemo neki cijeli broj a ne slovo. Taj broj označava šifru pod kojom je to slovo spremljeno u ASCII tablici. ASCII tablicu naravno nije potrebno pamtit i ukoliko se ukaže potreba za traženjem šifre pod kojom je neki znak spremljen uvijek to možete potražiti na *internetu*.

| Dec | Hx | Oct | Char                               | Dec | Hx | Oct | Html  | Chr   | Dec | Hx | Oct | Html  | Chr | Dec | Hx | Oct | Html   | Chr |
|-----|----|-----|------------------------------------|-----|----|-----|-------|-------|-----|----|-----|-------|-----|-----|----|-----|--------|-----|
| 0   | 0  | 000 | <b>NUL</b> (null)                  | 32  | 20 | 040 | ##32; | Space | 64  | 40 | 100 | ##64; | @   | 96  | 60 | 140 | ##96;  | `   |
| 1   | 1  | 001 | <b>SOH</b> (start of heading)      | 33  | 21 | 041 | ##33; | !     | 65  | 41 | 101 | ##65; | A   | 97  | 61 | 141 | ##97;  | a   |
| 2   | 2  | 002 | <b>STX</b> (start of text)         | 34  | 22 | 042 | ##34; | "     | 66  | 42 | 102 | ##66; | B   | 98  | 62 | 142 | ##98;  | b   |
| 3   | 3  | 003 | <b>ETX</b> (end of text)           | 35  | 23 | 043 | ##35; | #     | 67  | 43 | 103 | ##67; | C   | 99  | 63 | 143 | ##99;  | c   |
| 4   | 4  | 004 | <b>EOT</b> (end of transmission)   | 36  | 24 | 044 | ##36; | \$    | 68  | 44 | 104 | ##68; | D   | 100 | 64 | 144 | ##100; | d   |
| 5   | 5  | 005 | <b>ENQ</b> (enquiry)               | 37  | 25 | 045 | ##37; | %     | 69  | 45 | 105 | ##69; | E   | 101 | 65 | 145 | ##101; | e   |
| 6   | 6  | 006 | <b>ACK</b> (acknowledge)           | 38  | 26 | 046 | ##38; | &     | 70  | 46 | 106 | ##70; | F   | 102 | 66 | 146 | ##102; | f   |
| 7   | 7  | 007 | <b>BEL</b> (bell)                  | 39  | 27 | 047 | ##39; | '     | 71  | 47 | 107 | ##71; | G   | 103 | 67 | 147 | ##103; | g   |
| 8   | 8  | 010 | <b>BS</b> (backspace)              | 40  | 28 | 050 | ##40; | (     | 72  | 48 | 110 | ##72; | H   | 104 | 68 | 150 | ##104; | h   |
| 9   | 9  | 011 | <b>TAB</b> (horizontal tab)        | 41  | 29 | 051 | ##41; | )     | 73  | 49 | 111 | ##73; | I   | 105 | 69 | 151 | ##105; | i   |
| 10  | A  | 012 | <b>LF</b> (NL line feed, new line) | 42  | 2A | 052 | ##42; | *     | 74  | 4A | 112 | ##74; | J   | 106 | 6A | 152 | ##106; | j   |
| 11  | B  | 013 | <b>VT</b> (vertical tab)           | 43  | 2B | 053 | ##43; | +     | 75  | 4B | 113 | ##75; | K   | 107 | 6B | 153 | ##107; | k   |
| 12  | C  | 014 | <b>FF</b> (NP form feed, new page) | 44  | 2C | 054 | ##44; | ,     | 76  | 4C | 114 | ##76; | L   | 108 | 6C | 154 | ##108; | l   |
| 13  | D  | 015 | <b>CR</b> (carriage return)        | 45  | 2D | 055 | ##45; | -     | 77  | 4D | 115 | ##77; | M   | 109 | 6D | 155 | ##109; | m   |
| 14  | E  | 016 | <b>SO</b> (shift out)              | 46  | 2E | 056 | ##46; | .     | 78  | 4E | 116 | ##78; | N   | 110 | 6E | 156 | ##110; | n   |
| 15  | F  | 017 | <b>SI</b> (shift in)               | 47  | 2F | 057 | ##47; | /     | 79  | 4F | 117 | ##79; | O   | 111 | 6F | 157 | ##111; | o   |
| 16  | 10 | 020 | <b>DLE</b> (data link escape)      | 48  | 30 | 060 | ##48; | 0     | 80  | 50 | 120 | ##80; | P   | 112 | 70 | 160 | ##112; | p   |
| 17  | 11 | 021 | <b>DC1</b> (device control 1)      | 49  | 31 | 061 | ##49; | 1     | 81  | 51 | 121 | ##81; | Q   | 113 | 71 | 161 | ##113; | q   |
| 18  | 12 | 022 | <b>DC2</b> (device control 2)      | 50  | 32 | 062 | ##50; | 2     | 82  | 52 | 122 | ##82; | R   | 114 | 72 | 162 | ##114; | r   |
| 19  | 13 | 023 | <b>DC3</b> (device control 3)      | 51  | 33 | 063 | ##51; | 3     | 83  | 53 | 123 | ##83; | S   | 115 | 73 | 163 | ##115; | s   |
| 20  | 14 | 024 | <b>DC4</b> (device control 4)      | 52  | 34 | 064 | ##52; | 4     | 84  | 54 | 124 | ##84; | T   | 116 | 74 | 164 | ##116; | t   |
| 21  | 15 | 025 | <b>NAK</b> (negative acknowledge)  | 53  | 35 | 065 | ##53; | 5     | 85  | 55 | 125 | ##85; | U   | 117 | 75 | 165 | ##117; | u   |
| 22  | 16 | 026 | <b>SYN</b> (synchronous idle)      | 54  | 36 | 066 | ##54; | 6     | 86  | 56 | 126 | ##86; | V   | 118 | 76 | 166 | ##118; | v   |
| 23  | 17 | 027 | <b>ETB</b> (end of trans. block)   | 55  | 37 | 067 | ##55; | 7     | 87  | 57 | 127 | ##87; | W   | 119 | 77 | 167 | ##119; | w   |
| 24  | 18 | 030 | <b>CAN</b> (cancel)                | 56  | 38 | 070 | ##56; | 8     | 88  | 58 | 130 | ##88; | X   | 120 | 78 | 170 | ##120; | x   |
| 25  | 19 | 031 | <b>EM</b> (end of medium)          | 57  | 39 | 071 | ##57; | 9     | 89  | 59 | 131 | ##89; | Y   | 121 | 79 | 171 | ##121; | y   |
| 26  | 1A | 032 | <b>SUB</b> (substitute)            | 58  | 3A | 072 | ##58; | :     | 90  | 5A | 132 | ##90; | Z   | 122 | 7A | 172 | ##122; | z   |
| 27  | 1B | 033 | <b>ESC</b> (escape)                | 59  | 3B | 073 | ##59; | ;     | 91  | 5B | 133 | ##91; | [   | 123 | 7B | 173 | ##123; | {   |
| 28  | 1C | 034 | <b>FS</b> (file separator)         | 60  | 3C | 074 | ##60; | <     | 92  | 5C | 134 | ##92; | \   | 124 | 7C | 174 | ##124; |     |
| 29  | 1D | 035 | <b>GS</b> (group separator)        | 61  | 3D | 075 | ##61; | =     | 93  | 5D | 135 | ##93; | ]   | 125 | 7D | 175 | ##125; | }   |
| 30  | 1E | 036 | <b>RS</b> (record separator)       | 62  | 3E | 076 | ##62; | >     | 94  | 5E | 136 | ##94; | ^   | 126 | 7E | 176 | ##126; | ~   |
| 31  | 1F | 037 | <b>US</b> (unit separator)         | 63  | 3F | 077 | ##63; | ?     | 95  | 5F | 137 | ##95; | _   | 127 | 7F | 177 | ##127; | DEL |

Source: [www.LookupTables.com](http://www.LookupTables.com)

Slika 51 - ASCII tablica<sup>6</sup>

<sup>6</sup> izvor : <http://www.asciitable.com/>

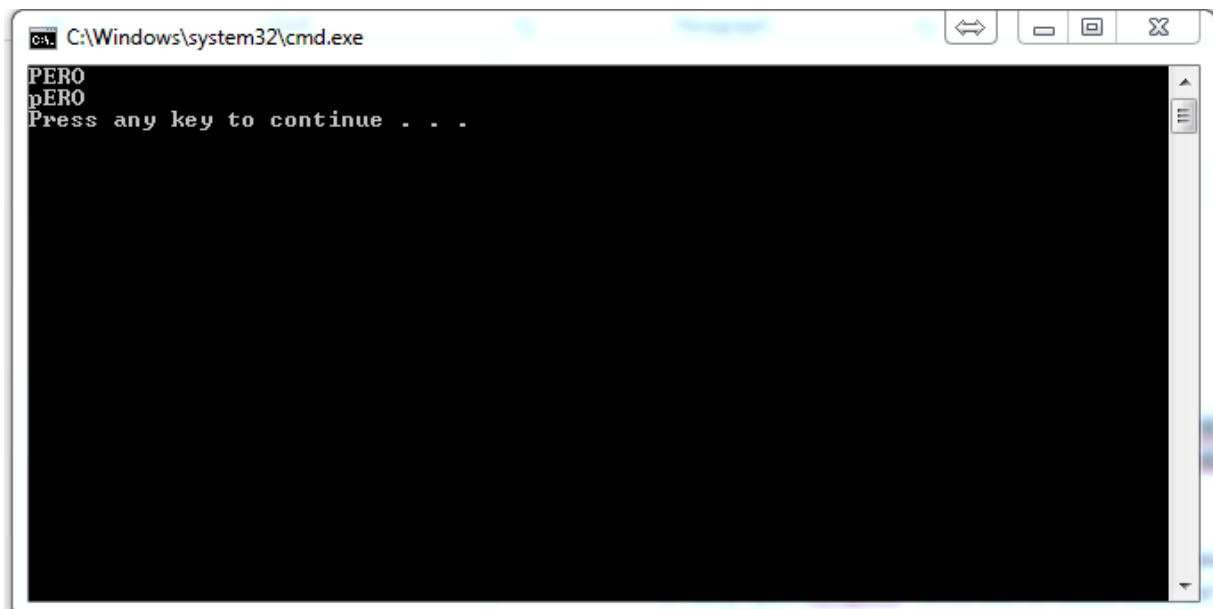
### Sintaksa :

```
tolower(karakter(jedno_slovo)_kojeg_funkcija_prima_kao_argument);  
toupper(karakter(jedno_slovo)_kojeg_funkcija_prima_kao_argument);
```

Iako funkcija vraća numeričku vrijednost nekog znaka mi to možemo iskoristiti jer ćemo svakom slovu u našoj riječi „izmjeniti“ šifru. Ako se veliko slovo „A“ nalazi pod šifrom 65 a malo slovo „a“ nalazi pod šifrom 97 tada ćemo napraviti izmjenu tako da šifru 65 zamijenimo šifrom 97.

### Primjer 1 :

```
#include <iostream>  
#include <string>  
  
using namespace std;  
  
int main()  
{  
    string rijec="PERO";  
    cout << rijec<<endl;  
    rijec[0] = tolower(rijec[0]);  
    cout << rijec << endl;  
  
    return 0;  
}
```



```
C:\Windows\system32\cmd.exe  
PERO  
pERO  
Press any key to continue . . .
```

Slika 52 - izmjena prvog slova u riječi korištenjem funkcije tolower

### Objašnjenje :

Deklariramo varijablu imena „rijec“ tipa *string* i prijedlimo joj vrijednost „PERO“ te ju ispišemo u konzolu. Zatim slovo koje se nalazi na nultom indeksu te riječi želimo prebaciti u „malo“ slovo. Ono kako *compiler* sada razmišlja je : Pronađi varijablu imenom „rijec“ i element na nultom indeksu zamijeni onime što izađe kao rezultat kada obavim funkciju „tolower“. Zatim pogleda što je predani



argument toj funkciji i shvati da treba naći nulti indeks riječi spremljene u varijablu „rijec“. Pronađe tu varijablu i dohvati znak koji se nalazi na nultom indeksu. Pogleda njegovu šifru u ASCII tablici i vidi da je to slovo „P“ te traži po ASCII tablici koja šifra odgovara malom slovu „p“. Nakon što nađe koja je šifra nju zapamti i onu vrijednost koja se nalazila na nultom indeksu riječi spremljene u varijablu „rijec“ zamijeni s novom vrijednosti.

### **Primjer 2 :**

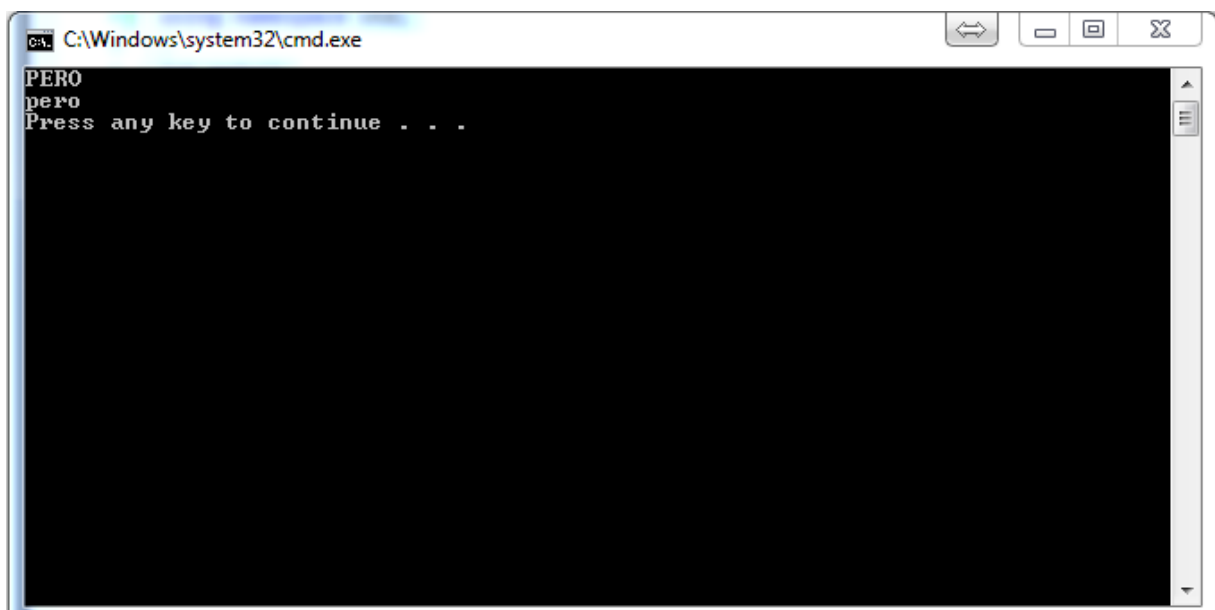
Sada ćemo promijeniti cijelu riječ tako što ćemo koristiti „for“ petlju da prođemo kroz cijelu riječ, funkciju „length“ da odredimo do kuda da „for“ petlja broji i u tijelu petlje ćemo pozivati funkciju „tolower“ koja će kao parametar primiti „i-ti“ indeks varijable „rijec“ i mijenjati vrijednost na tom indeksu.

```
#include <iostream>
#include <string>

using namespace std;

int main()
{
    string rijec="PERO";
    cout << rijec << endl;
    for( int i=0;i<rijec.length();i++ )
    {
        rijec[i]=tolower(rijec[i]);
    }
    Cout << rijec << endl;

    return 0;
}
```



```
C:\Windows\system32\cmd.exe
PERO
pero
Press any key to continue . . .
```

*Slika 53 - korištenje tolower funkcije za pretvorbu slova u "mala" slova*

## f. atoi funkcija

**Pretvorba iz *string* oblika u numerički** je također moguća. Naravno, morat ćemo biti sigurni da tekst koji pretvaramo zaista se može pretvoriti u broj. To znači da ćemo *string* zapisan kao „123“ moći pretvoriti u broj ali *string* koji je zapisan kao „1a2“ nećemo jer ćemo napraviti logičku pogrešku. Pokušamo li vrijednost „1a2“ pretvoriti u cjelobrojnu vrijednost naš *debugger* nam neće javiti pogrešku ali će rezultat biti pogrešan. Pretvorbu znaka „1“ u broju 1 (jedan) će uspješno prebaciti no kada naiđe na slovo „a“ prekinut će izvršavanje i u našoj varijabli u koju smo željeli spremi bročanu vrijednost će se spremi samo brojka 1 (jedan).

### Sintaksa :

```
atoi( string_varijabla.c_str() );
```

**Uočite** „c\_str()“ dio iza imena varijable. Ovo je jedna od onih stvari koju ćete morati zapamtiti da prilikom korištenja „atio“ funkcije za pretvorbu *string* vrijednosti u *integer* vrijednost morate dodati taj dodatak. Ovaj dodatak je funkcija (raspoznavamo je pomoću otvorene i zatvorene oble zagrade) koju je obavezno potrebno dodati. Razlog tome je što „atoi“ funkcija kao argument prima tzv. „C-style *string*“<sup>7</sup>. Što to točno je nije potrebno znati da bi se koristila „atoi“ funkcija već je potrebno samo zapamtiti da je dodavanje „c\_str()“ funkcije obavezno.

### Primjer 1:

```
int main()
{
    string rijec="123";
    int broj=atoi(rijec.c_str());
    cout << rijec*2;
    cout << broj*2<<endl;
    return 0;
}
```

Pokušamo li ovaj kôd pokrenuti dobit ćemo *sintaksnu* pogrešku jer pokušavamo *string* pomnožiti cijelim brojem što nije moguće. Ovaj primjer služi upravo da biste vidjeli da takva operacija nije dozvoljena te da je za matematičke operacije s cijelim brojevima neophodna „atoi“ funkcija. U primjeru 2 ćemo tu liniju kôda izbrisati i program će se uspješno provesti a kako bi se bolje uočila uspješnost izvedbe našeg kôda prilikom ispisa ćemo vrijednost varijable „broj“ pomnožiti s brojem 2 (dva).

---

<sup>7</sup> Generates a null-terminated sequence of *characters* (*c-string*) with the same content as the *string* object and returns it as a *pointer* to an array of *characters*.

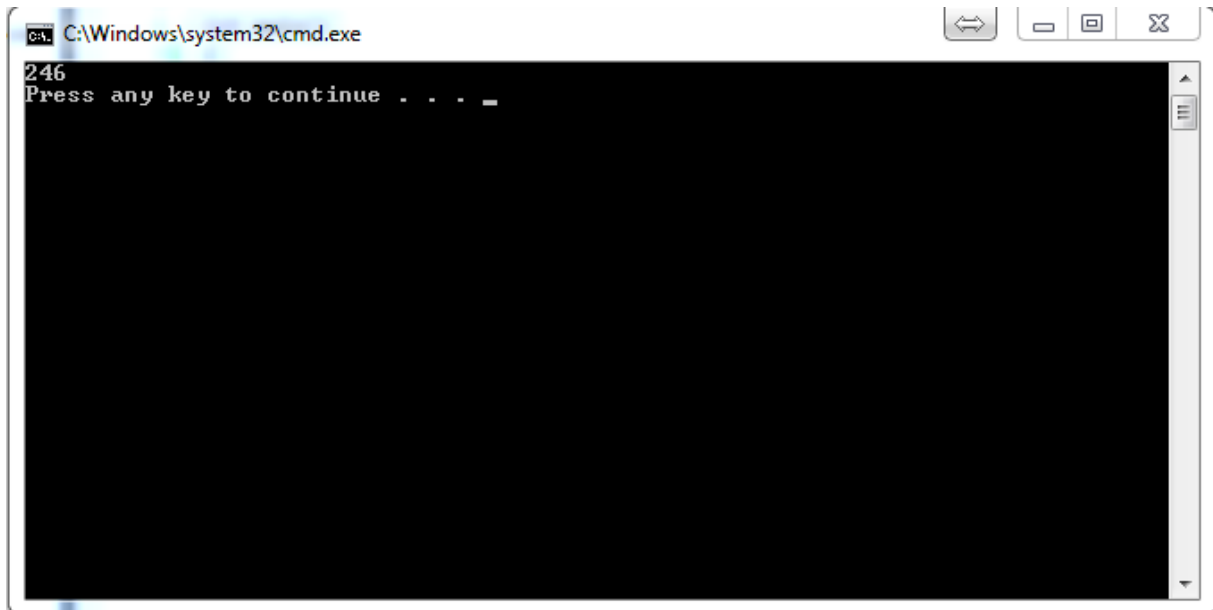
A terminating null *character* is automatically appended.

The returned array *points* to an *internal* location with the required storage space for this sequence of *characters* plus its terminating null-*character*, but the values in this array should not be modified in the program and are only guaranteed to remain unchanged until the next call to a non-constant member function of the *string* object.

### Primjer 2 :

```
int main()
{
    string rijec = "123";
    int broj = atoi(rijec.c_str());

    cout << broj*2 << endl;
    return 0;
}
```



Slika 54 - Rezultat nakon izvršenja kôda u primjeru 2

### g. Slične funkcije za pretvorbu iz *string* tipa u numerički tip – atof(), atol()

Uz već spomenutu „atoi“ funkciju postoje i funkcije „atof“ i „atol“. Funkcija „atoi“ predstavlja pretvorbu u *integer* tip podatka, funkcija „atof“ u *float* tip podatka a funkcija „atol“ u *long integer* tip podatka. *Sintaksa* je identična u sve tri funkcije pa time i kod „atof“ i kod „atol“ funkcije trebamo koristiti funkciju „c\_str()“ kako je opisano u korištenju kod „atoi“ funkcije. Vrijednosti koju vraća „atof“ funkcija je naravno *float* a vrijednost koju vraća „atol“ funkcija je *long integer*.

### h. Funkcija find()

Funkcija „find“ omogućuje pretraživanje unutar nekog *stringa*. Možemo pretraživati jedan znak ili više znakova, a funkcija vraća poziciju prvog pojavljivanja traženog znaka odnosno skupa znakova. Ako funkcija ne pronađe traženi znak ili skup znakova, kao rezultat vraća „-1“.

### i. Funkcija insert()

Funkcija „insert“ omogućuje ubacivanje jednog ili više znakova u već postojeći *string*. Kao argument predajemo lokaciju odnosno indeks na koji želimo umetnuti znak, skup znakova ili neki drugi *string*, te što želimo umetnuti. Funkcija ne vraća ništa.

### j. Funkcija replace()

Kao što samo ime kaže, ova funkcija će omogućiti zamjenu nekih vrijednosti. Kao argumente predajemo indeks znaka koji želimo zamijeniti te s kojom vrijednosti želimo zamijeniti.

### k. Funkcija clear()

Funkcija *clear* će isprazniti cijeli *string* odnosno od *stringa* nad kojim pozivamo funkciju će napraviti prazan *string*. Funkcija ne vraća ništa.

### l. Funkcija empty()

Funkcija „empty“ provjerava da li je neki *string* prazan. Moguće vrijednosti su, logično *True* ili *False*. To znamo jer se radi o funkciji koja vraća *bool* vrijednosti, a znamo da postoje jedino *True* i *False* vrijednosti.

## m. Zadaci

- 1) Napraviti aplikaciju koja pita korisnika za unos imena i prezimena ali sve u jednom retku te vraća vrijednost koliko je znakova unio.
- 2) Napraviti aplikaciju koja pita korisnika za unos dva broja (između 1 i 20) i prema tim brojevima stvara tablicu množenja. Prvi broj simbolizira od kojeg broja da započne tablica množenja a drugi broj simbolizira krajnji broj u tablici množenja. Potrebno je provjeriti da li je korisnik zaista unio samo brojeve i da li su u dozvoljenom rasponu. Ako nisu treba izbaciti pogrešku.
- 3) Napraviti aplikaciju koja pita korisnika za unos imena zaposlenika. Potrebno je unaprijed definirati moguće korisnike (primjerice „Ivan“, „Marko“, „Pero“, „Ivana“, „Maja“ i „Anita“). Aplikacija treba moći prepoznati bez obzira na velika i mala slova ime zaposlenika (unese li korisnik „IVAN“ ili „ivan“ ili bilo koju moguću varijaciju aplikacija treba prepoznati da se radi o zaposleniku imena „Ivan“)

## 25) Izrada vlastitih funkcija

Iako C++ sadrži veliki broj ugrađenih funkcija, vrlo često radimo vlastite funkcije. Ukoliko imamo identičan kôd koji se ponavlja, preporuka je napraviti funkciju koja će izvršavati taj kôd. Na ovaj način ćemo samo pozvati tu funkciju umjesto da imamo nekoliko puta identičan kôd što će rezultirati puno čitljivijim programom i znatno će se olakšati održavanje takvog kôda. Zamislite da morate na deset mjesta mijenjati nešto jer ste naknadno shvatili kako ste pogriješili prilikom izrade nekog dijela koda za izračun. Postavite li taj kôd za računanje unutar funkcije i kasnije na deset mjesta pozivate funkciju, prepravku kôda za izračun ćete morati napraviti samo na jednom mjestu.

Sve funkcije pišemo na identičan način. Prvo navodimo tip podatka koji funkcija vraća (izuzetak je naravno VOID funkcija koja ne vraća ništa), zatim ime funkcije te unutar otvorenih i zatvorenih obliha zagrada navodimo parametre koje funkcija prima. Na kraju, unutar vitičastih zagrada navodimo kôd koji će se izvršiti kada se pozove ta funkcija.

### a. Rekurzija

Unutar jedne funkcije moguće je pozivati druge funkcije ili napraviti da funkcija zove samu sebe. Ukoliko funkcija poziva samu sebe, to se zove **REKURZIJA**. Ideja iza rekurzije je da se funkcija ponavlja, odnosno poziva samu sebe određeni broj puta dok se neki uvjet ne zadovolji, nakon čega se prekida daljnje pozivanje te iste funkcije.

Dobar primjer za proučiti rekurziju je rješavanje mita o „Hanojevim tornjevima“ (*engl. Tower of Hanoi*). Proučite na *internetu* o čemu se radi i probajte riješiti ovaj mit. Uzmite u obzir da korisnik mora sam upisati koliko je koluta postavljeno na početni stup nakon čega aplikacija sama rješava ovu zagonetku. Nemojte pokušavati s velikim brojevima (nema potrebe za više od pet koluta) jer će se aplikacija izvršavati poprilično dugo. Uz malo veće brojeve, možete očekivati da se nekoliko dana vrti aplikacija.

### b. Prva vlastita funkcija

Napraviti ćemo vrlo jednostavnu funkciju za provjeru ispravnosti Email adrese. U ovom primjeru ćemo samo provjeravati sadrži li upisana mail adresa znak „@“. Ako sadrži, smatrat ćemo da je email adresa valjana. Ako ne sadrži znak „@“, onda smatramo da adresa nije valjana.

Unutar „main“ funkcije tražimo od korisnika da unese email adresu te pozivamo funkciju za provjeru ispravnosti email adrese.

```
int main()
{
    string emailAdresaKorisnika;
    cout << "Unesi mail adresu:"; cin >> emailAdresaKorisnika;

    provjeraMailAdrese(emailAdresaKorisnika);

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

Pogledajmo kako je složena funkcija za provjeru ispravnosti email adrese:

```
void provjeraMailAdrese(string mail) {  
    if (mail.find("@") != -1) {  
        cout << "Email adresa je valjana\n";  
    }  
    else {  
        cout << "Email adresa nije valjana\n.";  
    }  
};
```

Funkcija prima jedan argument tipa *string*. Ovaj argument je nazvan „mail“. Provjeravamo sadrži li varijabla „mail“ simbol „@“. Ako sadrži, funkcija „find“ vraća indeks na kojem se znak nalazi, a ako ne postoji, vraća „-1“. Naša provjera je izvedena tako da provjeravamo rezultat koji vraća funkcija „find“ pozvana nad varijablom „mail“. Ako rezultat nije „-1“, smatramo da je mail adresa valjana. Kako znamo da funkcija nije valjana funkcija „find“ vrati „-1“, sve ostalo što vrati nama odgovara u smislu da je mail adresa ispravna.

Ako nije zadovoljen uvjet da funkcija „find“ ne vrati kao rezultat „-1“, ulazimo u „else“ dio gdje se bez ikakve provjere ispisuje da email adresa nije valjana. Ovo možemo napraviti jer u prvom dijelu „provjera s if-om) testiramo jeli email adresa ispravna. Ako ta provjera ne prođe jedino što onda može biti je da je unesena email adresa neispravna.

### c. O izradi vlastitih funkcija

Jako često ćete se naći u situaciji da trebate raditi vlastite funkcije. Pokušajte ih što češće koristiti u situacijama kada imate isti kôd na više od jednog mjesta. Način slaganja funkcija je uvijek isti. Prvo navodimo tip podatka koji funkcija vraća, zatim ime funkcije te parametre koje prima funkcija (u slučaju da funkcija ne prima parametre, unutar obliha zagrada ne upisujemo ništa).

### d. Preopterećenje funkcija

Možda ste već uočili kako postoji više verzija iste funkcije. Primjerice, funkcija „find“ može pretraživati na način da samo predate koji simbol da traži. U tom slučaju će funkcija krenuti s pretraživanjem od nultog indeksa. Drugi način je da kažemo „find“ funkciji od kojeg indeksa da krene. To je sada već druga verzija iste funkcije. Iako se obje funkcije isto zovu i imaju isti tip podataka koji vraćaju (*integer*), razlika je u prametrima.

Prilikom izrade funkcija, mora se poštovati osnovno pravilo da ako imamo funkciju s istim povratnim tipom i imenom, broj, vrsta i raspored parametara ne smiju biti nikada identični u dvije funkcije. Napravite li svoju funkciju „mojaFunkcija“ koja u jednoj verziji ne prima parametre, a u drugom slučaju prima jedna parametar, to je sasvim legitiman način pisanja funkcija. Zamislimo da imamo upravo takav slučaj gdje imamo istu funkciju samo jednom bez ikakvih parametara, a drugi put s jednim *string* parametrom. Prilikom pozivanja funkcije važno je hoćemo li predati *string* argument ili ne. Ako ne predamo nikakav argument, pozvati će se ona funkcija koja nema nikakvih parametara. Suprotno tome, pozovemo li u kodu funkciju kojoj predajemo jedan *string* argument, pozvati će se funkcija koja treba primiti jedan *string* argument.

Isto tako, imamo li funkciju čiji je prvi argument tipa „*int*“ a drugi tipa „*string*“, možemo napraviti funkciju čiji će prvi tip biti „*string*“, a drugi „*int*“. Važno je samo da se nikada ne smije poklopiti točno isti broj, raspored i tip argumenata. Sve ostalo je dozvoljeno.

Kakve funkcije ćete vi sami raditi je isključivo na vama, a ovisiti će o situaciji koju trebate riješiti.



## 26) Objektno orijentirano programiranje

U ovom poglavlju se neće detaljno, pa čak niti površinski obrađivati objektno orijentirano programiranje. Svrha ovog poglavlja je samo spomenuti da ovaj način kôdiranja postoji i istaknuti da je danas, objektno orijentirano programiranje jedan od najzastupljenijih načina pisanja programa.

Objektno orijentirano programiranje (OOP) je jedan od mogućih načina kako pristupiti nekom problemu. Uz C++ često korišteni OOP jezici su Java i C#. Ideja koja leži iza OOP je da radimo s objektima nad kojima možemo izvršavati neke operacije i svaki objekt neke klase zovemo **instancom** te klase. Postupak kreiranja objekta se naziva **instanciranje**. Svaki objekt ima određeni tip podatka (kao i varijable primjerice), a tipove podatka nerijetko sam programer kreira (ne koristi već predifinirane poput *integer*, *Flat*, *String* i slično). Tipovi podataka se zovu **klase** (*engl. class*). Svaka klasa ima funkcije koje se u slučaju korištenja unutar klase zovu metode. Klasa može imati i svoje podklase (*engl. subclass*) koja može naslijediti svojstva (primjerice metode). Ovo svojstvo se zove **nasljeđivanje** (*engl. inheritance*). Uz nasljeđivanje još su tri glavna svojstva OOP-a; **enkapsulacija** i **polimorfizam** i **apstrakcija**. Polimorfizam (*engl. polymorphism*) nam omogućuje da imamo primjerice jednu funkciju koja može primiti više različitih tipova podataka i generalno nam polimorfizam omogućuje ponovno iskorištavanje napisanog kôda. Enkapsulacija (*engl. encapsulation*) nam omogućuje skrivanje dijelova unutar naše klase (metoda ili varijabli) odnosno može naš kôd zaštititi od neželjenog pristupa. Apstrakcija (*engl. abstraction*) nam omogućuje da prikazemo samo određene dijelove naše klase koji nam mogu zatrebati za neki objekt, a ostale dijelove klase možemo sakriti.

OOP je izrazito važno i korisno no potrebno je još upoznati i usvojiti dosta gradiva kako bi se moglo krenuti na učenje OOP-a. Svakako prije učenja OOP treba proučiti što su to **strukture** i **pokazivači**, a preporučeno je upoznati i rad s tekstualnim datotekama (upis i čitanje iz datoteka). Iako rad s tekstualnim datotekama nije striktno povezano s uvodom u OOP, preporuča se radi korištenja funkcija i širenja načina razmišljanja.

**Objektno orijentirano programiranje otvara sasvim novi svijet načina programiranja, prilika i mogućnosti.**

## 27) Popis slika

|  |    |
|--|----|
| Slika 1 - Instalacija Microsoft Visual C++ 2010 Express razvojnog sučelja .....                      | 8  |
| Slika 2 - Početni izbornik nakon otvaranja Microsoft Visual C++ 2010 Express razvojnog sučelja ..... | 9  |
| Slika 3 - Stvaranje novog praznog projekta .....   | 10 |
| Slika 4 - Izgled nakon stvaranja novog praznog projekta .....  | 10 |
| Slika 5 - Dodavanje prazne C++ datoteke (ekstenzija je .cpp).....                                    | 11 |
| Slika 6 - Razvojno sučelje nakon što je dodana prazna C++ datoteka.....                              | 12 |
| Slika 7 - Prva aplikacija u C++ programskom jeziku.....  | 13 |
| Slika 8 - prikaz gumba za pokretanje procesa debugiranja .....                                       | 14 |
| Slika 9 - Konzolna aplikacija - Hello World .....  | 14 |
| Slika 10 - Popis ključnih riječi u jeziku C++ .....  | 21 |
| Slika 11 - Tipovi podataka u C++ programskom jeziku i njihova svojstva .....                         | 24 |
| Slika 12 - Prikaz komentara u editoru .....  | 29 |
| Slika 13 - Rješenje zadatka 03 u dijelu Uvodni zadatci.....  | 31 |
| Slika 14 - Rješenje zadatka 04 u dijelu Uvodni zadatci.....  | 31 |
| Slika 15 - Rješenje zadatka 05 u dijelu Uvodni zadatci.....  | 32 |
| Slika 16 - Rješenje zadatka 06 (prvi način) u dijelu Uvodni zadatci.....                             | 32 |
| Slika 17 - Rješenje zadatka 06 (drugi način) u dijelu Uvodni zadatci.....                            | 33 |
| Slika 18 - Simboli koji se koriste u dijagramu toka .....  | 35 |
| Slika 19 - Dijagram toka pri izračunu površine kuće.....   | 35 |
| Slika 20 - Primjer pseudo kôda i dijagrama toka .....  | 36 |
| Slika 21 - Rezultat nakon izvršenja kôda u primjeru 1.....   | 45 |
| Slika 22 - Rezultat nakon izvršenja kôda u primjeru 2.....   | 46 |
| Slika 23 - Rezultat nakon izvršenja kôda u primjeru 3.....   | 47 |
| Slika 24 - Rezultat nakon izvršenja kôda u primjeru 1.....   | 49 |
| Slika 25 - Rezultat nakon izvršenja kôda u primjeru 2 - beskonačna petlja .....                      | 50 |
| Slika 26 - Rezultat nakon izvršenja kôda u primjeru 1.....   | 52 |
| Slika 27 - Rezultat izvršavanja kôda u primjeru 2 .....  | 53 |
| Slika 28 - Formatiranje ispisa pomoću višestruke "endl" naredbe.....                                 | 60 |
| Slika 29 - Formatiranje ispisa pomoću "\n".....  | 61 |
| Slika 30 - Rezultat nakon izvršavanja kôda u primjeru broj 1 .....                                   | 67 |
| Slika 31 - Rezultat izvršavanja kôda u primjeru korištenja prototipa funkcije .....                  | 75 |
| Slika 32 - Rezultat izvršenja kôda jednostavne funkcije koja vraća integer tip podatka.....          | 79 |
| Slika 33 - Rezultat izvršavanja kôda jednostavne funkcije koja vraća float tip podatka .....         | 80 |
| Slika 34 - Rezultat izvođenja jednostavne funkcije tipa boolean .....                                | 81 |
| Slika 35 - Rezultat izvođenje kôda u void tipu funkcije .....  | 83 |
| Slika 36 - Rezultat izvođenja kôda uz void funkciji.....   | 83 |
| Slika 37 - Rješenje primjera 1 .....   | 86 |
| Slika 38 - Rezultat izvršavanja kôda u primjeru 1.....   | 87 |
| Slika 39 - Rezultat nakon izvršavanja kôda u primjeru 1 .....  | 88 |
| Slika 40 - Rezultat nakon izvođenja kôda u primjeru 2 .....  | 89 |
| Slika 41 - Rezultat nakon izvođenja kôda u primjeru 1 .....  | 92 |
| Slika 42 - Rezultat izvođenja kôda u primjeru 2 .....  | 92 |
| Slika 43 - Rezultat izvođenja kôda u primjeru 1 .....  | 93 |
| Slika 44 - rezultat kôda za unos imena i prezimena u string varijablu .....                          | 96 |
| Slika 45 - Rezultat ispisa vrijednosti iz varijable tipa string.....                                 | 96 |

|  |     |
|--|-----|
| Slika 46 - Rezultat korištenja kôda s getline funkcijom .....  | 97  |
| Slika 47 - Rezultat korištenja kôda s lenght funkcijom .....   | 98  |
| Slika 48 - Ispis svakog slova u riječi "auto" .....  | 99  |
| Slika 49 - Ispis slova u jednoj riječi pomoću for petlje, lenght funkcije i pristupa znakovima stringa preko indeksa ..... | 100 |
| Slika 50 - Rezultat nakon izvršenja kôda u primjeru 3.....   | 101 |
| Slika 51 - ASCII tablica .....   | 102 |
| Slika 52 - izmjena prvog slova u riječi korištenjem funkcije tolower .....   | 103 |
| Slika 53 - korištenje tolower funkcije za pretvorbu slova u "mala" slova .....   | 104 |
| Slika 54 - Rezultat nakon izvršenja kôda u primjeru 2.....   | 106 |

## 28) Popis tablica:

|  |    |
|--|----|
| Tablica 2 - Prikaz Unarnih operatora i njihovih naziva .....                         | 25 |
| Tablica 3 - Popis binarnih operatora i njihovi nazivi.....                           | 26 |
| Tablica 4 - Korištenje binarnih operatora i prikaz rezultata .....                   | 27 |
| Tablica 5 – Popis najčešće korištenih operatora pridruživanja i njihovi nazivi ..... | 27 |
| Tablica 6 - Korištenje operatora pridruživanja i prikaz rezultata .....              | 27 |
| Tablica 7 - Popis operatora uspoređivanja i njihovi nazivi .....                     | 28 |
| Tablica 8 - Simboli logičkih operatora i njihova značenja.....                       | 28 |