Univerzitet Singidunum



C++

Ranko Popović
Zona Kostić

Programski jezik

sa rešenim zadacima

Beograd, 2010.

UNIVERZITET SINGIDUNUM

Prof. dr Ranko Popović
Zona Kostić

PROGRAMSKI JEZIK C++ SA REŠENIM ZADACIMA

Beograd, 2010.

PROGRAMSKI JEZIK C++ SA REŠENIM ZADACIMA

Autori:
Prof. dr Ranko Popović
Zona Kostić

Recenzenti:
Prof. dr Dragan Cvetković
Dr Boško Nikolić, docent

Izdavač:
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32
www.singidunum.ac.rs

Za izdavača:
Prof. dr Milovan Stanišić

Tehnička obrada:
Ranko Popović

Dizajn korica:
Aleksandar Mihajlović

Godina izdanja:
2010.

Tiraž:
250 primeraka

Štampa:
Mladost Grup
Loznica

ISBN: 978-86-7912-286-5

Predgovor

Knjiga ”Programski jezik C++ sa rešenim zadacima”, nastala je kao prateći materijal za kurseve koje su autori držali na Univerzitetu Singidunum (Fakultet za informatiku i računarstvo i Fakultet za menadžment).

Knjiga je orijentisana ka poslovnim aplikacijama i sadrži veliki broj rešenih zadataka sa objašnjenjima.

U dodatku knjige detaljno su opisana Microsoftova integrisana razvojna okruženja Visual Studio C++ verzije 6 i 2008.

Autori su uložili veliki napor da ova knjiga ne sadrži greške. Sve sugestije, korekcije i primedbe su više nego dobrodošle.

Sadržaj

Predgovor ... 3
Sadržaj ... 4
1. Uvod ... 6
1.1 Programski jezik C++ ... 6
1.2 Proširenja programskog jezika C ... 7
1.3 Jednostavan C++ program ... 9
1.4 Celi brojevi i aritmetika ... 16
2. Realni brojevi, iteracije i donošenje odluka ... 23
2.1 Realni brojevi ... 23
2.2 Iteracije (petlje) ... 30
2.3 Donošenje odluke ... 38
2.4 Pravila prioriteta ... 42
3. Koncept funkcije ... 49
3.1 Definicija funkcije ... 49
3.2 Prosleđivanje parametara po vrednosti ... 52
3.3 Promenljive kao atributi ... 54
4. Nizovi ... 57
4.1 Definicija niza ... 57
4.2 Višedimenzionalni nizovi ... 64
5. Pokazivači ... 68
5.1 Deklarisanje i inicijalizacija pokazivača ... 68
5.2 Pokazivači i nizovi ... 71
6. C-stringovi, pokazivači, nizovi, funkcije, korisnički definisani tipovi podataka i tabele ... 76
6.1 Definicija i inicijalizacija stringa ... 76
6.2 String konstante i pokazivači ... 79
6.3 Nizovi stringova i pokazivača ... 83
6.4 Pokazivači, reference na promenljive i funkcije ... 86
6.5 Nizovi i funkcije ... 93
6.6 Stringovi i funkcije ... 98
6.7. Korisnički definisani tipovi podataka i tabele ... 110
6.8 Strukture ... 112
6.9 Nizovi struktura: tabele ... 117
7. Uvod u klase i objekte ... 119
7.1 Objekti, klase i objektno orijentisani sistemi ... 119

7.2 Uvod u string objekte 121
7.3 Donošenje odluka u radu sa stringovima 131
7.4 Funkcije i string objekti 133
7.5 Primeri rada sa string objektima 138
7.6 Nizovi stringova 147
8. Programerski definisane klase i objekti 148
8.1 Deklarisanje objekata i klasa 148
8.2 Konstruktor klase 150
8.3 Preklapanje konstruktora 160
8.4 Destruktori 164
8.5 Opšte preklapanje funkcija i šabloni funkcija 174
9. Rad sa objektima 178
9.1 Korišćenje nizova, pokazivača i dinamička alokacija memorije 178
9.2 Konstruktor kopije 185
9.3 Korišćenje rezervisane reči const u radu sa klasama 197
9.4 Objekti, funkcije i pokazivači 215
9.5 Dinamička alokacija objekata 241
9.6 Statički podaci članovi i funkcije 247
10. Nasleđivanje 256
10.1 Primeri nasleđivanja i osnovna terminologija 256
10.2 Polimorfizam 278
10.3 Apstraktne osnovne klase 297
11. Fajlovi 299
11.1 Ulazno/izlazni tokovi 299
11.2 Procesiranje fajla karakter po karakter 321
11.3 Slučajni pristup fajlu 325
11.4 Procesiranje binarnog fajla sekvencijalno 332
12. Specijalne teme: Prijateljske funkcije, preklapanje operatora, makroi i inline funkcije 338
12.1 Prijateljske (friend) funkcije 338
12.2 Preklapanje osnovnih aritmetičkih operatora 344
12.3 Makroi i inline funkcije 359
Dodatak A: Integrisano razvojno okruženje Microsoft Visual C++ 365
Dodatak B: Integrisano razvojno okruženje Microsoft Visual C++ 2008 . 394
Reference: 412

1. Uvod

1.1 Programski jezik C++

Predmet „Uvod u programiranje II“ uvodi u osnovne koncepte objektno orijentisanog programiranja korišćenjem jezika C++.

C++ jezik je jedan od najpopularnijih programskih jezika korišćenih za razvoj računarskih aplikacija. Programski jezici se mogu svrstati u jezike niskog i visokog nivoa. Jezici niskog nivoa su asembleri. Asembleri dozvoljavaju programeru maksimalnu kontrolu računarskog sistema. Asembleri se teško uče zato što su simbolički i programer mora da vodi računa o svim detaljima programa. Asemblerski programi su, takođe, teški za održavanje i debagovanje. Asemblerski programi nisu portabilni, oni su specifični za hardver i operativni sistem računarske platforme za koju su razvijeni. Na primer, ne može se direktno izvršiti asemblerski programski jezik za PC na Macintosh platformi. Suprotno, jezici visokog nivoa, kao što su COBOL, Visual Basic i Java su lakši za učenje, održavanje i debagovanje u odnosu na asemblere.

C++ je jezik "srednjeg nivoa". Zadržava se kontrola nad računarom kao kod asemblera i zadržavaju se dobre strane jezika "visokog nivoa". C++ je portabilan-prenosiv u smislu da se može izvršiti, sa minimalnim promenama, na više različitih računarskih sistema bez modifikacija. C++ je "mali" programski jezik. On ne sadrži ugrađene karakteristike prisutne u drugim programskim jezicima (npr. Visual Basic, ima oko 150 rezervisanih reči) koje dalje povećavaju portabilnost. C++ ima oko 70 ključnih reči.

C++ je objektno orijentisano proširenje C programskog jezika. Dennis Ritchie je razvio C jezik u Bell-ovim laboratorijama 1972. godine kao pomoć pri pisanju UNIX operativnog sistema za PDP-11 miniračunare. Bjarne Stroustrup je 1980. godine u AT&T Bell laboratorijama razvio C++ jezik. Dok je C primer proceduralnog jezika, C++ vidi problem kao skup objekata (svaki sa svojim osobinama i ponašanjima) koji međusobno interaguju.

Razlozi za uvođenje novog jezika bili su: zahtevi korisnika su se dratično povećali; bilo je neophodno povećati produktivnost programera; ponovna upotreba ranije napisanog koda; drastično povećani troškovi održavanja, ...

Objektno orijentisano programiranje i C ++ kao odgovor daju: apstraktne tipove podataka, enkapsulaciju, nasleđivanje, polimorfizam itd.

Uvođenjem C ++ programskog jezika ne sprečava se pisanje loših programa, već se omogućava pisanje boljih programa i uvodi se drugačiji način razmišljanja u programiranje. Mnogo više vremena se troši na projektovanje nego na samu implementaciju (kodovanje). Razmišlja se najpre o problemu, a tek naknadno o programskom rešenju. Razmišlja se o delovima sistema (objektima) koji nešto rade, a ne o tome kako nešto radi (algoritmi). Pažnja se prebacuje sa realizacije na međusobne veze između delova, gde je cilj smanjenje interakcije između softverskih delova. Kasnije su dodate nove mogućnosti kao što su višestruko nasleđivanje, apstraktne klase, mehanizmi za sastavljanje generičkih klasa i za rukovanje izuzecima (obradu grešaka). ANSI standard za jezik C++ usvojen je 1997. godine.

Knjiga je podeljena na dva dela. U prvom delu, poglavlja od 1 do 6, dati su aspekti C++ jezika koji nisu objektno orijentisani. Ovo daje mogućnost studentima da nauče osnovnu strukturu C++ jezika. U drugom delu knjige, poglavlja od 7 do 12, uvode se objektno orijentisane ideje kroz C++ ugrađene string klase i daju se sve važnije objektno orijentisane konstrukcije koje čine C++ programski jezik jednim od najsnažnijih alata za razvoj objektno orijentisanog softvera. Na kraju knjige kao dodatak dat je kratak uvod u Microsoft Visual C++ okruženje kao integrisano razvojno okruženje grupe Microsoft-ovih prevodilaca različitih programskih jezika. Obrađene su dve verzije okruženja koje je Microsoft razvio: Visual Studio 6.0 i Visual studio 2008.

1.2 Proširenja programskog jezika C

Pored komentara /* i */ u jeziku C++ mogu da se koriste komentari koji počinju sa //. Ovi komentari se završavaju na kraju reda bez posebnog označavanja završetka.

Spisak službenih reči, koje su rezervisane reči i kao takve ne mogu da se koriste kao identifikatori u programima, produžen je za 30 novih reči.

Ne preporučuje se korišćenje identifikatora koji počinju sa dva znaka podvučeno (__). U standardnoj biblioteci funkcija jezika C++ koriste se takvi identifikatori. Treba izbegavati i identifikatore koji počinju jednim znakom podvučeno (_) jer se takvi identifikatori koriste u standardnoj biblioteci funkcija jezika C.

Tip bool spada u grupu celobrojnih tipova, i postoji automatska konverzija između tipa bool i numeričkih tipova. Kada se koristi u izrazima, logička vrednost false se pretvara u celobrojnu vrednost nula, a *true* u jedan.

Ako se promenljivoj tipa bool dodeljuje numerička vrednost, vrednost nula se pretvara u false, a bilo koja nenulta vrednost u *true*.

Znakovne konstante, na primer 'A', u jeziku C++ su tipa char dok u jeziku C su tipa int. To dovodi do toga da se A i 65 različito tretiraju, bez obzira što u slučaju korišćenja ASCII koda imaju istu brojčanu vrednost. U jeziku C te dve vrednosti ne mogu da se razlikuju ni pod kojim uslovima.

U jeziku C++, dodavanjem modifikatora const na početku naredbi za definisanje podataka dobijaju se prave konstante, pod uslovom da su inicijalizatori konstantni izrazi. Tako uvedeni identifikatori predstavljaju simboličke konstante koje mogu da se koriste i na mestima na kojima se izričito traže konstante.

Deklarativne naredbe u jeziku C++ smatraju se izvršnim naredbama i mogu da se koriste bilo gde u programu gde su naredbe dozvoljene, a ne samo na početku pojedinih blokova. Doseg uvedenih identifikatora se proteže od mesta definisanja do kraja bloka unutar kojeg su definisani.

Identifikatori nabrajanja, struktura i unija u jeziku C++ mogu da se koriste kao identifikatori tipa bez službenih reči enum, struct, odnosno union, pod uslovom da su jedinstveni u svojim dosezima.

Pokazivaču na podatke poznatih tipova u jeziku C++ ne može da se dodeli vrednost generičkog pokazivača (void*).

U jeziku C argumenti su prenošeni funkciji isključivo po vrednosti. Da bi neka funkcija mogla da promeni vrednost neke spoljne promenljive, trebalo je preneti pokazivač na tu promenljivu. U jeziku C++ moguć je i prenos po referenci. Referenca ili upućivač je alternativno ime za neki podatak. Reference mogu da budu formalni argumenti i vrednosti funkcije.

Za potrebe dodele memorije u dinamičkoj zoni memorije u vreme izvršavanja programa, u jeziku C++ uvedeni su operatori new i delete. Veličina dodeljenog prostora pomoću operatora new automatski se određuje na osnovu veličine podatka (kod funkcija iz C jezika malloc, calloc i realloc, veličina mora da se navede, obično primenom operatora sizeof).

Za vrlo male funkcije, pozivanje funkcije i povratak iz funkcije može da traje znatno duže od izvršavanja samog sadržaja funkcije. Za takve funkcije je efikasnije da se telo funkcije ugradi neposredno u kôd na svakom mestu pozivanja. To, pored uštede u vremenu, često znači i manji utrošak memorije. Treba dodati modifikator inline na početku definisanja funkcije.

U jeziku C nije bilo moguće definisati dve funkcije sa istim imenom. Mehanizam preklapanja imena funkcija (eng. *function overloading*) u jeziku C++ omogućava da se srodnim funkcijama daju ista imena. Pod preklapanjem imena funkcija podrazumeva se definisanje više funkcija sa

istim modifikatorima. Sve te funkcije mora da se međusobno razlikuju po broju i/ili tipovima argumenata na način koji obezbeđuje njihovu jednoznačnu identifikaciju.

Imenski prostori (namespace) služi za grupisanje globalnih imena u velikim programskim sistemima u više dosega.

Objekat je neko područje u memoriji podataka tokom izvršavanja programa. To može biti eksplicitno definisana promenljiva (globalna ili lokalna), ili privremeni objekat koji nastaje pri izračunavanju izraza. Objekat je primerak nekog tipa (ugrađenog ili klase), ali ne i funkcije.

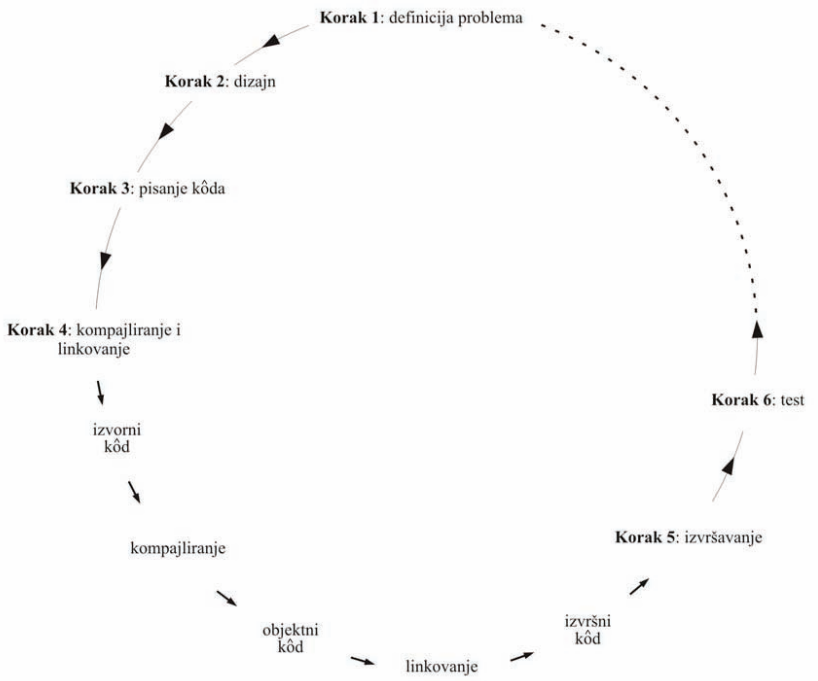
C++ je strogo tipizirani jezik, odnosno svaki objekat ima tačno određeni tip. Kada se očekuje objekat jednog tipa, a koristi objekat drugog tipa, potrebno je izvršiti konverziju tipova.

1.3 Jednostavan C++ program

Ciklus razvoja programa

Da biste razvili program treba da uradite nekoliko koraka. Sledećih šest koraka su tipični kada razvijate C++ program.

Prvi korak: definisanje problema - veoma je važno da razumete problem koji pokušavate da rešite. Budite sigurni da ste pročitali pažljivo specifikacije programa. Treba da znate koji su ulazni podaci i u kojoj formi ti podaci treba da budu. Takođe, treba da znate kako treba procesiranje tih podataka da bude urađeno. Na kraju treba da odredite koje očekivane rezultate i u kakvoj formi dobijate.



Drugi korak: dizajn programa - postoji više načina da se opiše dizajn programa. Dva prosta metoda su dijagram toka i pseudokod. Dijagram toka je grafički opis dizajna programa koji koristi kombinaciju različitih simbola. Pseudokod opisuje dizajn programa izbegavajući sintaksu programskog jezika naglašavajući dizajn rešenja problema.

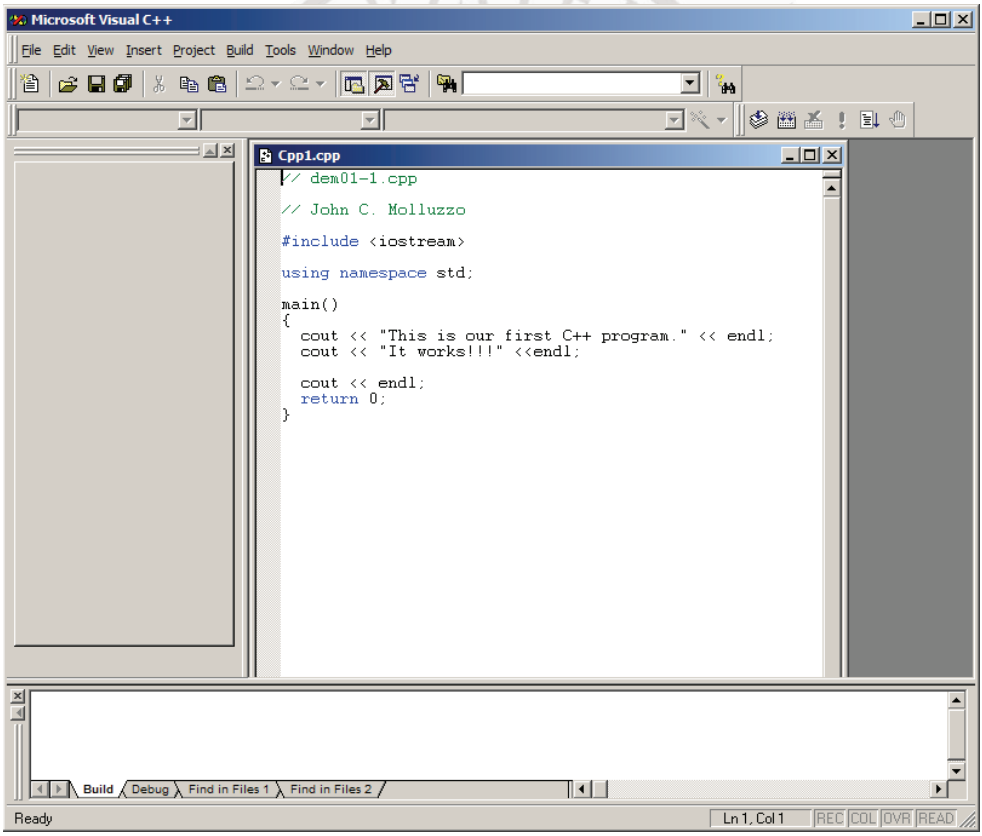
Kada dizajnirate program, važno je da proverite njegovu logiku pre nego što počnete da pišete sam programski kôd. Dizajner programa je često suviše blizak sa dizajnom da bi video neke greške.

Treći korak: pisanje koda - koraci 3, 4, 5 i 6 - se izvršavaju u integrisanom razvojnom okruženju (eng. *Integrated Development Environment*, IDE). Primeri ovih okruženja su: Visual Studio.NET, Borland Enterprise Studio (windows platforma), Code Forge (Linux okruženje), KDevelop (Unix okruženje).

Pisanje koda programa u tekst editoru odgovarajućeg programskog jezika, kao i kompajliranje, odnosno prevođenje izvornog programa ili sors koda (eng. *source*) su sledeći koraci.

Korak 4: kompajliranje (prevođenje) i linkovanje (povezivanje) programa - jedini jezik koji dati računar razume je njegov sopstveni mašinski jezik. Instrukcije mašinskog jezika su binarno kodirane instrukcije

(sastavljene od nula i jedinica) koje kažu računaru da treba da uradi određeni zadatak, kao što je add (dodaj) 1 u registar. Svaki računar ima svoj sopstveni mašinski jezik koji se međusobno razlikuju. Kako je mašinski jezik binarni, teško je pisati kod i pronaći greške u mašinskom programskom jeziku. Pošto računar razume samo svoj sopstveni mašinski jezik, on ne može da direktno izvršava instrukcije programskih jezika srednjeg i visokog nivoa. Računar koristeći program koji se zove kompajler (prevodilac), treba da prevede programe programskih jezika u ekvivalentne programe mašinskog jezika. PC bazirani C++ kompajler prevodi C++ u PC mašinski jezik. Kompajler ne može da prevede različite jezike (kao što je Java) i ne može se koristiti na različitim tipovima računara (kao što je Macintosh).



Kompajler ima dve glavne funkcije. Prvo, on proverava sintaksne greške sors programa. Sintaksne greške su greške u pravopisu-gramatici

programskog jezika. Ako kompajler ne nađe fatalne greške, on prevodi svaku instrukciju jezika visokog nivoa u jednu ili više instrukcija mašinskog jezika. Ova verzija sors programa na mašinskom jeziku se zove objektni program.

Mada je na mašinskom jeziku, objektni program nije spreman za izvršenje. Program može da sadrži reference, nazvane spoljne reference, prema drugim programima koje je korisnik napisao ili koje C++ kompajler obezbeđuje na korišćenje svim korisnicima sistema. U oba slučaja treba da koristite program koji se zove linker da bi razrešili ove spoljne reference. Linker nalazi druge programe koji su deo vašeg programa. Ovaj proces je linkovanje-povezivanje. Linker proizvodi izvršivi kôd (executable code). Ovo je program koji računar izvršava.

Korak 5: Izvršavanje programa - kada se program kompajlira i linkuje on može i da se izvrši preko ikone, menija ili komandne linije.

Korak 6: testiranje i debagovanje programa - čak i kada se program izvršava ne znači da je korektan-ispravan. Program treba testirati pod raznim uslovima i sa raznim podacima.

Greške u ciklusu

Greške koje se mogu pojaviti u različitim koracima su: sintaksne greške i upozorenja (eng. *wornings*), greške prilikom izvršavanja (eng. *run-time*) i logičke greške.

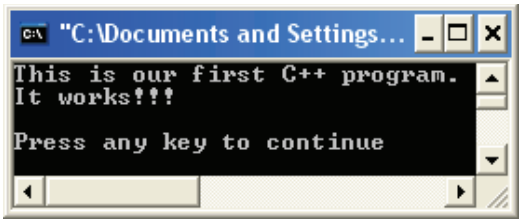
Sledi kôd prvog programa prikaz izlaza na displeju.

```
// Vas prvi program u C++ koji ispisuje na monitoru (konzolni izlaz) tekstaulne
//poruke

#include <iostream>
using namespace std;

int main()
{
  cout << "This is our first C++ program." << endl;
  cout << "It works!!!" <<endl;

  cout << endl;
  return 0;
}
```



Prve dve linije koda su komentari. Komentar počinje sa dva karaktera // i nastavlja do kraja linije. Komentar nije izvršna naredba, sledi, kompajler ignoriše sve od // do kraja linije.

Prazne linije ili blank možete da postavite bilo gde u programu kako bi povećali preglednost programa.

C++ program je kolekcija funkcija koje rade zajedno u cilju rešenja problema. Svaki program mora da sadrži funkciju main(). Ova funkcija kontroliše C++ program izvršavajući C++ naredbe i druge funkcije. Reč int koja prethodi reči main() kaže C++ kompajleru da main() proizvodi (vraća) celobrojnu vrednost.

Svaki postupak se naziva izraz (eng. *expression*). Izraz koji se završava znakom tačka-zapeta ";" naziva se naredba (eng. *statement*). Zanemarivanje znaka ; izaziva sintaksnu grešku pri kompajliranju. Kada biste napisali glavni program na sledeći način, kompajler ne bi prijavio sintaksnu grešku.

```
int main(){ cout << "This is our first C++ program." << endl; cout << "It works!!!" <<endl;
cout << endl; return 0;}
```

C++ je case sensitive, tj. kada pišemo C++ naredbe postoji razlika između velikih slova i malih slova. Na primer: int Main() je nepravilno, treba da stoji int main().

Izlaz na monitor je tok podataka (eng. *data stream*), odnosno niz karaktera. Tok cout (konzolni izlaz "console output" i čita se kao "see-out") predstavlja standardni izlazni tok, koji se automatski prikazuje na korisnikovom monitoru kada se C++ program izvršava. Podaci se mogu poslati na izlazni tok cout koristeći operator umetanja (eng. *insertion operator*), <<, koji se čita kao "poslato je" ili "uzima se". String (niz karaktera) se postavlja unutar znaka navoda.

Završetak linije i skok u novi red se ostvaruje pomoću endl (eng. *end of line*) i odgovara specijalnoj sekvenci '\n'.

Poslednja naredba return završava izvršenje funkcije main() i šalje ceo broj čija je vrednost 0 operativnom sistemu da bi pokazala da se program završio normalno.

Kompajliranje C++ programa se izvodi u dva koraka. Prvo, program nazvan pretprocesor analizira C++ sors kod i izvršava sve pretprocesorske direktive. Svaka direktiva počinje karakterom #. Drugi korak kompajliranja je prevođenje programa u mašinski kod.

Direktiva #include kao rezultat ima zamenu linije koja sadrži direktivu #include tekstom sadržaja heder (eng. *header*) fajla koji je unutar zagrada < >, tako da on postaje deo vašeg programa i kao takav se prevodi sa naredbama vašeg programa.

Heder fajlovi obično sadrže informacije o konstantama, funkcijama, klasama i drugim C++ elementima koji se često pojavljuju u C++ programima. Heder fajl iostream (*input/output stream*), je jedan od standardnih skupova sistemskih heder fajlova kojim je svaki od C++ kompajlera snabdeven. Heder fajl iostream sadrži informacije o standardnim ulazno/izlaznim tokovima.

Treba obratiti pažnju da se pretprocesorska direktiva

```
#include <iostream>
```

ne završava sa ; zato što pretprocesorska direktiva nije naredba.

Mehanizam prostora imena *namespace* (using namespace std)

Kompajler C++ jezika koristi više fajlova da bi napravio C++ program koji se može izvršiti. Uz izvorni fajl, koji je program koji piše programer, kompajler koristi jedan ili više heder fajlova, kao što je iostream i biblioteke standardnih i korisnički definisanih funkcija za kreiranje izvršnog programa. Kako je moguće da više programera učestvuju u pisanju koda, moguće je da koriste ista imena za potpuno različite stvari. Ovo dovodi do kompajlerske greške. Da bi se to izbeglo, C++ obezbeđuje mehanizam prostora imena namespace, koji deli imena koje programeri mogu da koriste na dosege imena. Direktiva

```
using namespace std;
```

dozvoljava da koristimo u našim programima cout (i kasnije cin) bez potrebe da pišemo kôd kao std::cout (i std::cin). Direktiva using u osnovi kaže

da možemo direktno da koristimo sva imena unutar std prostora imena (cout i cin su dva takva imena).

Sva standardom predviđena zaglavlja sve globalne identifikatore stavljaju u imenski prostor std. Standardom je predviđena mogućnost da uskladištavanje standardnih zaglavlja ne bude (mada može) u vidu tekstualnih datoteka. Zbog toga imena standardnih zaglavlja ne sadrže proširenja imena .h, što je uobičajeno u jeziku C.

Rezervisane reči

Do sada smo koristili rezervisane reči (ključne, službene reči ili eng. *keywords*) kao što su int, using, namespace i return. One imaju posebno značenje i predstavljene su u tri grupe:

Prva grupa 32 službene reči iz C jezika:

auto	const	double	float	int	short	struct	unsigned
break	continue	else	for	long	signed	switch	void
case	default	enum	goto	register	sizeof	typedef	volatile
char	do	extern	if	return	static	union	while

Sledeća grupa 30 novih službenih reči:

asm	dynamic_cast	namespace	reinterpret_cast	try
bool	explicit	new	static_cast	typeid
catch	false	operator	template	typename
class	friend	private	this	using
const_cast	inline	public	throw	virtual
delete	mutable	protected	true	wchar_t

i poslednja grupa 11 službenih reči koje se najčešće koriste za rad sa operatorima:

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

Specijalne sekvence (escape sequences)

Koristeći endl kontrolišemo prostor između linija. Prostor između linija možemo da kontrolišemo koristeći specijalnu sekvencu-simbol za novi red. Svaka specijalna sekvenca počinje obrnutom kosom crtom (*backslash character* \). Specijalna sekvenca koja postavlja kursor na početak novog

reda na ekranu monitora je \n (eng. *newline character*). Sledeće dve linije koda proizvode isti efekat kao u prethodnom primeru:

```
cout << "This is our first C++ program.\n";
cout << "It works!!!\n";
```

Postoji veliki broj specijalnih simbola koji se mogu koristiti u stringu koji se šalje na cout. Simboli treba da budu u stringu unutar znaka navoda. Sledi spisak specijalnih sekvenci-simbola:

\a	upozorenje
\b	*backspace*
\f	*formfeed*
\n	novi red
\r	*carriage return*
\t	horizontalni tabulator
\v	vertikalni tabulator
\\	obrnuta kosa crta
\?	znak pitanja
\'	jednostruki znaci navoda
\"	dvostruki znaci navoda
\000	oktalni broj
\xhh	heksadecimalni broj

1.4 Celi brojevi i aritmetika

Identifikatori

Svi programski jezici, uključujući i C++, zahtevaju da programer da imena različitim uzorcima (promenljive, funkcije ...) koje program koristi. To su identifikatori. Sastoje se od maksimalno 31 karaktera, koji mogu biti slova (mala i velika), brojevi i donja crta (_). Ne mogu da počnu brojem i ne mogu da budu istog oblika kao rezervisane reči.

Promenljiva je ime koje upućuje na lokaciju u glavnoj memoriji računara u kojoj se skladišti poseban tip podatka. U C++ programu definišemo promenljivu u deklaraciji. Deklaracija je iskaz koji uvodi neko ime u program i govori prevodiocu kojoj jezičkoj kategoriji pripada neko ime i šta sa tim imenom može da radi. Deklaracija promenljive počinje tipom podatka varijable. Postoji više tipova podataka, uključujući ugrađene

(ceo broj int, znak char i realni broj float i double) i korisnički definisane tipove. Iza tipa podatka dolazi ime promenljive i završni znak naredbe (tačka zarez ";").

Na primer, ovo je deklaracija promenljive:

```
int i1;
```

U ovoj deklaraciji, rezervisana reč int znači da je promenljiva sa imenom i1 tipa integer tj. ceo broj. Zavisno od platforme (platforma se sastoji od računarskog hardvera, operativnog sistema i C++ kompajlera), ceo broj može biti predstavljen sa dva bajta (16 bits, 2^{16} = 65536 celih brojeva) ili sa četiri bajta (32 bits). Kod računara koji skladište cele brojeve u četiri bajta, opseg int variabli je isti kao za tip long. Da bismo odredili koliko bajtova računar koristi za skladištenje promenljivih različitig tipa, koristimo operator sizeof.

Promenljiva može biti deklarisana bilo gde u programu pre njene upotrebe.

Vrednost promenljive je podatak uskladišten u određenoj lokaciji u memoriji računara. Deklarisanje promenljive ne daje promenljivoj određenu vrednost. Deklarisanje samo rezerviše prostor u memoriji za varijablu i pridružuje deklarisano ime toj lokaciji u memoriji. Pre nego što program smesti podatke u promenljive, vrednost promenljive nije važna za program. U ovom slučaju, kažemo da varijabla sadrži "đubre" (eng. *garbage*).

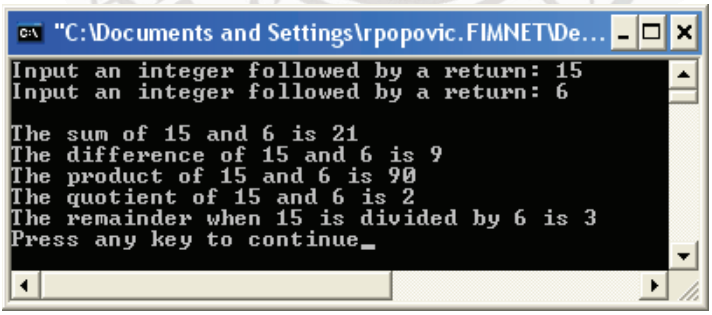
Sledeći program pita korisnika da ukuca ulazne vrednosti za cele brojeve, odradi određenu aritmetiku i prikaže rezultat na displeju. Program sadrži sledeće strukture: deklariše varijable, dobija podatke od korisnika, izvršava zahtevane aritmetičke operacije sa podacima i prikazuje rezultate.

```
// Ovaj program prikazuje koriscenje celih brojeva u C++.
// Korisnik treba da ukuca ulazne vrednosti za 2 cela broja.
// Rezultati aritmetickih operacija se prikazuju na izlazu.

#include <iostream>
using namespace std;

int main()
{
  // Deklaracija promenljivih
  int i1,
      i2,
      sum,
```

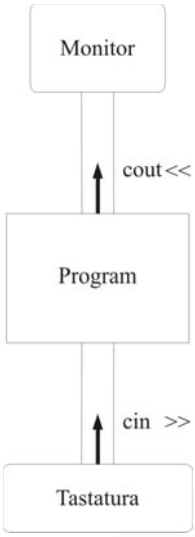
```
    difference,
    product,
    quotient,
    remainder;
  // Dobijanje podataka od korisnika
  cout << "Input an integer followed by a return: ";
  cin  >> i1;
  cout << "Input an integer followed by a return: ";
  cin  >> i2;
// Izvrsavanje aritmetickih operacija
  sum = i1 + i2;
  difference = i1 - i2;
  product = i1 * i2;
  quotient = i1 / i2;
  remainder = i1 % i2;
  // Izlazni rezultati
  cout << endl;
  cout << "The sum of " << i1 << " and " << i2 << " is " << sum << endl;
  cout << "The difference of " << i1 << " and " << i2 << " is "
       << difference << endl;
  cout << "The product of " << i1 << " and " << i2 << " is "
       << product << endl;
  cout << "The quotient of " << i1 << " and " << i2 << " is "
       << quotient << endl;
  cout << "The remainder when " << i1 << " is divided by " << i2
       << " is " << remainder << endl;
  return 0;
}
```



Ulazni tok cin

Ulazni tok cin (konzolni ulaz) dobija sledeći podatak sa tastature i smešta ga u varijablu posle koje sledi operator "izvlačenja" (*extraction*

operator, >>). Naredba `cin >> i1;` uzima ceo broj koji korisnik ukuca na tastaturi kao odziv na prompt i smešta vrednost celog broja u varijablu `i1`.



Celi brojevi koje korisnik ukuca na tastaturi se ne šalju odmah programu na procesiranje. Umesto toga računar smesti karaktere u privremenu lokaciju za skladištenje zvanu ulazni bafer. Računar šalje ove karaktere programu kada korisnik pritisne taster. Ovo nazivamo pražnjenje ulaznog bafera (eng. *flushing the input buffer*).

Operator dodele ili pridruženja (eng. *assignment operator*) =, radi na sledeći način: računar razvija izraz na desnoj strani operatora =, i dodeljuje rezultujuću vrednost varijabli na levoj strani znaka =. Ime varijable mora da bude na levoj strani operatora dodele. Izraz na desnoj strani je celobrojna konstanta, koja može biti pozitivna, negativna ili jednaka nuli.

Aritmetički operatori

Izraz na desnoj strani operatora dodele uključuje jedan ili više aritmetičkih operatora. Program koristi 5 osnovnih aritmetičkih operatora: sabiranje (+), oduzimanje (-), proizvod (*), deljenje (/), i ostatak ili modul (%), čiji su prioriteti i redosledi izračunavanja dati u sledećoj tabeli:

Operatori	Asocijativnost
()	sa leva na desno
+ (unarni) - (unarni)	sa desna na levo
* / %	sa leva na desno
+ -	sa leva na desno
=	sa desna na levo

Rešavanje problema sa celim brojevima

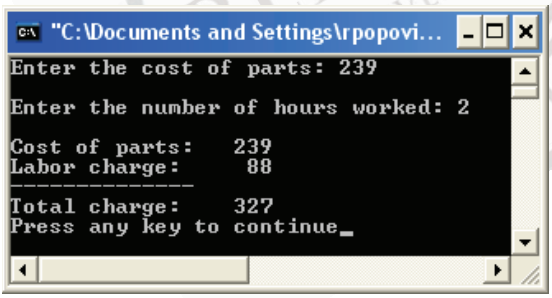
```
// Napisati program koji izracunava ukupnu cenu delova i cenu rada

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

```
int main()
{
  const int HOURLY_RATE = 44;
  int parts,              // Cena delova
      hours,              // Broj radnih sati
      labor,              // Cena rada
      total;              // Ukupna cena za korisnika
  cout << "Enter the cost of parts: ";
  cin >> parts;
  cout << endl;
  cout << "Enter the number of hours worked: ";
  cin >> hours;
  cout << endl;

  // Trazena izracunavanja
  labor = HOURLY_RATE * hours;
  total = parts + labor;

  // Prikaz rezultata
  cout << "Cost of parts: " << setw(5) << parts << endl;
  cout << "Labor charge: " << setw(6) << labor << endl;
  cout << "--------------" << endl;
  cout << "Total charge: " << setw(6) << total << endl;
  return 0;
}
```



Naredba cout sadrži novu karakteristiku, setw() I/O manipulator. I/O manipulator je poruka na koji način izlazni tok treba da bude prikazan na displeju. Manipulator setw() postavlja širinu polja displeja sledećeg prikaza izlaza. Izlaz je desno poravnat, na primer:

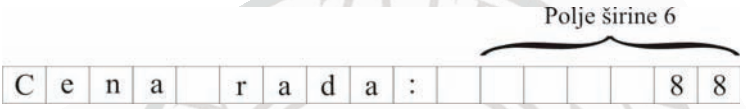
```
cout << "Cena delova: " << setw(5) << parts << endl;
```

prvo prikazuje string "Cena delova: " ("Cost of parts: ") sa blanko pozicijom nakon dvotačke. Zatim, manipulator setw(5) kaže cout-u da smesti sledeći uzorak u polje širine 5. Na taj način variabla parts, čija je vrednost 239, je prikazana u poslednje tri pozicije od pet pozicija polja kao na donjoj slici:



Za drugu naredbu:

```
cout << "Cena rada: " << setw(6) << labor << endl;
```



Da bi se iskoristio manipulator setw(), treba da smestite #include <iomanip> na početak vašeg programa.

Na kraju ove sekcije treba da iskoristite operator sizeof da bi pokazali tačno koliko bajtova se zahteva za skladištenje tipova podataka koje koristimo, pri čemu važi: sizeof(short) ≤ sizeof(int) ≤ sizeof(long).

Ako aritmetički izraz sadrži podatke različitog tipa, da bi se odradile operacije, C++ konvertuje sve vrednosti u tip koji je najopštiji, tj. u tip za koji je predviđeno u memoriji najviše bajtova.

Na primer, pretpostavimo sledeće deklaracije:

```
short k = 3;
int i = 8;
long j = 56;
```

Da bi razvio izraz, i + j + k, C++ konvertuje vrednost promenljive i, koja je tipa int, u ekvivalent long, zato što tip long može da uskladišti veće vrednosti od int. Zatim, C++ konvertuje vrednost k, koja je tipa short, u ekvivalent long. Konačno, C++ dodaje rezultujuću vrednost promenljivoj j.

Prethodna naredba će prouzrokovati da kompajler izda upozorenje (eng. *warning message*).

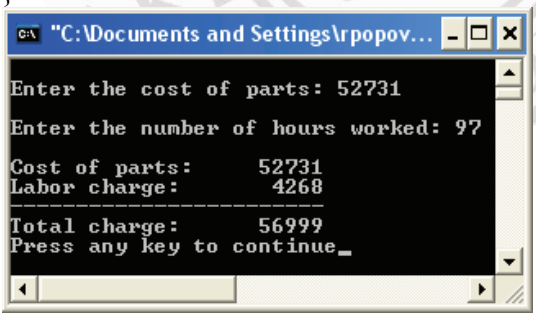
```
// Ovaj program izracunava ukupne naknade za delove i rad

#include <iostream>
```

```
using namespace std;

int main()
{
  const int HOURLY_RATE = 44;
  long parts,           // cena delova
       hours,           // broj radnih sati
       labor,           // cena rada
       total;           // ukupna cena za korisnika
  // Unos ulaznih podataka
  cout << endl;
  cout << "Enter the cost of parts: ";
  cin >> parts;
  cout << endl;
  cout << "Enter the number of hours worked: ";
  cin >> hours;
  // Izracunavanje
  labor = HOURLY_RATE * hours;
  total = parts + labor;

  // Prikaz rezultata
  cout << endl;
  cout << "Cost of parts: " << setw(9) << parts << endl;
  cout << "Labor charge: " << setw(10) << labor << endl;
  cout << "-------------------------"          << endl;
  cout << "Total charge: " << setw(10) << total << endl;
  return 0;
}
```



2. Realni brojevi, iteracije i donošenje odluka

2.1 Realni brojevi

Realni broj, ili *floating point*, je broj koji ima decimalni deo, tj. decimalnu tačku. Postoje dva tipa *floating point* C++ podataka float i double. Sledeće deklaracije definišu varijablu fl koja je tipa float i varijablu db koja je tipa double.

```
float  fl;
double db;
```

Dva *floating point* tipa se razlikuju u preciznosti (broj značajnih cifara koje svaki tip skladišti) i količini memorije koju svaki tip zahteva. Na većini računara, promenljiva tipa float je veličine 4 bajta i ima preciznost od 7 bitova. Promenljiva tipa double okupira 8 bajtova i ima preciznost 15 cifara.

```
C:\...\Primeri\prb01-1.cpp *
#include <iostream>
using namespace std;
int main()
{
  cout << "Na ovom racunaru broj bajtova odvojenih za tip float je
       << sizeof(float) << endl;
  cout << "On this computer a double occupies this many bytes "
       << sizeof(double) << endl;
  return 0;
}
```

```
"C:\Documents and Settings\rpopovic.FIMNET\Desktop\Knjiga 1\Primer...
Na ovom racunaru broj bajtova odvojenih za tip float je: 4
Na ovom racunaru tip double zahteva sledeci broj bajtova: 8
Press any key to continue_
```

Floating point promenljive mogu da se pojave u aritmetičkim izrazima na isti način kao i celobrojne varijable. Na primer, posmatrajmo sledeće deklaracije i dodele:

```
double db1 = 3.14159,
       db2 = -83.01267,
       db3;
db3 = 2 * db1 + db2;
```

Operator dodele daje varijabli db3 rezultat sa desne strane 76.72949. Izraz sa desne strane operatora dodele sadrži različite tipove podataka. Pre nego što uradi aritmetičke operacije, računar automatski konvertuje celobrojnu vrednost 2 u realan broj 2.0. Zatim računar množi vrednost db1 sa 2.0, što daje 6.28318. Konačno, računar dodaje ovaj rezultat vrednosti db2, što daje 76.72949. Računar tada izvršava operaciju dodele vrednosti koja je na desnoj strani, promenljivoj db3 na levoj strani.

Tip float ima preciznost samo sedam cifara, pa se ne može tačno raditi sa vrednostima većim od 99.999,99. Zbog toga što većina modernih poslovnih transakcija uključuje brojeve veće od 100.000,00 evra, tip float je neadekvatan.

Problemi prilikom prikaza na displeju decimalnih brojeva koristeći cout

Objasnićemo zbog čega se uključuju sledeće tri linije koda.

```
cout << setprecision(2)
     << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint);
```

Ponekad cout ne prikazuje na displeju broj u obliku koji se očekuje. Na primer:

```
cout << 345.60 << endl;
```

prikazuje na displeju broj 345.6, bez završne nule.

Drugi problem sa cout je što se češće prikazuje broj u naučnoj notaciji nego decimalnoj notaciji. Na primer, cout naredba

```
cout << 3456789.70 / 2 << endl;
```

prikazuje rezultat deljenja kao 1.72839 e + 006. Ovaj oblik ne odgovora izlaznom zapisu kod poslovnih aplikacija. Umesto toga, želeli bismo da prikaz na izlaznom displeju bude oblika 1728394.85.

Konačno, trebalo bi da kontrolišemo broj cifara sa desne strane decimalne tačke. Na primer, ako izračunamo porez na promet kao 8.25% od količine 19.95, želeli bismo da cout prikaže na displeju proizvod 0.0825 * 19.95 do najmanje vrednosti izražene u centima ili dinarima.

```
cout << 0.0825 * 19.95
```

Međutim, cout prikazuje proizvod kao 1.64588.

Problemi: zanemarivanja završnih nula, prebacivanja na naučnu notaciju i kontrola broja cifara sa desne strane decimalne tačke su međusobno povezani. Rešenje ovih problema je korišćenje I/O (Input/Output) manipulatora. Podsetimo se da o I/O manipulatoru možete da razmišljate kao o cout strimu koji prikazuje brojeve na određen način. Manipulatori koje koristimo su: setprecision(), koji postavlja broj cifara sa desne strane decimalne tačke, i setiosflags(), koji kontroliše različita postavljanja za cout. Mora da uključimo heder fajl iomanip koristeći pretprocesorsku direktivu #include <iomanip>. Ubacujemo sledeće naredbe na početak funkcije main() bilo kog programa:

```
cout << setprecision(2)                // dva decimalna mesta
     << setiosflags(ios::fixed)        // prikaz u fiksnoj notaciji
     << setiosflags(ios::showpoint);   // stampanje zavrsnih nula
```

Manipulator setiosflags() ima složenu sintaksu. U zagradama nakon imena manipulatora setiosflags nalazi se ime klase ios. Nakon toga slede dve dvotačke :: i ime zastavice (eng. *flag name*). Zastavica fixed kaže cout strimu da realni brojevi treba da se prikažu kao u formatu sa fiksnom tačkom a ne u naučnoj notaciji. Zastavica showpoint kaže cout strimu da štampa završne nule. Ovako definisan efekat ostaje sve dok ga ne promenimo šaljući druge vrednosti setiosflags() cout-u.

Ako se koristi setprecision() bez korišćenja setiosflags(ios::fixed), onda setprecision() određuje ukupan broj prikazanih cifara, ne samo broj cifara posle decimalne tačke.

Razmotrimo sledeći kôd:

```
cout << setprecision(2)
     << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint);
cout << 3456789.70 / 2 << endl;
cout << 0.0825 * 19.95 << endl;
cout << 356.70 << endl;
```

Kao izlaz dobijamo:

1728394.85
1.65
356.70

Treba obratiti pažnju da ako je vrednost promenljive db 356.091, prikaz sa jednom decimalom ne utiče na vrednost koja je uskladištena u db. Stvarna vrednost uskladištena u db je još uvek 356.091.

Napomena, preciznost i ios zastavice važe sve dok se ne resetuju. Da bismo promenili preciznost, prosto koristimo setprecision() ponovo sa različitom vrednošću. Da bismo resetovali ili ios::fixed ili ios::showpoint *flag*, treba da postavimo resetiosflags(ios::fixed) ili resetiosflags(ios::showpoint) kod cout.

```
// Ovaj program izracunava porez na promet i cenu stavke

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

int main()
{
 const double TAX_RATE = 0.0825;
 double base_price,   // cena jednog dela
       tax,           // porez
       price;         // prodajna cena (price + tax)

 //definisanje izlaza

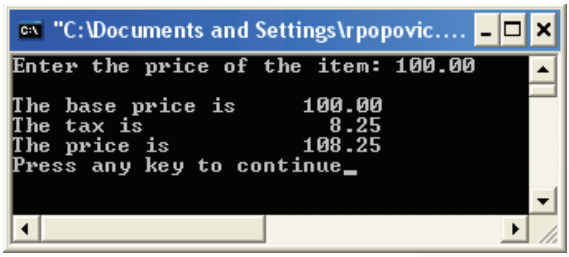
 cout << setprecision(2)
     << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint);

 cout << "Enter the price of the item: ";
 cin >> base_price;

 tax = base_price * TAX_RATE;
 price = base_price + tax;

 cout << endl;
 cout << "The base price is" << setw(11) << base_price << endl;
 cout << "The tax is" << setw(18) << tax << endl;
 cout << "The price is" << setw(16) << price << endl;
```

```
 return 0;
}
```



Rešavanje problema sa realnim brojevima

```
// Ovaj program simulira jednostavnu kasu

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

int main()
{
 const double SALES_TAX_RATE = 0.0825;
 double meal_price,     // cena obroka koju je uneo korisnik
        sales_tax,     // iznos poreza na promet
        total,     // ukupno: meal_price + sales_tax
        amt_tendered,     // iznos dobijen od korisnika
        change;     // kusur: amt_tendered - total

 // Definisanje izlaza
 cout << setprecision(2)
     << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint);

 // Prikaz banera i dobijanje cene
 cout << "*** C++-Side Restaurant ***" << endl << endl;
 cout << "Enter the price of the meal: $";
 cin >> meal_price;
 cout << endl;

 // Izracunavanje poreza i ukupne cene
 sales_tax = meal_price * SALES_TAX_RATE;
 total = meal_price + sales_tax;

 // Prikaz poreza ukupne cene
```

```
  cout << endl;
  cout << "Price of Meal: " << setw(6) << meal_price << endl;
  cout << "Sales Tax: " << setw(10) << sales_tax << endl;
  cout << "---------------------" << endl;
  cout << "Total Amount: " << setw(7) << total << endl;

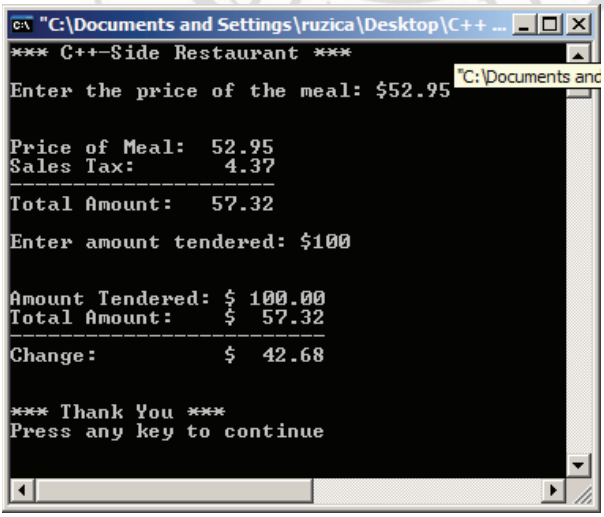
  // Dobijanje novca za naplatu
  cout << endl;
  cout << "Enter amount tendered: $";
  cin >> amt_tendered;
  cout << endl;

  // Izracunavanje kusura
  change = amt_tendered - total;

  // Prikaz količina i kusura
  cout << endl;
  cout << "Amount Tendered: $" << setw(7) << amt_tendered
       << endl;
  cout << "Total Amount:    $" << setw(7) << total << endl;
  cout << "-------------------------" << endl;
  cout << "Change:          $" << setw(7) << change << endl;

  // Stampanje zavrsnog banera racuna
  cout << endl << endl;
  cout << "*** Thank You ***" << endl;

  return 0;
}
```



Kada aritmetički izraz sadrži više promenljivih tipa int ili long i više promenljivih tipa float ili double, C++ automatski konvertuje sve promenljive u tip double pre nego što izvrši aritmetičke operacije. Sledi, rezultat je tipa double.

Tip cast (nakon koga sledi izraz u malim zagradama) menja tip podatka – vrednosti koja je uključena u određeni tip, na primer

```
int i = 5;
double(i);
```

kastuje-menja vrednost promenljive i, odnosno konvertuje vrednost i u double 5.0.

Kada se tip podatka sa desne strane znaka dodele razlikuje od tipa podatka promenljive sa leve strane, C++ konvertuje vrednost desne strane u tip podatka varijable sa leve strane.

Primeri iz aritmetike

Svaki izraz u C++ ima numeričku vrednost.

```
int h = 17;
        i,
        j,
        k;
i = j = k = h;

ili

sum = (count = 0);
sum = 0;
```

Složeni operatori dodele prikazani su: +=, -=, *= i /=.

Operatori	Asocijativnost
()	sa leva na desno
+ (unarni) - (unarni)	sa desna na levo
* / %	sa leva na desno
+ -	sa leva na desno
= += -= *= /=	sa desna na levo

Operatori uvećanja i umanjenja (eng. *increment*, *decrement*) povećavaju ili smanjuju vrednost promenljive za jedan. Mogu da se koriste pre ili posle varijable sa prefiksom pre ili post. To su unarni operatori. Na donjoj slici je dat primer.

	Izraz	
	j = 4 + -- i	**j = 4 + i --**
Inicijalna vrednost i je 7		
Upotrebljena vrednost i	6	7
Dodeljena vrednost j	10	11
Konačna vrednost i	6	6

2.2 Iteracije (petlje)

Relacioni uslovi

Razmotrićemo sledeće iteracije, odnosno petlje (while, do i for).

Najprostiji tip uslova je relacioni uslov, koji upoređuje vrednosti dva operanda, koristeći relacione operatore: ==, !=, <, <=, >, >=.

U jeziku C++, ako je relacioni uslov *true* (tačno), ceo uslov ima numeričku vrednost 1. Ako je relacioni uslov *false* (netačno), ceo uslov ima numeričku vrednost 0. Sledi, C++ uzima svaku vrednost različitu od nule kao *true*, a vrednost 0 kao *false*.

Primer: koja vrednost promenljive rezult će biti odštampana na displeju koristeći donju tabelu?

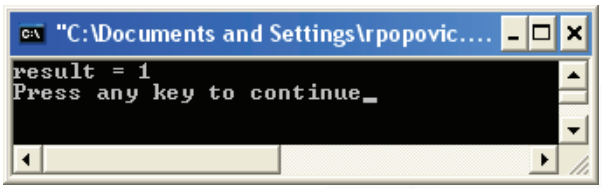
```
#include <iostream>
using namespace std;

int main()
{
  int i = 3,
      j = 8,
      k = 5,
      result;
  result = (i < j < k);
  cout << "result = " << result << endl;
  return 0;
}
```

Kompajler će upozoriti korisnika:

warning C4804: '<' : unsafe use of type 'bool' in operation,

ali će se program izvršiti i prikazati na ekranu rezultat.



Operatori	Asocijativnost
() [] -> .	Sa leva u desno
! ~ ++ -- + - * (*type*) sizeof	Sa desna u levo
* / %	Sa leva u desno
+ -	Sa leva u desno
<< >>	Sa leva u desno
< <= > >=	Sa leva u desno
== !=	Sa leva u desno
&	Sa leva u desno
^	Sa leva u desno
|	Sa leva u desno
&&	Sa leva u desno
||	Sa leva u desno
?:	Sa desna u levo
= += -= *= /= %= &= ^= |= <<= >>=	Sa desna u levo
,	Sa leva u desno

Beskonačne petlje: while i do

Petlja while

Naredba ima sledeći oblik:

```
while (kontrolni-izraz)
        telo-petlje
```

Kontrolni izraz, koji treba da bude u malim zagradama, može biti bilo koji ispravan C++ izraz odnosno relacioni uslov. Telo petlje može biti prosta C++ naredba završena sa ";" ili može biti sastavljeno iz više naredbi koje se nalaze unutar velikih zagrada.

Posmatramo sledeći kôd:

```
int count = 0;
while (count < 5)
{
  ++count;
  cout << "Hello" << endl;
}
cout << endl;
cout << "Hello je prikazan " << count << " puta." << endl;
```

Početna vrednost promenljive count je 0 i uslov petlje je ispunjen sve dok je vrednost promenljive count manja od 5. Telo while petlje je složena naredba sastavljena iz dve proste naredbe i izvršava se ako je uslov petlje ispunjen, u ovom slučaju pet puta:

```
Hello
Hello
Hello
Hello
Hello

Hello je prikazan 5 puta.
```

Primer neodređene petlje koja koristi char tip podatka

Pisanje koda za neodređenu petlju koristeći while naredbu je pogodno kada se nezna koliko će se puta telo petlje izvršiti. Zbog toga se ove petlje i nazivaju neodređene. Pretpostavimo da želimo da napišemo program koji

broji karaktere jedne linije koje korisnik unosi sa tastature. Linija je niz karaktera završena sa *return* odnosno *enter*. *Return* nije deo linije i ne broji se kao karakter. Program ne zna unapred broj karaktera koje će korisnik da unese.

Znakovna konstanta (eng. *character constant*) je prost karakter unutar jednostrukih znaka navoda. Na primer, 'A' i ';' su znakovne konstante. Možete da dodelite znakovnu konstantu promenljivoj tipa karakter (eng. *character variable*) na isti način kao što se dodeljuje brojna konstanta brojnoj promenljivoj. Možete da incijalizujete promenljivu tipa karakter u deklaraciji

```
char ch = 'A';
```

Korišćenje cin.get()

Ako deklarišemo promenljivu tipa karakter

```
char ch;
```

Naredba koja pridružuje sledeći karakter iz ulaznog toka varijabli ch je:

```
ch = cin.get();
```

Uslov u while naredbi određuje kada petlja prestaje sa izvršavanjem. U ovom programu, želimo da petlja prekine sa izvršavanjem kada korisnik unese karakter za novi red '\n'. Sledi, petlja se izvršava sve dok je ispunjen uslov (ch != '\n').

Prvi put kada program testira uslov petlje, promenljiva ch mora da ima ispravno deklarisanu i definisanu vrednost.

```
// Ovaj program broji karaktere sa ulaza koje korisnik unosi kao jednu liniju
// Linija se zavrsava ukucavanjem return, koji se ne broji kao karakter

#include <iostream>
using namespace std;

int main()
{
  char ch;                    // Koristi se da skladisti ulazni karakter
  int num_chars = 0;          // Broji karaktere
  cout << "Enter any number of characters terminated by return."
```

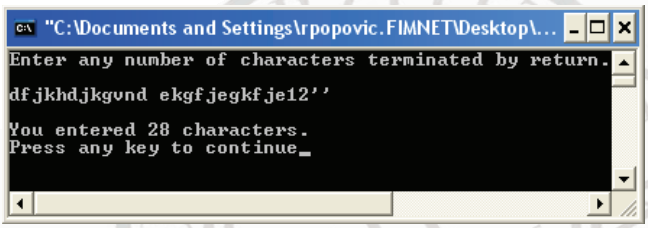
```
    << endl << endl;

  ch = cin.get();  // Dobija se prvi karakter

  while (ch != '\n')
  {
    ++num_chars;
    ch = cin.get();
  }

  cout << endl;
  cout << "You entered " << num_chars << " characters." << endl;

  return 0;
}
```



Program skladišti svaki karakter koji dobija znakovna promenljiva ch. Celobrojni brojač num_chars, koji program inicijalizuje na nulu u svojoj deklaraciji, skladišti broj karaktera koje unese korisnik.

Program pita korisnika da unese karaktere. Kada korisnik pritisne return taster, računar prenese celu liniju (uključujući *return* karakter) ulaznom strimu. Prvo izvršenje cin.get() vraća prvi karakter iz ulaznog strima i smešta ga u ch. Naredba while testira uslov (ch != '\n'). Ako korisnik nije pritisnuo odmah *return* taster već uneo određeni karakter, uslov je *true* i telo petlje se izvršava. Program povećava vrednost num_chars za jedan i pre nego što završi telo petlje dobija sledeći karakter iz ulaznog strima naredbom dodele

```
ch = cin.get();
```

Nakon završetka izvršenja tela petlje, program testira uslov petlje ponovo, ali sada sa novim karakterom dobijenim izvršenjem cin.get() u telu petlje.

Proces se nastavlja sve dok izvršenje ch = cin.get() u telu petlje ne dodeli ch karakter za novi red '\n', kada je uslov *false*.

Efikasnije rešenje umesto dela koda

```
ch = cin.get();  // Uzima se prvi karakter
 while (ch != '\n')
 {
  ++num_chars;
  ch = cin.get();
 }
```

je:

```
while ( (ch = cin.get()) != '\n')
   ++num_chars;
```

do-while petlja

while petlja testira uslov petlje pre nego što izvrši telo petlje. Sledi, ako je uslov petlje netačan- *false* prvi put kada se testira, program neće da izvršava telo petlje. Ponekad je potrebno da se program izvrši najmanje jednom pre nego što se testira uslov. Ova petlja se zove do-while, ili do, petlja.

Format naredbe je:

```
do
loop body
while (uslov petlje);
```

U prethodnom primeru deo koda napisan sa do-while naredbom je:

```
do
 {
  ch = cin.get();
  ++num_chars;
 }
 while (ch != '\n');
 --num_chars;
```

Određene petlje

Kada se zna tačno koliko često program treba da izvršava telo petlje koristi se određena iteraciona petlja, odnosno for petlja.

for petlja

Kada korisnik interaguje sa programom, ponekad je neophodno obrisati ekran monitora pre nastavka interakcije korisnika i programa. Jednostavan način (koji nije i najbolji) je da se prikaže onoliko novih linija koliko treba da se pokrije ekran. Ako vaš monitor prikaže 25 linija pomoću 25 karaktera za novi red rešava se problem. Nažalost, brojanje je jedini način da budemo sigurni da je sve u redu.

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

Umesto prethodnog rešenja problem možemo da posmatramo kao jednu iteraciju: prikazati karakter za novi red 25 puta koristeći petlju. Sledeća while petlja izvršava ovo:

```
int line_count = 1;

while (line_count <= 25)
{
  cout << endl;
  ++line_count;
}
```

Naredba for je posebno dizajnirana za određene iteracione petlje. Sledeća for naredba ima isti efekat kao prethodna while naredba.

```
int line_count;
for (line_count = 1; line_count <= 25; ++line_count)
  cout << endl;
```

Kada program izvršava prvu naredbu unutar zagrada (u ovom slučaju, line_count = 1), to je inicijalizacija petlje, koju program izvršava samo jednom nakon ulaska u petlju. Nakon toga, program razvija sledeću naredbu (uslov line_count <= 25). Ovaj izraz je test for petlje. Ako se kao rezultat dobije *true* (bilo koja nenegativna vrednost) telo petlje se izvršava jednom. Treća naredba je podešavanje petlje (++ line_count). Program nakon toga ponovo testira uslov petlje i nastavlja sa radom.

```
for (inicijalizacija_petlje; test_petlje; podesavanje_petlje)
     telo_petlje
```

Ugnježđene while petlje

Telo petlje može da sadrži bilo koju C++ naredbu. Petlje su ugnježđene kada telo petlje sadrži drugu petlju.

Kako da signaliziramo programu da ne želimo više da unosimo kataktere? C++ ima specijalnu simboličku konstantu EOF da bi se označio kraj ulaza koristeći fajl. EOF je mnemonik koji znači "end of file“, definisan u standardnom heder fajlu iostream. Vrednost EOF je sistemski zavisna, mada većina sistema koristi celobrojnu vrednost 1. U Windows-u, da bi se ubacila sa terminala korektna vrednost EOF, mora da ukucate ^Z; U UNIX/Linux-u, treba da ukucate ^D za vrednost EOF.

```
// Ovaj program koristi ugnjezdjene while-petlje za brojanje
// karaktera sa ulaza koje ukucava korisnik ali u vise linija

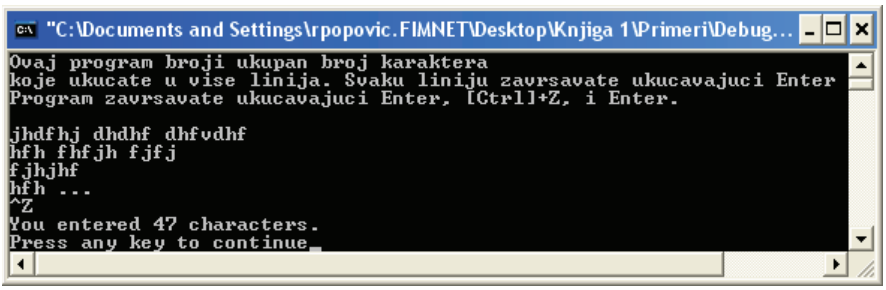
#include <iostream>
using namespace std;

int main()
{
 int ch;          // skladisti karakter koji unosi korisnik
 int num_chars = 0; // broji karaktere koji se unose

 cout << "Ovaj program broji ukupan broj karaktera" << endl
 << "koje ukucate u vise linija. Svaku liniju zavrsavate ukucavajuci Enter"<< endl
 << "Program zavrsavate ukucavajuci Enter, [Ctrl]+Z, i Enter." << endl << endl;

 while ((ch = cin.get()) != EOF)
 {
  ++num_chars;
  while ((ch = cin.get()) != '\n')
    ++num_chars;
 }
 //cout << endl;
 cout << "You entered " << num_chars << " characters." << endl;

 return 0;
}
```



Promenljiva ch u koju se smešta karakter koji korisnik unosi preko tastature je tipa int umesto tipa char. Jedina razlika u skladištenju karaktera recimo 'a', u varijablu tipa int, umesto u varijablu tipa char, je da int varijabla zahteva 2 ili 4 bajta memorije dok promenljiva tipa karakter zahteva 1 bajt (dva bajta za Unicode karaktere) memorije. U oba slučaja karakter 'a' je uskladišten kao ceo broj čija je vrednost 97.

2.3 Donošenje odluke

U jeziku C++ kôd za donošenje odluke se piše koristeći if naredbu čiji je opšti oblik:

```
if (kontrolni-izraz)
  true-deo
else
  false-deo
```

Reči if i else su C++ rezervisane reči. Takođe treba da ubacite kontrolni izraz u zagrade. True i false deo if naredbe može biti bilo koja prosta naredba završena sa ; ili složena naredba.

Složeni uslovi koji sadrže logičke operatore

Ponekad program mora da testira uslov koji je kombinacija drugih prostijih uslova. Možemo da kombinujemo proste uslove da bismo formirali složene uslove koristeći logičke operatore, koji su dati u tabeli:

Operator	Značenje
!	negacija
&&	i
||	ili

Operator not !

Podsetimo se da vrednosti istinitosti izraza mogu biti ili *true* (bilo koja nenulta vrednost) ili *false* (nulta vrednost). Operator not ili negacija (eng. *negation operator*, "!") menja reverzno vrednosti operanda. Pretpostavimo da imamo sledeće deklaracije i if naredbu

```
int i = 7;
char ch;

if (!(i == 3))
  ch = 'A';
else
  ch = 'B';
```

Kako je vrednost i jednaka 7, uslov (i == 3) je false. Operator not ! menja reverzno uslov u *true*. Zbog toga, if naredba dodeljuje vrednost 'A' varijabli ch.

and operator && (konjukcija)

Operator and && kombinuje dva uslova na sledeći način: (uslov 1) && (uslov 2). Tablica istinitosti za operator konjukcije je *true* samo ako su oba uslova *true*; inače, vrednost je *false*.

Tablica istinitosti za (uslov 1) && (uslov 2)

		Uslov 1	
	&&	T	F
Uslov 2	T	T	F
	F	F	F

Primer:

```
if ( (salary >= 500.00) && (salary < 1000.00) )
  tax_rate = 0.22;
```

Treba obratiti pažnju da if naredba sadrži dve male zagrade u kojima se nalazi složeni uslov i dva para zagrada u kojima se nalaze prosti uslovi.

Mada C++ često ne zahteva to, treba da koristite zagrade za svaki prost uslov zbog jasnijeg prikaza koda.

Sledeći program pita korisnika da ukuca jedan karakter i zatim return. Program testira karakter da li je broj i prikazuje odgovarajuću poruku.

```
// Ovaj program prikazuje koriscenje slozene if-naredbe
// koja odlucuje da li je taster koji je korisnik pritisnuo numericki.
// Program koristi cin.get() za unos jednog karaktera u jednom trenutku.

#include <iostream>
using namespace std;

int main()
{
  char ch;

  cout << "Please press a key, then press enter: ";

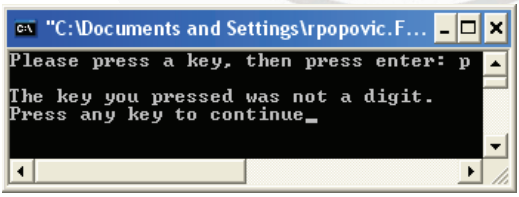
  ch = cin.get();

  cout << endl;

  if ( (ch >= '0') && (ch <= '9'))

    cout << "The key you pressed was a digit." << endl;
  else
    cout << "The key you pressed was not a digit." << endl;

  return 0;
}
```



or operator || (disjunkcija)

Operator || (disjunkcija) kombinuje dva uslova na sledeći način: (uslov 1) || (uslov 2). Vrednost u tablici istinitosti disjunkcije je true ako je ili jedan uslov, ili drugi uslov ili oba uslova *true*; inače, vrednost u tablici istinitosti (*truth* vrednost) je *false*. Drugim rečima, disjunkcija daje *false*

samo kada su oba uslova *false*. Donja tabela daje vrednosti istinitosti za disjunkciju za različite kombinacije uslova.

Tablica istinitosti za (uslov 1) || (uslov 2)

	||	Uslov 1	
		T	F
Uslov 2	T	T	T
	F	T	F

Sledeći primer, kao i prethodni, testira karakter koji korisnik unosi sa tastature da bi se videlo da li je on ceo broj (*digit*). Međutim, umesto da se koristi and operator (&&) za testiranje opsega brojeva, koristi se operator || za testiranje da li je karakter izvan validnog opsega. Karakter koji korisnik unosi nije validan ako je on manji od '0' ili ako je veći od '9'.

```
// Ovaj program prikazuje da li je uneti karakter ceo broj

#include <iostream>
using namespace std;

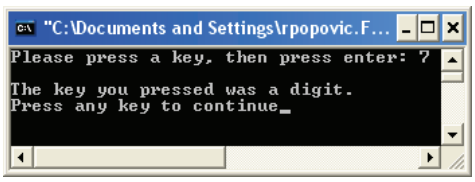
int main()
{
  char ch;

  cout << "Please press a key, then press enter: ";

  ch = cin.get();

  cout << endl;
  if ( (ch < '0') || (ch > '9'))
    cout << "The key you pressed was not a digit." << endl;
  else
    cout << "The key you pressed was a digit." << endl;

  return 0;
}
```



2.4 Pravila prioriteta

Prioritet i redosled izračunavanja

Ranije smo dali tabele koje definišu prioritet i asocijativnost aritmetičkih operatora, operatora dodele i relacionih operatora. Treba da ubacimo i logičke operatore u ovu hijerarhiju.

Operatori	Asocijativnost
()	sa leva na desno
! + (unarni) - (unarni) ++ --	sa desna na levo
* / %	sa leva na desno
+ -	sa leva na desno
< <= > >=	sa leva na desno
== !=	sa leva na desno
&&	sa leva na desno
||	sa leva na desno
= += -= *= /=	sa desna na levo

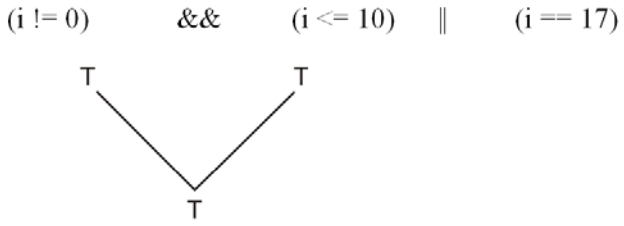
Negacija je najvećeg prioriteta zajedno sa ostalim unarnim operatorima. Operator konjukcije && ima veći prioritet od operatora disjunkcije ||, i oba operatora imaju asocijativnost sleva u desno.

Kao primer, pogledajmo sledeći kôd:

```
int i = 7,
    result;

if ( (i != 0) && (i <= 10) || (i == 17) )
  result = 1;
else
  result = 0;
```

Kako && ima veći prioritet od ||, računar izračunava, odnosno razvija konjukciju prvo. Uslov (i != 0) je istinit i uslov (i <= 10) je istinit. Zbog toga, konjukcija ova dva uslova je *true*. Ceo uslov u naredbi je *true*. Zbog toga kod postavlja vrednost result na 1. Donja slika prikazuje razvoj složenog uslova.



Ovaj primer pokazuje interesantan aspekt kako C++ razvija logičke izraze; C++ koristi razvoj kratkog spoja (eng. *short-circuit*). Kada računar zaključi da konjukcija (i != 0) && (i <= 10) daje *true*, on zna da konačan rezultat disjunkcije sa uslov (i == 17) mora biti *true*. Zbog toga računar ne testira drugi uslov da vidi da li je promenljiva i jednaka 17.

Ugnježđene naredbe

Ponekad je neophodno da se testira uslov ako postoji prethodno ispinjen uslov *true* ili *false*, odnosno ako postoji ugnježđena if naredba. Ako se ne koriste zagrade, kompajler pridružuje svaki else susednoj if naredbi koja ne sadrži else.

Sledeći karakter

Treba voditi računa o sadržaju ulaznog bafera za unos sa tastature u program karaktera i brojeva. Pretpostavimo da korisnik ukuca karakter 'A' a zatim *newline* karakter. Pretpostavimo da program dobija sledeći karakter iz ulaznog bafera izvršavanjem cin.get(). On će vratiti *newline* karakter koji je preostao u baferu nakon ubacivanja nove vrednosti od strane korisnika. Program neće dobiti karakter 'A'. Da bi odbacili ovaj *newline* karakter, program treba da izvrši cin.get() i odbaci karakter koji vraća. Program će izvršiti cin.get() ponovo i smestiti povratni karakter u promenljivu koju program kasnije testira. Kôd koji ovo izvršava je:

```
cin.get();
employee_code = cin.get();
```

Prva naredba cin.get() dobija sledeći karakter koji je na raspolaganju iz ulaznog bafera, ali ne dodeljuje ni jednoj varijabli ovu vrednost, odnosno odbacuje se *newline* character.

switch naredba

U C++ jeziku switch naredba implementira case strukturu strukturnog programiranja, odnosno višestruka grananja. Na taj način, switch naredba se može iskoristiti u određenim okolnostima umesto ugnježđenih naredbi. Naredba switch ima sledeći format.

```
switch (celobrojni izraz)
{
   case   case-value-1:
     statement-body-1
   case   case-value-2:
     statement-body-2
    .
    .
    .
   case   case-statement-n
     statement-body-n
   default:
     default-statement-body
}
```

Najčešći celobrojni izraz je prosta int ili char varijabla. Telo switch naredbe mora biti u velikim zagradama. Telo switch naredbe se sastoji od skupa slučajeva. Svaki slučaj počinje oznakom case, nakon toga sledi case vrednost i dvotačka. Vrednost slučaja je moguća vrednost koju celobrojni izraz može da dobije. Program izvršava naredbe ako je vrednost celobrojnog izraza jednaka vrednosti slučaja koja je u case oznaci. Napomena, telo naredbe može biti bez naredbe.

Naredba default (opciona je) sadrži naredbe koje program treba da izvrši ako vrednost celobrojnog izraza nije jednaka bilo kojoj vrednosti sa oznakom case.

```
// Ovaj program odredjuje vrstu prodaje nepokretnosti

#include <iostream>
#include <iomanip>
#include <cstdlib>

using namespace std;

int main()
{
```

```
const double    RESIDENTIAL_RATE = 0.060;
const double    MULTIDWELLING_RATE = 0.050;
const double    COMMERCIAL_RATE = 0.045;

int     property_code;
double  sale_price,
        commission_rate,
        commission;
cout << setprecision(2)
     << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint);

cout << "Enter the property's selling price: ";
cin >> sale_price;

cout << endl;
cout << "Enter the property code according to the following."
     << endl << endl;
cout << "Residential,       enter R" << endl;
cout << "Multiple Dwelling, enter M" << endl;
cout << "Commercial,        enter C" << endl << endl;
cout << "Please make your selection: ";

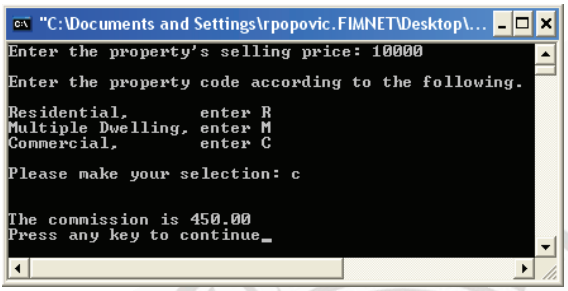
cin.get();          // Odbaci \n
property_code = cin.get();

switch (property_code)
{
  case 'R':
  case 'r':
    commission_rate = RESIDENTIAL_RATE;
    break;
  case 'M':
  case 'm':
    commission_rate = MULTIDWELLING_RATE;
    break;
  case 'C':
  case 'c':
    commission_rate = COMMERCIAL_RATE;
    break;
  default:
    cout << endl << endl
         <<"Invalid Property Code! Try Again" << endl;
    exit(1);
    break;
}
```

```
  commission = sale_price * commission_rate;

  cout << endl << endl;
  cout << "The commission is " << commission << endl;

  return 0;
}
```



Naredba break

Izvršenje break naredbe prouzrokuje da program završi while, do, switch ili for naredbu. Program zatim predaje kontrolu naredbi koja je nakon te kompletirane while, do, switch ili for naredbe. Dakle, break naredba proizvodi prevremen izlaz iz petlje.

Sledeći program prikazuje break naredbu korišćenu u petlji. Program broji karaktere koje korisnik unosi pre pojave prvog @ karaktera.

```
// Program broji karaktere koje korisnik unosi
// pre pojave prvog @ karaktera.

#include <iostream>

using namespace std;

int main()
{
  int ch,       // uneti karakter
      count = 0;  // broji karaktere pre pojave prvog  @ karaktera

  cout << "Enter a line containing @" << endl;

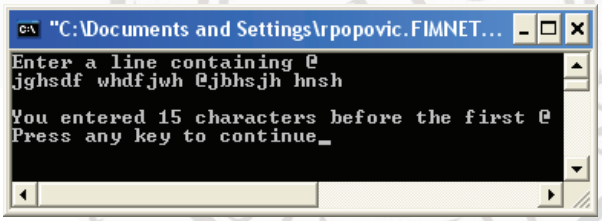
  while ( (ch = cin.get()) != '\n')
  {
    if (ch == '@')
```

```
    break;
  ++count;
 }

 if (ch == '\n')
 {
  cout << endl;
  cout << "You did not enter a @ character." << endl;
  cout << "You entered " << count << " characters." << endl;
 }

 else
 {
  cout << endl;
  cout << "You entered " << count
       << " characters before the first @"
       << endl;
 }

 return 0;
}
```



Naredba continue

Naredba continue prekida tok kontrole unutar while, do-while, ili for petlje. Međutim, umesto završetka kao kod naredbe break, naredba continue završava samo tekuću iteraciju. Sledeća iteracija petlje počinje odmah nakon toga.

Sledeći primer broji karaktere u jednoj liniji (velika slova).

```
// Ovaj program broji karaktere (velika slova) koje korisnik unese kao jednu liniju

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

```
{
  int ch,         // uneti karakter
      count = 0; // broj unetih velikih slova

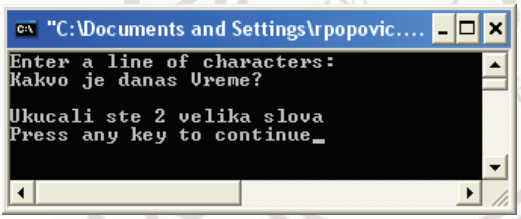
  cout << "Enter a line of characters:" << endl;

  while ( (ch = cin.get()) != '\n')
  {
    if ( (ch < 'A') || (ch > 'Z') )
      continue;
    ++count;
  }

  cout << endl;
  cout << "Ukucali ste " << count << " velika slova" << endl;

  return 0;
}
```

Ukucali ste " << count << " velika slova" << endl;



3. Koncept funkcije

Objasnićemo prirodu funkcija i naučiti kako da deklarišemo, definišemo i koristimo proste funkcije.

3.1 Definicija funkcije

Funkcija treba da izvrši jedan dobro definisan zadatak. Da bi to uradila funkcija može da zahteva više vrednosti, nazvanih argumenti, koji joj se prosleđuju kada je pozivajuća funkcija. Funkcija može i da ne sadrži argumente. Kada funkcija kompletira svoj zadatak, ona može da ne vrati vrednost ili da vrati jednu vrednost koja se zove povratna vrednost, pozivajućoj funkciji. Pozivajuća funkcija može onda da koristi povratnu vrednost u bilo kojoj od svojih naredbi.

Definicija funkcije:

```
return-type ime-funkcije(type parameter, type parameter, ...)
        {
        telo fukcije
        }
```

Funkcija se deklariše preko prototipa funkcije. Prototip ima sledeći format:

```
return-type ime-funkcije(argument type, argument type, . . .);
```

return-type je tip podatka koji funkcija vraća.

Prototip ne kaže šta funkcija radi ili kako funkcija izvršava svoj zadatak. Prototip daje kompajleru informacije o funkciji tako da kompajler može da proveri da li program koristi funkcije korektno. Na primer:

```
double Calc_Sales_Tax(double, double);          //prototip funkcije
```

Prototip kaže C++ kompajleru da funkcija koja izračunava porez na promet Calc_Sales_Tax() zahteva dva argumenta, koji su tipa double i vraća vrednost koja je tipa double.

Korišćenje void u deklaraciji

Ako funkcija ne vraća vrednost, koristi se rezervisana reč void kao povratni tip u prototipu. Ako funkcija nema argumenata, onda se koristi void za tip liste argumenata ili se zanemaruje lista tipova argumenata.

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

void Print_10_Xs();

int main()
{
 int i;
 cout << endl;

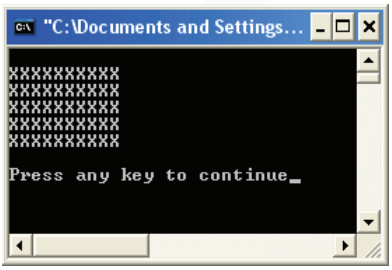
 for (i = 1; i <= 5; ++i)
 {
  Print_10_Xs();
  cout << endl;
 }

 cout << endl;
 return 0;
}

void Print_10_Xs()
{
 int j;

 for (j = 1; j <= 10; ++j)
  cout << 'X';

 return;
}
```



Argument/parametar

Kada se argumenti prosleđuju funkciji, prvom argumentu se dodeljuje prvi parametar u definiciji funkcije, i tako dalje. Na taj način argumenti i parametri se podešavaju samo na osnovu pozicije, a ne po drugom značenju.

```
#include <iostream>

using namespace std;

void Print_Char(char, int);

int main()
{
  int i,
      number_of_rows;
  char ch;

  cout << "Ukucajte karakter koji zelite da prikazete na displeju: ";
  cin >> ch;

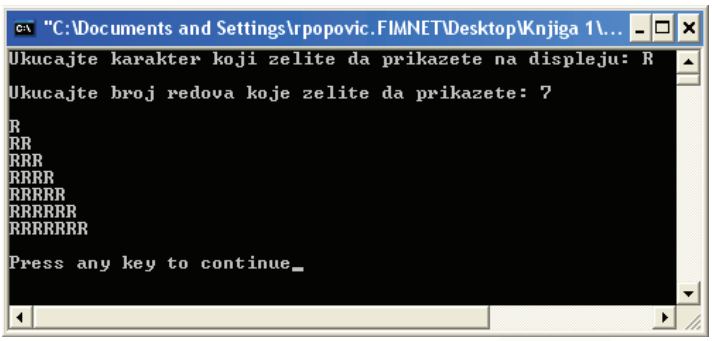
  cout << endl;
  cout << "Ukucajte broj redova koje zelite da prikazete: ";
  cin >> number_of_rows;

  cout << endl;
  for (i = 1; i <= number_of_rows; ++i)
  {
    Print_Char(ch, i);
    cout << endl;
  }

  cout << endl;
  return 0;
}

void Print_Char(char display_char, int count)
{
  int j;

  for (j = 1; j <= count; ++j)
    cout << display_char;
  return;
}
```



3.2 Prosleđivanje parametara po vrednosti

```
// Ovaj program prikazuje kako se argument
// prosledjuje po vrednosti.

#include <iostream>

using namespace std;

void PreInc(int);

int main()
{
  int i;

// Promenljiva i je inicijalizovana na  5.

  i = 5;

  cout << "Pre poziva funkcije PreInc(), i = " << i << endl;

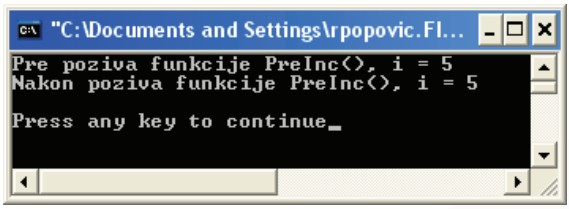
  PreInc(i);

  cout << "Nakon poziva funkcije PreInc(), i = " << i << endl;

  cout << endl;
  return 0;
}

void PreInc(int parm)
{
// Vrednost koja je dodeljena parametru parm se prosledjuje funkciji.
// Funkcija povecava vrednost parametra parm i vraća kontrolu funkciji main().
```

```
 ++parm;
 return;
}
```



Kada se izvršava PreInc(i) u glavnoj funkciji main(), vrednost i, koja iznosi 5, se prosleđuje funkciji PreInc() i dodeljuje se parametru parm. U funkciji PreInc(), naredba ++parm; povećava vrednost parm. Zbog toga, vrednost i u glavnoj funkciji main() ostaje nepromenjena.

C++ takođe dozvoljava prosleđivanje argumenata po referenci. Kada se argumenti prosleđuju po referenci, adrese argumenata u glavnoj memoriji se prosleđuju funkciji. Tako, funkcija može da promeni vrednost argumentu u programu koji poziva funkciju.

```
// Program izracunava konacne poene na osnovu poena sa kolokvijuma i
// konacnog ispita
// Koristi se funkcija Calc_Grade() za izracunavanje poena za konacnu ocenu
// Funkcija vraca vrednost konacnih poena funkciji main().

#include <iostream>
using namespace std;

int Calc_Grade(int, int);

int main()
{
 int midterm_exam,
     final_exam,
     final_grade;

 cout << "Unesite poene sa kolokvijuma: ";
 cin >> midterm_exam;
 cout << endl;

 cout << "Unesite poene sa zavrsnog ispita: ";
 cin >> final_exam;

 final_grade = Calc_Grade(midterm_exam, final_exam);
```

```
  cout << endl;
  cout << "The Final Grade is " << final_grade << endl;

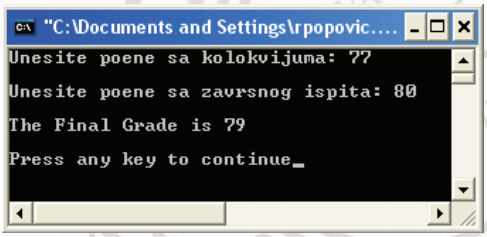
  cout << endl;
  return 0;
}

int Calc_Grade(int midterm, int final)
{
  const double MIDTERM_WEIGHT = 0.40;
  const double FINAL_WEIGHT   = 0.60;

  double grade;
  int rounded_grade;

  grade = MIDTERM_WEIGHT * midterm + FINAL_WEIGHT * final;
  rounded_grade = grade + 0.5;

  return rounded_grade;
}
```



3.3 Promenljive kao atributi

U C++ svaka promenljiva ima dva atributa koji određuju njenu oblast važenja-doseg i trajanje. Doseg (*scope*) promenljive je deo programa gde je moguć pristup promenljivoj. Trajanje (*duration*) promenljive je vreme za koje program drži memoriju zauzetom za tu promenljivu.

Doseg (scope) - oblast važenja

Promenljiva deklarisana unutar funkcije ima lokalni doseg (*local scope*)**.** Promenljiva je poznata samo unutar funkcije. Ni jedna naredba izvan funkcije ne može da promeni vrednost promenljive sa lokalnim dosegom. Varijabla deklarisana izvan tela funkcije ima globalni doseg (*global scope*) i poznata je svim funkcijama definisanim posle tačke u kojoj

se deklariše promenljiva. Na taj način, svaka funkcija čija definicija dolazi posle deklaracije globalne promenljive može da promeni vrednost globalne promenljive. Treba izbegavati korišćenje globalnih promenljivih deljenjem podataka između funkcija koristeći argumente i povratne vrednosti.

Trajanje (duration)

Klasa skladištenja promenljive (storage class) određuje njeno trajanje. Klasa skladištenja promenljive je ili auto ili static. Lokalna promenljiva, koja po *default*-u ima storage class auto (ponekad nazvana automatska varijabla) je alocirana memorija kada se funkcija u kojoj je ona deklarisana počne izvršavati. Kada se promenljivoj dodeli memorija, računar odvaja određenu količinu memorije (zavisno od tipa promenljive) za tu promenljivu i povezuje ime promenljive sa tom lokacijom. Kada se završi izvršavanje funkcije, dodeljena memorija automatskoj promenljivoj se dealocira i memorijski prostor se može ponovo koristiti od strane računara. Promenljiva u stvari prestaje da postoji. Ako main() ponovo pozove funkciju, varijabla se ponovo kreira, verovatno na drugoj lokaciji u memoriji računara.

Statičkoj promenljivoj (static variable) je dodeljena memorija kada program počne da se izvršava. Varijabla static postoji onoliko dugo koliko se program izvršava, nezavisno koja funkcija je trenutno aktivna. Globalna promenljiva ima static klasu skladištenja po *default*-u. Moguće je kreirati lokalnu statičku varijablu uključujući rezervisanu reč static u deklaraciji varijable. Takva varijabla je dostupna samo u funkciji u kojoj je deklarisana. Međutim, zbog njene klase skladištenja koja je statička, varijabla ostaje u važenju i zadržava svoju vrednost nakon završetka funkcije gde je ona deklarisana.

Izlazni tok cerr

Za prikaz poruke greške na monitoru koristi se cerr izlazni tok, koji ne može biti preusmeren.

C++ biblioteka matematičkih funkcija

C++ sadrži biblioteku matematičkih funkcija. Ove funkcije uključuju standardne trigonometrijske funkcije (sin(), cos(), tan(), ...), eksponencijalne i logaritamske funkcije (exp() i log()) ... Koristi se heder fajl <cmath>.

Prikazaćemo primere korišćenja dve proste matematičke funkcije.

Funkcija pow(osnova, eksponent) stepenuje prvi argument osnovu sa drugim argumentom tj. eksponentom. Oba argumenta moraju biti dvostruke preciznosti. Funkcija vraća double tip. Dakle

$p = b^e$.

Naredba

```
cout << pow(3.0, 1.35);
```

kao rezultat daje: 4.4067

Funkcija sqrt() vraća kvadratni koren njenog argumenta. Argument mora biti tipa double. Funkcija vraća tip double, na primer:

```
cout << sqrt(125.0);
```

4. Nizovi

4.1 Definicija niza

Niz je kolekcija konačnog broja objekata istog tipa, koji su smešteni u memoriji računara sekvencijalno. Ovi objekti su elementi niza. Veličina niza je fiksna za vreme trajanja programa. Često se nizovi zovu linearne strukture podataka ili jednodimenzionalne strukture podataka.

Deklarisanje niza

Format prema kome se deklariše niz je oblika:

```
tip-niza ime-niza [ velicina-niza ];
```

Primer niza čije je ime rate, čiji su elementi tipa double, koji može da memoriše 5 vrednosti je:

```
double rate[5];
```

Referenciranje i inicijalizacija elemenata niza

Svaki elemenat niza je definisan svojim pomerajem (*offset***)** od početka niza, koji se često zove indeks (*subscript* ili *index***).** Indeks prvog elementa niza je 0. Primer niza sa pet elemenata prikazan je preko sledeće naredbe:

```
double rate[5] = {0.075, 0.080, 0.082, 0.085, 0.088};
```

gde su članovi niza rate[0], rate[1], rate[2], rate[3] i rate[4] sa odgovarajućim vrednostima.

U nizu veličine N, opseg ofseta je od 0 do N-1. Da biste inicijalizovali niz u njegovoj deklaraciji, početne vrednosti treba da smestite u velike zagrade. Nakon svake vrednosti je zapeta, osim nakon poslednje. Ako lista ne sadrži dovoljno vrednosti, kompajler će dopuniti preostale članove liste nulama. Ako lista sadrži više elemenata od veličine niza, kompajler ignoriše suvišne elemente.

Na primer, posmatrajmo sledeću deklaraciju:

```
double rate[5] = {0.075, 0.080, 0.082};
```

C++ kompajler kreira niz rate[] i alocira dovoljno memorije za tačno 5 promenljivih tipa double. Nakon toga, kompajler inicijalizuje prva tri elementa na 0.075, 0.080, i 0.082, i smešta double 0.0 u preostale elemente niza.

Treba obratiti pažnju da niz može da se inicijalizuje koristeći inicijalizacionu listu samo u deklaraciji, ne kasnije dodelom. Sledi, primer neispravne dodele

```
double rate[5];
rate[] = {0.075, 0.080, 0.082, 0.085, 0.088}; // pogresna dodela
```

Sledeća deklaracija inicijalizuje sve elemente niza na nulu.

```
int grade[10] = {0};
```

U sledećoj deklaraciji sedam vrednosti je na inicijalizacionoj listi, ali je niz veličine 5. Sledi, kompajler smešta samo prvih pet brojeva u niz customer_count[].

```
int customer_count[5] = {3, 4, 5, 6, 7, 8, 9};
```

Ako inicijalizujete niz u deklaraciji niza i zanemarite veličinu niza, kompajler će uključiti veličinu sa liste elemenata. Na primer, sledeća deklaracija deklariše da veličina niza customer_count[] bude 5 zato što lista sadrži 5 elemenata:

```
int customer_count[] = {3, 4, 5, 6, 7};
```

U slučaju da izostavite veličinu niza, mora da inicijalizujete niz. Sledi primer nepravilne deklaracije:

```
int customer_count[];  // pogresno deklarisanje niza
```

Sledi jedan prost primer rada sa nizovima:

```
// Ovaj program izracunava troskove dostave naloga za narudzbenice preko poste
// na osnovu prodajne cene u regionu.
// Program koristi broj regiona za pristup elementima niza
```

```
#include <iostream>
#include <iomanip>

using namespace std;

int Get_Valid_Region();

int main()
{
 // Sledeca deklaracija definise niz rate[] sa 5 celobrojnih vrednosti

 double rate[5] = {0.075, 0.080, 0.082, 0.085, 0.088};
 int    region;
 double price,
        charge,
        total_price;

 cout << setprecision(2)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);

 cout << "Enter the price of the item: $";
 cin >> price;
 region = Get_Valid_Region();

 charge = price * rate[region - 1];
 total_price = price + charge;
 cout << endl << endl;
 cout << "Item Price:     " << setw(9) << price << endl;
 cout << "Shipping Charge: " << setw(9) << charge << endl;
 cout << "Total Price:     " << setw(9) << total_price << endl;

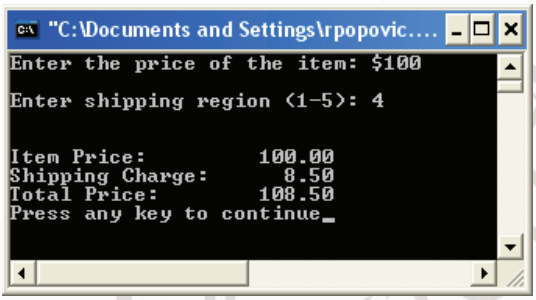
 return 0;
} //Kraj "main()" funkcije

int Get_Valid_Region()
{
 bool invalid_data;
 int region;

 do
 {
  cout << endl;
  cout << "Enter shipping region (1-5): ";
  cin >> region;
  if (region < 1 || region > 5)
```

```
    {
      cerr << endl;
      cerr << "Region in invalid - Please reenter.";
      invalid_data = true;
    }
    else
      invalid_data = false;
  }
  while(invalid_data);

  return region;
}
```



Korišćenje for petlje

Idealna kontrolna struktura za procesiranje niza je for petlja. Indeks niza se koristi kao brojač petlje. Kao primer dat je program koji procesira studentske ocene. Nastavnik želi da unese deset ocena (poena) sa kviza koje je student osvojio tokom semestra. Program treba da prikaže srednju ocenu studenta. Ocene se smeštaju u niz od deset elemenata grade[].

```
// Program prikazuje kako se dobija srednja vrednost elemenata niza

#include <iostream>
#include <iomanip>
using namespace std;
int main()
{
  const int NUM_QUIZZES = 10;
  int grade[NUM_QUIZZES]; //Niz za skladistenje ocena sa kviza
  int quiz,               // Indeks niza
     grade_sum = 0;
  double grade_avg;
```

```
  cout << setprecision(1)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

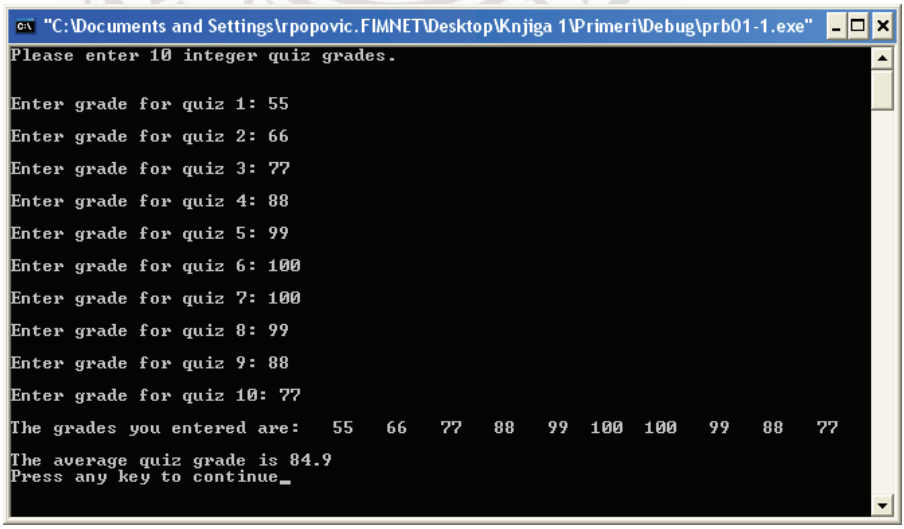
  cout << "Please enter " << NUM_QUIZZES
       << " integer quiz grades." << endl << endl;

  for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
  {
    cout << endl;
    cout << "Enter grade for quiz " << quiz + 1 << ": ";
    cin >> grade[quiz];
  }
  cout << endl;
  cout << "The grades you entered are:";
  for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
    cout << setw(5) << grade[quiz];

  for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
    grade_sum += grade[quiz];

  grade_avg = double(grade_sum) / NUM_QUIZZES;
  cout << endl << endl;
  cout << "The average quiz grade is " << grade_avg << endl;

  return 0;
}
```



Sortiranje niza

Sortiranje niza znači aranžiranje elemenata niza po rastućem ili opadajućem sistemu. Pretpostavimo da nakon unošenja deset ocena za studenta, program treba da prikaže ocene po rastućoj vrednosti. Odabraćemo *bubble sort* tehniku, koja se često koristi za male nizove. Postoji više varijacija ovog algoritma. Algoritam koji mi koristimo nije najefikasniji, ali radi i daje osnove tehnike.

```
// Ovaj program prikazuje kako se sortiraju elementi niza koristeci
// bubble sort algoritam.

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

int main()
{
 const int NUM_QUIZZES = 10;

 int grade[NUM_QUIZZES];     //niz za skladistenje ocena sa kviza
  int quiz,          // indeks niza „grade[]“
     temp,           // za zamenu elemenata niza
     pass,           // broj prolaza
     limit;          // vodi evidenciju o broju prolaza

  // Unos ocena

  cout << "Please enter " << NUM_QUIZZES
       << " integer quiz grades." << endl << endl;

  for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
  {
    cout << endl;
    cout << "Enter grade for quiz " << quiz + 1 << ": ";
    cin >> grade[quiz];
  }

  // Prikaz ocena sa kviza
  cout << endl << endl;
  cout << "The grades you entered are as follows:" << endl;

  for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
    cout << setw(6) << grade[quiz];
```

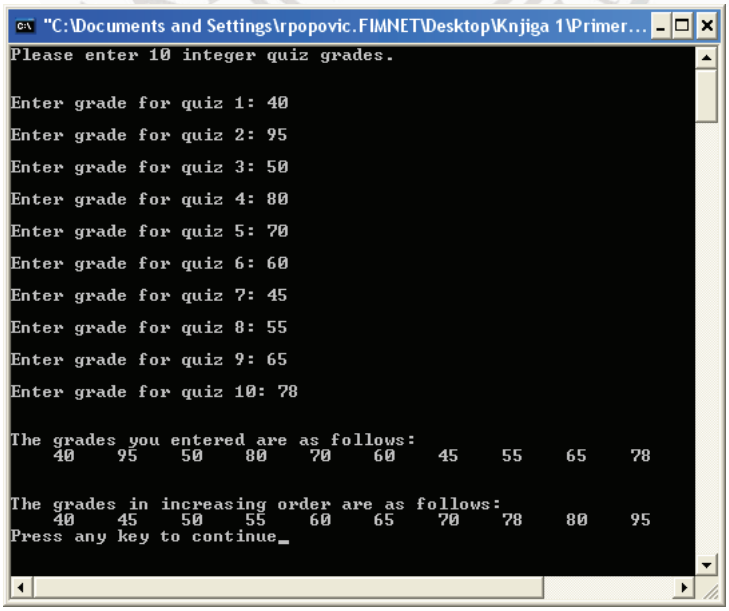
```
  cout << endl;

  // Primena  bubble sort algoritma
  limit = NUM_QUIZZES - 2;

  for (pass = 1; pass <= NUM_QUIZZES - 1; ++pass)
  {
    for (quiz = 0; quiz <= limit; ++quiz)
      if (grade[quiz] > grade[quiz + 1])
      {
        temp = grade[quiz];
        grade[quiz] = grade[quiz + 1];
        grade[quiz + 1] = temp;
      }
    limit;
  }
  // Prikaz sortiranih ocena sa kviza
  cout << endl << endl;
  cout << "The grades in increasing order are as follows:" << endl;
  for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
    cout << setw(6) << grade[quiz];

  cout << endl;

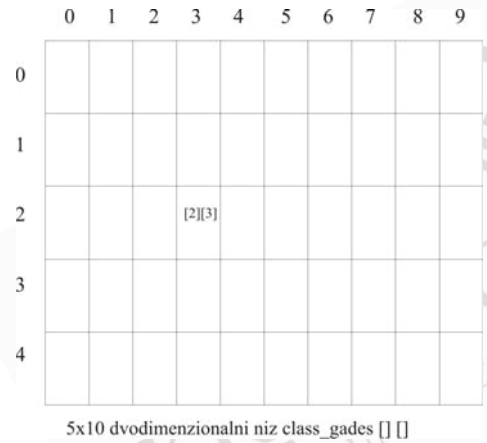
  return 0;
}
```



4.2 Višedimenzionalni nizovi

Razmatraćemo dvodimenzionalne nizove zato što ih programeri najviše koriste i laki su za opis.

Kao primer, posmatraćemo srednju ocenu sa kviza za pet studenata. Imamo ocene sa 10 kvizova za po pet studenata, ukupno 50 celobrojnih vrednosti. Možemo da klasifikujemo svaku ocenu prema dve kategorije: student kome ocena pripada i kviz na kome je ocena postignuta. Možemo da smestimo takve podatke u dvodimenzionalni niz, ili matricu. Slika prikazuje kako su ocene smeštene u dvodimenzionalni niz class_grades.



5x10 dvodimenzionalni niz class_gades [] []

Podaci su grupisani u redove i kolone. Pet redova predstavljaju pet studenata u klasi. Deset kolona predstavlja deset ocena sa kviza. Veličina niza je 5 x 10, odnosno 5 redova i 10 kolona sa indeksima za redove 0 do 4 i kolone od 0 do 9.

Da biste deklarisali dvodimenzionalni niz, odredite tip niza i njegovu veličinu. Smestite u 2 srednje zagrade broj redova i broj kolona respektivno. Na primer:

```
int class_grades[5][10];
```

naredba se čita: class_grades je niz od 5 elemenata od kojih je svaki niz od deset celih brojeva.

Jezik C++ skladišti elemente niza sekvencijalno po redovima. Moguće je inicijalizovati dvodimenzionalni niz u deklaraciji niza.

```
int class_grades[5][10] = {{50, 56, 87, 67, 98, 90, 68, 54, 67, 30},
                           {70, 68, 64, 78, 97, 57, 68, 90, 67, 74},
                           {64, 76, 87, 67, 95, 67, 56, 83, 60, 78},
                           {76, 65, 84, 47, 86, 65, 46, 66, 87, 65},
                           {76, 57, 65, 45, 90, 76, 76, 44, 67, 82}};
```

Svi elementi jednog reda se nalaze unutar velikih zagrada, i odvojeni su zapetom. Zapetama su odvojeni i svi redovi osim poslednjeg. Svi elementi se nalaze u velikim zagradama.

Ako inicijalizator redova sadrži manje elemenata, ostali elementi reda se automatski inicijalizuju na nulu, a ako inicijalizator redova sadrži više elemenata, kompajler ekstra elemente reda ignoriše.

Procesiranje dvodimenzionalnih nizova

Glavni alat za procesiranje dvodimenzionalnih nizova je ugnježđena for petlja. Pretpostavimo sledeću deklaraciju:

```
int array[BR_REDOVA][BR_KOLONA];
```

Da biste procesirali svaki element u ovom dvodimenzionalnom nizu, koristite ugnježđene for petlje strukturirane na sledeći način:

```
for (row = 0; row < BR_REDOVA; ++row)
  for (column = 0; column < BR_KOLONA; ++column)
    {procesiraj jedan elemenat niza array[row][column]}
```

Spoljna for petlja ulazi u prvu iteraciju za prvi red niza (indeks 0). Unutrašnja petlja procesira svaki elemenat niza reda određenog indeksom niza (u prvom slučaju 0), itd.

```
// Ovaj program koristi dvodimenzionalni niz za skladistenje
// rezultata kvizova studenata u nekoliko razreda. Program izracunava
// srednju vrednost poena razlicitih kvizova za svakog studenta.

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
```

```
  const int NUM_QUIZZES = 10;
  const int NUM_STUDENTS = 5;

  int class_grades[NUM_STUDENTS][NUM_QUIZZES];
  int student,
     quiz,
     quiz_sum;
  double quiz_average;

  cout << setprecision(1)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  // uzmi i sacuvaj ocene svakog studenta

  cout << "Enter exactly " << NUM_QUIZZES
       << " quiz grades for each student." << endl;
  cout << "Separate the grades by one or more spaces." << endl;

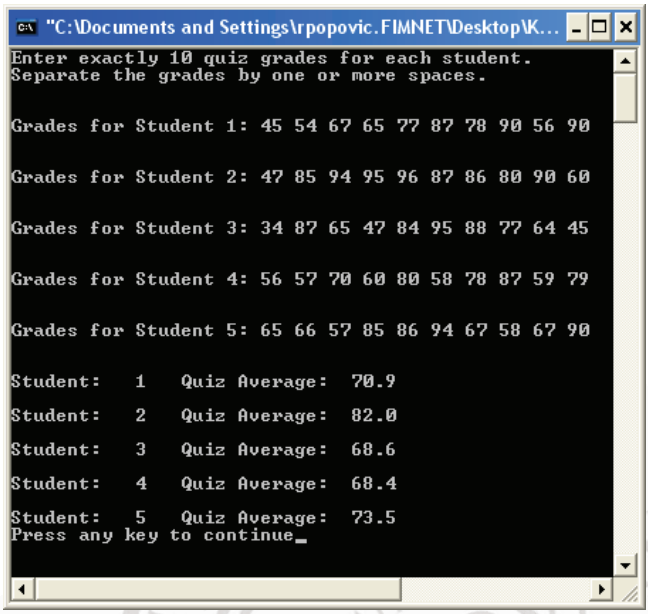
  for (student = 0; student < NUM_STUDENTS; ++student)
  {
    cout << endl << endl;
    cout << "Grades for Student " << student + 1 << ": ";
    for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
      cin >> class_grades[student][quiz];
  }
  // Izracunaj i prikazi prosecnu ocenu svakog studenta

  for (student = 0; student < NUM_STUDENTS; ++student)
  {
    quiz_sum = 0;
    for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
      quiz_sum += class_grades[student][quiz];
      quiz_average = (double) quiz_sum / NUM_QUIZZES;

      cout << endl << endl;
      cout << "Student: " << setw(3) << student + 1
           << "   Quiz Average: " << setw(5) << quiz_average;
  }

  cout << endl;

  return 0;
}
```

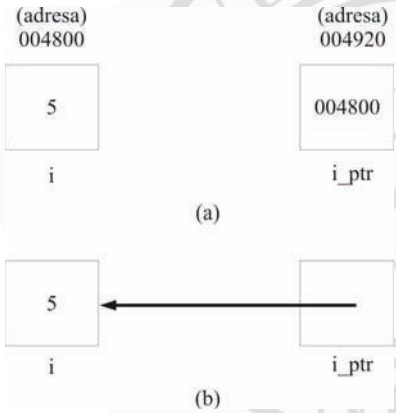

"C:\Documents and Settings\rpopovic.FIMNET\Desktop\K...
Enter exactly 10 quiz grades for each student.
Separate the grades by one or more spaces.
Grades for Student 1: 45 54 67 65 77 87 78 90 56 90
Grades for Student 2: 47 85 94 95 96 87 86 80 90 60
Grades for Student 3: 34 87 65 47 84 95 88 77 64 45
Grades for Student 4: 56 57 70 60 80 58 78 87 59 79
Grades for Student 5: 65 66 57 85 86 94 67 58 67 90
Student: 1 Quiz Average: 70.9
Student: 2 Quiz Average: 82.0
Student: 3 Quiz Average: 68.6
Student: 4 Quiz Average: 68.4
Student: 5 Quiz Average: 73.5
Press any key to continue

5. Pokazivači

U ovom poglavlju predstavljamo ideju pointera odnosno pokazivača, kako ih deklarisati, inicijalizovati itd. Takođe, objasnićemo kako se može menjati vrednost promenljive.

5.1 Deklarisanje i inicijalizacija pokazivača

Svaka promenljiva ima adresu, koja je broj koji određuje lokaciju varijable u glavnoj memoriji računara. Pointer je promenljiva čija je vrednost adresa druge promenljive. Na slici je prikazana promenljiva i čija je vrednost 5, a adresa (pretpostavimo) 004800. Promenljiva i_ptr je pointer čija je vrednost adresa promenljive i (tj. 004800). Zato kažemo da i_ptr pokazuje na i.



Kako adresa nije običan ceo broj, ne možemo ni pokazivač da deklarišemo kao običan ceo broj. Da bismo deklarisali pokazivač, počinjemo sa tipom podatka promenljive na koju će on pokazivati, za kojim sledi asterisk (*), za kojim sledi ime pokazivača. Na primer:

```
int*    i_ptr;   //   i_ptr je pokazivac na promenljivu tipa ceo broj - intidzer
int  *  i_ptr;   //   i_ptr je pokazivac na promenljivu tipa intidzer
int    *i_ptr;   //   i_ptr je pokazivac na promenljivu tipa intidzer
double* d_ptr;,  //   d_ptr je pokazivac na promenljivu tipa dabl
```

Deklarisanje pokazivača, kao i deklarisanje obične promenljive, ne daje pokazivaču validnu vrednost. Pre korišćenja pokazivača, budite sigurni

da ste dali validnu vrednost pokazivaču dodelom adrese promenljive na koju on pokazuje. Unarni operator adresa od (&), vraća adresu promenljive na koju je primenjen, na primer, ako je i celobrojna promenljiva, onda je &i njena adresa.

Inicijalizacija pokazivača

Možete da dodelite vrednost pokazivaču u deklaraciji, vodeći računa da je promenljiva prethodno deklarisana.

```
int      i = 5;
double   d = 3.14;
int*     i_ptr = &i;
double*  d_ptr = &d;
```

Sledeći kôd je pogrešan zato što je i deklarisano nakon deklaracije i inicijalizacije i_ptr. U trenutku kada se i_ptr deklariše i inicijalizuje, kompajler ne zna ništa o varijabli i.

```
int* i_ptr = &i;       // pogresna inicijalizacija
int  i;
```

Tip adrese treba da odgovara tipu pointera. Na primer:

```
int     i;
double* d_ptr = &i;      // pogresna inicijalizacija pokazivaca
```

Ne možete direktno dodeliti celobrojnu vrednost pokazivacu:

```
int* i_ptr = 004600;     // pogresna dodela
```

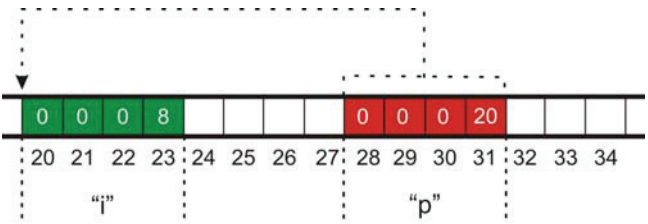
Jedini izuzetak je null pointer čija je vrednost 0. Možete koristiti i ranije definisanu konstantu NULL umesto nule. NULL je definasan u nekoliko heder fajlova uključujući iostream. Sledi:

```
int* i_ptr = 0;       // ispravna dodela
int* j_ptr = NULL;   // ispravna dodela
```

Inicijalizacijom pokazivača na nulu, pokazivač ne pokazuje ni na šta. Vrednost 0 se može dodeliti pokazivaču bilo kog tipa. Sledeće deklaracije su ispravne.

```
double* d_ptr = NULL;
char*   c_ptr = NULL;
```

Memorijski prikaz rada sa pokazivačima:




```
int i = 8
&i = 20
```

```
int *p = &i  (=20)
&p = 28
int j = *p(=8)
```

Operator indirekcije

Deklarišimo i i i_ptr:

```
int  i;
int* i_ptr = &i;
```

Direktna dodela je

```
i = 17;
```

dok je indirektna dodela koristeći operator indirekcije *

```
*i_ptr = 17;
```

Operator indirekcije * je unarni operator. Sledi, ima isti prioritet i asocijativnost (sa desna-u-levo), kao i drugi unarni operatori. Čita se kao target-cilj pokazivača je promenljiva na koju on pokazuje.

```
// Ovaj program  ilustruje koriscenje pokazivaca i operatora indirekcije

#include <iostream>

using namespace std;
```

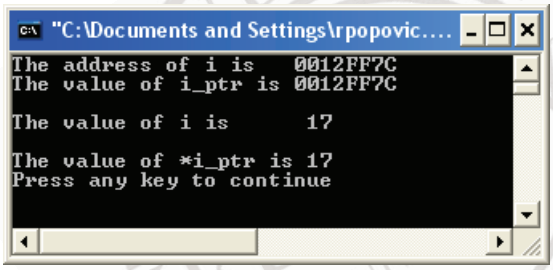
```
int main()
{
 int   i;
 int* i_ptr = &i;
 cout << "The address of i is   " << &i << endl;
 cout << "The value of i_ptr is " << i_ptr << endl << endl;

 // Smesti vrednost u promenljivu i koristeci indirekciju

 *i_ptr = 17;

 cout << "The value of i is      " << i << endl << endl;
 cout << "The value of *i_ptr is " << *i_ptr << endl;

 return 0;
}
```



5.2 Pokazivači i nizovi

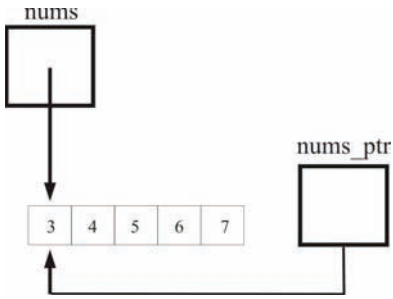
Ime niza je pokazivač na prvi elemenat niza. Na primer:

```
int  nums[5] = {3, 4, 5, 6, 7};
int* nums_ptr;
```

identifikator nums, koji je ime niza, je *integer pointer* na prvi elemenat niza. Tako, *nums, pokazuje na nums[0], čija je vrednost 3. Kako je nums pokazivač na ceo broj, sledeća dodela je legalna:

```
nums_ptr = nums;     // ispravna dodela pointeru
```

Nakon gornje dodele, *nums_ptr takođe upućuje na prvi elemenat niza nums[], odnosno nums[0].



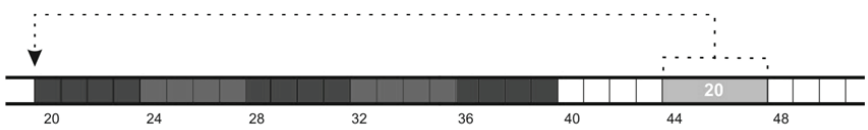
Sledeću dodelu ne smete da uradite:

```
nums_ptr = nums[2]; // neispravna dodela pointeru
```

Ime niza je konstantan pokazivač u smislu da ni jedna naredba C++ ne može da promeni njegovu vrednost. Sledi, neispravna dodela:

```
nums = nums_ptr;    // neispravna dodela
```

Memorijski prikaz veze pokazivača i niza:

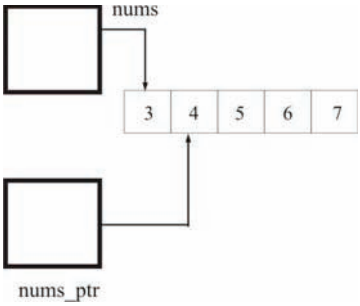



```
int array = [5]
int *p = array
```

Operacije sabiranja i oduzimanja mogu da se primene na pointere. Na primer, pretpostavimo sledeću dodelu

```
nums_ptr = nums;
```

Možemo sada da izvršimo ++nums_ptr. Ovo kao rezultat daje uvećanje vrednosti pokazivača num_ptr za 1, odnosno on sada pokazuje na sledeću celobrojnu vrednost elementa nums[1], ili 4, kao na donjoj slici. Treba napomenuti, međutim da ++nums je neispravno zato što je to pokušaj promene imena niza, što je konstantan pokazivač.



Važno je napomenuti da je aritmetika pokazivača unutar niza skalirana na tip niza. Na primer, pretpostavimo sledeću deklaraciju

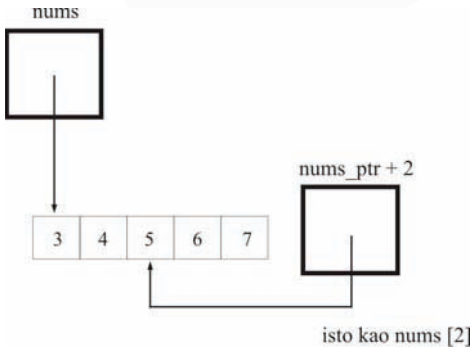
```
double  d_array[4] = {3.2, 4.3, 5.4, 6.5};
double* d_ptr = d_array;
```

Sledi, pokazivač d_ptr pokazuje na prvi elemenat niza d_array, čija je vrednost 3.2. Naredba ++d_ptr kao rezultat daje da d_ptr pokazuje na sledeći double u nizu, čija je vrednost 4.3. Ako ponovo izvršimo ++d_ptr, d_ptr će pokazivati na sledeći double niza, tj. 5.4.

Možete da koristite aritmetiku pointera na ime niza, ako to ne rezultuje promenom vrednosti imena niza, sledi:

```
nums_ptr = nums + 2;    // validna dodela pokazivacu
```

Ova dodela kao rezultat daje da pointer nums_ptr pokazuje na element niza nums[2], kao na slici. Generalno, izraz nums + i je pointer na i-ti element niza nums[i]. Zato, *(nums + i) je isti elemenat kao i nums[i].



Treba voditi računa da aritmetika pokazivača ne pređe broj elemenata niza. C++ kompajler neće vas upozoriti ako napravite ovakvu grešku. Pristup takvim lokacijama može da prouzrokuje *run-time* grešku u programu ili može da prouzrokuje nekorektne rezultate.

```
//Ovaj program prikazuje aritmetiku pokazivaca i nizova

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
    int  nums[5] = {3, 4, 5, 6, 7};
    int* nums_ptr;
    int  i;

    cout << "The array using nums[i] notation" << endl;

    for (i = 0; i < 5; ++i)
      cout << setw(4) << nums[i];

    cout << endl << endl;

    cout << "The array using *(nums + i) notation" << endl;

    for (i = 0; i < 5 ; ++i)
      cout << setw(4) << *(nums + i);

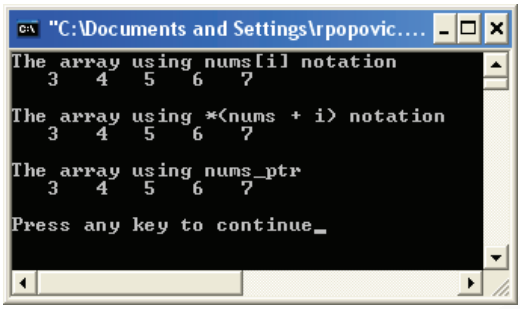
    cout << endl << endl;

    cout << "The array using nums_ptr" << endl;

    for (nums_ptr = nums; nums_ptr < nums + 5; ++nums_ptr)
      cout << setw(4) << *nums_ptr;

    cout << endl << endl;

    return 0;
}
```



UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

6. C-stringovi, pokazivači, nizovi, funkcije, korisnički definisani tipovi podataka i tabele

Ova sekcija razmatra rad sa stringovima. Počinjemo sa osnovnim idejama i tehnikama za rad sa stringovima.

6.1 Definicija i inicijalizacija stringa

U jeziku C++ stringovi se mogu implementirati na dva načina. Definisaćemo string kao niz karaktera koji je završen null karakterom. Null karakter je bajt čiji su svi bitovi nule. Ovaj bajt označavamo specijalnim simbolom '\0'. Ako niz karaktera ne sadrži na kraju null karakter, niz ne predstavlja string.

Ovako definisan string se naziva C-string zato što je tako definisan u C programskom jeziku. Podsetimo se da je C++ baziran na programskom jeziku C i da sadrži sve mogućnosti C programskog jezika (C++ je uz minimalne izuzetke nadskup jezika C). Kasnije ćemo videti kako se stringovi mogu implementirati koristeći string klasu.

Sledeća deklaracija i inicijalizacija kreiraju string sastavljen od reči "Hello". C-string je oblika:

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

Drugi pristup za inicijalizaciju stringa koristi string konstantu, koja je niz karaktera pod znacima navoda, kao u primeru koji sledi:

```
char greeting[6] = "Hello";
```

Programer ne smešta null character na kraj string konstante. C++ kompajler automatski smešta '\0' na kraj stringa kada inicijalizuje niz. Prethodna deklaracija kao rezultat daje da kompajler inicijalizuje memoriju na sledeći način

greeting

H	e	l	l	o	\0

Takođe, podsetimo se da ako inicijalizujete niz u njegovoj deklaraciji, ne treba da postavite veličinu niza. C++ kompajler će izvesti

zaključak o veličini niza iz inicijalizovanih vreednosti. Sledi, ova deklaracija je ekvivalentna prethodnoj deklaraciji.

```
char greeting[] = "Hello";
```

String i ulaz sa tastature i izlaz na displej

Pretpostavimo da želimo da napišemo jednostavan program koji pita korisnika za naziv proizvoda i njegovu cenu. Program treba nakon toga da prikaže na displeju naziv proizvoda, porez na cenu i konačnu cenu koštanja (cena + porez).

Podsetimo se da za dobijanje jednog karaktera sa tastature i ulaznog bafera, koristimo cin.get(), a za dobijanje stringa sa tastature i ulaznog bafera, koristimo cin.getline(). Funkcija član getline() zahteva dva argumenta. Prvi argument mora biti ime niza u koji se smešta string. Drugi argument treba da bude veličina niza koji je prvi argument. Funkcija član getline() uzima sve karaktere iz ulaznog bafera tastature uključujući i karakter za novi red, ili za jedan manje od broja karaktera koji je određen drugim argumentom. Dodatna pozicija je ostavljena za završni *null* karakter. Funkcija smešta karaktere, završene null karakterom u niz sa imenom određenim prvim argumentom. Na primer, pretpostavimo sledeću deklaraciju i naredbu:

```
char buffer[6];
cin.getline(buffer, 6);
```

Naredba cin.getline() čita sve karaktere koje korisnik unese sa tastature zaključno sa karakterom za novi red do 5 karaktera. Fukcija smešta string, sa dodatnim *null* karakterom u niz buffer []. Ako se '\n' pročita sa cin.getline(), ne smešta se u bafer.

```
// Ovaj program prikazuje koriscenje cin.getline() za ubacivanje string-a
// i prikazuje na displeju stringove na nekoliko načina

#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
  double const TAX_RATE = 0.0825;
```

```
    char item_name[51];
    double price,
           tax,
           total;

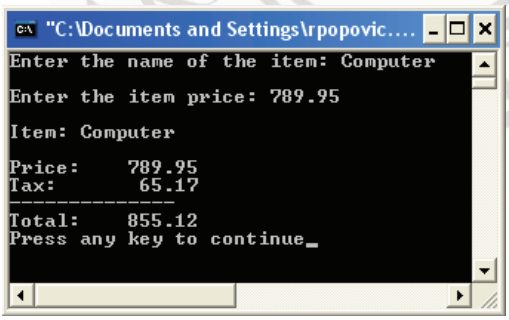
    cout << setprecision(2)
         << setiosflags(ios::fixed)
         << setiosflags(ios::showpoint);

    cout << "Enter the name of the item: ";
    cin.getline(item_name, 51);
    cout << endl;
    cout << "Enter the item price: ";
    cin >> price;

    tax = price * TAX_RATE;
    total = price + tax;

    cout << endl;
    cout << "Item: " << item_name << endl << endl;;
    cout << "Price: " << setw(9) << price << endl;
    cout << "Tax:   " << setw(9) << tax << endl;
    cout << "--------------" << endl;
    cout << "Total: " << setw(9) << total << endl;

    return 0;
}
```



Pretpostavljamo da je ime uzorka veličine najviše do 50 karaktera. Veličina niza item_name[] je 51 zbog dodatnog prostora za *null* karakter.

item_name

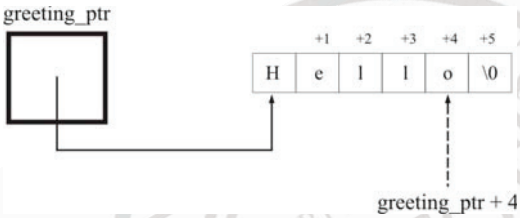
C	o	m	p	u	t	e	r	\0

6.2 String konstante i pokazivači

Važna veza između pokazivača i stringa je da je string konstanta, kao što je "Hello", ustvari pokazivač na niz karaktera. Kada kompajler radi sa string konstantom, on tretira string konstantu kao pokazivač na prvi karakter stringa tako da je sledeća deklaracija i inicijalizacija legalna:

```
char* greeting_ptr = "Hello";
```

Na donjoj slici izraz *(greeting_ptr + 4) pokazuje na karakter '0' u stringu.



U nastavku ovog poglavlja dati su primeri koji koriste pokazivače i određenu aritmetiku za rad sa stringovima.

Brojanje karaktera u stringu

Sledeći primer broji karaktere u stringu koji unosi korisnik, koristeći while petlju i pokazivač za kretanje kroz niz.

```
// Ovaj program broji karaktere jedne linije sa ulaza koje unosi korisnik.
// Koristi se pointer za kretanje kroz string i while petlja za brojanje karaktera

#include <iostream>

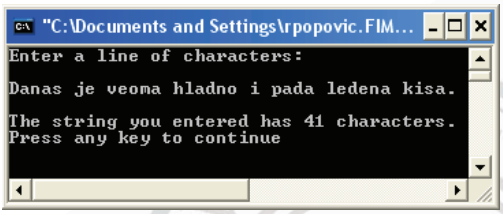
using namespace std;

int main()
{
  char line[81];
  char* ch_ptr = line;
  int  count = 0;

  cout << "Enter a line of characters:" << endl << endl;
  cin.getline(line, 81);
```

```
 while ( *ch_ptr != '\0' )
 {
  ++count;
  ++ch_ptr;
 }
 cout << endl;
 cout << "The string you entered has " << count << " characters." << endl;

 return 0;
}
```



Kako je normalna veličina ekrana monitora 80 karaktera, deklarišemo niz line[] da bude veličine 81. Deklarišemo karakter pointer ch_ptr, koji inicijalizujemo da pokazuje na prvi član niza line[], i inicijalizujemo count na nulu. Povećanjem vrednosti pokazivača ch_ptr za jedan, tako da pokazuje na sledeći karakter stringa, brojač se povećava za jedan, i petlja se izvršava do trenutka kada pointer ch_ptr ne pokaže na null karakter.

Reverzno prikazivanje stringa

Nakon unosa sa tastature stringa, program ispisuje obrnutim redosledom karaktere stringa.

```
// Ovaj program ispisuje obrnutim redosledom karaktere stringa koje korisnik unese sa tastature

#include <iostream>

using namespace std;
int main()
{
 char line[81];
 char* ch_ptr = line;
```

```
 cout << "Enter a line of characters:" << endl << endl;
 cin.getline(line, 81);

 // trazi kraj stringa

 while ( *ch_ptr != '\0' )
   ++ch_ptr;

 // ch_ptr sada pokazuje na null karakter
 --ch_ptr;

 // ch_ptr sada pokazuje na poslednji karakter u stringu

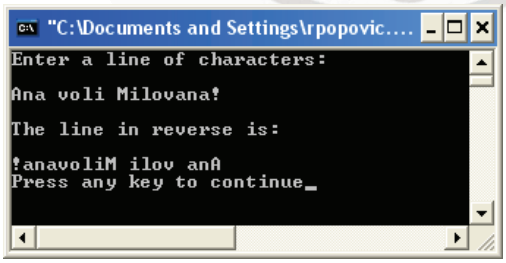
 cout << endl;
 cout << "The line in reverse is:" << endl << endl;

 // while petlja prikazuje sve karaktere osim prvog

 while ( ch_ptr != line )
 {
   cout << *ch_ptr;
   --ch_ptr;
 }
 // prikazuje se prvi karakter

 cout << *ch_ptr;

 cout << endl;
 return 0;
}
```



Brojanje reči u stringu

Posmatramo program za brojanje reči u stringu:

```
// Ovaj program broji reci u stringu koji unosi korisnik
```

```
#include <iostream>

using namespace std;

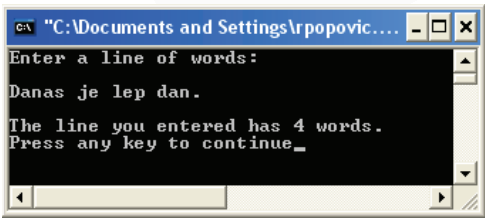
int main()
{
  char line[81];
  char* ch_ptr;
  int word_count = 0;

  cout << "Enter a line of words:" << endl << endl;
  cin.getline(line, 81);

  // Preskace sve prazne pozicije na pocetku linije
  for (ch_ptr = line; *ch_ptr == ' '; ++ch_ptr)
    ;
  // Procesiranje ostatka linije
  while (*ch_ptr != '\0')
  {
    ++word_count;              // Brojanje reci

    // Preskakanje reci
    for ( ; (*ch_ptr != ' ') && (*ch_ptr != '\0'); ++ch_ptr)
      ;
    // Preskakanje praznih pozicija nakon reci
    for ( ; *ch_ptr == ' '; ++ch_ptr)
      ;
  }
  cout << "\nThe line you entered has " << word_count << " words."
       << endl;

  return 0;
}
```



6.3 Nizovi stringova i pokazivača

U jeziku C++ elementi niza mogu biti podaci bilo kog tipa. U ovom poglavlju posmatraćemo nizove stringova i nizove pokazivača.

Definisanje niza stringova

Podsetimo se da je dvodimenzionalni niz ustvari niz čiji su svi elementi takođe nizovi. Na primer, niz

```
char dani_u_nedelji[7][10];
```

je niz od sedam elemenata od kojih je svaki niz od deset karaktera. Niz karaktera može da skladišti string. Zbog toga, dvodimenzionalni niz karaktera, kao što je dani_u_nedelji[][], je sposoban za smeštanje stringova. Razmotrimo, na primer, sledeću deklaraciju i inicijalizaciju:

```
char dani_u_nedelji [7][10] = {"Ponedeljak", "Utorak", "Sreda",
                               "Cetvrtak", "Petak", "Subota", "Nedelja"};
```

Niz karaktera dani_u_nedelji[][] veličine 7 x 10 karaktera prikazan je na slici:

dani_u_nedelji

0	P	o	n	e	d	e	lj	a	k	\0
1	U	t	o	r	a	k	\0			
2	S	r	e	d	a	\0				
3	C	e	t	v	r	t	a	k	\0	
4	P	e	t	a	k	\0				
5	S	u	b	o	t	a	\0			
6	N	e	d	e	lj	a	\0			

Sada kada smo smestili imena dana u nedelji, kako im pristupiti u programu? Svaki element niza dani_u_nedelji[][] je niz karaktera. Sledi, referenciranje dani_u_nedelji[3] pristupa elementu broj tri niza, što je niz karaktera koji sadrži string "Cetvrtak."

Korišćenje niza stringova - primer

Menadžer prodaje želi da zna ukupnu prodaju za svakog prodavca u svom departmanu i da pronađe dan u nedelji kada neko iz prodaje ima najveću prodaju. Sledi rešenje:

```
// Ovaj program pronalazi dan najvece prodaje u nedelji.
// Koristi se niz stringova za skladistenje imena dana u nedelji

#include <iostream>
#include <iomanip>

using namespace std;
int main()
{
 char dani_u_nedelji[7][10] = {"Ponedeljak", "Utorak", "Sreda",
                "Cetvrtak", "Petak", "Subota", "Nedelja"};
 double sales[7];
 char   salesperson[41];
 double max_sales,
       total_sales;
 int    day,
       max_day;

 cout << setprecision(2)
    << setiosflags(ios::fixed)
    << setiosflags(ios::showpoint);

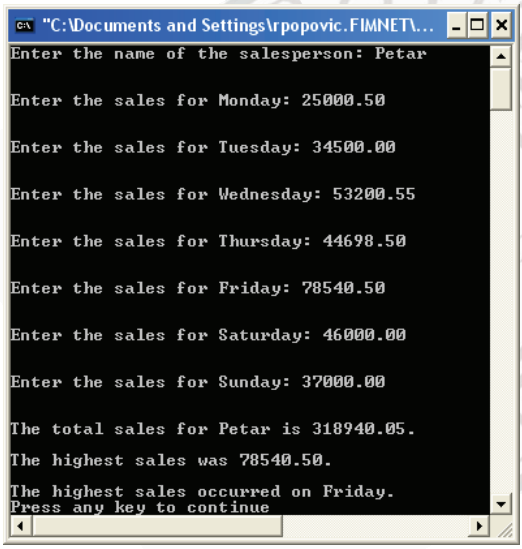
 cout << "Enter the name of the salesperson: ";
 cin.getline(salesperson, 41);

 for (day = 0; day < 7; ++day)
 {
  cout << endl << endl;
  cout << "Enter the sales for "<< dani_u_nedelji[day] << ": ";
  cin >> sales[day];
 }

 total_sales = 0;
 max_day = 0;
 max_sales = sales[0];

 for (day = 0; day < 7; ++day)
 {
  if (sales[day] > max_sales)
```

```
    {
     max_sales = sales[day];
     max_day = day;
    }
   total_sales += sales[day];
  }
  cout << endl << endl;
  cout << "The total sales for " << salesperson
       << " is " << total_sales << "." << endl << endl;
  cout << "The highest sales was " << max_sales << "." << endl << endl;
  cout << "The highest sales occurred on "
       << dani_u_nedelji[max_day]<< "." << endl;
  return 0;
}
```

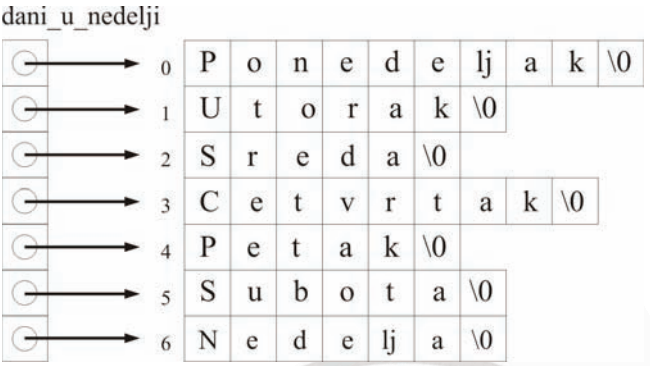


Korišćenje niza pokazivača za skladištenje stringova

Metod koji smo koristili za skladištenje stringova u dvodimenzionalni niz je veoma neefikasan. Niz dani_u_nedelji[][], veličine 7 x 10 okupira 70 bajtova memorije. Svaki red je veličine 10 karaktera bez obzira na veličinu stringa. U ovom slučaju 19 % veličine niza se gubi. Ponekad su neka imena veoma duga pa je problem još veći.

Efikasniji način skladištenja imena je definisanje niza imena kao niza karakter pointera.

```
char* dani_u_nedelji [7] = {"Ponedeljak", "Utorak", "Sreda",
                            "Cetvrtak", "Petak", "Subota", "Nedelja"};
```



Ova deklaracija dani_u_nedelji[] kao niza pokazivača, čuva memoriju zato što svaki string dobija onoliko memorije kolike je dužine.

6.4 Pokazivači, reference na promenljive i funkcije

Funkcija može da ima pokazivač kao argument i pozivajućoj funkciji može da se vrati pokazivač. Može da vrati pokazivač pozivajućoj funkciji. Ovo poglavlje objašnjava tu problematiku.

Poziv po adresi – pokazivači kao argumenti funkcija

Argumenti funkcija u C++ se prosleđuju po vrednosti. Ovo znači da se vrednost argumenta, ne sam argument, prosleđuje funkciji. Razmotrimo sledeći program:

```
// Ovaj program prikazuje prosledjivanje argumenata po vrednosti

#include <iostream>

using namespace std;

int Neg_By_Value(int);

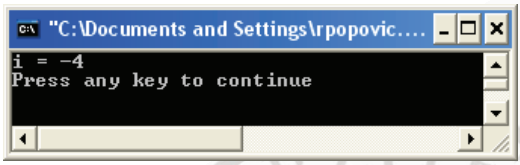
int main()
{
  int i = 4;

  i = Neg_By_Value(i);
```

```
 cout << "i = " << i << endl;

 return 0;
}

int Neg_By_Value(int a)
{
 return -a;
}
```



Funkcija main() u naredbi dodele poziva funkciju Neg_By_Value() i prosleđuje vrednost varijable i, tj. 4, funkciji. Funkcija nakon poziva vraća negativnu vrednost svog argumenta. Na taj način funkcija Neg_By_Value() ne menja vrednost svog argumenta. Funkcija vraća vrednost -4. Naredba dodele u glavnoj funkciji main() zatim dodeljuje -4 promenljivoj i.

U jeziku C++, promenljiva bilo kog tipa može da se prosledi funkciji kao argument, čak i pokazivač. Kada je pokazivač na promenljivu prosleđen kao argument funkcije, funkcija dobija adresu promenljive. Koristeći operator indirekcije, funkcija može da manipuliše sa vrednošću varijable kao da je varijabla lokalna funkciji. Sledeći program daje negativnu vrednost celobrojnoj promenljivoj, ali koristi funkciju koja ima pointer argument.

```
// Ovaj program prikazuje prosledjivanje argumenta po adresi

#include <iostream>

using namespace std;

void Neg_By_Address(int*);

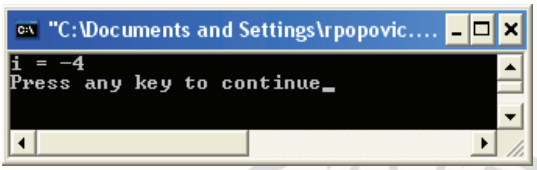
int main()
{
 int i = 4;

 Neg_By_Address(&i);
```

```
  cout << "i = " << i << endl;

  return 0;
}

void Neg_By_Address(int* a_ptr)
{
  *a_ptr = -(*a_ptr);
}
```



Funkcija Neg_By_Address() menja znak svom celobrojnom argumentu. Tip (int*) u prototipu funkcije kaže kompajleru da je argument pokazivač na ceo broj. Tako, kada koristimo funkciju, argument mora biti adresa celog broja. Treba obratiti pažnju da funkcija ne vraća vrednost.

Da bi promenila znak vrednosti i, funkcija main() izvršava sledeću naredbu.

```
Neg_By_Address(&i);
```

Ovaj poziv funkcije prosleđuje adresu varijable i, ne vrednost promenljive i, funkciji Neg_By_Address(). U funkciji Neg_By_Address(), parametar a_ptr je deklarisan kao pokazivač na ceo broj. Kada funkcija main() pozove funkciju, funkcija smešta adresu koju main() prosleđuje pointeru a_ptr.

Pokazivač a_ptr sada pokazuje na varijablu i. Zbog toga, funkcija može da koristi operator indirekcije da manipuliše targetom pokazivača a_ptr, nazvanim i. Telo funkcije sadrži samo sledeću naredbu:

```
*a_ptr = -(*a_ptr); // dodela sadrzaja lokacije na koju pokazuje pokazivac sa
                    // promenjenim znakom
```

Ova naredba zamenjuje vrednost cilja pointera a_ptr, odnosno vrednost i u memoriji, sa negativnom vrednošću targeta na koji pokazuje a_ptr. Sledi, funkcija Neg_By_Address() menja vrednost i u funkciji main().

Prosleđivanje adrese varijable funkcije se označava kao poziv po adresi (eng. *call by address*).

Funkcija može da vrati najviše jednu vrednost pozivajućoj funkciji. Međutim, koristeći poziv po adresi, moguća je promena dve ili više varijabli u pozivajućoj funkciji. Podsetimo se da je funkcija zamene dve celobrojne promenljive (Swap_Int()) definisana ranije kao deo *bubble sort* algoritma.

```
#include <iostream>

using namespace std;

void Swap_Int(int*, int*);

int main()
{
  int i = 3,
      j = 5;
  cout << "Before swapping: i = " << i<< " j = " << j << endl;
  Swap_Int(&i, &j);

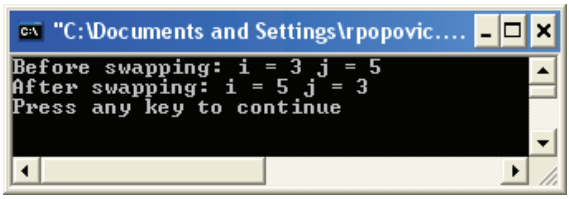
  cout << "After swapping: i = " << i<< " j = " << j << endl;

  return 0;

}

void Swap_Int(int* a_ptr, int* b_ptr)
{
  int temp;

  temp = *a_ptr;
  *a_ptr = *b_ptr;
  *b_ptr = temp;
}
```



Prototip funkcije Swap_Int() deklariše njenu povratnu vrednost da bude tipa void zato što funkcija ne vraća vrednost. Umesto toga, Swap_Int() zamenjuje vrednosti dve celobrojne promenljive u pozivajućoj funkciji koristeći pokazivače. Oba argumenta funkcije su deklarisana kao integer pointeri. Sledi, mora da prosledimo adrese dve celobrojne promenljive funkciji.

Deklaracija varijabli i i j u main() funkciji inicijalizuje varijable na 3 i 5, respektivno. Naredba cout prikazuje početne vrednosti za i i j. Zatim main() funkcija izvršava Swap_Int() funkciju izvršavajući sledeću naredbu, koja prosleđuje adrese za i i j funkciji Swap_Int().

```
Swap_Int(&i, &j);
```

Funkcija Swap_Int() dodeljuje ove adrese parametrima a_ptr i b_ptr. Swap_Int() koristi operator indirekcije za razmenu vrednosti targeta a_ptr i b_ptr, sa imenima i i j. Na taj način funkcija menja vrednosti dve promenljive u funkciji main().

Reference na varijable i poziv po referenci

Reference na promenljive i argumente su ponekad alternativa pokazivačima. Referenca na promenljivu (izvedeni tip) je alias, odnosno drugo ime za varijablu koja već postoji. Na primer, pretpostavimo da imamo sledeće deklaracije

```
int    i = 7;
double d = 1.2;
```

Možemo da deklarišemo reference na varijable za i i d na sledeći način:

```
int&    r = i;
double&    s = d;
```

Znak & u ovim deklaracijama se čita kao "referenca". Prva deklaracija se čita kao "r je integer referenca (referenca na promenljivu tipa integer) inicijalizovana na i" a "s je double referenca inicijalizovana na d."

Razmišljajte o imenu varijable kao o oznaci pridruženoj lokaciji varijable u memoriji, odnosno o drugoj oznaci pridruženoj memorijskoj

lokaciji. Sledi, možete pristupiti sadržaju varijable ili preko originalnog imena varijable ili preko reference.

Možemo da promenimo sadržaje promenljive i koristeći identifikator i ili referencu r. Na primer, bilo koja od ovih naredbi promeniće vrednost promenljive i na 9.

```
i = 9;
r = 9;
```

Kao i kod pokazivača, dodajemo & u deklaraciji reference na kraju tipa podataka da bi naglasili da je referenca drugi tip. Znak *ampersand* možemo smestiti bilo gde između tipa podatka i imena reference na varijablu ispravno je:

```
double & s = d;  //ispravna deklaracija reference

int &r = i;      // ispravna deklaracija reference
```

Pokazivači i reference

Sličnosti između pokazivača i reference:

- -pristup do objekta i preko pokazivača i preko reference je posredan,
- -mnoga pravila, a naročito pravila konverzije, važe i za pokazivače i za reference.

Razlike između pokazivača i reference:

- -pokazivač se može preusmeriti tako da ukazuje na drugi objekat, dok je referenca od trenutka svog nastanka, tj. od inicijalizacije, trajno vezana za isti objekat,
- -pokazivač može da ukazuje ni na šta (vrednost 0), dok referenca uvek, od početka do kraja svog životnog veka, upućuje na jedan (isti) objekat,
- -pristup do objekta preko pokazivača vrši se preko operatora *, dok je pristup do objekta preko reference neposredan.

Ne postoje nizovi referenci, pokazivači na reference, ni reference na reference. U mnogim slučajevima, lakše je koristiti reference od pokazivača

kao argumente funkcija. U ovim slučajevima, prosleđujemo argumente po referenci ili koristeći poziv po referenci. Kao primer poziva po referenci posmatrajmo sledeći program:

```
// Ovaj program prikazuje prosledjivanje argumenata po referenci

#include <iostream>

using namespace std;

void Neg_By_Reference(int&);

int main()
{
  int i = 4;

  Neg_By_Reference(i);

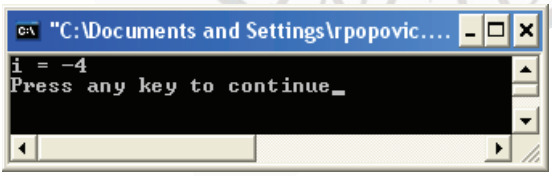
  cout << "i = " << i << endl;

  return 0;
}

void Neg_By_Reference(int&  a)

{

  a = -a;
}
```



Kao drugi primer poziva po referenci posmatramo sledeći program:

```
// Ovaj program prikazuje prosledjivanje argumenata po referenci

#include <iostream>

using namespace std;

void Swap_Int(int&, int&);
```

```
int main()
{
  int i = 3,
      j = 5;

  cout << "Before swapping: i = " << i<< " j = " << j << endl;

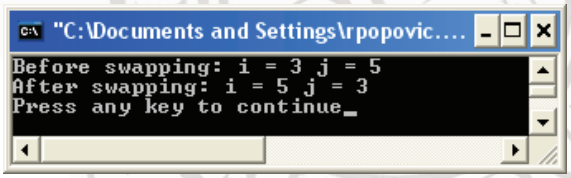
  Swap_Int(i, j);

  cout << "After swapping: i = " << i<< " j = " << j << endl;

  return 0;
}

void Swap_Int(int& a, int& b)
{
  int temp;

  temp = a;
  a    = b;
  b    = temp;
}
```



6.5 Nizovi i funkcije

Ime niza je pokazivač na prvi član niza. C-string je karakter pointer. Sledi, prosleđivanje niza ili C-stringa funkciji, ekvivalentno je prosleđivanju pokazivača.

Prosleđivanje niza funkciji

Da bi se prosledio niz funkciji, treba da se urade dve stvari u programu, jedna u prototipu funkcije i jedna u hederu definicije funkcije.

Pretpostavimo da funkcija Avg() treba da pronađe srednju vrednost celobrojnih članova niza. Prosleđujemo niz funkciji i to preko jednog parametra od dva parametra funkcije. Drugi parametar je veličina niza. Prototip funkcije je oblika:

```
double Avg(int [], int);
```

Funkcija Avg() vraća vrednost tipa double i ima dva parametra. Prvi parametar je niz promenljivih tipa int. Srednje zagrade [] kažu kompajleru da se radi o nizu. Definicija funkcije koja odgovara prethodnom prototipu je:

```
double Avg(int arr[], int size)
```

Sledeći program nalazi srednju vrednost skupa od 10 poena (ocena) sa kviza koje korisnik unosi sa tastature, ali koristi funkciju Avg().

```
// Ovaj program pokazuje kako se nalazi srednja vrednost elemenata niza
// prosledjivanjem niza funkciji

#include <iostream>
#include <iomanip>

using namespace std;

double Avg(int [], int);

int main()
{
  const int NUM_QUIZZES = 10;

  int    grade[NUM_QUIZZES]; // niz za skladistenje ocena sa kviza
  int    quiz;               // Indeks niza
  double grade_avg;

  cout << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint)
       << setprecision(1);

  cout << "Please enter " << NUM_QUIZZES << " integer quiz grades."
       << endl << endl;

 for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
 {
  cout << "\nEnter grade for quiz " << quiz + 1 << ": ";
  cin >> grade[quiz];
 }

 grade_avg = Avg(grade, NUM_QUIZZES);

 cout << endl;
```

```
  cout << "The average quiz grade is " << grade_avg << endl;

  return 0;

}   // Kraj main() funkcije

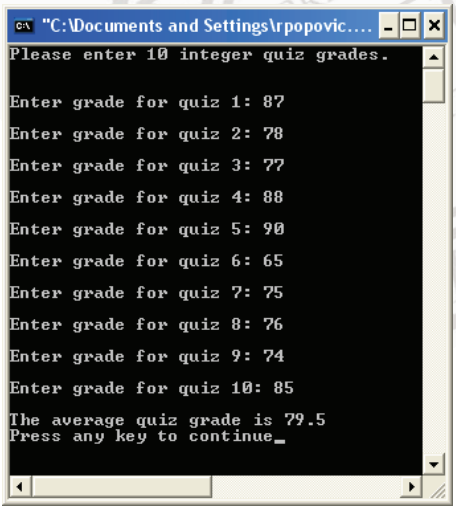
double Avg(int arr[], int size)
{
  int    i,              // Indeks niza
         sum = 0;        // Suma ocena
  double avg;            // Prosecna ocena

  for (i = 0; i < size; ++i)
    sum += arr[i];

  avg = double(sum) / size;

  return avg;

}   // Kraj Avg() funkcije
```



Kada prosleđujemo ime niza funkciji, prosleđujemo adresu niza, koja je pokazivač na prvi element niza. Ne prosleđujemo vrednost elemenata niza ili kopiju celog niza.

Sortiranje niza

Prosleđivanjem niza funkciji koristeći ime niza kao argument, prosleđuje se adresa niza funkciji. Sledi, bilo koja promena koju funkcija načini nad parametrom niz menja niz u pozivajućoj funkciji.

U sledećem programu korisnik sa tastature unosi elemente niza (10), program prikazuje niz, pa se zatim koristi *bubble sort* algoritam za sortiranje po rastućem redosledu. Konačno, program prikazuje sortiran niz.

```
// Ovaj program prikazuje kako se sortiraju elementi niza.
// Program koristi funkcije za rad sa nizovima.
// Funkcija Swap_Int() koristi poziv po referenci

#include <iostream>
#include <iomanip>

using namespace std;

void Get_Grades(int [], int);
void Display_Array(int [], int);
void Bubble_Sort(int [], int);
void Swap_Int(int&, int&);
int main()
{
  const int NUM_QUIZZES = 10;

  // Niz za skladištenje ocena sa kviza
  int grade[NUM_QUIZZES];

  Get_Grades(grade, NUM_QUIZZES);

  cout << endl << endl;
  cout << "The grades are as follows:";
  Display_Array(grade, NUM_QUIZZES);

  Bubble_Sort(grade, NUM_QUIZZES);

  cout << endl << endl;
  cout << "The grades in increasing order are as follows:";
  Display_Array(grade, NUM_QUIZZES);

  cout << endl;
  return 0;

}    // Kraj funkcije main()
```

```
void Get_Grades(int arr[], int size)
{
 int quiz;

 cout << endl;
 cout << "Please enter " << size << " integer quiz grades."
      << endl << endl;

 for (quiz = 0; quiz < size; ++quiz)
 {
  cout << endl;
  cout << "Enter grade for quiz :" << quiz + 1 << ": ";
  cin >> arr[quiz];
 }

 return;

}   // Kraj funkcije Get_Grades()

void Display_Array(int arr[], int size)
{
 int i;

 cout << endl;
 for (i = 0; i < size; ++i)
  cout << setw(6) << arr[i];
 cout << endl;

 return;

}   // Kraj funkcije Display_Array()

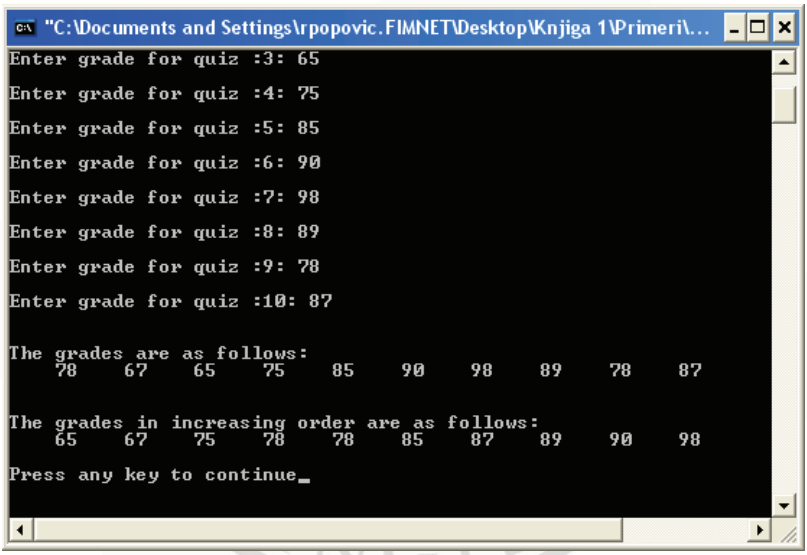
void Bubble_Sort(int arr[], int size)
{
 int i,                    // Indeks niza
     pass,                 // Broj prolaza
     limit = size - 2;   // Vodi racuna koliko daleko ici u jednom prolazu

 for (pass = 1; pass <= size - 1; ++pass)
 {
  for (i = 0; i <= limit; ++i)
    if (arr[i] > arr[i + 1])
      Swap_Int(arr[i], arr[i + 1]);
  --limit;
 }
```

```
 return;

}   // Kraj funkcije Bubble_Sort()

void Swap_Int(int& a, int& b)
{
 int temp;      // privremena promenljiva za zamenu celobrojnih vrednosti
 temp = a;
 a = b;
 b = temp;
 return;
}   // Kraj Swap_Int() funkcije
```



Kada glavna funkcija main() poziva funkciju Bubble_Sort(), ona prosleđuje adresu niza grade[]. Sledi, Bubble_Sort() manipuliše elementima niza grade[]. Na taj način kada funkcija Bubble_Sort() završi sortiranje niza, elementi niza grade[] su uređeni po rastućem redosledu. Nakon toga, glavna funkcija main() prikazuje na displeju poruku i koristi funkciju Display_Array() da bi prikazala elemente niza grade[] u rastućem redosledu.

6.6 Stringovi i funkcije

Pošto je C-string ekvivalentan pokazivaču na karaktere, prosleđivanje stringa funkciji je ekvivalentno prosleđivanju pokazivača funkciji. U ovoj sekciji prikazaćemo kako se prosleđuju stringovi

funkcijama. Takođe, prikazaćemo kako se definiše i koristi funkcija koja vraća pokazivač.

Korišćenje funkcije za brojanje karaktera u stringu

Program pita korisnika da unese sa tastature string a zatim program prikazuje broj karaktera u stringu. Program koristi funkciju Len_String() koja uzima string kao svoj jedini argument i vraća ceo broj jednak broju karaktera u stringu. Prototip funkcije je definisan na sledeći način:

```
int Len_String(char*);

// Ovaj program izracunava broj karaktera jedne linije koju korisnik unese sa
// tastature
// Za brojanje karaktera koristi se funkcija Len_String()

#include <iostream>
using namespace std;

int Len_String(char*);

int main()
{
 char line[81];

 cout << "Enter a line of characters:" << endl;
 cin.getline(line,81);

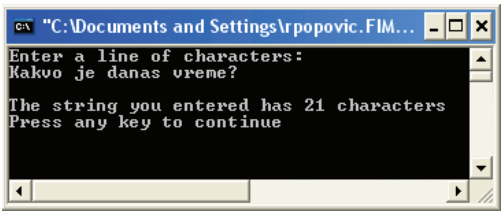
 cout << endl;
 cout << "The string you entered has " << Len_String(line)
      << " characters" << endl;

 return 0;
}     // Kraj funkcije main()

int Len_String(char* ch_ptr)
{
 int length = 0;

 while (*ch_ptr != '\0')
 {
  ++length;
  ++ch_ptr;
 }
```

```
 return length;
}      // Kraj funkcije Len_String()
```



Parametar deklarisan u hederu funkcije Len_String() je karakter pointer ch_ptr. Kada glavna funkcija main() izvršava poziv funkcije Len_String(line), ona prosleđuje vrednost line (što je pokazivač na prvi elemenat niza line[]) funkciji Len_String(). Vrednost line se dodeljuje ch_ptr. Sledi, u funkciji Len_String(), ch_ptr inicijalno pokazuje na prvi karakter stringa koji je smešten u line[]. Lokalna promenljiva length, koja predstavlja dužinu stringa, inicijalizovana je na 0.

Petlja while u funkciji Len_String() testira target pokazivača ch_ptr. Ako target nije null karakter, koji je kraj stringa, onda se u telu petlje broji taj karakter povećavajući vrednost varijable length. Zatim, petlja povećava pokazivač za jedan.

Korišćenje funkcije za reverzno smeštanje elemenata stringa

Ova funkcija menja elemente niza reverzno, tj. prikazuje obrnuto elemente niza.

```
// Ovaj program koristi funkciju za reverzni prikaz stringa.

#include <iostream>

using namespace std;

void Reverse_String(char*);

int main()
{
 char line[81];

 cout << "Enter a string:" << endl;
 cin.getline(line, 81);
```

```
  Reverse_String(line);
  cout << endl;
  cout << "The string in reverse is as follows:"  << endl;
  cout << line << endl;

  return 0;
}
void Reverse_String(char* ch_ptr)
{
  char* front;  // Pokazuje na pocetak stringa
  char* end;    // Pokazuje na kraj stringa
  char temp;    // Potrebno za zamenu karaktera

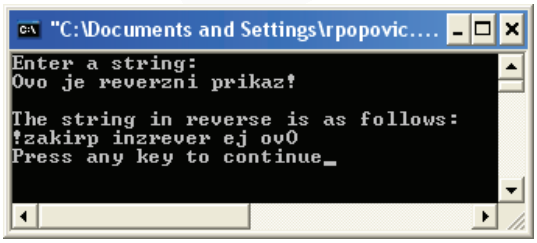
  // inicijalizacija pokazivaca
  front = end = ch_ptr;

  // pronalazenje kraja stringa
  while (*end != '\0')
    ++end;

  --end;

  // zamena karaktera
  while (front < end)
  {
    temp   = *front;
    *front = *end;
    *end   = temp;
    ++front;
    --end;
  }

  return;
}
```



Nakon dobijanja stringa od korisnika, program pravi sledeći poziv funkcije:

```
Reverse_String(line);
```

Funkcija Reverse_String() reverzno menja string "u mestu". Da bi uradila ovo funkcija deklariše dva karakter pointera front i end. Druga petlja while reverzno menja string zamenjujući karaktere na koje pokazuju front i end.

Standardna biblioteka string funkcija

C++ jezik sadrži oko dvadeset bibliotečkih funkcija koje manipulišu C-stringovima. Ove funkcije nam dozvoljavaju da izvršimo neke proste string operacije.

Prototipovi svih funkcija za rad sa stringovima nalaze se u heder fajlu cstring (u starijim implementacijama C++ jezika, koristio se heder fajl string.h). Sledi, treba uključiti sledeću pretprocesorsku direktivu:

```
#include <cstring>
```

Dužina stringa - funkcija strlen()

Bibliotečka funkcija strlen() nalazi dužinu stringa. Na primer:

```
int length;
char greeting[6] = "Hello";
length = strlen(greeting);
```

Dodela u trećem redu smešta 5 u varijablu length.

Dodela stringa - funkcija strcpy()

Ako želimo da smestimo kopiju stringa greeting1 u string greeting2

```
char greeting1[6] = "Hello";
char greeting2[10];
```

dodela sledećeg oblika je pogrešna (kompajler prijavljuje grešku), zato što je ime niza konstantan pointer čija se vrednost ne može menjati:

```
greeting2 = greeting1; // pogresna dodela
```

Bibliotečka funkcija strcpy() dodeljuje jedan string drugom kopirajući izvorni string, karakter po karakter u target string, čija veličina je odgovarajuća. Funkcija ima sledeći format:

```
strcpy(target-string-name, source-string-name)
```

Kopiranje stringa greeting1 u string greeting2 je pomoću sledeće naredbe:

```
strcpy(greeting2, greeting1);
```

Funkcija strcpy() vraća pokazivač na prvi karakter u *target* stringu. Ako pretpostavimo sledeću deklaraciju na početku programa:

```
char* ch_ptr;
```

sledeća naredba je ispravna:

```
ch_ptr = strcpy(greeting2, greeting1);
```

Na taj način, ch_ptr pokazuje na prvi karakter greeting2[].
Možete napisati i sledeći kôd koji prikazuje string koji je u greeting2[]:

```
cout << strcpy(greeting2, greeting1);
```

Sledeća naredba kopira "llo" u greeting2:

```
strcpy(greeting2, greeting1 + 2);
```

Poređenje stringova - funkcija strcmp()

Za poređenje dva stringa koristi se bibliotečka funkcija strcmp(), sa sledećim formatom:

```
strcmp(string1, string2)
```

Kada funkcija strcmp() upoređuje dva stringa, ona počinje poređenje od prvog karaktera svakog stringa. Ako su karakteri jednaki, upoređuje se sledeći par karaktera, itd.

Poređenje karakter po karakter nastavlja se dok se ne naiđe na različite karaktere (funkcija vraća negativan ceo broj ako prvi string prethodi drugom, a pozitivan ceo broj ako prvi string sledi drugi, prema vrednostima karaktera u tabeli ASCII koda) ili dok se ne dođe do kraja karaktera koji su isti (funkcija vraća 0).

Pretpostavimo sledeće deklaracije i inicijalizacije:

```
char string1[5] = "abcd";
char string2[5] = "abCd";
char string3[7] = "abcd  ";
char string4[5] = "abCd";
```

strcmp(string1, string2) vraća positivan ceo broj.
strcmp(string1, string3) vraća negativan ceo broj.
strcmp(string2, string4) vraća 0.

Dodavanje stringova - funkcija strcat()

Ponekad je neophodno dodati jedan string na kraj drugog. Na primer:

```
char string1[27] = "abcdef";
char string2[27] = "ghij";
```

Pretpostavimo da želimo da dodamo sadržaj string2 na kraj sadržaja string1. Sledi, nakon dodavanja, string1 treba da sadrži string "abcdefghij". Format poziva funkcije je:

```
strcat(target-string, source-string)
```

pa se funkcija poziva na sledeći način:

```
strcat(string1, string2);
```

Funkcije za klasifikaciju prema karakterima i funkcije za konverziju

Standardna C++ biblioteka sadrži nekoliko funkcija koje su korisne za rad sa stringovima.

Funkcije za klasifikaciju prema karakterima

Ponekad je neophodno klasifikovati karaktere prema zajedničkim osobinama, npr. da li je karakter iz skupa znakova interpunkcije. U donjoj tabeli su date standardne bibliotečke funkcije klasifikovane prema karakterima. Imena svih funkcija počinju sa "*is*" (da li je) i imaju jedan argument tipa int. Treba zapamtiti da možete da memorišete karakter u varijablu tipa int. Ove funkcije takođe rade ispravno ako im se prosledi argument tipa char. Svaka funkcija vraća vrednost različitu od nule (*true*) ako je testirani karakter u određenoj kategoriji, i vraća nulu (*false*) ako karakter nije u određenoj kategoriji. Prototipovi funkcija su u heder fajlu **cctype**.

Ime funkcije	Kategorija
isalnum(ch)	broj ili slovo (veliko ili malo)
isalpha(ch)	slovo (veliko ili malo)
isdigit(ch)	broj od 0 do 9
islower(ch)	malo slovo
ispunct(ch)	znak interpunkcije (karakter koji nije prazan, nije broj ni slovo)
isspace(ch)	*white-space* karakter (*blank, newline, carriage return, vertical tab*, ili *form feed*)
isupper(ch)	veliko slovo

Funkcije za konverziju karaktera

Standardna C++ biblioteka sadrži nekoliko funkcija za konverziju individualnih karaktera.

Ako int varijabla ch sadrži malo slovo, funkcija toupper(ch) vraća odgvarajuće veliko slovo kao integer. Ako int varijabla ch sadrži veliko slovo, funkcija tolower(ch) vraća odgovarajuće malo slovo kao integer. U ostalim slučajevima, funkcija vraća nepromenjeni argument. Funkcije se nalaze u cctype.

Primer palindroma koji koristi više funkcija

Reči "otto" i "mom" su palindromi - isto se čitaju unapred i unazad. String može biti palindrom ako se ignorišu prazna mesta i znaci interpunkcije. Na primer string "Madam, I'm Adam" je palindrom.

Napisaćemo program koji pita korisnika da ubaci string (do 80 karaktera) i odlučuje da li je string palindrom.

Veoma je prosto da se odluči da li je reč palindrom. Proverava se da se vidi da li su prvi i poslednji karakter isti. Ako nisu, string nije palindrom. Ako su isti, proverava se sledeći par karaktera. Kada se stigne do sredine reči i ako su svi karakteri jednaki, reč je palindrom.

String mora biti očišćen od praznih karaktera, sa malim slovima i bez znakova interpunkcije.

Primer koristi string "Madam, I'm Adam!" i program skladišti string u niz line[]. Da bi se očistio string od blanko znakova i znakova interpunkcije, program kopira string u drugi niz buffer[], kopira samo slovne karaktere. Program zatim menja sve karaktere u nizu buffer[] u mala slova. Konačno, program proverava da li je rezultujuća reč palindrom.

```
// Ovaj program koristi bibliotecke funkcije za rad sa stringovima i karakterima
// kako bi ocenio da li je string koji korisnik unosi palindrom.
// Palindrom je string koji se isto cita unapred i unazad
// String mora biti ociscen od praznih karaktera, sa malim slovima i bez znakova
// interpunkcije

#include <iostream>
#include <cctype>

using namespace std;

void Copy_Alpha(char*, char*);
void Convert_To_Lower(char*);
bool  Palindrome(char*);

int main()
{
  char line[81];
  char buffer[81];

  // Unost stringa

  cout <<"Enter a string:" << endl;
  cin.getline(line, 81);

  // Kopiranje samo slovnih karaktera

  Copy_Alpha(buffer, line);

  // Konvertuje string u mala slova
```

```
  Convert_To_Lower(buffer);
  // Testira se rezultat da li je palindrom

  if ( Palindrome(buffer) )
  {
    cout << endl;
    cout <<"The string you entered is a palindrome." << endl;
  }
  else
  {
    cout << endl;
    cout <<"The string you entered is not a palindrome." <<endl;
  }

  return 0;

} // Kraj main() funkcije
void Copy_Alpha(char* target, char* source)
{
  while (*source != '\0')
  {
    if (isalpha(*source))
    {
      *target = *source;
      ++target;
    }
    ++source;
  }
  *target = '\0';

  return;

} // Kraj Copy_Alpha() funkcije
void Convert_To_Lower(char* source)
{
  while (*source != '\0')
  {
    *source = tolower(*source);
    ++source;
  }

  return;

}  // Kraj Convert_To_Lower() funkcije

bool Palindrome (char* source)
```

```
{
  char* start;
  char* end;

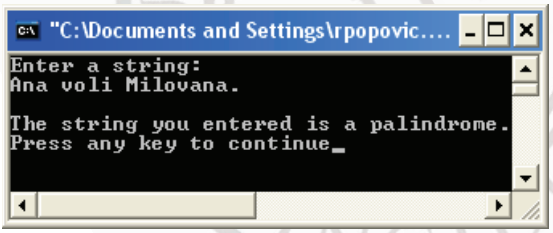
  start = end = source;

  while (*end != '\0')
    ++end;

  --end;

  while (start < end)
    if (*start != *end)
      return false;
    else
    {
      ++start;
      --end;
    }

  return true;
}  // Kraj Palindrome() funkcije
```



Funkcije za numeričku konverziju i numeričku proveru

Sledeće funkcije konvertuju numerički string u broj. U svakom slučaju, funkcija vraća broj koji numerički string predstavlja. Svaka funkcija prestaje sa konverzijom stringa kada naiđe na znak koji nije numerički, funkcija tada vraća nulu. Mora da se uključi i #include <cstdlib> da bi se koristile funkcije.

atoi() konvertuje numerički string u integer.
atol() konvertuje numerički string u long.
atof() konvertuje numerički string u double.

Dinamička alokacija memorije
Hip (heap) memorija

Pretpostavimo da želimo da napišemo program koji treba da memoriše imena pet klijenata, koje treba korisnik da unese sa tastature. Pretpostavimo da je najveće ime dužine 80 karaktera. Možete da deklarišete niz na sledeći način:

```
char client[5][81];
```

Niz client[][] zahteva 5 x 81 = 405 bajtova memorije. Međutim, imena su proseku duga 15 do 20 karaktera. Verovatno se gubi oko 300 ili više od 405 bajtova dodeljenih nizu client[][]. U slučaju da koristimo niz pokazivača kao u prethodnom poglavlju, problem je što unapred ne znamo dužinu niza.

Kad god je memorija potrebna programu, moguće je u C++ alocirati memoriju za promenljive, nizove itd. iz specijalne zone memorije koja se zove hip (*heap*). Hip memorija je kompletno pod kontrolom korisnika, tj. korisnik može da alocira i dealocira memoriju kada je to neophodno. Proces alokacije i dealokacije memorije kada se program izvršava zove se dinamička alokacija memorije.

C++ operator new alocira hip memoriju, a operator delete dealocira hip memoriju. Operator new ima jedan argument, tip podatka skladišta koje želimo da alociramo, i vraća pointer na alocirani prostor. Na primer:

```
int* i_ptr;
i_ptr = new int;
```

Operator new alocira dovoljno hip memorije za skladištenje intidžera i vraća pointer na tu memoriju. Naredba dodele dodeljuje taj pointer i_ptr. Ako nema dovoljno slobodne heap memorije da bi se zadovoljio alokacioni zahtev, new vraća null pokazivač.

Kao drugi primer posmatramo dinamičku alokaciju memorije niza od 20 celih brojeva.

```
int* arr_ptr;
arr_ptr = new int [20];
```

Memorija za statičke i automatske promenljive i hip memorija

Hip memorija se razlikuje od drugih zona memorije koje vaš C++ program koristi, kao što su memorija za statičke i automatske (static i automatic) promenljive. Ako se deklariše promenljiva kao static ili ako se deklariše promenljiva van funkcije, prostor za promenljivu se alocira iz static memorije pre nego što program počne. Samo kada se main() završi računar dealocira prostor za static varijable. Automatska (automatic) varijabla, tj. varijabla deklarisana unutar bloka, kreira se u zoni zvanoj *stack* kada program ulazi u blok. Takve varijable se dealociraju kada program napušta blok. Prostor za automatic varijable se alocira i dealocira kada se program izvršava. Međutim, alokacija i dealokacija su dati automatski. Programer nema kontrolu nad alokacijom i dealokacijom automatskih varijabli.

Pre izlaska iz funkcija operator oslobađanja memorije delete treba da dealocira hip memoriju koja je alocirana operatorom new.

6.7. Korisnički definisani tipovi podataka i tabele

Naredbe koje koriste typedef i enum

U ovoj sekciji razmotrićemo typedef za definisanje sinonima za postojeći tip, i enum za definisanje novog tipa čije promenljive mogu biti male celobrojne vrednosti.

Naredba typedef

Naredba typedef ne definiše novi tip podatka nego drugo ime za postojeći tip. Naredba je sledećeg oblika:

```
typedef old_type new_type;
```

Na primer:

```
double vrednost, kolicina;
```

Ako se ove varijable odnose na novac, deklarišimo promenljive kao

```
typedef double NOVAC_KOLICINA;
NOVAC_KOLICINA vrednost, kolicina;
```

Ako je:

```
typedef char NAME[51];
```

možemo da deklarišemo nove nizove first i last da budu nizovi sa 51 karakterom:

```
NAME first, last;
```

Naredba enum

Tip nabrajanja koji koristi rezervisanu reč enum, definiše celobrojne konstante koje su predstavljene identifikatorima na sledeći način:

```
enum enumerated-type-name {identifier list}
```

Na primer, sledeća naredba deklariše tip nabrajanja RESPONSE:

```
enum RESPONSE {cancel, ok};
```

Jednom kada deklarišete tip nabrajanja, možete da deklarišete promenljive tog tipa. Na primer, možemo da napišemo kôd za sledeću deklaraciju:

```
RESPONSE result;
```

Sledi, promenljivoj se može dodeliti:

```
result = ok;
```

Tip nabrajanja C++ kompajler posmatra kao novi tip.

Nabrojani tipovi vrednosti su celi brojevi (na primer, u tipu RESPONSE type, ok ima vrednost 1). Međutim, C++ kompajler izdaje poruku upozorenja za sledeću naredbu:

```
result = 1;
```

Varijabla result je tipa RESPONSE a konstanta 1 je ceo broj. Mada naredba dodeljuje korektnu vrednost varijabli result, kompajler izdaje upozoravajuću (warning) poruku da tipovi možda nisu kompatibilni.

U deklaraciji tipa nabrajanja ne treba da se da lista identifikatora vrednosti koje su uzastopne od 0 pa naviše. U deklaraciji

```
enum COLOR {red = 3, green, blue, white = 8, black};
```

red ima vrednost 3, green 4, blue 5, white 8, black 9.

6.8 Strukture

Strukture su složeni tipovi podataka koji se sastoje od uređenih nizova elemenata koji mogu da budu međusobno različitih tipova. Ti elementi nazivaju se polja strukture. Polja struktura se obeležavaju identifikatorima. Opšti oblik opisa strukture u naredbama za definisanje podataka je:

```
struct ime_strukture {niz_deklaracija}
```

Pretpostavimo da treba da procesiramo informacije o inventaru. Na primer, treba da uskladištimo broj dela (kôd od sedam karaktera), količinu (ceo broj) i jediničnu cenu (promenljiva tipa double). Možemo da deklarišemo promenljivu za svaku osobinu. Međutim, gubimo informaciju da se osobine odnose na isti deo. Potrebna nam je jedna promenljiva gde ćemo da smestimo sve tri veličine. Većina programskih jezika koristi varijablu nazvanu zapis-*record* za ovaj problem. U jeziku C++, struktura može da reši problem.

Sledeća definicija strukture rešava problem inventara.

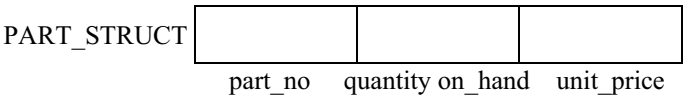
```
struct PART_STRUCT
{
  char   part_no[8];
  int    quantity_on_hand;
  double unit_price;
};
```

Definicija počinje sa struct (rezervisana reč). Identifikator PART_STRUCT je ime strukture. U velikim zagradama se nalaze članovi strukture. Definicija strukture se završava sa tačkom zarez.

Važno je shvatiti da definicija strukture PART_STRUCT ne deklariše promenljivu. Ona ne rezerviše bilo koji prostor u memoriji. Definicija strukture ustvari definiše šablon za kompajler kada deklarišemo

PART_STRUCT promenljive. Kao i kod enum, naredba struct ustvari definiše drugi tip podatka, sama po sebi ne deklariše promenljivu. Takođe, part_no, quantity_on_hand, i unit_price članovi strukture su prosto imena strukture i nisu promenljive.

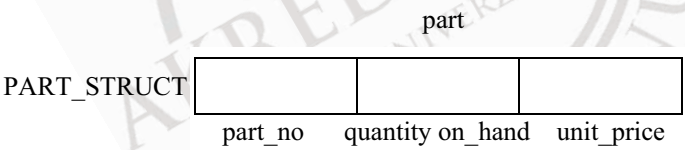
Struktura PART_STRUCT je podeljena na tri dela odnosno tri člana, part_no, quantity_on_hand, i unit_price, po tom redosledu.



Da bismo deklarisali promenljivu koja je deo tipa PART_STRUCT, pišemo sledeći kôd:

```
PART_STRUCT part;
```

Ova deklaracija definiše promenljivu part, koja je podatak tipa PART_STRUCT. Promenljiva part ima strukturu koju smo definisali u naredbi struct za PART_STRUCT. Na donjoj slici je prikazana struktura promenljive part



Mada tip podatka PART_STRUCT ne rezerviše memoriju računara, varijabla part to čini, odnosno zahteva neophodnu veličinu memorije za skladištenje niza od osam karaktera, celog broja i promenljive tipa double.

Pristup članovima strukture

Članovima strukture se pristupa koristeći operator člana (*member operator*) koji se kodira kao tačka između imena promenljive strukture i člana strukture kome se pristupa, na primer:

```
part.quantity_on_hand = 62;
```

Ova dodela se čita kao "62 se dodeljuje promenljivoj quantity_on_hand članu strukture part".

Sledeći program prikazuje kako pristupiti članu strukture:

```
// Ovaj program ilustruje deklaraciju i koriscenje promenljive tipa strukture
// i operatora clan za pristup varijablama koje su clanovi strukture.

#include <iostream>
#include <iomanip>

using namespace std;

struct PART_STRUCT
{
  char   part_no[8];
  int    quantity_on_hand;
  double unit_price;
};

int main()
{
  PART_STRUCT part;

  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

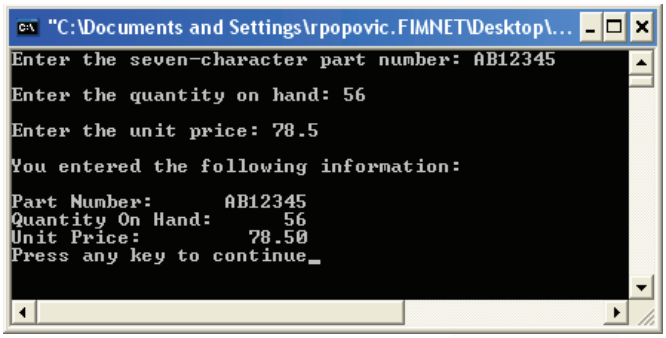
  cout << "Enter the seven-character part number: ";
  cin.getline(part.part_no, 8);

  cout << endl;
  cout << "Enter the quantity on hand: ";
  cin >> part.quantity_on_hand;

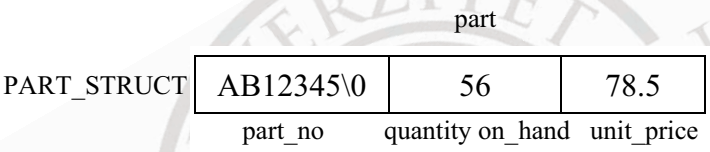
  cout << endl;
  cout << "Enter the unit price: ";
  cin >> part.unit_price;

  cout << endl;
  cout << "You entered the following information:" << endl << endl;
  cout << "Part Number:      " << part.part_no << endl;
  cout << "Quantity On Hand: " << setw(7) << part.quantity_on_hand
       << endl;
  cout << "Unit Price:       " << setw(7) << part.unit_price << endl;

  return 0;
}
```



Nakon dobijanja podataka sa tastature od korisnika, možemo da označimo sadržaj promenljivih part tipa strukture kao na slici:



Inicijalizacija promenljive tipa structure

Promenljiva tipa struktura može da se inicijalizuje slično inicijalizaciji niza. Na primer, koristeći PART_STRUCT tip podatka koji je definisan ranije, možemo da deklarišemo i inicijalizujemo promenljivu old_part na sledeći način:

```
PART_STRUCT old_part = {"XY98765", 17, 99.99};
```

Složene strukture

Struktura može, takođe, da bude složena, odnosno može da sadrži druge ugnježđene strukture. Na primer, pretpostavimo da želimo da memorišemo sledeće informacije o zaposlenima kompanije: ime, adresu, broj socijalnog osiguranja i vrednost radnog časa. Kako su ime i adresa sastavljeni iz više delova definisaćemo ih preko struktura.

```
struct NAME_STRUCT
{
  char first_name[31];
  char mid_initial;
```

```
  char last_name[31];
};

struct ADDRESS_STRUCT
{
  char st_adress[31];
  char city[31];
  char state[3];
  char zip[6];
};

struct EMPLOYEE_STRUCT
{
  NAME_STRUCT    name;
  ADDRESS_STRUCT address;
  char           soc_sec_no[10];
  double         pay_rate;
};

EMPLOYEE_STRUCT employee;
```

Promenljiva employee tipa struktura ima četiri člana, kao na slici. Prvi član name je struktura tipa NAME_STRUCT. Drugi član address je struktura tipa ADDRESS_STRUCT, treći član soc_sec_no je niz kraktera. Četvrti član je tipa double.

Ako želimo da inicijalizujemo kod za državu od dva karaktera za promenljivu employee na "NY", treba da koristimo operator člana dva puta:

```
employee.address.state
```

Da bismo kopirali string "NY" u niz, koristimo funkciju strcpy() na sledeći način:

```
strcpy(employee.address.state, "NY");
```

Ako želimo da pristupimo prvom karakteru koda države, koristićemo sledeću referencu:

```
employee.address.state[0]
```

6.9 Nizovi struktura: tabele

Tabele su podaci organizovani u redove i kolone. Na primer donja slika prikazuje listu delova. Svaki red tabele predstavlja deo. Prva kolona sadrži broj dela, druga sadrži količinu, a treća jediničnu cenu dela.

Part Number	Quantity on Hand	Unit Price
A123456	123	12.99
A987654	53	52.99
D001234	93	22.95
B109897	44	13.95
A035467	15	59.95
C837289	0	57.99
D346246	35	25.99
C253146	104	110.99
B194791	33	14.95
A826749	56	41.95

Definisanje tabele koristeći strukture

Tabela u jeziku C++ je niz struktura. Na primer, posmatrajmo strukturu PART_STRUCT:

```
struct PART_STRUCT
{
  char   part_no[8];
  int    quantity_on_hand;
  double unit_price;
};
```

Možemo da definišemo tabelu za skladištenje podataka deset delova na sledeći način:

```
PART_STRUCT part_table[10];
```

Dakle, part_table je niz od 10 elemenata od kojih je svaki tipa PART_STRUCT.

Ako želimo da dodelimo vrednost 102 članu quantity_on_hand u petom redu tabele (sa indeksom 4), pisaćemo:

```
part_table[4].quantity_on_hand = 102;
```

Kada se vrednosti u tabeli ne menjaju često, možete na sledeći način ubaciti vrednosti tabele u program.

```
PART_STRUCT partial_part_table[4] = { {"A123456", 123, 12.99},
                                     {"A987654", 53, 52.95},
                                     {"D001234", 93, 22.95},
                                     {"B109897", 44, 13.95}
                                     };
```

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

7. Uvod u klase i objekte

7.1 Objekti, klase i objektno orijentisani sistemi

Klasa je realizacija apstrakcije koja ima svoju internu predstavu (svoje atribute) i operacije koje se mogu vršiti nad njenim primercima. Klasa definiše tip. Jedan primerak takvog tipa (instanca klase) naziva se objektom te klase.

Podaci koji su deo klase nazivaju se podaci članovi klase. Funkcije koje su deo klase nazivaju se funkcije članice klase.

Objekat klase ima unutrašnje stanje, predstavljeno vrednostima atributa, koje menja pomoću operacija. Funkcije članice nazivaju se još i metodima klase, a poziv ovih funkcija – upućivanje poruke objektu klase. Objekat klase menja svoje stanje kada se pozove njegov metod, odnosno kada mu se uputi poruka. Moguće je preklopiti (overload) funkcije članice.

U objektno orijentisanom programiranju aplikaciju vidimo kao sistem objekata koji međusobno interaguju. Svaki objekat je odgovoran za svoje akcije. Objekti interaguju slanjem poruka i ponašanjem na odgovarajući način kao odziv na poruke.

Kao primer, pretpostavimo da želite da podignete €100 sa računa u banci na ATM (*automatic teller machine*). Da biste podigli €100, šaljete poruku ATM objektu (koristeći touch screen) za izvršenje podizanja €100. ATM objekat šalje poruku vašem štednom računu (savings account objektu koji se nalazi u bazi podataka banke) da se podigne €100. ATM objekat može da uradi ovo šaljući poruku Withdraw (100.00) vašem savings account objektu. Objekat savings account sada određuje da li je njegovo stanje (količina novca na vašem savings account) veće ili jednako €100.00. Ako jeste, objekat smanjuje njegovo stanje za €100 i šalje poruku ATM objektu da isplati €100. ATM mašina kao odgovor na tu poruku, isplaćuje pet novčanica od €20 i prikazuje poruku da je novac spreman. Vaš odziv odnosno odgovor na tu poruku je da uzmete novac iz automata.

Ispitajmo sistem malo detaljnije. Objekat štedni račun koji predstavlja vaš račun je jedan od mnogih takvih bankovnih računa. Svi oni imaju stanje i svi se ponašaju na sličan način. Možemo da kažemo da je svaki objekat za čuvanje računa član klase Savings_Account. U terminologiji objektno orijentisanog programiranja, kažemo da pojedinačni račun savings account predstavlja instancu-primerak klase Savings_Account. Svi članovi klase mora da se ponašaju na isti način. Podizanje novca sa računa poziva

isti metod (niz akcija) kao što je podizanje novca sa vašeg računa. Ono što se razlikuje su stanja, odnosno vrednosti podataka koji se nalaze u objektu. Stanje savings account objekta može da uključi korisnikov ID član, balance – stanje i interest rate - kamatnu stopu.

Ovo je princip skrivanja informacija – pošiljalac poruke ne treba da zna način na koji se primalac brine o zahtevu.

Koja je razlika između objektno orijentisanog pristupa slanja poruka i proceduralnog pristupa koji koristi funkcije? Prvo, kod objektno orijentisanog programiranja, poruke imaju prijemnik. Funkcije nemaju prijemnik. Funkcije obično rade na nečemu. Na primer, proceduralna verzija ATM sistema za podizanje €100 izvršava poziv funkcije na sledeći način:

```
Withdraw(your_account, 100.00);
```

Funkcija Withdraw() će ustvari raditi na računu. S druge strane, kod objektno orijentisanog programiranja ATM objekat zahteva vaš account objekat za izvršenje servisa, za podizanje €100 sa svog sopstvenog stanja.

Druga razlika između slanja poruka i korišćenja funkcija je da kod objektno orijentisanog pristupa prijemnik određuje interpretaciju poruke. Zbog toga interpretacija može da varira kod različitih tipova prijemnika.

Sličnost klasa i objekata

U prethodnom poglavlju predstavili smo standardne ulazne i izlazne strimove cin i cout. C++ jezik dolazi sa nekoliko predefinisanih klasa. Dve od njih su klasa koja definiše ulaz istream i klasa ostream, koja definiše izlaz. Standardni ulazni strim cin je objekat klase istream. Definicije klasa istream i ostream i definicije objekata cin i cout se nalaze u heder fajlu iostream.

U program dovodimo podatke sa tastature primenjujući operator ekstrakcije-izvlačenja >> na cin objekat. Naredba

```
cin >> price;
```

kaže cin objektu da "izvuče" broj iz ulaznog strima i smesti njegovu vrednost u varijablu price. Na taj način, šaljemo poruku cin objektu i on odgovara na odgovarajući način na tu poruku.

Na sličan način možemo da prikažemo podatke na monitoru primenjujući operator insertovanja-ubacivanja << na cout objekat. Možemo da vidimo naredbu

```
cout << "The value is " << price;
```

kao slanje poruke cout objektu da prikaže na ekranu monitora string "The value is" za kojim sledi vrednost varijable price.

Koristimo get() i cin na sledeći način

```
ch = cin.get();
```

gde je ch varijabla tipa karakter.

Funkcija get() je metod istream klase. Kada pišemo kod sa cin.get(), šaljemo get() poruku cin objektu. Kao odziv na ovu poruku, objekat vraća programu vrednost sledećeg karaktera u ulaznom baferu. Treba obratiti pažnju da koristimo isti operator . (tačka) koji smo koristili kod struktura.

Isto je i sa getline(). Kada napišemo

```
cin.getline(name, 81);
```

šaljemo getline() poruku cin objektu. Kao odgovor na poruku, cin objekat "izvlači" celu liniju teksta iz ulaznog bafera i smešta njenu kopiju u niz name.

7.2 Uvod u string objekte

C-string smo definisali kao niz karaktera. Objekti klase string su lakši za rad, bezbedniji i prirodniji za korišćenje od C-stringova. Nije važno da znate kako su string objekti ili operacije na njima implementirani. Važno je da znate kako da koristite string objekte.

Klasa string je ugrađena u C++, tako da je nije potrebno deklarisati na bilo koji način osim uključenjem direktive

```
#include <string>
```

Kreiranje string objekata

Pre nego što naučimo kako da koristimo string objekte, mora da znamo kako da napravimo primerak (instancu) objekta, odnosno kako da ga kreiramo. Mada string objekat predstavlja više, u suštini on je skup karaktera. Skup karaktera string objekta zvaćemo vrednost string objekta. Ovaj skup karaktera je smešten u string objekat, moguće zajedno sa drugim

podacima kao što je dužina stringa. Ovi podaci uključuju stanje string objekta. Obratite pažnju da je string objekat više nego njegova vrednost.

Treba napomenuti da reč string nije C++ ključna reč, to je ime klase.

U C++, bilo koja klasa, uključujući i string, je validan tip podatka. Sledi, mogu se deklarisati string objekti skoro na isti način kao što se deklarišu numerički tipovi podataka. Posmatrajmo sledeće deklaracije:

```
string str1;
string str2 = "Hello";
string str3("Hello");
string str4(50, '*');
```

String koji nije inicijalizovan u svojoj deklaraciji automatski se postavlja na prazan string po *default*, ili null string, tj. string koji nema karaktere, kao što je slučaj sa str1 u prvoj deklaraciji. Ovo je suprotno u odnosu na deklaraciju neinicijalizovanog intidžera ili karakter pointera, koji ima garbage vrednost. Kada se kreira jedan objekat, izvršava se specijalan metod, nazvan konstruktor. Glavni cilj konstruktora je inicijalizacija objekta. Kako ni jedna početna vrednost nije određena u prvoj deklaraciji, objekat izvršava takozvani konstruktor bez argumenata, koji inicijalizuje vrednost stringa na prazan string.

Druga i treća deklaracija definišu str2 i str3 kao string objekte koji imaju vrednost "Hello" (bez znaka navoda). U oba ova slučaja, konstruktor sa jednim argumentom inicijalizuje vrednost stringa na "Hello". Treba obratiti pažnju da je treća deklaracija slična pozivu funkcije, mada str3 nije ime funkcije.

Praktično nema ograničenja za broj karaktera koje string objekat može da uskladišti.

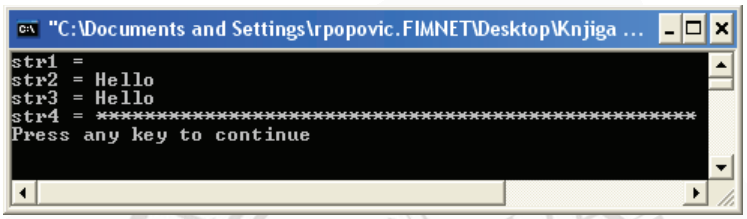
Četvrta deklaracija definiše str4 kao string od 50 zvezdica (asteriska *). Ova deklaracija koristi konstruktor sa dva argumenta za inicijalizaciju vrednosti stringa. Prvi argument (ceo broj) je broj ponavljanja drugog argumenta (karakter) u stringovoj vrednosti.

```
//Ovaj program prikazuje razlicite nacine deklarisanja string objekata
// i prikazuje na izlazu string objekte

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

```
  string str1;
  string str2 = "Hello";
  string str3("Hello");
  string str4(50, '*');

  cout << "str1 = " << str1 << endl;
  cout << "str2 = " << str2 << endl;
  cout << "str3 = " << str3 << endl;
  cout << "str4 = " << str4 << endl;
  return 0;
}
```



U ovom programu string "Hello" nije string objekat - on je string literal. Zato, konstruktor sa jednim argumentom mora da konvertuje string literal u ekvivalentan objekat.

Kako objekat str1 nije inicijalizovan, prva linija na izlazu ne prikazuje ništa sa desne strane znaka dodele (str1 je prazan string). Objekti str2 i str3 imaju istu vrednost "Hello", a str4 je string od 50 zvezdica.

String Input

Možete ubaciti vrednosti string objekata koristeći cin >> operator, kao u primeru:

```
string first_name;
cout << "Enter your first name: ";
cin >> first_name;
```

Sledeći program primenjuje ovaj deo koda:

```
//Ovaj program koristi cin za unos imena i prezimena u program

#include <iostream>
#include <string>

using namespace std;
```

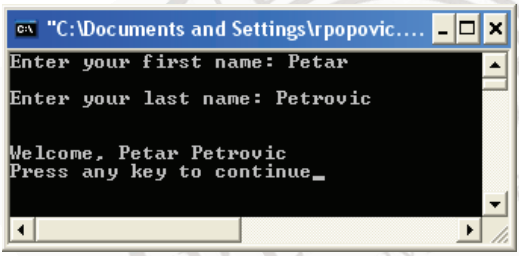
```
int main()
{
  string first_name;
  string last_name;

  cout << "Enter your first name: ";
  cin >> first_name;

  cout << endl;
  cout << "Enter your last name: ";
  cin >> last_name;

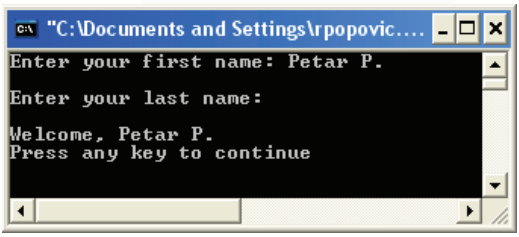
  cout << endl << endl;
  cout << "Welcome, " << first_name << " " << last_name << endl;

  return 0;
}
```



U poslednjoj cout naredbi, smeštamo blanko prostor kao string literal "" u izlazni strim između imena i prezimena.

Šta se dešava kada na pitanje za unos imena unesemo pored imena i srednje slovo sa tačkom?



Kada se pritisne *enter* taster nakon ukucavanja Petar P., program ne čeka na nas da ubacimo *last name*, već prikazuje na displeju pozdravnu poruku. Program se ponaša ovako zbog toga što kada se ubaci string

podatak cin koristi *white space* (*blank*, ili *enter*, ili *tab*) da odvoji stringove. Sledi cin može da se iskoristi za ulaz samo jedne reči u jednom trenutku.

Za unos stringa sastavljenog od više reči, mora da koristimo funkciju getline(). Na primer:

```
string state;

cout << "Enter your state of residence: ";

getline(cin, state);
```

Funkcija getline() pita cin da uzme jednu liniju iz ulaznog strima tj. sve što korisnik ukuca dok ne pritisne taster *enter*. Tekst koji je ukucan (bez *enter* karaktera) se predaje string objektu state, koji uzima tekst kao njegovu vrednost. Funkcija getline() nije metod string klase.

```
//Ovaj program ilustruje koriscenje funkcije getline() za unos stringa sastavljenog
//iz vise reci

#include <iostream>
#include <string>

using namespace std;

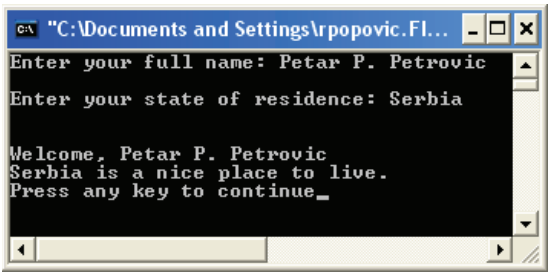
int main()
{
 string full_name;
 string state_name;

 cout << "Enter your full name: ";
 getline(cin, full_name);

 cout << endl;
 cout << "Enter your state of residence: ";
 getline(cin, state_name);

 cout << endl << endl;
 cout << "Welcome, " << full_name << endl;
 cout << state_name << " is a nice place to live." << endl;

 return 0;
}
```




```
//Program izracunava srednju vrednost rezultata sa tri testa

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

int main()
{
   int  quiz1,          // Ocena sa prvog testa
        quiz2,          // Ocena sa drugog testa
        quiz3;          // Ocena sa treceg testa

  double average;     // Prosecna ocena sa sva tri testa
  string name;         // String koji cuva ime studenta

  // Postaviti izlazno stampanje sa jednim decimalnim mestom

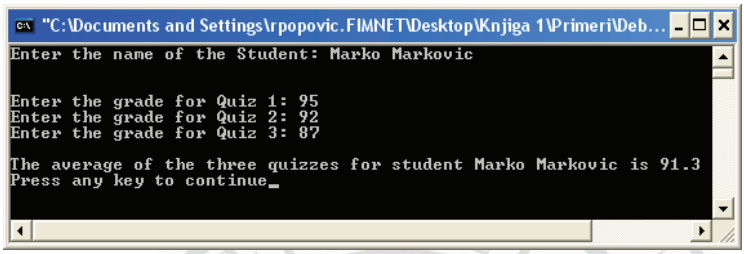
  cout << setprecision(1)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);
  cout << "Enter the name of the Student: ";
  getline(cin, name);

  cout << endl;
  cout << "Enter the grade for Quiz 1: ";
  cin >> quiz1;
  cout << "Enter the grade for Quiz 2: ";
  cin >> quiz2;
  cout << "Enter the grade for Quiz 3: ";
  cin >> quiz3;

// Tip cast double() se zahteva za konvertovanje sume ocena sa kviza, koja je
// integer u tip floating point. Greska u konverziji izraza sa desne strane dodele kao
// rezultat ce dati da je srednja vrednost ceo broj bez decimalnog broja.
```

```
  average = double (quiz1 + quiz2 + quiz3) / 3;

  cout << endl;
  cout << "The average of the three quizzes for student "
       << name << " is " << average << endl;
  return 0;
}
```



Rad sa string objektima

Klasa string ima više metoda koji vam dozvoljavaju da dobijete informacije o stringu i da radite sa vrednostima string objekta. Takođe, možete da koristite više operatora u radu sa string objektima skoro na isti način na koji ih koristite u radu sa numeričkim tipovima podataka.

Dodela stringova

Ispravno je dodeliti jedan string objekat drugom na sledeći način:

```
string str1 = "Hello";
string str2;

str2 = str1;
```

Nakon dodele, vrednost string objekta str2 je kopija vrednosti string objekta str1. To je suprotno u odnosu na dodelu C-stringova.

Možete da inicijalizujete jedan string objekat drugim kao u sledećoj deklaraciji:

```
string str1 = "Hello";
string str2 = str1;
```

ili na sledeći način:

```
string str1 = "Hello";
string str2(str1);
```

Spajanje dva stringa

Možete spojiti dva stringa koristeći operator +, isti operator koji smo koristili za sabiranje brojeva. Na primer, sledeća deklaracija smešta string "Hello, World" na kraj vrednosti string objekta str3:

```
string str1 = "Hello, "
string str2 = "World!"
string str3 = str1 + str2;
```

U C++ moguće je definisati operator, na primer kao operator sabiranja (+), da radi različite stvari zavisno od tipova podataka na koje se operator odnosi. Ovo se naziva preklapanje operatora (*overloading*). Sledi, operator sabiranja u izrazu i + j, gde su i i j celi brojevi, radiće drugačije od operatora sabiranja u izrazu str1 + str2. Preklapanje operatora ne radi automatski za bilo koju kombinaciju objekata. Preklapanje mora biti negde isprogramirano. U slučaju klase string, preklapanje je isprogramirano u heder fajlu <string>.

Na primer:

```
string str3 = "Hello, ";

str3 += "World!";
```

Deklaracija inicijalizuje str3 na string "Hello, ". Druga naredba je ekvivalentna dodeli str3 = str3 + "World!";, što kao rezultat daje da str3 sadrži string "Hello, World!".

Sledi, operatori =, +, += se mogu preklopiti za string objekte.

Dužina stringa

Metod length() klase string služi za određivanje broja karaktera u vrednosti string objekta (dužina stringa). Da bismo primenili metod na objekat str1 koristimo operator tačka (*dot* .) na sledeći način:

```
str1.length()
```

odnosno,

class_object.method_name()

Sledeći program prikazuje operacije koje smo do sada razmatrali:

```
//Ovaj program ilustruje dodelu i spajanje stringova
//kao i length() metod.

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

int main()
{
  string first_name;
  string last_name;
  string full_name;

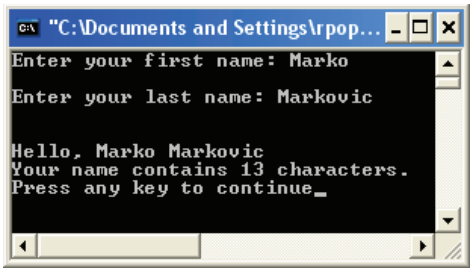
  int name_length;

  cout << "Enter your first name: ";
  getline(cin, first_name);

  cout << endl;
  cout << "Enter your last name: ";
  getline(cin, last_name);

  full_name = first_name + " " + last_name;
  name_length = full_name.length() - 1;
  cout << endl <<endl;
  cout << "Hello, " << full_name << endl;
  cout << "Your name contains " << name_length << " characters." << endl;

  return 0;
}
```



Pristup određenim karakterima u stringu

Da bi se pristupilo pojedinim karakterima string objekta, treba da se zna da su karakteri u vrednosti string objekta numerisani počevši od nule (indeksi).

0	1	2	3	4
H	e	l	l	o

Prvi karakter ima indeks 0, dok poslednji karakter ima indeks koji je jednak dužini stringa -1.

Dobiti ili promeniti individualni karakter u string objektu možemo koristeći metod at() string klase. Ako je str1 string objekat, onda izvršavanjem str1.at(i) pitamo string objekat str1 da vrati karakter sa indeksom i u stringovoj vrednosti.

Posmatrajmo sledeći kôd:

```
string str1 = "Hippopotamus";
int last_index = str1.length() - 1;
cout << "Prvi karakter u stringu je " << str.at(0) << endl;
cout  << "Poslednji karakter u stringu je " << str.at(last_index)
     << endl;
```

String objekat je inicijalizovan na "Hippopotamus". Promenljiva last_index je inicijalizovana na dužinu stringa -1. Dve cout naredbe prikazuju prvi karakter (na index-u 0) i poslednji karakter (sa indeksom last_index).

Ovaj metod se može iskoristiti za promenu karaktera u string objektu str1.at(i). Na primer sledeća naredba menja prvi karakter u stringu str u karakter X:

```
str.at(0) = 'X';
```

Sledeći program ilustruje ovaj primer:

```
//Koriscenje metoda at() klase string

#include <iostream>
#include <string>

using namespace std;
```

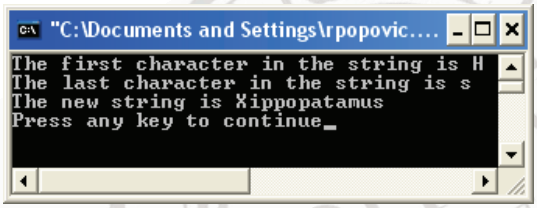
```
int main()
{
 string str = "Hippopatamus";

 int last_index = str.length() - 1;

 cout << "The first character in the string is " << str.at(0) << endl;
 cout << "The last character in the string is " << str.at(last_index)
     << endl;

 str.at(0) = 'X';

 cout << "The new string is " << str << endl;
 return 0;
}
```



7.3 Donošenje odluka u radu sa stringovima

Operatori koje smo koristili za poređenje brojeva su preklopljeni za korišćenje sa string objektima.

Jednakost

Operator jednakosti (= =) se može iskoristiti za poređenje da li su dva stringa jednaka. Dva string objekta su jednaka samo ako su njihove vrednosti (karakteri koje sadrže) identične, karakter po karakter.

```
string str1 = "Hello";
string str2 = str1;
```

Onda test

```
if (str1 == str2)
```

daje *true*.

Nejednakost

Testiranje nejednakosti između dva string objekta može da se obavi koristeći uobičajene operatore !=, >, <, >=, i <=. Na primer, ako su str1 i str2 string objekti, onda izraz str1 < str2 upoređuje vrednosti str1 i str2. Za string objekte operator < treba da se čita kao "predhodi" i operator > treba da se čita kao "sledi", upoređujući vrednosti (karakter po karakter) string objekata koristeći ASCII tabelu.

Sledeći program koristi više različitih koncepata rada sa string klasom.

```
// Ovaj program koristi nekoliko string koncepata

#include <iostream>
#include <string>

using namespace std;

int main()
{
  string first_name;
  string last_name;
  string full_name;
  string full_name_chk;

  cout << "Please enter your first name: ";
  getline(cin, first_name);

  cout << endl;
  cout << "Please enter your last name:" ;
  getline(cin, last_name);

  full_name = first_name + " " + last_name;

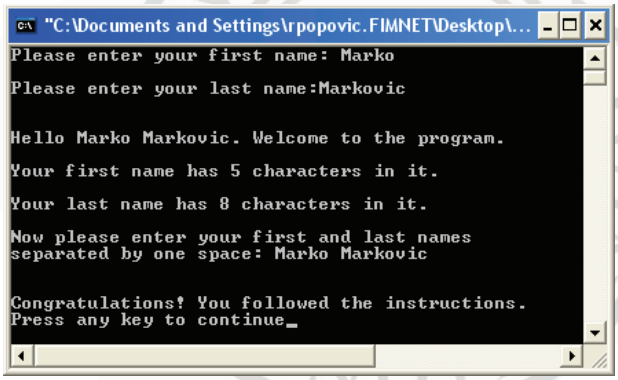
  cout << endl << endl;
  cout << "Hello " << full_name << ". Welcome to the program."
     << endl << endl;
  cout << "Your first name has " << first_name.length()
     << " characters in it." << endl << endl;
  cout << "Your last name has "<< last_name.length()
     << " characters in it." << endl << endl;

  cout << "Now please enter your first and last names" << endl;
```

```
  cout << "separated by one space: ";

  getline(cin, full_name_chk);

  if (full_name == full_name_chk)
  {
   cout << endl << endl;
   cout << "Congratulations! You followed the instructions." << endl;
  }
  else
  {
   cout << endl;
   cout << "You did not follow the instructions!" << endl;
   cout << "Please try the program again." << endl;
  }
  return 0;
}
```



7.4 Funkcije i string objekti

Klasa je složeni tip podatka u jeziku C++. Objekat string može da se koristi kao argument funkcije i funkcija može da vrati string objekat.

Testiranje da li je string prazan

Ponekad je potrebno testirati da li je vrednost string objekta prazna (*null*), odnosno da objekat nema karaktere. Ako je string prazan, metod vraća true, inače, vraća false. Na primer, ako imamo sledeću deklaraciju:

```
string str1;
```

gde string objekat nije inicijalizovan, prazan je, odnosno ima vrednost 0. Sledi, str1.empty() daje true. Ako imamo sledeću deklaraciju:

```
string str2 = "Hello";
```

Sledi, str2.empty() daje false.

Funkcije koje koriste string objekte

Sledeći program koristi funkciju koja ima string argument i vraća string kao rezultat kako bi se proverilo da li je ulazni string prazan.

```
//Ovaj program koristi funkciju koja ima string argument i vraća string kao rezultat
//za proveru da li je ulazni string prazan ili ne

#include <iostream>
#include <string>

using namespace std;

string Get_Name(string);

int main()
{
  string first_name;
  string last_name;
  string full_name;
  string full_name_chk;
  string first_initial;
  string last_initial;
  first_name = Get_Name("Please enter your first name: ");

  last_name = Get_Name("Please enter your last name: ");

  full_name = first_name + " " + last_name;
  first_initial = first_name.at(0);
  last_initial = last_name.at(0);

  cout << endl << endl;
  cout << "Hello " << full_name << ". Welcome to the program."
      << endl << endl;
  cout << "Your first name has " << first_name.length()
      << " characters in it." << endl << endl;
  cout << "Your last name has "<< last_name.length()
```

```
      << " characters in it." << endl << endl;
  cout << "Your initials are " << first_initial + last_initial
      << endl << endl;

  cout << "Now please enter your first and last names" << endl;
  cout << "separated by one space: ";

  getline(cin, full_name_chk);

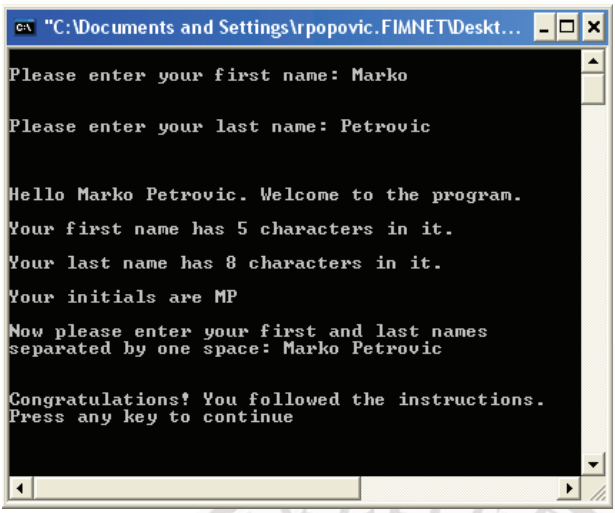
  if (full_name == full_name_chk)
  {
   cout << endl << endl;
   cout << "Congratulations! You followed the instructions." << endl;
  }
  else
  {
   cout << endl;
   cout << "You did not follow the instructions!" << endl;
   cout << "Please try the program again." << endl;
  }
  return 0;
}

string Get_Name(string prompt)
{
  string name;

  bool invalid_data;

  do
  {
   cout << endl << prompt;
   getline(cin, name);
   cout << endl;

   if (name.empty())
   {
    cerr << "You did not enter a name - Please try again." << endl;
    invalid_data = true;
   }
    else
      invalid_data = false;
   }
   while(invalid_data);
   return name;
  }
```



Prototip funkcije Get_Name() je:

```
string Get_Name(string);
```

i pokazuje da će funkcija vratiti string objekat i da funkcija uzima prost string objekat kao argument. Vrednost koju funkcija vraća nije prazan string objekat. Objekat koji prosleđujemo funkciji je *prompt* koji funkcija koristi da dobije ime. Zato, da bi iskoristili funkciju za dobijanje imena koje nije prazno, treba da prosledimo *prompt* kao argument funkcije:

```
first_name = Get_Name("Please enter your first name: ");
```

String objekat koji se koristi kao argument funkcije prosleđen je po vrednosti, kao obična brojna varijabla. Pozvana funkcija radi sa kopijom string objekta. Originalni string objekat u pozivajućoj funkciji ne može biti promenjen od strane pozivajuće funkcije.

Napomena, string objekti se mogu proslediti funkciji po referenci. U tom slučaju funkcija može da promeni string objekat.

Sledeći program ilustruje prosleđivanje stringova po referenci. Koristi se funkcija Swap_Strings() koja ima kao argumente dve string reference.

```
//Ovaj program ilustruje prosledjivanje string objekata po referenci.
//On zamenjuje vrednosti dva string objekta koristeci funkciju
```

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

using namespace std;

void Swap_Strings(string&, string&);

int main()
{
  string str1;
  string str2;

  cout << "Enter string 1: ";
  getline(cin, str1);

  cout << "Enter string 2: ";
  getline(cin, str2);

  cout << endl;

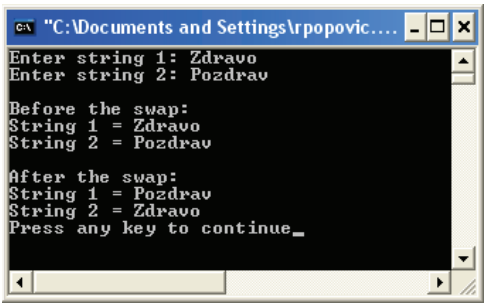
  cout << "Before the swap:" << endl;
  cout << "String 1 = " << str1 << endl;
  cout << "String 2 = " << str2 << endl;

  Swap_Strings(str1, str2);

  cout << endl;
  cout << "After the swap:" << endl;
  cout << "String 1 = " << str1 << endl;
  cout << "String 2 = " << str2 << endl;

  return 0;
}
void Swap_Strings(string& s1, string& s2)
{
  string temp;

  temp = s1;
  s1 = s2;
  s2 = temp;
}
```



Funkciji se može proslediti i pokazivač na string objekat.

7.5 Primeri rada sa string objektima

U ovoj sekciji dati su primeri rada sa metodima string klase i string objektima. Kao primer koristi se string sa slike:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
T	h	i	s		i	s		a	n		e	x	a	m	p	l	e		s	t	r	i	n	g

Pronalaženje stringa u stringu

Prvi metod koji razmatramo je find(). On locira poziciju podstringa u stringu. Na primer, pretpostavimo sledeće deklaracije:

```
string str1 = "This is an example string";
int position;
```

Neka string objekat str1 ima vrednost označenu na slici, onda naredba

```
position = str1.find("an");
```

pita string objekat str1 da pronađe prvo pojavljivanje podstringa "an" u njegovoj vrednosti i vraća vrednost indeksa te pozicije. Naredba će smestiti broj 8 u varijablu position, zato što podstring "an" startuje u lokaciji 8 u stringu str1. U slučaju da string objekat ne locira podstring koji se traži, metod vraća -1.

Druga verzija metoda find() dozvoljava da se startuje pretraživanje lokacije podstringa nakon date pozicije. Na primer, pronaći lokaciju prve blanko pozicije u str1 na ili nakon 11 pozicije.

```
position = str1.find(" ", 11)
```

Naredba smešta broj 18 u varijablu position zato što je sledeća blanko pozicija nakon 11 pozicije u vrednosti str1 na poziciji 18. I ova verzija metoda find() takođe vraća -1 ako podstring nije nađen.

Obe verzije metoda find() počinju pretraživanjem stringa sleva u desno, dok metod rfind() (r u rfind znači "reverse", tj. obrnuto) pretražuje string sdesna u levo.

Pronalaženje određenog karaktera sa date liste

Ponekad je neophodno pronaći prvi karakter u stringu iz grupe karaktera. Na primer, pretpostavimo da želimo da nađemo znak interpunkcije (,.:;!?) u stringu. Metod koji koristimo za pronalaženje prve pojave bilo kog karaktera iz skupa znakova interpunkcije je find_first_of(). Na primer posmatrajmo sledeće deklaracije:

```
string str2 = "Hello, my name is John.";
string punctuation = ",.:;!?";
```

Naredba

```
str2.find_first_of(punctuation);
```

vraća 5, tj. lokaciju zapete (,) u stringu. Ako metod find_first_of() ne pronađe bilo koji od karaktera sa liste, metod vraća -1.

Verzija find_first_of() koja startuje sa pretraživanjem stringa od određene pozicije stringa je

```
str2.find_first_of(punctuation, 7);
```

vraća 22, što je pozicija tačke (.) u stringu

Izvlačenje podstringa iz stringa

Metod substr() izvlači određeni broj karaktera iz vrednosti string objekta počevši od određene pozicije. Na primer, posmatrajmo sledeću deklaraciju:

```
string str2;
```

Sledeća naredba (izvlači dva karaktera-podstring iz vrednosti str1 počevši od lokacije 5) smešta podstring "is" u str2.

```
str2 = str1. substr(5, 2);
```

String metodi se mogu kombinovati međusobno.

Insertovanje i uklanjanje podstringova

Postoje string metodi koji vam dozvoljavaju da insertujete i uklanjate podstring iz stringa. String metod replace() zamenjuje jedan podstring drugim. Na primer, sledeća naredba zamenjuje 4 karaktera (drugi argument) počevši od lokacije 0 (prvi argument) stringom "Here" (treći argument):

```
str1.replace(0, 4, "Here");
```

Metod insert() može da se koristi za insertovanje stringa u vrednost string objekta. Ovo se razlikuje od metoda replace(). Metod replace() zamenjuje podstring drugim stringom. S druge strane metod insert() smešta string na određenu lokaciju u drugom stringu sa karakterima koji slede na desno. Na primer, ako je str1 string čija je vrednost prikazana na ranije iznad rezultat izvršenja naredbe

```
str1.insert(19, "of a ");
```

daje vrednost stringa:

```
"This is an example of a string"
```

Metod append() dodaje string na kraj stringa na koji se primenjuje metod.

String metod erase() briše podstring iz stringa. Na primer, brisanje reči "is" iz str1

```
str1.erase(5, 3);
```

Prvi argument je startna pozicija, a drugi argument je broj karaktera koji se brišu. Nakon izvršenja prethodne narebe, str1 sadrži string

```
"This an example string".
```

Metod erase() može da se koristi sa jednim argumentom. Naredba

```
str1. erase(10);
```

briše sve karaktere iz stringa od pozicije 10 do kraja stringa.

Sledeći program ilustruje string metode koje smo definisali u ovoj sekciji

```
//Ovaj program ilustruje neke od raspolozivih metoda za rad sa string objektima

#include <iostream>
#include <string>

using namespace std;

int main()
{
   string str1 = "This is an example string";
   string str2 = "sample";
   string str3;
   string punctuation = ".,:;!?";

   int position;

   cout << "The original string, str1, is : "
        << str1 << endl << endl;

   //Nadji prvo pojavljivanje dela stringa "is" u stringu str1
   position = str1.find("is");

   cout << "The first occurrence of \"is\" is at position "
        << position << endl << endl;
```

```
// Nadji prvo pojavljivanje dela stringa "is" u str1 nakon pozicije 3
position = str1.find("is", 3);

cout << "The first occurrence of \"is\" after position 3 is "
    << "at position " << position << endl << endl;

// Nadji prvo pojavljivanje dela stringa "is" od kraja stringa str1
position = str1.rfind("is");

cout << "The first occurrence of \"is\" from the end of str1 is "
    << "at position " << position << endl << endl;

//Pokusaj da nadjes podstring koji nije u stringu str1.
position = str1.find("Hello");

if (position == -1)
   cerr << "The substring \"Hello\" was not found in str1."
       << endl << endl;

//prosiri dati string str1 od pozicije 11 podstringom duzine 8 karaktera
str3 = str1.substr(11, 8);
cout << "The 8-charater substring of str1 beginning at position 11 is: "
    << str3 << endl << endl;

//Zameni 7 karaktera na pocetku str1 od pozicije 11 stringom str2
str1.replace(11, 7, str2);

cout << "After replacing the word \"example\" " << endl
    << "by the word \"sample\", in str1 the string is: "
    << str1 << endl << endl;

//Obrisi podstring duzine 7 karaktera stringa str1 od njegove
//11 pozicije od pocetka
strl.erase(11, 7);

cout << "After erasing the word \"sample \", in str1 the string is: "
    << str1 << endl << endl;

//Insertuj novi string "example " na početak stringa od pozicije 11
strl.insert(11, "example ");

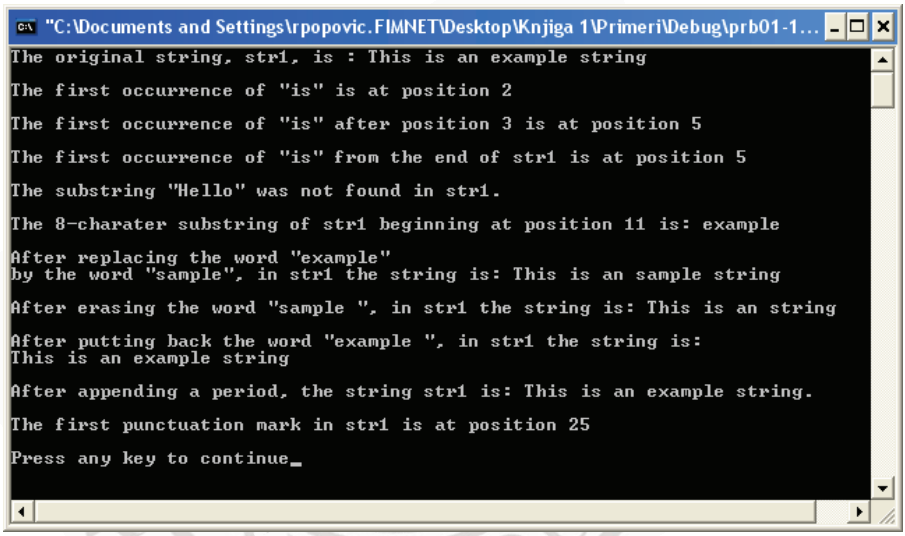
cout << "After putting back the word \"example \", in str1 the string is: "
    << endl << str1 << endl << endl;

//Dodaj tacku na kraj stringa
strl.append(".");
```

```
    cout << "After appending a period, the string str1 is: "
         << str1 << endl << endl;

    // Pronadji prvo pojavljivanje bilo kog znaka interpunkcije u stringu str1
    position = strl.find_first_of(punctuation);

    cout << "The first punctuation mark in str1 is "
         << "at position " << position << endl << endl;
    return 0;
}
```



Slede dva složenija primera rada sa stringovima.

Uklanjanje i štampanje reči iz stringa

```
//Ovaj program pita korisnika da unese string sastavljen iz vise reci
//i zatim stampa jednu rec u jednoj liniji.
//Program koristi find(), length(), substr() i erase() metode klase string

#include <iostream>
#include <string>

using namespace std;

int main()
{
  string the_string;
```

```
  string word;

  int space_location;

  cout << "Enter a string. Separate words by spaces: " << endl << endl;
  getline(cin, the_string);

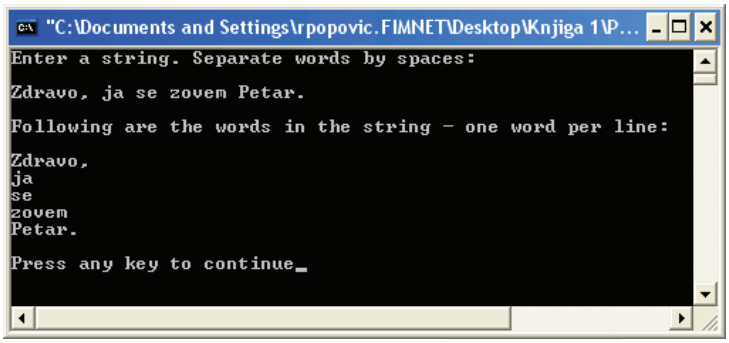
  cout << endl;
  cout << "Following are the words in the string - one word per line: "
       << endl << endl;
//Nadji prvi blanko karakter.
//Ako ne postoji blanko karakter, posalji na izlaz samo tu rec kao string

space_location = the_string.find(" ", 0);
if (space_location == -1)
  cout << the_string << endl << endl;

//Petlja se izvrsava dok postoji blanko karakter
while (space_location != -1)
{
  if (space_location == 0) //Ako je pocetni prostor blanko, preskoci ga
    if (the_string.length() == 1) //Ako je string jedan razmak
      break;
    else
      the_string = the_string.substr(1, the_string.length());
  else
  {
    //inace, dohvati prvu rec i prikazi je
    word = the_string.substr(0, space_location);
    cout << word << endl;

    //Ukloni rec
    the_string.erase(0, word.length());
  }
  //Nadji sledecu blanko poziciju
  space_location = the_string.find(" ", 0);

  //Ako ne postoji blanko karakter, posalji na izlaz taj string
  if (space_location == -1)
    cout << the_string << endl << endl;
 }
 return 0;
}
```



Problem palindroma

```
// Ovaj program koristi bibliotecke funkcije koje rade sa string objektima
// i karakterima i odredjuje da li je uneti string palindrom

#include <iostream>
#include <cctype>
#include <string>

using namespace std;

void Copy_Alpha(string&, string);
void Convert_To_Lower(string&);
bool Palindrome(string);

int main()
{
 string line;
 string buffer;

 // Unos stringa u program

 cout <<"\nEnter a string:\n";
 getline(cin, line);
 // Kopiranje samo slovnih karaktera

 Copy_Alpha(buffer, line);

 // Konverzija stringa u mala slova

 Convert_To_Lower(buffer);

 // Provera da li je string palindrom
```

```
  if ( Palindrome(buffer) )
    cout <<"\nThe string you entered is a palindrome.";
  else
    cout <<"\nThe string you entered is not a palindrome.";

  cout << endl;
  return 0;
} // Kraj funkcije main()

void Copy_Alpha(string& target, string source)
{
  for (int i = 0; i < source.length(); ++i)
    if (isalpha(source.at(i)))
       target.append(1, source.at(i));

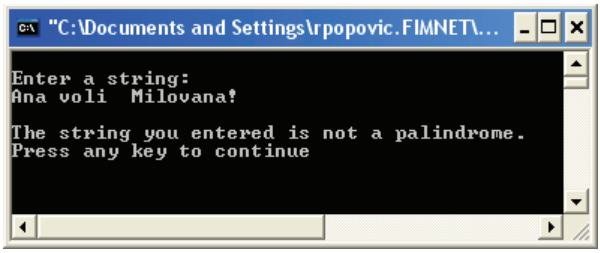
  return;
} // Kraj funkcije Copy_Alpha()

void Convert_To_Lower(string& source)
{
  for (int i = 0; i < source.length(); ++i)
    source.at(i) = tolower(source.at(i));

  return;
} // Kraj funkcije Convert_To_Lower()

bool Palindrome (string source)
{
  int end = source.length() - 1;
  int start = 0;

  while (start < end)
    if (source.at(start) != source. at(end))
      return false;
    else
      {
        ++start;
        end;
      }
  return true;
} // Kraj funkcije Palindrome()
```



7.6 Nizovi stringova

Možete da deklarišete niz objekata istog tipa, odnosno možete da deklarišete niz string objekata. Sledi deklaracija niza string objekata i inicijalizacija niza sa imenima dana u nedelji:

```
string dani_u_nedelji [] = {"Ponedeljak", "Utorak", "Sreda", "Cetvrtak",
                            "Petak", "Subota", "Nedelja"};
```

8. Programerski definisane klase i objekti

8.1 Deklarisanje objekata i klasa

Definišimo klasu Savings_Account (štedni račun):

```
class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;
 public:
   Savings_Account(string, double, double);
   double Calc_Interest();
};

Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no = id;
 balance = bal;
 rate = rt;
}

double Savings_Account::Calc_Interest()
{
 return balance * rate;
}
```

Definicija klase počinje rezervisanom reči class nakon koje je ime klase. Nakon imena klase u velikim zagradama se definiše klasa, nakon čega sledi tačka zarez.

Delovi deklarisani u klasi su članovi klase. Klasa ima tri podatka člana (ili instance-primerka promenljivih) id_no (identifikacioni broj), balance (stanje), i rate (kamatna stopa). To su instance promenljivih zato što svaki Savings_Account (štedni račun) objekat čiju instancu pravimo sadrži svoje sopstvene kopije ova tri podatka člana. Vrednosti instanci varijabli za određeni objekat određuju stanje tog objekta. Klasa Savings_Account takođe sadrži dva metoda, Savings_Account(), koji je konstruktor klase, i Calc_Interest(), koji izračunava kamatu. Podsetimo se da metodi klase određuju ponašanje svih objekata te klase. Svaki objekat klase ima isto ponašanje.

Postoje dve nove rezervisane reči u definiciji klase. Specifikatori pristupa private i public određuju vidljivost članova klase. Članovima klase koji su deklarisani posle oznake private mogu da pristupe samo metodi te klase. Ako su id_no, balance, i rate, privatne instance varijabli, njima pristupaju samo metodi Savings_Account() i Calc_Interest(). Ni jedna druga funkcija nema pristup id_no, balance, ili rate. Ako deklarišemo jedan Account objekat u funkciji main(), ne možemo direktno da referenciramo te objektove privatne instance varijabli zato što main() nije metod klase Savings_Account. Zbog toga je sledeća naredba u funkciji main() neispravna.

```
cout << acc1.rate;
```

Privatni članovi klasa su način da se u jeziku C++ primeni princip skrivanja informacija. Stvari koje čine jedan objekat, stanje objekta, održavaju se skriveni od spoljneg sveta. Jedini način da se promeni privatna instanca varijable jednog objekta je kroz korišćenje metoda klase. Skrivanje informacija kontroliše objekte, odnosno objekte možemo da koristimo samo kroz metode klasa.

Svim članovima klase koji su nakon oznake public, u našem primeru Calc_Interest() i Savings_Account(), može se pristupiti iz svih funkcija. Javni pristup znači da main(), ili bilo koja druga funkcija ili klasa, može da koristi javne metode za slanje odgovarajućih poruka objektima te klase. Ponekad se javni metodi klase zovu javni interfejs klase. Međutim, kako metodi rade, odnosno stvarni kod metoda nije poznat izvan klase.

Skoro uvek su instance varijabli klase privatni, a metodi klase su javni. Moguće je, mada neuobičajeno, imati javne instance varijabli i privatne metode klase.

Default specifikator pristupa je private. Sledeća deklaracija je ekvivalentna prethodnoj deklaraciji klase:

```
class Account
{
  string id_no;     //Ovo su privatni clanovi po defaultu
  double balance;
  double rate;

  public:
    Savings_Account(string, double, double);
    double Calc_Interest();
};
```

Uobičajeno je da se drže svi public članovi zajedno i svi private članovi zajedno. Ali, to nije neophodno.

Svi metodi klase mora da budu deklarisani unutar definicije klase, uključujući one metode koji su definisani izvan deklaracije klase.

Metodi klase

Bilo koji metod koji je definisan izvan deklaracije klase mora da koristi operator rezolucije opsega (scope resolution) ::, u hederu svoje definicije:

```
return-type ClassName::MethodName(parameter-list)
```

Čita se Calc_Interest() kao "Savings_Account-ov Calc_Interest()".

Veoma je važno zapamtiti da metodi klase imaju pristup svim primercima varijabli klase. Nema potrebe deklarisati primerke varijabli klase unutar metoda klase - samo koristite instance varijabli u metodu.

8.2 Konstruktor klase

Konstruktor je metod klase koji se automatski izvršava kada se kreira objekat te klase. Ime konstruktora mora da bude isto kao ime klase i on ne može imati return type (zato što on uvek kao rezultat daje kreiranje objekta klase) i mora biti public. Ako konstruktor nije eksplicitno deklarisan u definiciji klase, po *default*-u, C++ obezbeđuje *default* constructor. *Default* konstruktor samo kreira objekat klase, on ne inicijalizuje objektove podatke članove.

Operator razrešenja dosega mora biti korišćen u definiciji konstruktora zato što je definicija izvan deklaracije klase. Treba zapamtiti da ako deklarišemo konstruktor tako da ima jedan ili više argumenata (kao što Savings_Account() konstruktor ima tri argumenta) deafult konstruktor bez argumenata nije više na raspolaganju programu. U ovom slučaju, svaki objekat koji je deklarisan mora biti inicijalizovan. Ako postoji potreba za konstruktorom bez argumenata, možete sami da napravite kod.

Da bismo koristili konstruktor u funkciji main(), treba da napišemo sledeći deo koda:

```
void main()
{
```

```
  int id;
  double bal;
  double rt;

  cout << "\nEnter Account ID: ";
  cin >> id;

  cout << "\nEnter Balance: ";
  cin >> bal;

  cout << "\nEnter Interest Rate: ";
  cin >> rt;

  Savings_Account acc1(id, bal, rt);
  .
  .
  .
}
```

Naredba

```
Savings_Account acc1(id, bal, rt);
```

kreira primerak acc1 objekta i daje primercima varijabli određene vrednosti. Treba obratiti pažnju da konstruktor nije eksplicitno pozvan; kada se objekat deklariše konstruktor se automatski primenjuje na objekat koristeći obezbeđene argumente. Sledi, argumenti id, bal, i rt se prosleđuju konstruktoru Savings_Account. Konstruktor dodeljuje vrednost id parametra id_no, vrednost bal parametra balance, i vrednost rt dodeljuje rate. Slika prikazuje objekat koji je konstruktor kreirao ako korisnik ubaci sa tastature 5678, 200.00, i 0.07.

id_no	balance	rate
5678	200.00	0.07

Metodi

U prethodnom poglavlju videli smo kako da primenimo metode na objekte. Objekat cin nije objekat istream klase. Funkcije get() i getline() su public metodi klase istream. Kada pišemo kod cin.get() ili cin.getline(), primenjujemo ove funkcije članice na cin objekat. Koristimo *dot* (tačka) operator da bismo primenili metod na objekat. Ove tehnike se takođe

primenjuju na klase koje smo deklarisali u našim programima. Jednom kada deklarišemo Savings_Account objekat, možemo da koristimo *dot* operator sa metodima klasa za rad sa objektima.

U naredbi

```
cout << "The interest on the account is " << acc1.Calc_Interest() << endl;
```

primenjujemo metod Calc_Interest() na acc1 objekat šaljući poruku objektu acc1 da izračuna interest, odnosno, kamatu. Podsetimo se da definicija metoda Calc_Interest() sadrži samo jednu naredbu

```
return balance * rate;
```

Sada možemo da prikažemo kompletan program:

```
//Ovaj program ilustruje koriscenje obicne klase i funkcija članica

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

class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;
 public:
   Savings_Account(string, double, double);
   double Calc_Interest();
};

Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no   = id;
 balance = bal;
 rate    = rt;
}

double Savings_Account::Calc_Interest()
{
 return balance * rate;
}
```

```
int main()
{
 cout << setprecision(2)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);

 string id;     // Varijable za skladistenje ulaznih podataka
 double bal;
 double rt;

 //Dobijanje informacija o novom racunu od korisnika

 cout << "\nEnter Account ID: ";
 getline(cin, id);

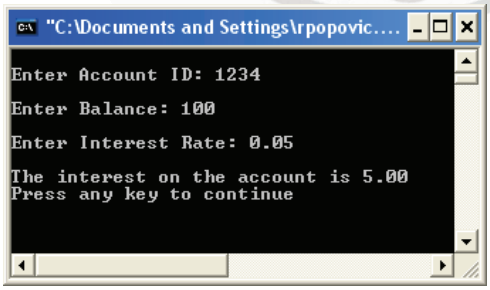
 cout << endl;
 cout << "Enter Balance: ";
 cin >> bal;

 cout << endl;
 cout << "Enter Interest Rate: ";
 cin >> rt;

 Savings_Account acc1(id, bal, rt);    //Inicijalizacija objekta

 cout << endl;
 cout << "The interest on the account is " << acc1.Calc_Interest() << endl;

 return 0;
}
```



Struktura programa koji koristi klase je:

```
//Pretprocesorske direktive
//Deklaracije klasa
```

```
//Definicije metoda
//Definicija funkcije main()
```

Za složenije programe možemo podeliti program na nekoliko fajlova, jedan fajl koji sadrži main() i drugi koji sadrže definicije klasa. Nakon kompajliranja izvornog fajla, linker kombinuje objektne fajlove u jedan izvršni program.

Objasnićemo detalje prethodnog programa. Unutar glavne funkcije main(), deklarišemo varijable id, bal, i rt da bismo koristili lokacije skladištenja za ulazne podatke koje unosi korisnik. Nakon unošenja ulaznih vrednosti, program pravi instancu acc1 objekta koristeći konstruktor sa tri argumenta. Podsetimo se da u C++ možete da deklarišete varijablu ili objekat bilo gde u programu.

Poslednja naredba cout u funkciji main() sadrži referencu na acc1.Calc_Interest(). Metod Calc_Interest() vraća vrednost. Sledi, naredba cout prikazuje vrednost kamate 5.00.

Veza između klase i strukture

Klasa je kao i struktura agregatni tip podataka u smislu da njeni podaci članovi mogu biti različitog tipa. U stvari, u C++ postoji mala razlika između klasa i struktura. C++ strukture mogu da imaju funkcije članice, kao i klase. Koja je onda razlika između klasa i struktura? *Default* pristup kod klase je private. Difolt pristup kod strukture je public.

U C jeziku za razliku od C++, strukture ne mogu da imaju funkcije članice i ne postoje specifikatori pristupa kao što su public i private. Takođe, rezervisana reč class ne postoji u C jeziku jer ne postoje klase u C jeziku.

Accessor i mutator metodi klase

Klasa Savings_Account iz prethodne sekcije nije mnogo korisna iz više razloga. Prvo, metod Calc_Interest() izračunava samo kamatu. Metod ne primenjuje kamatu na stanje računa, što želimo da uradimo u stvarnoj aplikaciji. Takođe, ne postoji način da dodamo ili oduzmemo novac sa računa nakon kreiranja računa. Konačno, ne postoji način da vidimo koliko je novca na računu. Ponovo ćemo definisati metod Calc_Interest() i dodati metode klase Get_Balance() koji prikazuje stanje na računu, Deposit() koji omogućuje stavljanje novca na račun, i Withdraw() koji omogućuje podizanje novca sa računa.

```
class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;

 public:
   Savings_Account(string, double, double);
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
};
```

Nova verzija Calc_Interest() ne izračunava samo kamatu za određeni račun, nego primenjuje kamatu na stanja računa. Metodi Deposit() i Withdraw() imaju jedan argument tipa double. Prosleđujemo metodu Deposit() količinu novca koji želimo da uložimo. Metodu Withdraw() prosleđujemo količinu novca koji želimo da podignemo.

Sledi nova definicija Calc_Interest():

```
double Savings_Account::Calc_Interest()
{
 double interest;

 interest = balance * rate;
 balance += interest;
 return interest;
}
```

Definicija Calc_Interest() sadrži deklaraciju lokalne promenljive interest. Lokalne promenljive koje su deklarisane unutar metoda klase ponašaju se po istim pravilima za doseg i trajanje kao i sve druge lokalne promenljive. Sledi, lokalna promenljiva je poznata samo unutar metoda u kome je deklarisana. Lokalna promenljiva počinje da postoji kada se deklariše u metodu, i prestaje da postoji kada se metod završi. Promenljivoj interest se daje vrednost prvom naredbom dodele u metodu. Drugom naredbom dodele povećava se vrednost primerka promenljive balance za vrednost interest. Kao kod originalne verzije metoda, metod vraća vrednost promenljive interest.

Metod Get_Balance() je veoma prost.

```
double Savings_Account::Get_Balance()
{
  return balance;
}
```

Pretpostavimo da je acc1 objekat klase Savings_Account koja ima član balance čija je vrednost 100.00. Sledeća naredba prikazuje vrednost člana balance objekta acc1

```
cout << "The balance in the account is " << acc1.Get_Balance() << endl;
```

Metod čiji je jedini cilj da pristupi vrednosti privatne instance promenljive date klase se ponekad zove accessor metod. Get_Balance() je primer accessor metoda čija svrha je pristup članu balance objekta Account. Accessor metodi su neophodni zato što su private instance varijabli skrivene od svih funkcija osim sopstvenih metoda date klase.

Definišimo sada metod Deposit().

```
void Savings_Account::Deposit(double amount)
{
  balance += amount;
}
```

Ponovo naglašavamo da su sve instance promenljivih date klase na raspolaganju metodu klase bez potrebe za ponovnom deklaracijom. Zato, Deposit() ima pristup primerku promenljive balance. Naredba u definiciji metoda povećava balance za vrednost amount depozita. Da bi uplatili depozit €55.42 na račun, recimo acc1, treba da napišemo sledeći kôd:

```
acc1.Deposit(55.42);
```

Metod Deposit() menja vrednost instanci promenljive i naziva se mutator metod. Definicija metoda Withdraw() mora da uzme u obzir da li vrednost promenljive amount prelazi stanje na računu. Sledi definicija:

```
bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
```

```
   balance -= amount;
   result = true;
  }
  else
    result = false;

  return result;
}
```

Sada koristimo novu definiciju i ubacujemo je u program:

```
//Ovaj program koristi prosirenu verziju klase Savings_Account

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
  private:
    string id_no;
    double balance;
    double rate;

  public:
    Savings_Account(string, double, double);
    double Calc_Interest();
    double Get_Balance();
    string Get_ID();
    void Deposit(double);
    bool Withdraw(double);
};
Savings_Account::Savings_Account(string id, double bal, double rt)
{
  id_no   = id;
  balance = bal;
  rate    = rt;
}

double Savings_Account::Get_Balance()
{
  return balance;
}
```

```
string Savings_Account::Get_ID()
{
  return id_no;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}
void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;
  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;
  return result;
}

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);
  string id;
  double bal;
  double rt;
  double amount;

  cout << endl;
  cout << "Enter Account ID: ";
  getline(cin, id);
```

```
cout << endl;
cout << "Enter Balance: ";
cin >> bal;

cout << endl;
cout << "Enter Interest Rate: ";
cin >> rt;

Savings_Account acc1(id, bal, rt);

cout << endl;
cout << "Account ID Number is " << acc1.Get_ID() << endl << endl;

cout << "The balance in the account is now " << acc1.Get_Balance()
    << endl << endl;

cout << "Enter an amount to deposit: ";
cin >> amount;

acc1.Deposit(amount);

cout << endl << endl;
cout << "A deposit of " << amount << " was made." << endl;
cout << "The balance in the account is now " << acc1.Get_Balance();

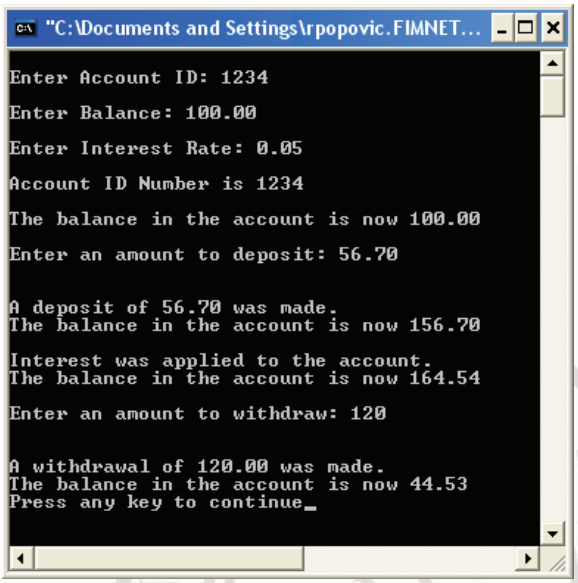
acc1.Calc_Interest();

cout << endl << endl;
cout << "Interest was applied to the account." << endl;
cout << "The balance in the account is now " << acc1.Get_Balance();

cout << endl << endl;
cout << "Enter an amount to withdraw: ";
cin >> amount;

if (acc1.Withdraw(amount))
{
 cout << endl << endl;
 cout << "A withdrawal of " << amount << " was made.";
}
else
{
 cout << endl << endl;
 cout << "WITHDRAWAL NOT MADE: Insufficient funds.";
}
cout << endl;
```

```
 cout << "The balance in the account is now " << acc1.Get_Balance() << endl;
 return 0;
}
```



Do sada smo koristili konstruktor sa tri argumenta u definiciji klase Savings_Account. Sada ćemo razmotriti korišćenje više konstruktora u klasi, kao i destruktora koji se izvršava kada se jedan objekat uništava.

8.3 Preklapanje konstruktora

C++ dozvoljava definisanje više konstruktora klase. Ovo se zove preklapanje funkcija (function overloading). Preklapanje dozvoljava da koristimo konstruktore u različitim okolnostima. U klasi Savings_Account, imamo konstruktor koji dozvoljava da kreiramo štedni račun koristeći identifikacioni broj računa account ID number, stanje na računu balance, i kamatnu stopu interest rate. Možemo da definišemo konstruktor koji dozvoljava da kreiramo štedni račun dajući samo broj računa account number, sa stanjem i kamatom sa *default*-nim vrednostima od 0.00 i 0.02 respektivno, ili možda želimo konstruktor koji nam dozvoljava da kreiramo račun dajući broj računa i stanje sa kamatom koja je data unapred 0.02. Mora da deklarišete sva tri konstruktora u definiciji klase kao što sledi:

```
class Savings_Account
{
 .
 .
 .
 public:
   Savings_Account(string, double, double);
   Savings_Account(string, double);
   Savings_Account(string);
 .
 .
 .
};
```

Konstruktor je sada preklopljen zato što imamo tri različita konstruktora zavisno od broja argumenata datih u deklaraciji objekta štedni račun Savings_Account. Slede tri definicije konstruktora:

```
Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no = id;
 balance = bal;
 rate = rt;
}

Savings_Account::Savings_Account(string id, double bal)
{
 id_no = id;
 balance = bal;
 rate = 0.02;
}

Savings_Account::Savings_Account(string id)
{
 id_no = id;
 balance = 0.0;
 rate = 0.02;
}
```

U funkciji main(), možemo sada da napišemo sledeće deklaracije:

```
Savings_Account acc3("3333", 100.00, 0.06);
Savings_Account acc2("2222", 200.00);

Savings_Account acc1("1111");
```

U jeziku C++ kompajler razlikuje tri konstruktora po broju i tipu argumenata obezbeđenih u deklaraciji objekta. Prva deklaracija koristi konstruktor sa tri argumenta i kreira račun 3333 sa stanjem 100.00 i kamatom 0.06. Druga deklaracija kreira račun 2222 sa otvorenim stanjem od 200.00 i unapred datom kamatom od 0.02. Treća deklaracija koristi konstruktor sa jednim argumentom i kreira račun 1111 sa unapred definisanim stanjem 0.00 i kamatom 0.02.

Sledeći program prikazuje preklapanje konstruktora. Dodali smo cout naredbu za svaki konstruktor da bismo verifikovali da se odgovarajući konstruktori izvršavaju. Treba obratiti pažnju da funkcija main() ima samo jednu cout naredbu koja objavljuje kraj programa. Drugi izlazi potiču od konstruktora.

```
//Ovaj program koristi prosirenu verziju klase Savings_Account
//On prikazuje preklapanje konstruktora

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;

 public:
   Savings_Account(string, double, double);
   Savings_Account(string, double);
   Savings_Account(string);
   double Calc_Interest();
   double Get_Balance();
   string Get_ID();
   void Deposit(double);
   bool Withdraw(double);
};
Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no   = id;
 balance = bal;
 rate    = rt;
```

```
  cout << "Three-arg constuctor for " << id_no << endl;
}

Savings_Account::Savings_Account(string id, double bal)
{
  id_no = id;
  balance = bal;
  rate = 0.02;

  cout << "Two-arg constuctor for " << id_no << endl;
}

Savings_Account::Savings_Account(string id)
{
  id_no = id;
  balance = 0.0;
  rate = 0.02;

  cout << "One-arg constuctor for " << id_no << endl;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

string Savings_Account::Get_ID()
{
  return id_no;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

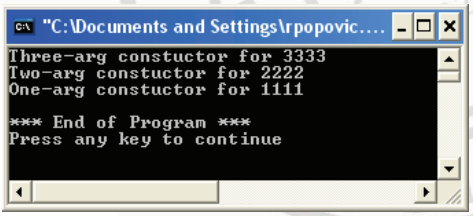
  return interest;
}
void Savings_Account::Deposit(double amount)
{
  balance += amount;
}
```

```
bool Savings_Account::Withdraw(double amount)
{
  bool result;
  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;
  return result;
}

int main()
{
  Savings_Account acc3("3333", 100.00, 0.06);
  Savings_Account acc2("2222", 200.00);
  Savings_Account acc1("1111");

  cout << endl;
  cout << "*** End of Program ***" << endl;

  return 0;
}
```



8.4 Destruktori

Kada jedan objekat izlazi iz dosega, C++ uništava objekat. Na primer, ako deklarišemo objekat unutar funkcije, to je lokalni objekat i zato je klasa skladištenja automatska. Kada se funkcija završi, objekat se uništava. Klasi je po *default*-u dat destruktor - *default* destruktor, koji je metod koji uništava objekat. Kao i konstruktor, *default* destruktor se ne poziva eksplicitno već se automatski izvršava kada objekat ispadne iz dosega. Kao i u slučaju konstruktora, možemo da napišemo naš sopstveni destruktor za klasu. Ime destruktora je isto kao i ime klase sa prefiksnim

karakterom "tilde" ~. Destruktor ne može imati argumente, ne može imati povratni tip i mora biti public. Ne može biti preklopljen.

Sledi primer kako destruktor radi:

```
//Ovaj program koristi prosirenu verziju klase Savings_Account.
//Program ilustruje korišćenje destruktora

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;

 public:
   Savings_Account(string, double, double);
   Savings_Account(string, double);
   Savings_Account(string);
   ~Savings_Account();
   double Calc_Interest();
   double Get_Balance();
   string Get_ID();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no   = id;
 balance = bal;
 rate    = rt;

 cout << "Three-arg constuctor for " << id_no << endl;
}

Savings_Account::Savings_Account(string id, double bal)
{
 id_no = id;
 balance = bal;
```

```
  rate = 0.02;

  cout << "Two-arg constuctor for " << id_no << endl;
}

Savings_Account::Savings_Account(string id)
{
  id_no = id;
  balance = 0.0;
  rate = 0.02;

  cout << "One-arg constuctor for " << id_no << endl;
}

Savings_Account::~Savings_Account()
{
  cout << "Destructor executed for " << id_no << endl;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

string Savings_Account::Get_ID()
{
  return id_no;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
```

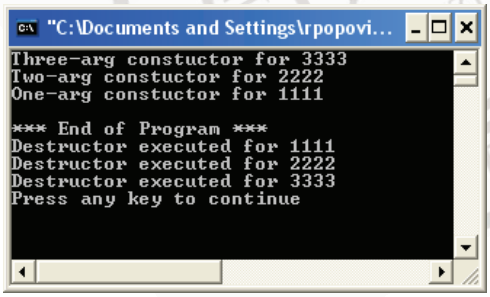
```
  bool result;
  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

int main()
{
  Savings_Account acc3("3333", 100.00, 0.06);
  Savings_Account acc2("2222", 200.00);
  Savings_Account acc1("1111");

  cout << endl;
  cout << "*** End of Program ***" << endl;

  return 0;
}
```



Redosled prikazanih poruka je važan. Kada se acc3 deklariše, konstruktor se izvršava za taj objekat. To isto važi i za acc2 i za acc1. Sva tri objekta prestaju da važe kada se funkcija main() završi. Zato, main() prikazuje poruku da se program završava. Zatim se destruktor izvršava za svaki objekat obrnutim redom od kreiranja objekta.

Default argumenti

U prethodnoj sekciji preklopili smo konstruktor štedni račun Savings_Account da bismo deklarisali Savings_Account objekat na bilo koji od tri

načina. Ponekad možemo postići efekat preklapanja konstruktora koristeći jedan konstruktor sa *default* argumentima. Potrebno je da specificirate *default* argumente u deklaraciji konstruktora, a ne u definiciji. Sledi, u deklaraciji klase treba da napišete sledeći kôd:

```
public:
   Savings_Account(string id, double bal = 0.00, double rt = 0.02);
                  //konstruktor sa default veličinama
                  //koji služi kao konstruktor sa jednim argumentom
                  //i kao konstruktor sa dva argumenta
```

Ova deklaracija znači da ako izostavimo poslednja dva argumenta kada deklarišemo Savings_Account objekat, *default* vrednost 0.00 biće iskorišćena kao drugi argument i *default*-na vrednost 0.02 biće iskorišćena kao treći argument. Sledeća deklaracija kreira Savings_Account objekat sa identifikacionim brojem ID čija je vrednost 1111, stanje 0.00 i kamatna stopa 0.02:

```
Savings_Account acc1("1111");
```

Samo završni argumenti se mogu izostaviti. Konstruktor bez argumenata nije iskoristljiv.

Sledeći program ilustruje korišćenje *default* argumenata u konstruktoru klase štedni račun Savings_Account.

```
//Ovaj program koristi proširenu verziju klase Savings_Account.
//Program ilustruje default argumente u deklaraciji konstruktora

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;

 public:
```

```
   Savings_Account(string id, double bal = 0.00, double rt = 0.02);
   ~Savings_Account();
   double Calc_Interest();
   double Get_Balance();
   string Get_ID();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no = id;
 balance = bal;
 rate = rt;

 cout << "Default-argument constuctor for " << id_no << endl;
 cout << "Value of balance is " << balance << endl;
 cout << "Value of rate is " << rate << endl << endl;
}

Savings_Account::~Savings_Account()
{
 cout << "Destructor executed for " << id_no << endl;
}

double Savings_Account::Get_Balance()
{
 return balance;
}

string Savings_Account::Get_ID()
{
 return id_no;
}

double Savings_Account::Calc_Interest()
{
 double interest;

 interest = balance * rate;
 balance += interest;
 return interest;
}

void Savings_Account::Deposit(double amount)
{
```

```
  balance += amount;
}

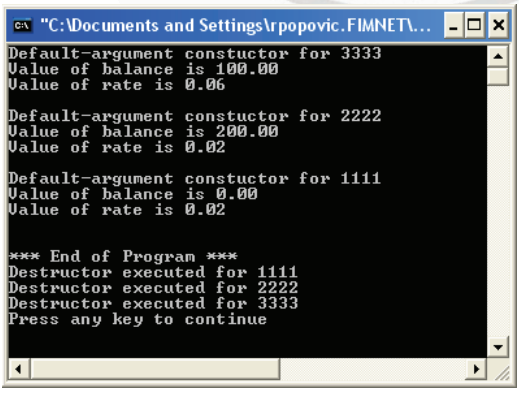
bool Savings_Account::Withdraw(double amount)
{
  bool result;
  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;
  return result;
}

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  Savings_Account acc3("3333", 100.00, 0.06);
  Savings_Account acc2("2222", 200.00);
  Savings_Account acc1("1111");

  cout << endl;
  cout << "*** End of Program ***" << endl;

  return 0;
}
```



Default argumenti u deklaraciji konstruktora nam dozvoljavaju da deklarišemo objekat Savings_Account na isti način kao ranije, gde smo imali tri preklopljena konstruktora. Dodali smo cout naredbe u konstruktoru da bismo verifikovali vrednosti primeraka promenljivih.

Dodela jednog objekta drugom

Moguće je dodeliti jedan objekat drugom objektu ako pripadaju istoj klasi. U ovom slučaju dodela je data po principu član po član. Na primer:

```
Savings_Account acc1("1111", 100.00, 0.03);
Savings_Account acc2("2222", 200.00);

acc2 = acc1;
```

Objekat acc1 ima svoje članove id, balance i rate inicijalizovane na "1111", 100.00 i 0.03, respektivno. Objekat acc2 ima svoj id i balance inicijalizovane na "2222" i 200.00. Vrednost za rate za objekat acc2 je difoltna 0.02. Kada se dodela izvršava, id član objekta acc1 se kopira u id član objekta acc2; balance član acc1 se kopira u balance član acc2; rate član acc1 se kopira u rate član acc2.

Ovo ćemo ilustrovati u sledećem programu. Treba obratiti pažnju da smo izbacili destruktor i cout narebe iz konstruktora. Takođe smo dodali Get_Rate() metod, tako da možemo da prikažemo vrednosti svih primeraka varijabli.

```
//Ovaj program koristi prosirenu verziju klase Savings_Account
//Program ilustruje dodelu objekata
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   string id_no;
   double balance;
   double rate;
 public:
   Savings_Account(string id, double bal = 0.00, double rt = 0.02);
```

```
   double Calc_Interest();
   double Get_Balance();
   double Get_Rate();
   string Get_ID();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(string id, double bal, double rt)
{
 id_no   = id;
 balance = bal;
 rate    = rt;
}

double Savings_Account::Get_Balance()
{
 return balance;
}

string Savings_Account::Get_ID()
{
 return id_no;
}

double Savings_Account::Get_Rate()
{
 return rate;
}

double Savings_Account::Calc_Interest()
{
 double interest;
 interest = balance * rate;
 balance += interest;

 return interest;
}

void Savings_Account::Deposit(double amount)
{
 balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
```

```
  bool result;
  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;
  return result;
}

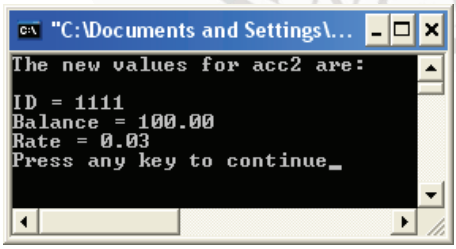
int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  Savings_Account acc1("1111", 100.00, 0.03);
  Savings_Account acc2("2222", 200.00);

  acc2 = acc1;

  cout << "The new values for acc2 are: " << endl << endl;
  cout << "ID = " << acc2.Get_ID() << endl;
  cout << "Balance = " << acc2.Get_Balance() << endl;
  cout << "Rate = " << acc2.Get_Rate() << endl;

  return 0;
}
```



Savings_Account konstruktor se izvršava dva puta za acc1 i jednom za acc2. Program zatim dodeljuje vrednost acc1 desnoj strani, tj. objektu acc2. Dodela je data po principu član po član. Zato se podaci članovi objekta acc1 kopiraju u podatke članove objekta acc2. Program verifikuje dodelu kada prikaže vrednosti podataka članova objekta acc2.

8.5 Opšte preklapanje funkcija i šabloni funkcija

Bilo koja funkcija može biti preklopljena u C++. Preklopljene verzije funkcija mora da imaju isti povratni tip i mora da se razlikuju u broju i/ili tipu argumenata. Sledeća funkcija zamenjuje dva celobrojna argumenta.

```
void Swap(int& i, int& j)
{
  int temp;

  temp = i;
  i = j;
  j = temp;
}
```

U C++ jeziku možemo da preklopimo integer verzije funkcije Swap() sa sledećom double verzijom Swap().

```
void Swap(double& x, double& y)
{
  double temp;
  temp = x;
  x = y;
  y = temp;
}
```

Šablon funkcije dozvoljava programeru da napiše kod za funkciju čija je definicija nezavisna od tipa podatka njenih argumenata i/ili njenog return tipa. Na taj način, šablon funkcije se ustvari koristi da kompajler automatski preklopi funkcije kao što je potrebno programu. Šablon funkcije Swap() je:

```
template <class Type>
void Swap(Type& x, Type& y)
{
  Type temp;
  temp = x;
  x = y;
  y = temp;
}
```

Definicija počinje rezervisanom reči template za kojom sledi rezervisana reč class i jedan identifikator Type. Reč template kaže kompajleru da napravi definiciju šablona. Identifikator Type se koristi kao ime proizvoljne klase ili tipa podatka. Type se ponaša kao varijabla čija vrednost može biti bilo koja klasa ili tip podatka.

Nakon deklaracije šablona sledi definicija funkcije. Jedino pravilo koga treba da se držite kada pišete funkciju je da koristite identifikator Type kad god želite da referencirate na tip podatka parametra ili kada deklarišete varijablu. U primeru koji sledi u hederu funkcije imamo Swap(Type& x, Type& y) deklarisanje parametara bilo kog tipa. Takođe, u telu funkcije deklarišemo promenljivu temp kao što sledi

```
Type temp;
```

Sledeći program sadrži nekoliko primera šablona funkcija:

```
//Ovaj program prikazuje sablone funkcija

#include <iostream>
using namespace std;

template <class Type>
Type Max(Type x, Type y)
{
 return (x > y)? x: y;
}
template <class Type>
void Swap(Type& x, Type& y)
{
 Type temp;
 temp = x;
 x = y;
 y = temp;
}
template <class TypeA, class TypeB>
TypeA Func(TypeA x, TypeB y)
{
 TypeA r;
 r = 3 * x + 2 * y;
 return r;
}
template <class Type>
Type Sum(Type a[], int n)
{
```

```
  Type t = 0;

  for (int i = 0; i < n; ++i)
    t += a[i];
  return t;
}

int main()
{
  int i = 3,
      j = 5;
  double d = 8.62,
         e = 4.14;

  float f_arr[6] = {1.2, 2.3, 3.4, 4.5, 5.6, 6.7};
  int   i_arr[4] = {4, 5, 6, 7};

  cout << "i = " << i << " and j = " << j << endl;
  cout << "d = " << d << " and e = " << e << endl << endl;

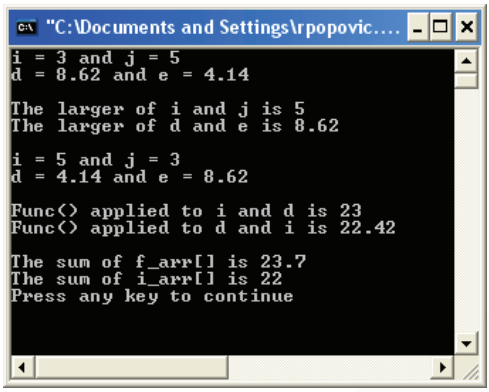
  cout << "The larger of i and j is " << Max(i, j) << endl;
  cout << "The larger of d and e is " << Max(d, e) << endl << endl;

  Swap(i, j);
  Swap(d, e);

  cout << "i = " << i << " and j = " << j << endl;
  cout << "d = " << d << " and e = " << e << endl << endl;

  cout << "Func() applied to i and d is " << Func(i, d) << endl;
  cout << "Func() applied to d and i is " << Func(d, i) << endl << endl;
  cout << "The sum of f_arr[] is " << Sum(f_arr, 6) << endl;
  cout << "The sum of i_arr[] is " << Sum(i_arr, 4) << endl;

  return 0;
}
```



Šabloni funkcija su definisani pre funkcije main(), što zahteva većina kompajlera. Ako pokušate da iskoristite šablon funkcije za primer u kome kompajler ne može da podesi tipove, program se neće kompajlirati.

9. Rad sa objektima

9.1 Korišćenje nizova, pokazivača i dinamička alokacija memorije

Da bismo napravili klasu Savings_Account da bude što realnija, menjamo tip podatka id_no, tj. član koji skladišti broj računa i uvodimo pokazivač kao član za skladištenje imena vlasnika računa.

Niz i pokazivač kao članovi klase

U većini aplikacija broj računa (identifikacioni broj) ili ID ima određen broj karaktera. Želimo da naš ID računa bude 4 karaktera, pa ga smeštamo u niz karaktera veličine 5 (dodata je pozicija za null karakter, '\0'). Pretpostavimo da smo deklarisali sledeće u private sekciji definicije Savings_Account:

```
char id_no[5];
```

Dodaćemo i definiciju primerka varijable name klase Savings_Account, koja predstavlja ime osobe koja je vlasnik računa. Ime čuvamo u stringu. Međutim, da bi ilustrovali određene karakteristike jezika C++, napravićemo da podatak član name bude karakter pointer (C-string) i dinamički alocira tačno onoliko memorije koliko zahteva ime vlasnika kada kreiramo štedni račun.

```
char* name;
```

Sledi nova deklaracija štednog računa Savings_Account, koja uključuje deklaraciju novog konstruktora sa 4 argumenta, sa 2 *default* argumenta i deklaracijom destruktora. Takođe, dodajemo metode za prikaz varijabli id_no i name.

```
class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;
```

```
 public:
  Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
  ~Savings_Account();
  double Calc_Interest();
  double Get_Balance();
  void Deposit(double);
  int Withdraw(double);
  void Display_ID();
  void Display_Name();
};
```

Novi konstruktor pretpostavlja da su prvi i drugi argument karakter stringovi. Podsetimo se da je ime niza bez srednjih zagrada pointer na prvi elemenat niza. Tretiraćemo prvi od stringova kao niz karaktera fiksne veličine, a drugi kao karakter pointer. Ako želimo da kreiramo tekući račun sa ID brojem 1111 koji pripada Jane Doe sa otvorenim stanjem od 200.00 i *default* kamatnom stopom od 0.04, deklarisaćemo sledeće:

```
Account acc("1111", "Jane Doe", 200.00);
```

Konstruktor mora da kopira prvi argument "1111" u član niz id_no[], i napravi član pointer name koji pokazuje na karakter string "Jane Doe". Da bismo ovo ostvarili koristimo funkcije za rad sa C-string-om i dinamičku alokaciju memorije. Kod za konstruktor je:

```
Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);   //kopira prvi argument u id_no[]

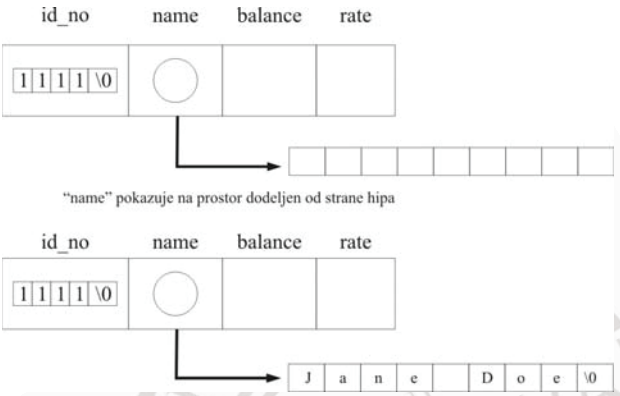
 name = new char[strlen(n_p) + 1];  //kreira prostor u memoriji za pokazivac name

 strcpy(name, n_p);  //kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;
}
```

Prva naredba u konstruktoru koristi bibliotečku funkciju strcpy() da bi se kopirao prvi argument u član niza id_no[] sa pet karaktera. Da bi član name pokazivao na string koji je drugi argument potrebne su dve naredbe. Prva naredba koristi new operator za alokaciju dovoljnog prostora za niz karaktera koji će da drži drugi argument i dodeli name-u pokazivač koji new vraća.

String funkcija strlen(), vraća broj karaktera u drugom argumentu, plus 1 za null karakter. Nakon izvršenja dodele, name pokazuje na niz karaktera koji je tačno toliko velik da drži string koji je drugi argument konstruktora. Druga naredba kopira drugi argument, na koji pokazuje n_p, u prostor na koji pokazuje name.



Sledi definicija destruktor funkcije:

```
Savings_Account::~Savings_Account()
{
 cout << "\nAccount " << id_no << " terminated.";
 delete [] name;
}
```

Kada program kreira Savings_Account objekat, konstruktor alocira prostor u hip memoriji za ime vlasnika štednog računa. Kada jedan objekat izlazi iz dosega, tj. kada se objekat uništava, mora da dealociramo tu hip memoriju izvršenjem naredbe delete u destruktoru. Proces dealociranja hip prostora kada više nije potreban se naziva "garbage collection". Greška pri dealokaciji hip prostora prouzrokuje da prostor ostane alociran ali beskoristan ("đubre") zato što pokazivač na taj prostor, name, više ne postoji. Ako se ovo desi za više Savings_Account objekata, biće problem sa neiskorišćenim prostorom u glavnoj memoriji.

Ubacićemo sada promenjenu klasu u program.

```
//Ovaj program koristi prosirenu verziju klase Savings_Account.
//Program koristi konstruktor sa default argumentima.
//ID broj je uskladisten kao niz a ime klase Savings_Account je pokazivac.
//Konstruktor koristi dinamicku alokaciju memorije.
```

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;
 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   ~Savings_Account();
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
   void Display_ID();
   void Display_Name();
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);   // kopira prvi argument u id_no[]

 name = new char[strlen(n_p) + 1];  // kreira prostor na unos imena

 strcpy(name, n_p);  // kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;
}
Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
 delete [] name;
}

double Savings_Account::Get_Balance()
{
 return balance;
}
double Savings_Account::Calc_Interest()
{
```

```
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

void Savings_Account::Display_ID()
{
  cout << id_no;
}

void Savings_Account::Display_Name()
{
  cout << name;
}

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  char   id[5];
  char   buffer[81];  //Privremeno skladisti ime
```

```
double bal;
double rt;
double amount;

cout << endl;
cout << "Enter Account ID: ";
cin.getline(id, 5);

cout << endl;
cout << "Enter Account Holder's Name: ";
cin.getline(buffer, 81);

cout << endl;
cout << "Enter Balance: ";
cin >> bal;

cout << endl;
cout << "Enter Interest Rate: ";
cin >> rt;
Savings_Account acc1(id, buffer, bal, rt);

cout << endl;
cout << "The Account number is ";
acc1.Display_ID();
cout << endl;
cout << "The owner of the Account is ";
acc1.Display_Name();

cout << endl << endl;
cout << "The balance in the Account is now " >> acc1.Get_Balance();

cout << endl << endl;
cout << "Enter an amount to deposit: ";
cin >> amount;

acc1.Deposit(amount);
cout << endl << endl;
cout << "A deposit of " << amount << " was made." << endl;
cout << "The balance in the Account is now "
    << acc1.Get_Balance() << endl;

acc1.Calc_Interest();

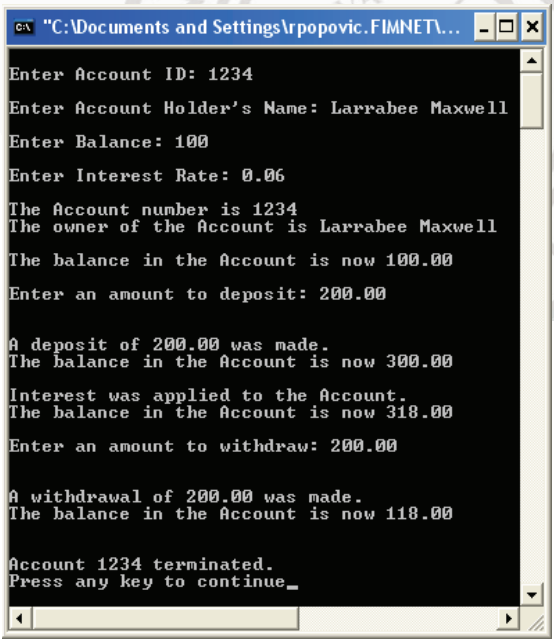
cout << endl;
cout << "Interest was applied to the Account." << endl;
cout << "The balance in the Account is now "
```

```
    << acc1.Get_Balance() << endl << endl;

  cout << "Enter an amount to withdraw: ";
  cin >> amount;

  if (acc1.Withdraw(amount))
  {
    cout << endl <<endl;
    cout << "A withdrawal of " << amount << " was made.";
  }
  else
  {
    cout << endl << endl;
    cout << "WITHDRAWAL NOT MADE: Insufficient funds.";
  }
  cout << endl;
  cout << "The balance in the Account is now " << acc1.Get_Balance() << endl;

  return 0;
}
```



Treba obratiti pažnju da destruktor prikazuje poruku kada se program završi.

9.2 Konstruktor kopije

Jedan objekat može da se dodeli drugom objektu kada se kreira objekat, kao u sledećem slučaju:

```
Savings_Account acc1("1111", "Larrabee Maxwell", 100.00, 0.02);
Savings_Account acc2  = acc1;
```

Kao što se može videti, kada se jedan objekat kreira kroz inicijalizaciju, (kao što je objekat acc2), konstruktor klase se ne izvršava. Metod koji se izvršava za vreme inicijalizacije je konstruktor kopije. U ovoj sekciji, objasnićemo konstruktor kopije i okolnosti pod kojima treba da napišete kod za konstruktor kopije za klasu.

Konstruktor kopije: prost primer

U jeziku C++, postoji razlika između dodele i inicijalizacije. Pri dodeli, *receiving*-objekat mora da postoji pre dodele. Pri inicijalizaciji, objekat je inicijalizovan za vreme procesa kreiranja. Na primer:

```
int i = 7,
    j;
    j = i;
```

Ceo broj i je deklarisan i inicijalizovan na 7. Kada se kreira, kreira se sa vrednošću 7. Ne postoji vreme za koje i postoji bez ispravne vrednosti. Sa druge strane, kada se j deklariše, on ne prima vrednost. Zbog toga, on ima *garbage* vrednost dok mu se validna vrednost ne dodeli. Operacija dodele kopira vrednost i u postojeći objekat j, dajući objektu j validnu vrednost.

Posmatrajmo primer:

```
//Ovaj program ilustruje dodelu jednog objekta drugom objektu.
//Treba obratiti paznju da inicijalizacija jednog objekta ne koristi
//konstruktor sa jednim argumentom

#include <iostream>

using namespace std;
```

```
class Test_Class
{
 private:
   int n;
 public:
   Test_Class(int);
   ~Test_Class();
   int Get_Value();
};

Test_Class::Test_Class(int i)
{
 n = i;

 cout << endl;
 cout << "Constructor Executed: Data member initialized to " << n << endl;
}

Test_Class::~Test_Class()
{
 cout << endl;
 cout << "Destructor Executed for object with data member " << n << endl;
}

int Test_Class::Get_Value()
{
 return n;
}

int main()
{
 Test_Class tc_object1(4);
 Test_Class tc_object2(0);

 cout << endl;
 cout << "After object creation:" << endl <<endl;
 cout << "Object 1: " << tc_object1.Get_Value() << endl;
 cout << "Object 2: " << tc_object2.Get_Value() << endl << endl;

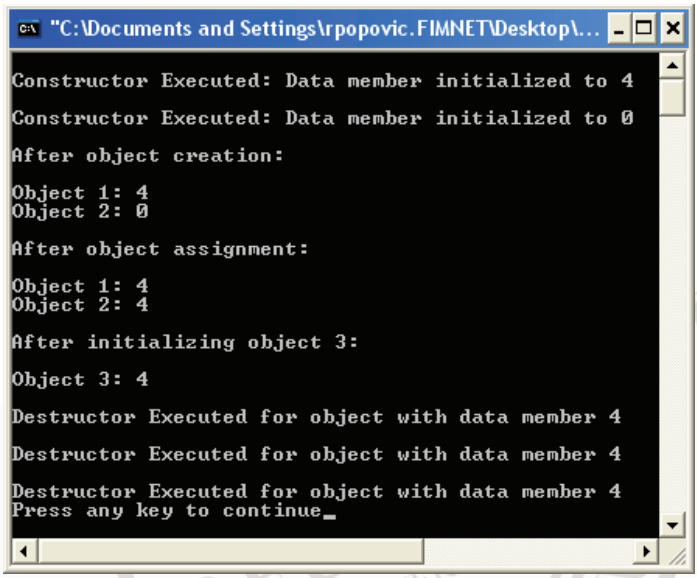
 tc_object2 = tc_object1;

 cout << "After object assignment:" << endl << endl;
 cout << "Object 1: " << tc_object1.Get_Value() << endl;
 cout << "Object 2: " << tc_object2.Get_Value() << endl << endl;

 Test_Class tc_object3 = tc_object1;
```

```
 cout << "After initializing object 3:" << endl << endl;
 cout << "Object 3: " << tc_object3.Get_Value() << endl;

 return 0;
}
```



Test_Class konstruktor izvršava se dva puta – jednom za tc_object1 i jednom za tc_object2. Nakon kreiranja objekta, program prikazuje vrednosti podataka članova objekata. Program zatim dodeljuje vrednost tc_object1 objektu tc_object2. Dodela je data na osnovu "member-by-member". Zbog toga se, podatak n član objekta tc_object1 kopira u n podatak član objekta tc_object2. Program verifikuje dodelu kada prikaže vrednosti podataka članova objekata.

Napomena: konstruktor sa jednim argumentom se ne izvršava kada program kreira i inicijalizuje tc_object3 izvršavajući sledeću naredbu:

```
Test_Class tc_object3 = tc_object1;
```

Konstruktorove poruke se ne prikazuju u ovoj tački programa! Vrednost tc_object3 se prikazuje u funkciji main(), ali ne od strane konstruktora. Konačno, na kraju programa, destruktor se izvršava tri puta, jednom za svaki objekat koji kreira program.

Zato što se operacija inicijalizacije razlikuje od dodele, specijalan konstruktor se koristi u C++ za inicijalizaciju konstruktora kopije (*copy constructor*). Kao i kod običnog konstruktora, ako eksplicitno ne napišete kôd za konstruktor kopije, klasi se daje *default* konstruktor kopije. *Default* konstruktor kopije kopira podatke članove iz tc_object1 i inicijalizuje ih, na bazi član po član, u odgovarajuće podatke članove objekta tc_object3.

Kod konstruktora kopije koji ste vi kreirali za Test_Class je:

```
class Test_Class
{
  .
  .
  .
 public:
   Test_Class(Test_Class&);    // Deklaracija konstruktora kopije
  .
  .
  .
};
```

Konstruktor kopije mora, kao i drugi konstruktori, da ima isto ime kao i klasa. On se razlikuje od drugih konstruktora po formi svog argumenta, koji je referenca na jedan objekat klase za koju je on konstruktor kopije.

Sintaksa deklaracije konstruktora kopije je oblika:

```
Class_Name (Class_Name& r_object);
```

Kada se piše kod za konstruktor kopije, razmišljajte o konstruktorovom argumentu kao o objektu sa desne strane operatora dodele = pri inicijalizaciji. U kodu konstruktora kopije, da bi uputili (referencirali) na članove ovog objekta, koristite parametar name r_object, sa *dot* operatorom, za kojim sledi primerak promenljive name.

Konstruktor kopije se primenjuje na objekat koji se kreira. Zbog toga, primenjen je na objekat sa leve strane znaka = pri inicijalizaciji.

```
Test_Class::Test_Class(Test_Class& tc_r)
{
  n = tc_r.n;  //Kopira podatak n clan objekta sa desne strane operatora dodele
               // u inicijalizaciji u podatak n clana objekta koji se kreira

  cout << endl << endl;
```

```
  cout << "Copy Constructor executed: "
       << "Data member initialized to " << n << endl;
}
```

Konstruktor kopije se poziva kada se jedan objekat inicijalizuje u deklaraciji. Zbog toga, druga deklaracija koja sledi prouzrokuje da se konstruktor kopije izvrši

```
Test_Class tc_object1(3);              //regularan konstruktor se izvršava
Test_Class tc_object2 = tc_object1;  //konstruktor kopije se izvršava
```

O parametru tc_r u definiciji konstruktora kopije razmišljate kao o predstavljanju desne strane operatora dodele = pri inicijalizaciji (u prethodnoj drugoj deklaraciji, tc_object1). Na taj način konstruktor kopije dodeljuje n data članu novog objekta koji se kreira (objekat tc_object2 u ovoj inicijalizaciji) odgovarajući data član objekta na desnoj strani operatora dodele =, nazvanom, tc_r.n (koji predstavlja n data član objekta tc_object1).

Objasnićemo u narednoj sekciji zašto argument konstruktora kopije mora da bude referenca. Sledeći program ilustruje korišćenje konstruktora kopije.

```
//Ovaj program ilustruje koriscenje konstruktora kopije.

#include <iostream>

using namespace std;

class Test_Class
{
  private:
    int n;

  public:
    Test_Class(int);  //Konstruktor sa jednim argumentom
    Test_Class(Test_Class&);  //Konstruktor kopije
    ~Test_Class();
    int Get_Value();
};

Test_Class::Test_Class(int i)
{
  n = i;
```

```
  cout << endl;
  cout << "Constructor Executed: Data member initialized to " << n << endl;
}

Test_Class::Test_Class(Test_Class& tc_r)
{
  n = tc_r.n;

  cout << endl << endl;
  cout << "Copy constructor executed: "
            << "Data member initialized to " << n << endl << endl;
}

Test_Class::~Test_Class()
{
  cout << endl << endl;
  cout << "Destructor Executed for object with data member " << n << endl;
}

int Test_Class::Get_Value()
{
  return n;
}

int main()
{
  Test_Class tc_object1(4);
  Test_Class tc_object2(0);

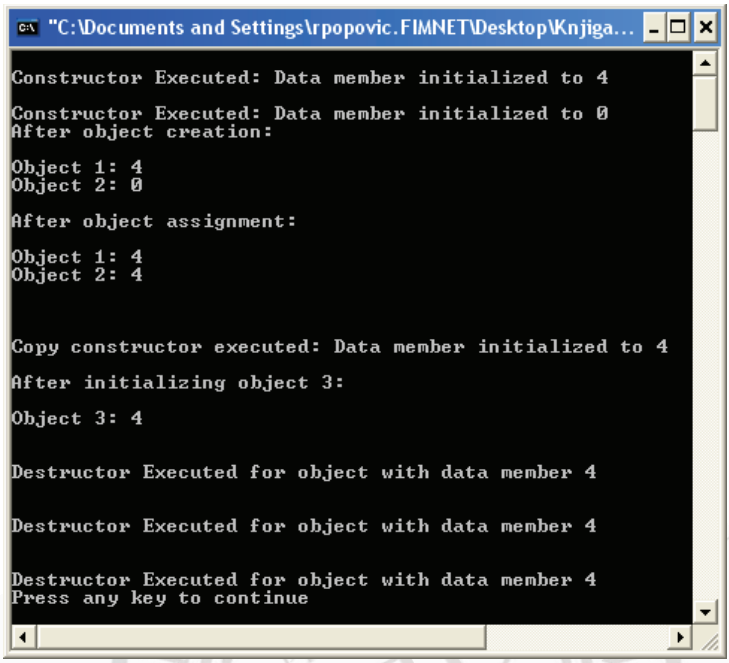
  cout << "After object creation:" << endl << endl;
  cout << "Object 1: " << tc_object1.Get_Value() << endl;
  cout << "Object 2: " << tc_object2.Get_Value() << endl << endl;

  tc_object2 = tc_object1;
  cout << "After object assignment:" << endl << endl;
  cout << "Object 1: " << tc_object1.Get_Value() << endl;
  cout << "Object 2: " << tc_object2.Get_Value() << endl << endl;

  Test_Class tc_object3 = tc_object1;

  cout << "After initializing object 3:" << endl << endl;
  cout << "Object 3: " << tc_object3.Get_Value() << endl;

  return 0;
}
```



Obratite pažnju da se konstruktor kopije izvršava kada se program kreira i inicijalizuje objekat tc_object3.

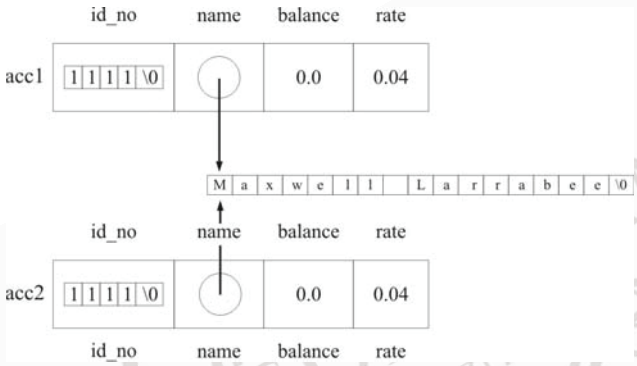
Konstruktor kopije za klasu Savings_Account

Mada nije neophodno da se napiše kôd za konstruktor kopije za Test_Class (*default* konstruktor kopije takođe radi dobro), važno je da napišete vaš sopstveni kôd za konstruktor kopije kad god klasa sadrži pokazivač kao instancu promenljive. Da biste videli zašto, posmatrajmo Savings_Account, koji ima character pointer name, kao primerak promenljive. Zato što klasa nema eksplicitno kodiran konstruktor kopije, *default* konstruktor kopije klase biće korišćen pri inicijalizaciji bilo kog objekta. Pretpostavimo da smo u programu koji koristi Savings_Account napravili sledeće deklaracije (možemo da napravimo takvu dodelu da bi eksperimentisali sa kopijom objekta acc1, a da ne menjamo original):

```
Account acc1("1111", "Maxwell Larrabee");
Account acc2 = acc1;
```

Zato što je *default* konstruktor kopije korišćen pri inicijalizaciji, acc2 je "član-po-član" kopija od acc1. Zbog toga, pointer acc1.name koji pokazuje na string "Maxwell Larrabee" biva kopiran u pointer acc2.name. Obratite pažnju da se pokazivač, koji je adresa, kopira, a ne kopira se string na koji pokazivač pokazuje. Na taj način, imamo name pokazivač dva različita objekta koji pokazuje na istu zonu hip memorije.

Sa slike sledi da promena cilja jednog od ovih pokazivača kao rezultat daje promenu oba. Na taj način ako promenimo string na koji pokazuje name član objekta acc2, takođe menjamo string na koji pokazuje name član objekta acc1. Ovo je upravo ono što nismo želeli da se desi!



Kako bismo rešili ovaj problem, napisali smo konstruktor kopije za Savings_Account koji kopira cilj na koji pokazuje pointer name.

```
Savings_Account::Savings_Account(Savings_Account& acc_r)
{
  strcpy(id_no, acc_r.id_no);            // kopira prvi argument u id_no[]

  name = new char[strlen(acc_r.name) + 1];       // kreira prostor na unos imena

  strcpy(name, acc_r.name);              // kopira drugi argument u novi prostor

  balance = acc_r.balance;

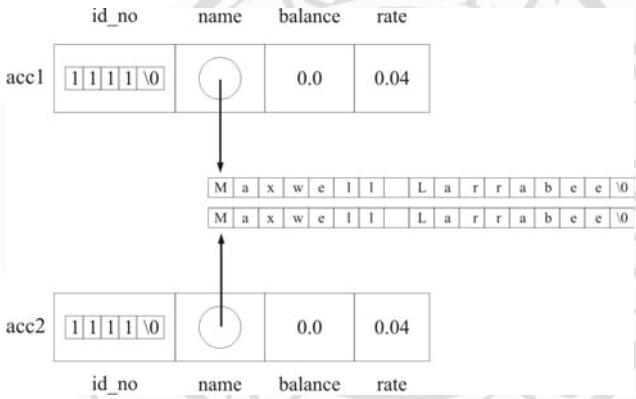
  rate    = acc_r.rate;
}
```

Zapamtite da je argument u konstruktoru kopije objekat na desnoj strani operatora = u inicijalizaciji i da je konstruktor kopije primenjen na objekat koji se kreira.

Obratite pažnju na sličnost između koda konstruktora sa četiri argumenta i konstruktora kopije. Prvo, id_no član objekta koji se kopira, kopira se u id_no člana objekta koji se kreira. Zatim se prostor alocira iz hip memorije za string dovoljno velik da drži kopiju od name člana objekta koji se kopira. Nakon kopiranja name u novi hip prostor, stanje i kamatna stopa članovi objekta koji se kopira se kopiraju u odgovarajuće članove objekta koji se kreira. Na taj način, ako napravimo prethodni konstruktor kopije člana od Savings_Account, sledeće deklaracije kao rezultat daju situaciju označenu na slici:

```
Savings_Account acc1("1111", "Maxwell Larrabee");

Savings_Account acc2 = acc1;
```



Sledi program koji ilustruje korišćenje konstruktora kopije u klasi Savings_Account.

```
//Ovaj program ilustruje koriscenje konstruktora kopije
//u klasi koja ima pokazivac kao podatak clan.

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
  private:
    char   id_no[5];
```

```
    char*  name;
    double balance;
    double rate;

  public:
    Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
    Savings_Account(Savings_Account&);
    ~Savings_Account();
    double Calc_Interest();
    double Get_Balance();
    void Deposit(double);
    bool Withdraw(double);
    void Display_ID();
    void Display_Name();
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
  strcpy(id_no, id);   // kopira prvi argument u id_no[]

  name = new char[strlen(n_p) + 1];  //kreira prostor za pokazivac name

  strcpy(name, n_p);   //kopira drugi argument u novi prostor

  balance = bal;
  rate    = rt;

  cout << endl << endl;
  cout << "Constructor executed." << endl;
}

Savings_Account::Savings_Account(Savings_Account& acc_r)
{
  strcpy(id_no, acc_r.id_no);  // kopira prvi argument u id_no[]

  name = new char[strlen(acc_r.name) + 1]; // kreira prostor na unos imena

  strcpy(name, acc_r.name);   // kopira drugi argument u novi prostor

  balance = acc_r.balance;
  rate    = acc_r.rate;

  cout << endl << endl;
  cout << "Copy constructor executed." << endl;
}
```

```
Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
 delete [] name;
}

double Savings_Account::Get_Balance()
{
 return balance;
}

double Savings_Account::Calc_Interest()
{
 double interest;

 interest = balance * rate;
 balance += interest;

 return interest;
}

void Savings_Account::Deposit(double amount)
{
 balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
 bool result;

 if (amount <= balance)
 {
  balance -= amount;
  result = true;
 }
 else
  result = false;

 return result;
}

void Savings_Account::Display_ID()
{
 cout << id_no;
}
```

```
void Savings_Account::Display_Name()
{
 cout << name;
}

int main()
{
 cout << setprecision(2)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);

 char id[5];
 char buffer[81];  //Privremeno skladisti ime
 double bal;
 double rt;
 cout << "Enter Account ID: ";
 cin.getline(id, 5);

 cout << endl;
 cout << "Enter Account Holder's Name: ";
 cin.getline(buffer, 81);

 cout << endl;
 cout << "Enter Balance: ";
 cin >> bal;

 cout << endl;
 cout << "Enter Interest Rate: ";
 cin >> rt;

 Savings_Account acc1(id, buffer, bal, rt);

 cout << endl << endl;
 cout << "Account created with Account number ";
 acc1.Display_ID();
 cout << endl;
 cout << " and owner name ";
 acc1.Display_Name();

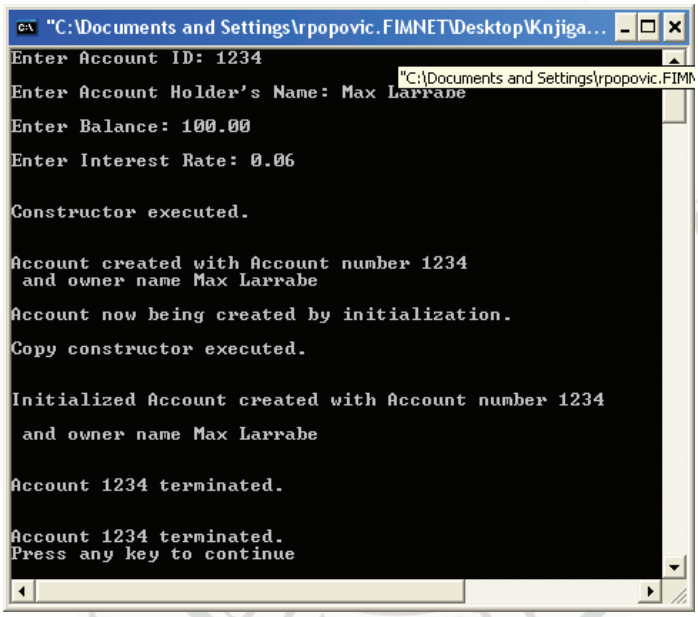
 cout << endl << endl;
 cout << "Account now being created by initialization.";
 Savings_Account acc2 = acc1;

 cout << endl << endl;
 cout << "Initialized Account created with Account number ";
 acc2.Display_ID();
```

```
 cout << endl;
 cout << "\n and owner name ";
 acc2.Display_Name();

 cout << endl;

 return 0;
}
```



9.3 Korišćenje rezervisane reči const u radu sa klasama

Rezervisana reč const može da se iskoristi na više načina zavisno od objekata i metoda klasa.

Konstantni objekti i metodi

Moguće je deklarisati konstantne objekte, kao što je moguće deklarisati konstantne cele brojeve, promenljive tipa double, itd. Konstantan objekat je onaj čije vrednosti primeraka promenljivih ne mogu da se menjaju. Na primer, posmatrajmo Test_Class, definisanu u prethodnoj sekciji sa dodatnim metodom Change_Value().

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

class Test_Class
{
 private:
   int n;

 public:
   Test_Class(int i = 0);    //Konstruktor sa jednim argumentom i default vrednoscu
   int Get_Value();
   void Change_Value(int);
};

Test_Class::Test_Class(int i)
{
 n = i;

 cout   << "\n\nConstructor Executed: Data member initialized to " << n;
}

int Test_Class::Get_Value()
{
 return n;
}

void Test_Class::Change_Value(int i)
{
 n = i;
}
```

Suppose we declare the following in main().

```
const Test_Class tc_object1(4);

Test_Class tc_object2(7);
```

Sledeća naredba dodele nije ispravna zato što je objekat tc_object1 konstantan i ne možemo da modifikujemo konstantan objekat u funkciji main().

```
tc_object1 = tc_object2;     //pogresno - ne mozemo da modifikujemo konstantan
                             //objekat
```

Kada je objekat konstantan, ne možemo da menjamo vrednost njegove instance promenljive, n. Ne postoji način da kompajler sazna

razliku između aksesor metoda (koji ne menja vrednost instance varijable) i mutator metoda (koji menja vrednost instance promenljive). Kompajler će izbaciti poruku: ili warning ili error, ako primenimo bilo koji od metoda klase Test_Class na objekat tc_object1.

```
tc_object1.Get_Value();      //Kompajler izbacuje poruku
tc_object.Change_Value();    // Kompajler izbacuje poruku
```

Veoma je ozbiljno ograničenje ne dozvoliti bilo kom metodu da ne radi sa konstantnim objektom. Želimo da možemo da primenimo aksesor metod na konstantan objekat zato što takav metod ne pokušava da promeni primerke varijabli te klase. Da bismo rekli kompajleru da je metod tipa aksesor metod i da treba da mu bude dozvoljeno da radi sa konstantnim objektima, on treba da bude deklarisan kao konstantan. Ovo je dato dodavanjem rezervisane reči const nakon zagrada koje slede nakon imena aksesor metoda i u deklaraciji i u definiciji funkcije.

Konstantni metodi

Bilo koji metod klase koji je *accessor* (znači, ne menja vrednost bilo koje instance promenljive) treba da bude deklarisan kao konstantan. Ovo je dato dodavanjem rezervisane reči const nakon zagrada koje slede nakon imena metoda i u deklaraciji i u definiciji. Ovo dozvoljava da takav metod bude primenjen na konstantan objekat.

Sledi korektna verzija deklaracije Test_Class.

```
#include <iostream.h>

using namespace std;

class Test_Class
{
 private:
   int n;

 public:
   Test_Class(int i = 0);      // Konstruktor sa jedim argumentom koji ima default
                               // vrednost
   int Get_Value() const;    //Accessor funkcija deklarisana kao konstanta - const
   void Change_Value(int);
};
```

```
Test_Class::Test_Class(int i)
{
 n = i;
 cout  <<  "\n\nConstructor Executed: Data member initialized to " << n;
}

int Test_Class::Get_Value() const          //const se zahteva
{
 return n;
}

void Test_Class::Change_Value(int i)
{
 n = i;
}
```

Sledeći program ilustruje korišćenje konstantnih metoda. Treba napomenuti da ako uključite naredbu u main() koja je u komentaru, kompajler će izdati "error message" zato što naredba pokušava da promeni konstantan objekat.

```
//Ovaj program ilustruje konstantne funkcije članice.
//Napomena: naredba u komentaru u funkciji main() prouzrokuje
//kompajlerovu poruku o gresci

#include <iostream>

using namespace std;

class Test_Class
{
 private:
 int n;

 public:
   Test_Class(int i = 0);    // Konstruktor sa jedim argumentom koji ima default
                             // vrednost
   int Get_Value() const;
   void Change_Value(int);
};
Test_Class::Test_Class(int i)
{
 n = i;

 cout << endl;
```

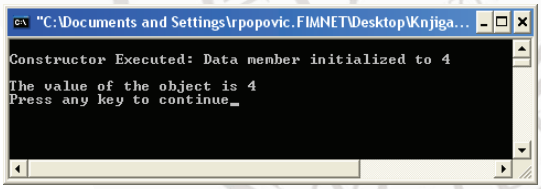
```
  cout << "Constructor Executed: Data member initialized to "
       << n << endl << endl;
}
int Test_Class::Get_Value() const
{
  return n;
}
void Test_Class::Change_Value(int i)
{
  n = i;
}

int main()
{
  const Test_Class tc_object1(4);

  cout << "The value of the object is " << tc_object1.Get_Value() << endl;

  //Dodavanje sledeceg reda prouzrokovalo bi gresku u kompajliranju
  //tc_object1.Change_Value(7);

  return 0;
}
```



const u deklaracijama pokazivača

Rezervisana reč const može da se koristi na dva načina u radu sa pokazivačima, prvo, pokazivač može da se deklariše kao konstanta, i cilj na koji pokazuje pokazivač može da bude konstantan. Ako const neposredno predhodi imenu pokazivača u deklaraciji, onda je pokazivač konstantan. Sledi, jednom inicijalizovan pokazivač ne može da se promeni, tj. ne može da pokazuje na nešto drugo. Posmatrajmo kao primer sledeće deklaracije:

```
int i = 4,
    j = 7;
int* const i_p = &i; //pointer i_p je konstantan, zato, njegova vrednost &i, ne moze
                     //da se menja. Ona mora uvek da pokazuje na i.
```

Deklaracija pokazivača i_p kaže da je pokazivač konstantan. Tako, jednom inicijalizovana u deklaraciji, vrednost pokazivača ne može da se menja. Sledeća dodela je zato neispravna:

```
i_p = &j; //pogresno jer je pokazivac i_p konstanta
```

Međutim, ispravno je promeniti vrednost targeta i_p.

```
*i_p = 5; //ispravno – cilj pokazivaca i_p NIJE konstanta.
```

Ako se rezervisana reč const pojavi pre tipa podatka u deklaraciji pokazivača, cilj na koji on pokazuje je konstanta. Ovo znači da cilj pokazivača ne može da se menja preko pokazivača. Razmotrimo sledeće deklaracije:

```
int n = 3,
    m = 5;

const int* n_p = &n;       //Cilj od n_p, sa imenom n, ne može da se promeni
                           //preko n_p.
```

Deklaracija pokazivača n_p deklariše target pokazivača kao konstantu; tj. n_p je pokazivač na konstantan podatak objekta. Kako je cilj pokazivača n_p konstantan, možete razmišljati o pokazivaču n_p kao o "read-only" pokazivaču. Zato, sledeća dodela je neispravna:

```
*n_p = m;     //neispravno - target pointera n_p je konstantan
```

S druge strane, n_p nije konstanta. Zato možemo da napravimo sledeću dodelu

```
n_p = &m;     //ispravno - sam pokazivac nije konstanta
```

Mada smo promenili cilj na koji pointer pokazuje, n_p je još uvek pokazivač na konstantni objekat. Zato, neispravno je promeniti *target* pointera n_p

```
*n_p = 6;   //neispravno:ne moze se promeniti konstantan objekat na koji pokazuje
//pointer.
```

Međutim, možemo još uvek promeniti vrednost m kao što sledi:

```
m = 6;
```

Mešanje konstantnih pokazivača i pokazivača koji nisu konstantni

Važan princip mešanja pokazivača koji treba zapamtiti je: deklaracija može da uvede restrikcije koje se tiču konstante, ali se ne mogu ukinuti. Razmotrimo sledeće deklaracije:

```
int i = 9;
int* i_p = &i;
const int* j_p = i_p;    //ispravno
```

Pokazivač i_p nije konstantan niti je njegov cilj. Međutim, deklaracija j_p, koja inicijalizuje j_p na i_p, kaže da je target od j_p konstantan. Ograničenje koje se odnosi na j_p je da je njegov target konstantan, restrikcija koja nije *true* za i_p. Zbog toga, ispravno je promeniti i, target od i_p, preko i_p ali ne preko j_p.

```
*i_p = 2;    //ispravno: target od i_p, i, nije konstantan

*j_p = 1;    //neispravno: target od j_p, i, je konstanta kada je
             //referencirana preko j_p
```

S druge strane, razmotrimo sledeće deklaracije.

```
int n = 8;

const int* n_p = &n;

int* m_p = n_p;    //neispravno – ne moze se ukloniti restrikcija za konstantu
```

Poslednja deklaracija je neispravna. Pointer n_p je deklarisan kao pointer na objekat sa konstantnim podacima. Deklaracija m_p je jedan običan pointer. Zato je neispravna inicijalizacija m_p, pokazivača koji nije konstantan, na n_p, konstantan pokazivač.

Konstantni argumenti u funkciji

Rezervisana reč const može takođe da se pojavi sa pokazivačem i referencom kao argumentima funkcije. Na primer, razmotrimo sledeću deklaraciju funkcije

```
void Func(char* p, const char* q);
```

U ovom slučaju, const u drugom argumentu znači da je *target* od q konstantan. Zbog toga, podaci na koje pokazuje argument q ne mogu da se promene preko pokazivača.

```
// Ovaj program prikazuje kako da se nadje srednja vrednost elemenata u jednom
// nizu
// Program nalazi  srednju vrednost prosledjivanjem niza funkciji koja izracunava
// srednju vrednost.

#include <iostream>
#include <iomanip>

using namespace std;

double Avg(const int [], int);
int main()
{
 const int NUM_QUIZZES = 10;
 int    grade[NUM_QUIZZES];       // Niz sa ocenama iz kviza
 int    quiz;                     // Indeks niza
 double grade_avg;

 cout << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint)
     << setprecision(1);

 cout << "Please enter " << NUM_QUIZZES << " integer quiz grades."
     << endl << endl;

 for (quiz = 0; quiz < NUM_QUIZZES; ++quiz)
 {
  cout << endl;
        cout << "Enter grade for quiz " << quiz + 1 << ": ";
  cin >> grade[quiz];
 }
 grade_avg = Avg(grade, NUM_QUIZZES);
```

```
  cout << endl;
  cout << "The average quiz grade is " << grade_avg << endl;
  return 0;

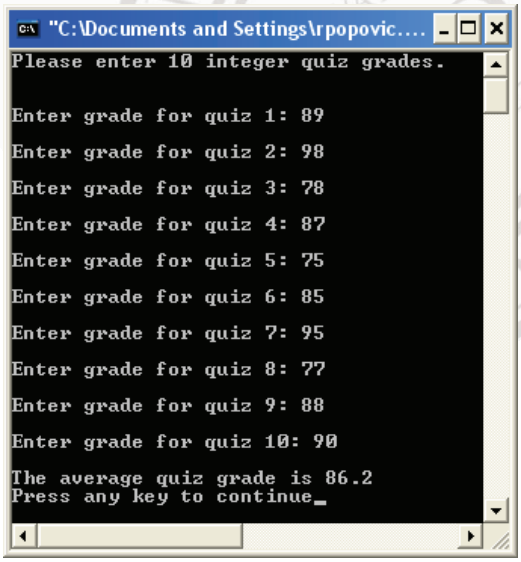
}    // Kraj main() funkcije

double Avg(const int arr[], int size)
{
  int    i,             // Indeks niza
         sum = 0;       // Zbir ocena
  double avg;           // Prosecna ocena

  for (i = 0; i < size; ++i)
    sum += arr[i];

  avg = double(sum) / size;

  return avg;
}   // Kraj funkcije Avg()
```



Konstantni argumenti i konstruktor kopije

Ako uključite konstruktor kopije u definiciju klase, argument treba da bude deklarisan sa const da bi se obezbedilo da se kod konstruktora kopije ne menja ništa u vezi sa argumentom. Ovo je poželjno zbog toga što

konstruktor kopije treba samo da kopira i ništa drugo. Slede deklaracija i definicija za konstruktor kopije za štedni račun Savings_Account sa bezbednijim kodom:

Deklaracija:

```
Savings_Account(const Savings_Account&);
```

Definicioni heder:

```
Savings_Account::Savings_Account (const Savings_Account& acc_r)
```

Aksesor metodi koji vraćaju konstantne pokazivače

U verzijama klase Savings_Account koje smo razvijali u ovom poglavlju, nismo imali aksesor funkcije koje vraćaju id_no ili name članove klase. Umesto toga koristili smo funkcije Display_ID() i Display_Name(), koji prosto prikazuju na displeju odgovarajuće podatke članove. Ako pišemo kod za aksesor funkciju za name kao što sledi

```
char* Savings_Account::Get_Name()
{
  return name;
}
```

onda, kada funkcija main() primenjuje ovo na Savings_Account objekat, main() će imati pristup članu name objekta kroz pokazivač koji vraća Get_Name(). Ovo može da naruši princip skrivanja podataka i ne uvaži činjenicu da je name deklarisano kao private u deklaraciji klase. Na primer, main() koristi pokazivač koji vraća Get_Name() da bi promenio ime objekta u string "XXX".

```
//Ovaj program ilustruje koriscenje konstruktora kopije u klasi
//koja ima pokazivač na podatak clan klase.
//Klasa sadrzi aksesor funkcije koje vracaju konstantan pokazivac.

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

class Savings_Account
{
  private:
    char   id_no[5];
```

```
    char*  name;
    double balance;
    double rate;

  public:
    Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
    Savings_Account(const Savings_Account &);
    ~Savings_Account();
    double Calc_Interest();
    double Get_Balance();
    char*  Get_Name();
    void Deposit(double);
    bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
  strcpy(id_no, id);   //kopira prvi argument u „id_no[]“

  name = new char[strlen(n_p) + 1];  //kreiranje prostora za ime

  strcpy(name, n_p);  //kopiranje drugog argumenta u novi prostor

  balance = bal;
  rate    = rt;

  cout << endl << endl;
  cout << "Constructor executed." << endl << endl;
}
Savings_Account::Savings_Account(const Savings_Account& acc_r)
{
  strcpy(id_no, acc_r.id_no);  // kopira prvi argument u „id_no[]“

  name = new char[strlen(acc_r.name) + 1]; // kreiranje prostora za ime

  strcpy(name, acc_r.name);  // kopiranje drugog argumenta u novi prostor

  balance = acc_r.balance;
  rate    = acc_r.rate;

  cout << endl << endl;
  cout << "Copy constructor executed." << endl << endl;
}

Savings_Account::~Savings_Account()
{
```

```
  cout << endl << endl;
  cout << "Account " << id_no << " terminated." << endl;
  delete [] name;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

char* Savings_Account::Get_Name()
{
  return name;
}
double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

int main()
{
```

```
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  char id[5];
  char buffer[81]; //Privremeno skladisti ime
  double bal;
  double rt;

  cout << "Enter Account ID: ";
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

  cout << endl << endl;
  cout << "Account now being created." << endl << endl;

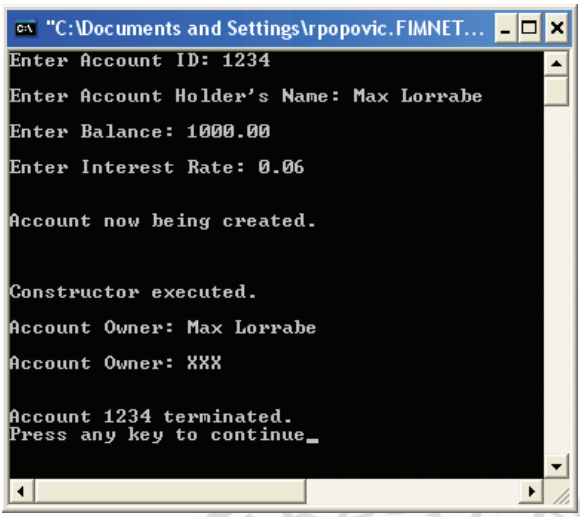
  Savings_Account acc1(id, buffer, bal, rt);

  cout << "Account Owner: " << acc1.Get_Name() << endl << endl;

  char* name = acc1.Get_Name();
  strcpy(name, "XXX");

  cout << "Account Owner: " << acc1.Get_Name() << endl;

  return 0;
}
```



Da bismo napisali siguran kod za aksesor metod koji vraća pokazivač, treba da napravimo da metod vraća konstantan pokazivač. Ovo obezbeđuje da objekat na koji se pokazuje ne može biti promenjen kroz pokazivač od strane pozivajuće funkcije.

Na primer, u klasi Savings_Account, bezbedan metod koji vraća vrednost ID broja je:

```
const char* Get_Name()
{
 return name;
}
```

U funkciji main(), možemo da napišemo kôd koji sledi, ali nam onda neće biti dozvoljeno da promenimo ime-name objekta acc1 kroz pointer.

```
const char* name_p = acc1.Get_Name();
```

Obratite pažnju da je sledeća naredba neispravna zato što uklanja restrikcije koje postavlja const. Pokazivač koji vraća Get_Name() je pokazivač na konstantu, dok je name_p pokazivač koji nije konstantan.

```
char* name_p = acc1.Get_Name();
```

Sledeći program ilustruje ovu ideju.

```
//Ovaj program ilustruje koriscenje konstruktora kopije
//u klasi koja ima pokazivac kao podatak clan.
//Klasa sadrzi accessor funkcije koje vracaju konstantan pokazivac.

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   Savings_Account(const Savings_Account&);
   ~Savings_Account();
   double Calc_Interest();
   double Get_Balance();
   const char* Get_Id();
   const char* Get_Name();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
           {
            strcpy(id_no, id);   //kopira prvi argument u „id_no[]“

            name = new char[strlen(n_p) + 1];  //kreiranje prostora za ime

            strcpy(name, n_p);  //kopiranje drugog argumenta u novi prostor

            balance = bal;
            rate    = rt;

            cout << endl << endl;
            cout << "Constructor executed." << endl;
           }
           Savings_Account::Savings_Account(const Savings_Account& acc_r)
           {
```

```
  strcpy(id_no, acc_r.id_no);  // kopira prvi argument u „id_no[]“

  name = new char[strlen(acc_r.name) + 1]; // kreiranje prostora za ime

  strcpy(name, acc_r.name);  // kopiranje drugog argumenta u novi prostor

  balance = acc_r.balance;
  rate    = acc_r.rate;

  cout << endl << endl;
  cout << "Copy constructor executed." << endl;
}

Savings_Account::~Savings_Account()
{
  cout << endl << endl;
  cout << "Account " << id_no << " terminated." << endl;
  delete [] name;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

const char* Savings_Account::Get_Id()
{
  return id_no;
}

const char* Savings_Account::Get_Name()
{
  return name;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}
void Savings_Account::Deposit(double amount)
{
```

```
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  char id[5];
  char buffer[81];  //Privremeno skladisti ime
  double bal;
  double rt;

  cout << "Enter Account ID: ";
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

  cout << endl <<endl;
  cout << "Account now being created." << endl <<endl;
```

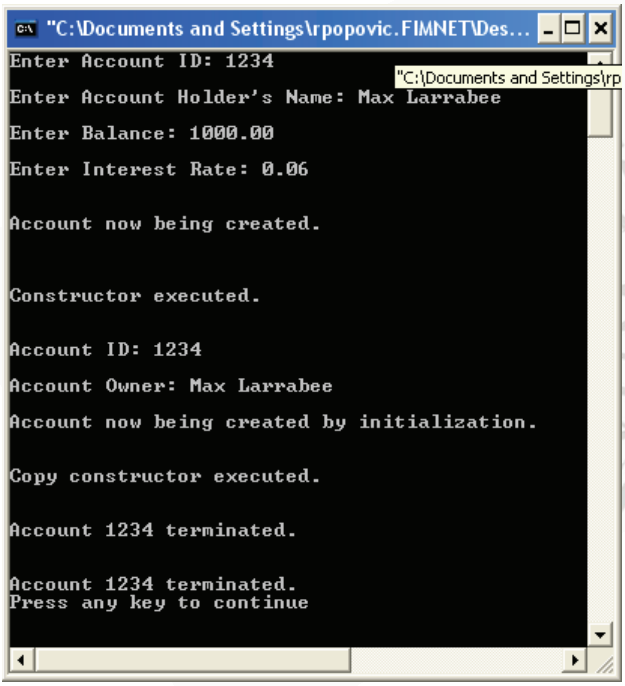
```
    Savings_Account acc1(id, buffer, bal, rt);

    cout << endl << endl;
    cout << "Account ID: " << acc1.Get_Id() << endl <<endl;
    cout << "Account Owner: " << acc1.Get_Name() << endl <<endl;

    cout << "Account now being created by initialization." << endl;

    Savings_Account acc2 = acc1;

    return 0;
}
```



Aksesor metodi Get_Id() i Get_Name() vraćaju const char*. Ovo garantuje da pozivajuća funkcija, main() u ovom slučaju, ne može da menja cilj pokazivača koji vraća svaki metod.

9.4 Objekti, funkcije i pokazivači

Klasa je tip podatka. Sledi, funkcija može da vrati objekat i možemo da prosledimo objekat kao argument funkcije. Postoje tri načina na koji se funkciji može proslediti objekat: po vrednosti, po pokazivaču i po referenci.

Funkcije koje vraćaju objekat

Napisaćemo kôd funkcije Get_Account() koji vraća glavnoj funkciji main() objekat Savings_Account. U hederu definicije Get_Account(), nije potrebno koristiti ime klase i operator dosega ::, zato što funkcija nije član klase Savings_Account. Funkcija vraća Savings_Account objekat, što je razlog zbog čega se identifikator Savings_Account pojavljuje pre imena funkcije.

```
Savings_Account Get_Account()
{
  char id[5];
  char buffer[81];
  double bal;
  double rt;

  cout << endl;
  cout << "Enter Account ID: ";
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

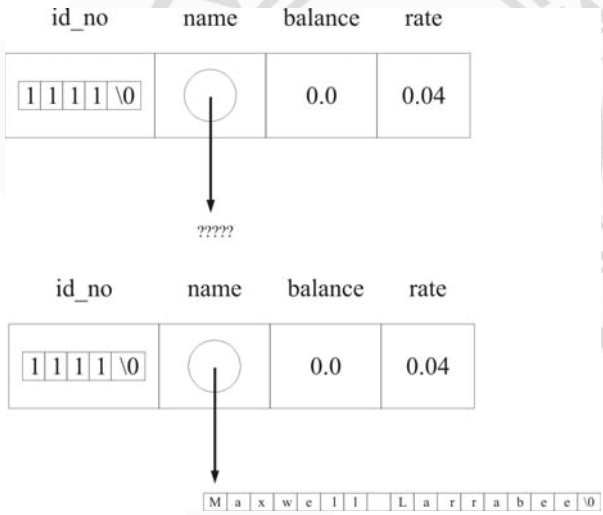
  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

  Savings_Account acc(id, buffer, bal, rt);

  return acc;
}
```

Ova funkcija sadrži veći deo koda koji se pojavljuje u glavnoj funkciji main() u programima koji koriste Savings_Account.

Get_Account() kreira lokalni Savings_Account objekat, acc, koristeći korisnikove ulazne podatke. Zato, name član objekta acc pokazuje na prostor koji je alociran na hipu od strane Savings_Account konstruktora. Naredba return šalje privremenu kopiju objekta acc pozivajućoj funkciji. Kada se funkcija završi, lokalni objekat acc izlazi iz dosega i, zbog toga, se uništava. Ovo prouzrokuje da se Savings_Account destruktor izvršava, i on dealocira prostor na koji je name član objekta acc pokazivao. Sada, ako je naredba return poslala član-po-član kopiju objekta acc nazad pozivajućoj funkciji, name član te kopije pokazivaće na nealocirani prostor. Ovaj pogrešan pokazivač može da prouzrokuje da se program sruši. Da bismo obezbedili da se ovo ne desi, C++ koristi Savings_Account konstruktor kopije da bi napravio privremenu kopiju objekta acc. Zbog toga, name član privremene kopije pokazuje na različito mesto na hipu od onoga koje je bilo za name član objekta acc.



Sledi kompletan program koji koristi novu funkciju.

```
//Ovaj program ilustruje koriscenje konstruktora kopije
//u klasi koja ima pokazivac kao data clan.
//Konstruktor kopije se koristi da bi napravio privremenu kopiju
//vracenog objekta u funkciji Get_Account().

#include <iostream>
#include <iomanip>
```

```
#include <string>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   Savings_Account(const Savings_Account&);
   ~Savings_Account();
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);  //kopira prvi argument u „id_no[]“

 name = new char[strlen(n_p) + 1]; //kreira prostor za ime

 strcpy(name, n_p);  //kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;

 cout << endl << endl;
 cout << "Constructor executed." << endl;
}

Savings_Account::Savings_Account(const Savings_Account& acc_r)
{
 strcpy(id_no, acc_r.id_no);  // kopira prvi argument u „id_no[]“

 name = new char[strlen(acc_r.name) + 1]; // kreira prostor za ime

 strcpy(name, acc_r.name);  // kopira drugi argument u novi prostor

 balance = acc_r.balance;
```

```
  rate    = acc_r.rate;

  cout << endl << endl;
  cout << "Copy constructor executed." << endl;
}
Savings_Account::~Savings_Account()
{
  cout << endl <<endl;
  cout << "Account " << id_no << " terminated." << endl;
  delete [] name;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}
```

```
Savings_Account Get_Account();    //Prototip za Get_Account()

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  cout << endl << endl;
  cout << "Account now being created.";

  Savings_Account acc1 = Get_Account();

  cout << endl << endl;
  cout << "Account now being created by initialization." << endl;

  Savings_Account acc2 = acc1;

  return 0;
}

Savings_Account Get_Account()
{
  char id[5];
  char buffer[81];
  double bal;
  double rt;

  cout << endl;
  cout << "Enter Account ID: ";
  cin.getline(id, 5);

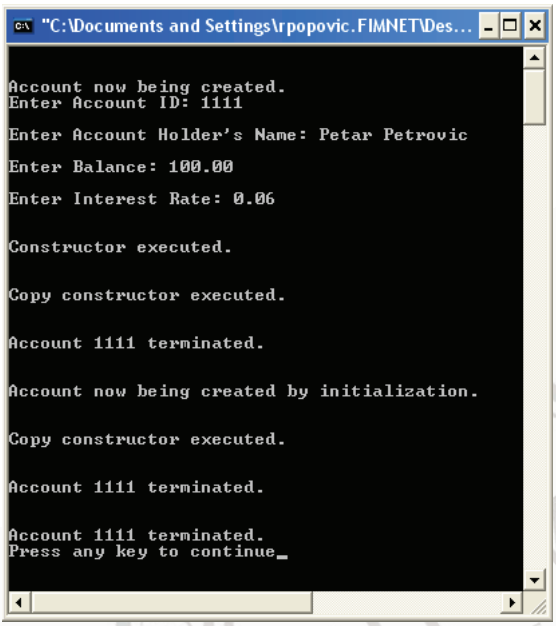
  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

  Savings_Account acc(id, buffer, bal, rt);
```

```
 return acc;
}
```



Poruku "Konstruktor izvršen" je proizveo konstruktor kada se kreira lokalni objekat acc u funkciji Get_Account(). Kada se return nardba izvršava, ona načini privremenu kopiju objekta acc, koju izvršava konstruktor kopije. Prva poruka "Account 1111 terminated" se izvršava kada se lokalni objekat acc uništi kada se funkcija završi. Nazad u funkciji main(), privremena kopija objekta acc koja je bila kreirana od strane funkcije dodeljena je objektu acc1. Treba obratiti pažnju da ovo ne prouzrokuje da se kopija konstruktora izvrši. Objekat acc1 postaje privremena kopija koja je bila vraćena od funkcije. Sledeći korak u funkciji main() je kreiranje inicijalizacijom objekta acc2. Ovo prouzrokuje da se konstruktor kopije izvršava. Konačno, kada se funkcija main() završava, acc1 i acc2 se uništavaju, što kao rezultat daje da se destruktor izvršava za svaki od njih.

Napomena, ako definišete funkciju koja vraća objekat klase koju je definisao korisnik, obezbedite klasu sa konstruktorom kopije.

Prosleđivanje objekta po vrednosti

Pretpostavimo da želimo da prosledimo objekat po vrednosti funkciji Display_Account() na takav način da izvršavanje Display_Account(acc1) prikazuje informacije o računu za objekat acc1. Pretpostavimo definiciju hedera funkcije Display_Account() na sledeći način:

```
Display_Account(Savings_Account acc)
```

Kada program izvršava Display_Account(acc1), parametri funkcije objekta acc su inicijalizovani na vrednost objekta acc1. Zato, konstruktor kopije za Savings_Account se koristi da prosledi argument funkciji. Podsetimo se da treba da napišete kôd za sopstveni konstruktor kopije kad god klasa ima član podatak koji je pokazivač. Zbog toga, kada prosleđujemo objekat po vrednosti, moramo da budemo sigurni da naša klasa sadrži konstruktor kopije.

Sledi program koji ilustruje korišćenje konstruktora kopije kada prosleđuje jedan objekat po vrednosti.

```
//Ovaj program ilustruje prosledjivanje objekta po vrednosti.
//Konstruktor kopije se zahteva da bi se korektna kopija objekta
//prosledila funkciji.

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   ~Savings_Account();
   Savings_Account(const Savings_Account&);
   const char* Get_Id()
   const; const char* Get_Name() const;
```

```
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);   //kopira prvi argument u „id_no[]“

 name = new char[strlen(n_p) + 1];  //kreira prostor za ime

 strcpy(name, n_p);  //kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;
}

Savings_Account::Savings_Account(const Savings_Account& acc_r)
{
 strcpy(id_no, acc_r.id_no);  // kopira prvi argument u „id_no[]“

 name = new char[strlen(acc_r.name) + 1];  // kreira prostor za ime

 strcpy(name, acc_r.name);  // kopira drugi argument u novi prostor

 balance = acc_r.balance;
 rate    = acc_r.rate;
 cout << endl << endl;
 cout << "Copy constructor executed." << endl;
}

Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
 delete [] name;
}

const char* Savings_Account::Get_Id() const
{
 return id_no;
}

const char* Savings_Account::Get_Name() const
{
```

```
  return name;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

void Display_Account(Savings_Account);     //prototip funkcije

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);
```

```
char id[5];
char buffer[81];  //privremeno skladisti ime
double bal;
double rt;
double amount;

cout << endl;
cout << "Enter Account ID: ";
cin.getline(id, 5);

cout << endl;
cout << "Enter Account Holder's Name: ";
cin.getline(buffer, 81);

cout << endl;
cout << "Enter Balance: ";
cin >> bal;

cout << endl;
cout << "Enter Interest Rate: ";
cin >> rt;

Savings_Account acc1(id, buffer, bal, rt);

Display_Account(acc1);

cout << endl << endl;
cout << "Enter an amount to deposit: ";
cin >> amount;

acc1.Deposit(amount);

cout << endl << endl;
cout << "A deposit of " << amount << " was made.";

Display_Account(acc1);

acc1.Calc_Interest();

cout << endl << endl;
cout << "Interest was applied to the account.";

Display_Account(acc1);

cout << endl << endl;
cout << "Name: " << acc1.Get_Name();
```

```
  cout << endl << endl;
  cout << "Enter an amount to withdraw: ";
  cin >> amount;
  if (acc1.Withdraw(amount))
  {
    cout << endl << endl;
    cout << "A withdrawal of " << amount << " was made.";
  }
  else
  {

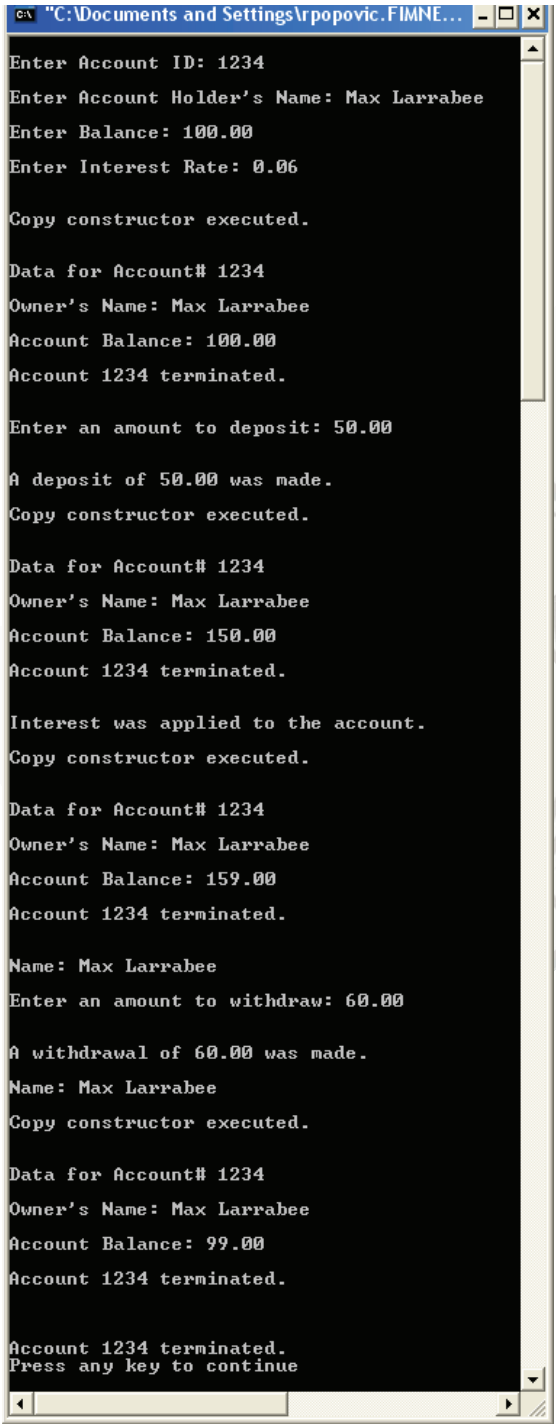
    cout << endl << endl;
    cout << "WITHDRAWAL NOT MADE: Insufficient funds.";
  }
  cout << endl << endl;
  cout << "Name: " << acc1.Get_Name();

  Display_Account(acc1);

  cout << endl;
  return 0;
}
void Display_Account(Savings_Account acc)
{
  cout << endl << endl;
  cout << "Data for Account# " << acc.Get_Id();

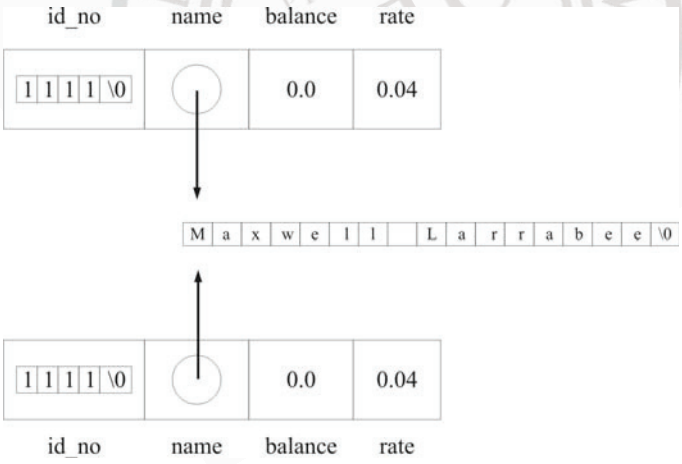
  cout << endl << endl;
  cout << "Owner's Name: " << acc.Get_Name();

  cout << endl << endl;
  cout << "Account Balance: " << acc.Get_Balance();
}
```


"C:\Documents and Settings\rpopovic.FIMNE...
Enter Account ID: 1234
Enter Account Holder's Name: Max Larrabee
Enter Balance: 100.00
Enter Interest Rate: 0.06
Copy constructor executed.
Data for Account# 1234
Owner's Name: Max Larrabee
Account Balance: 100.00
Account 1234 terminated.
Enter an amount to deposit: 50.00
A deposit of 50.00 was made.
Copy constructor executed.
Data for Account# 1234
Owner's Name: Max Larrabee
Account Balance: 150.00
Account 1234 terminated.
Interest was applied to the account.
Copy constructor executed.
Data for Account# 1234
Owner's Name: Max Larrabee
Account Balance: 159.00
Account 1234 terminated.
Name: Max Larrabee
Enter an amount to withdraw: 60.00
A withdrawal of 60.00 was made.
Name: Max Larrabee
Copy constructor executed.
Data for Account# 1234
Owner's Name: Max Larrabee
Account Balance: 99.00
Account 1234 terminated.
Account 1234 terminated.
Press any key to continue

Treba obratiti pažnju da odmah pre nego što Display_Account() prikaže liniju, program prikazuje poruku iz konstruktora kopije. Konstruktor kopije se izvršava kada se parametar u Display_Account() inicijalizuje na vrednost računa koji se prosleđuje funkciji. Kada se funkcija završava, parametri funkcije, koji su Savings_Account objekti, izlaze iz dosega. Sledi, parametri se uništavaju i destruktori klase izvršavaju, što prouzrokuje prikaz poruke destruktora.

Šta se događa ako programer ne obezbedi konstruktor kopije i umesto toga se osloni na difolt konstruktor kopije? Kada se funkcija Display_Account() poziva po prvi put, difolt konstruktor će napraviti kopiju (član po član) argumenta i smestiti je u parametar funkcije. Zbog toga će name član parametra pokazivati na isti hip prostor argumenta. Problem nastaje kada se funkcija završava zbog toga što se destruktor klase izvršava. Ovo prouzrokuje da se prostor na koji pokazuje parametrov name član dealocira. Ali ovo je isti prostor na koji pokazuje argumentov name član. Rezultat je da argumentov name član pokazuje na nezaštićeni hip prostor - tj. prostor koji može biti ponovo korišćen od strane sistema.



Napomena u vezi pitanja kada koristiti konstruktor kopije:

-Kada se jedan objekat kreira od postojećeg objekta te klase kroz inicijalizaciju.

-Jedan objekat se prosleđuje po vrednosti funkciji.

-Privremena kopija jednog objekta se kreira da bi držala *return* vrednost objekta u funkciji koja vraća objekat te klase.

Kada se konstruktor kopije ne koristi:

-Kada se objekat dodeljuje drugom objektu iste klase.

Pokazivači na objekte

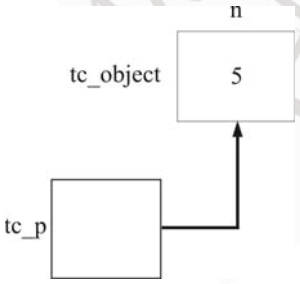
Možete da deklarišete pokazivač na objekat na isti način kao da deklarišete pokazivač na bilo koji tip. Na primer, posmatrajmo Test_Class:

```
class Test_Class
{
 private:
   int n;
 public:
   Test_Class(int i = 0);   //konstruktor sa jednim argumentom sa
                            //default-nom vrednossću
   int Get_Value() const;
   void Change_Value(int);
};
```

U funkciji main(), možemo da načinimo sledeće deklaracije, koje su ilustrovane na donjoj slici.

```
Test_Class tc_object(5);

Test_Class* tc_p = &tc_object;
```



Metodi se mogu primeniti na ciljni objekat koristeći operator indirektnog pristupa članu -> koji se koristi i za pokazivače kod struktura. Dakle, tc_p -> Get_Value() primenjuje Get_Value() metod na cilj tc_p. Podsetimo se da tc_p -> Get_Value() je ekvivalentno (*tc_p).Get_Value(). Podsetimo se da mora da koristimo zagrade oko *tc_p zato što tačka operator ima veći prioritet od operatora dereferenciranja *.

Sledeći kod prikazuje vrednost cilja tc_p pokazivača, menja vrednost cilja na 7, i prikazuje novu vrednost

```
cout << "The value of the object is " << tc_p -> Get_Value();

tc_p -> Change_Value(7);

cout << end;

cout << "\nThe new value of the object is " << tc_p -> Get_Value();
```

U sledećem programu koristimo pokazivač na objekat da bi prikazali vrednost objekta, promenili njegovu vrednost i zatim prikazali novu.

```
//Ovaj program prikazuje koriscenje pokazivaca na objekte
//i koriscenje operatora -> za poziv metoda.

#include <iostream>

using namespace std;

class Test_Class
{
 private:
   int n;

 public:
   Test_Class(int i = 0);    //Konstruktor sa jednim argumentom i
                             //default-nom vrednoscu
   int Get_Value() const;
   void Change_Value(int);
};

Test_Class::Test_Class(int i)
{
 n = i;

 cout << endl << endl;
 cout << "Constructor Executed: Data member initialized to " << n << endl;
}

int Test_Class::Get_Value() const
{
 return n;
}
```

```
void Test_Class::Change_Value(int i)
{
  n = i;
}

int main()
{
  Test_Class tc_object(5);
  Test_Class* tc_p = &tc_object;

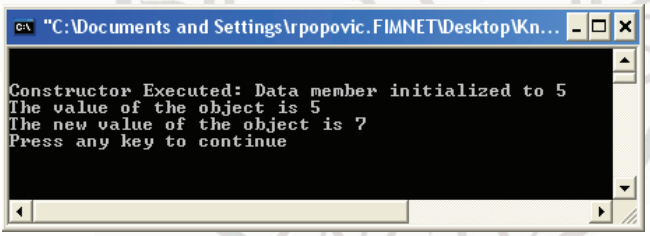
  cout << "The value of the object is " << tc_p -> Get_Value();

  tc_p -> Change_Value(7);              //Promena vrednosti promenljive
                                        //na koju pokazivac pokazuje

  cout << endl;

  cout << "The new value of the object is " << tc_p -> Get_Value();

  cout << endl;
  return 0;
}
```



Prosleđivanje objekta po pokazivaču

U prethodnoj sekciji, koristili smo funkciju Display_Account() za prikaz podataka Savings_Account objekta. Sledeći program koristi pokazivače u definiciji Display_Account().

```
//Ovaj program koristi prosirenu verziju Savings_Account.
//Program koristi konstruktor sa default argumentima.
//ID broj je uskladisten kao niz a ime vlasnika racuna character pointer.
//Konstruktor koristi dinamicku alokaciju memorije.
//Objekat Savings_Account je prosledjen po pointeru funkciji Display_Account()

#include <iostream>
```

```
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   ~Savings_Account();
   const char* Get_Id() const;
   const char* Get_Name() const;
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{

 strcpy(id_no, id);   // kopira prvi argument u „id_no[]“

 name = new char[strlen(n_p) + 1];  // kreira prostor za ime

 strcpy(name, n_p);  // kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;
}

Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
 delete [] name;
}

const char* Savings_Account::Get_Id() const
{
```

```
  return id_no;
}

const char* Savings_Account::Get_Name() const
{
  return name;
}

double Savings_Account::Get_Balance()
{
  return balance;
}
double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

void Display_Account(Savings_Account*);     //prototip funkcije

int main()
{
```

```
cout << setprecision(2)
    << setiosflags(ios::fixed)
    << setiosflags(ios::showpoint);

char id[5];
char buffer[81];  //Privremeno skladisti ime
double bal;
double rt;
double amount;

cout << "Enter Account ID: ";
cin.getline(id, 5);

cout << endl;
cout << "Enter Account Holder's Name: ";
cin.getline(buffer, 81);

cout << endl;
cout << "Enter Balance: ";
cin >> bal;

cout << endl;
cout << "Enter Interest Rate: ";
cin >> rt;

Savings_Account acc1(id, buffer, bal, rt);

Display_Account(&acc1);

cout << endl << endl;
cout << "Enter an amount to deposit: ";
cin >> amount;

acc1.Deposit(amount);

cout << endl <<endl;
cout << "A deposit of " << amount << " was made." << endl;

Display_Account(&acc1);

acc1.Calc_Interest();

cout << endl << endl;
cout << "Interest was applied to the account." << endl;
Display_Account(&acc1);
```

```
  cout << endl << endl;
  cout << "Name: " << acc1.Get_Name();

  cout << endl << endl;
  cout << "Enter an amount to withdraw: ";
  cin >> amount;

  if (acc1.Withdraw(amount))
  {

    cout << endl << endl;
    cout << "A withdrawal of " << amount << " was made." << endl;
  }
  else
  {
    cout << endl << endl;
    cout << "WITHDRAWAL NOT MADE: Insufficient funds." << endl;
  }
  cout << endl << endl;

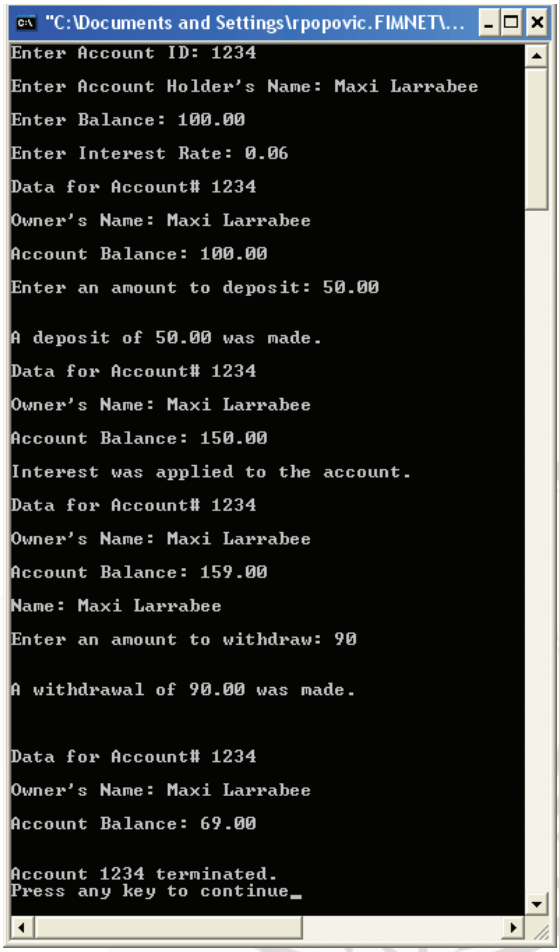
  Display_Account(&acc1);

  cout << endl;
  return 0;
}
void Display_Account(Savings_Account* acc_p)
{
  cout << endl;

  cout << "Data for Account# " << acc_p -> Get_Id() << endl << endl;

  cout << "Owner's Name: " << acc_p -> Get_Name() << endl << endl;

  cout << "Account Balance: " << acc_p -> Get_Balance();
}
```



Reference na objekte

Mogu se definisati reference na varijable objekta na isti način kao kada definišete reference na varijable za ugrađene (*native*) tipove podataka. Posmatrajmo sledeću deklaraciju:

```
Savings_Account acc1("1111", "Maxwell Larrabee", 100.00, 0.06);

Savings_Account& acc_r = acc1;
```

Promenljiva acc_r je referenca, odnosno drugo ime za varijablu acc1. Rad sa acc_r je ekvivalentan radu sa acc1. Sledi:

```
acc_r.Deposit(50.00);
```

povećava za 50.00 vrednost koja se uplaćuje za acc1.

Prosleđivanje objekata po referenci

Možemo da prosledimo objekat funkciji po referenci. Posmatramo sledeći primer:

```
//Ovaj program koristi prosirenu verziju klase Savings_Account.
//Program koristi konstruktor sa default argumentima.
//ID broj je uskladisten kao niz a ime vlasnika racuna je
//pokazivac na promenljivu tipa karakter.
//Konstruktor koristi dinamicku alokaciju memorije.
//Savings_Account objekat je prosledjen po referenci funkciji Display_Account()

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   ~Savings_Account();
   const char* Get_Id() const;
   const char* Get_Name() const;
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);   // kopira prvi argument u „id_no[]“
```

```
  name = new char[strlen(n_p) + 1];  //kreira prostor za ime

  strcpy(name, n_p);  //kopira drugi argument u novi prostor

  balance = bal;
  rate    = rt;
}

Savings_Account::~Savings_Account()
{
  cout << endl << endl;
  cout << "Account " << id_no << " terminated." << endl;
  delete [] name;
}

const char* Savings_Account::Get_Id() const
{
  return id_no;
}

const char* Savings_Account::Get_Name() const
{
  return name;
}
double Savings_Account::Get_Balance()
{
  return balance;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
```

```
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

void Display_Account(Savings_Account&);     //prototip funkcije

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  char id[5];
  char buffer[81];  //Privremeno skladisti ime
  double bal;
  double rt;
  double amount;
  cout << "Enter Account ID: ";
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

  Savings_Account acc1(id, buffer, bal, rt);

  Display_Account(acc1);

  cout << endl << endl;
```

```
  cout << "Enter an amount to deposit: ";
  cin >> amount;

  acc1.Deposit(amount);

  cout << endl << endl;
  cout << "A deposit of " << amount << " was made." << endl;

  Display_Account(acc1);

  acc1.Calc_Interest();

  cout << endl << endl;
  cout << "Interest was applied to the account." << endl;

  Display_Account(acc1);

  cout << endl << endl;
  cout << "Name: " << acc1.Get_Name();

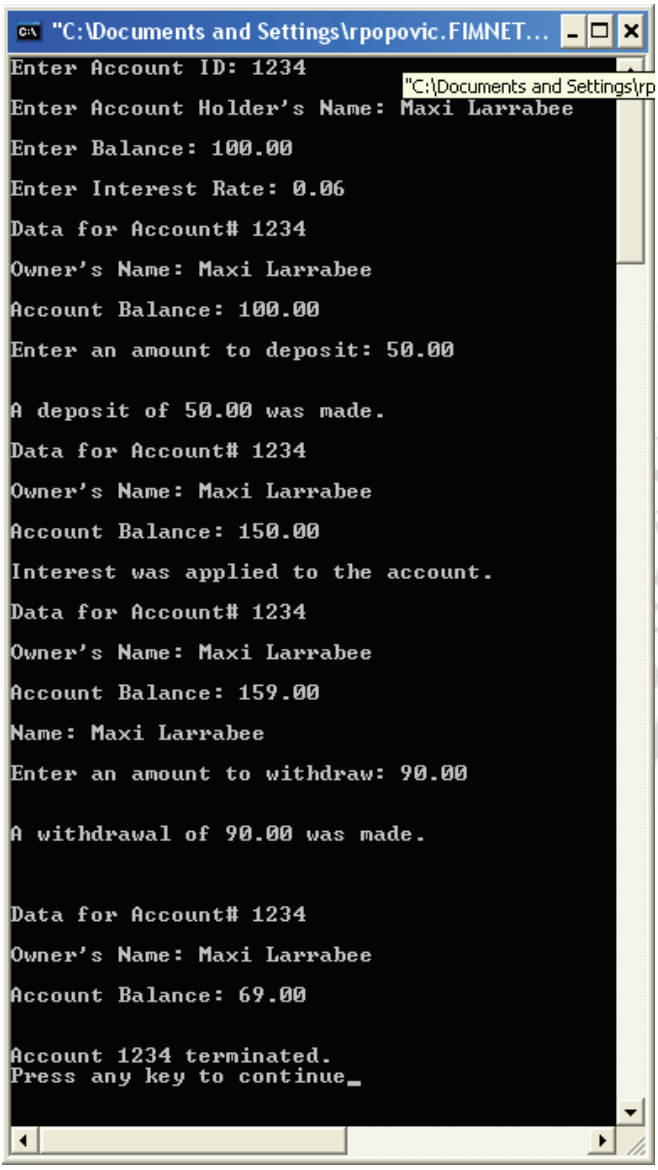
  cout << endl << endl;
  cout << "Enter an amount to withdraw: ";
  cin >> amount;

  if (acc1.Withdraw(amount))
  {
   cout << endl << endl;
   cout << "A withdrawal of " << amount << " was made." << endl;
  }
  else
  {
   cout << endl << endl;
   cout << "WITHDRAWAL NOT MADE: Insufficient funds." << endl;
  }
  cout << endl << endl;
  Display_Account(acc1);

  cout << endl;
  return 0;
}
void Display_Account(Savings_Account& acc)
{
  cout << endl;
  cout << "Data for Account# " << acc.Get_Id() << endl << endl;

  cout << "Owner's Name: " << acc.Get_Name() << endl << endl;
```

```
  cout << "Account Balance: " << acc.Get_Balance();
}
```



9.5 Dinamička alokacija objekata

Podsetimo se da celi brojevi, realni brojevi dvostruke preciznosti i drugi ugrađeni objekti podataka mogu biti kreirani i uništeni dinamički koristeći new i delete operatore. Takođe možemo da koristimo ove operatore za dinamičko kreiranje i uništavanje objekata.

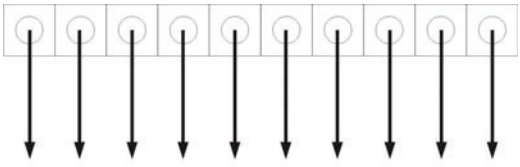
Objekti mogu dinamički da se alociraju koristeći operator new. Na primer:

```
Test_Class* t_p = new Test_Class(5);
Savings_Account* acc_p = new Savings_Account("3456", "Smith", 250.00, 0.062);
```

Prva naredba deklariše Test_Class pokazivač i postavlja ga da je jednak adresi Test_Class objekta sa podatkom članom čija je vrednost 5. Konstruktor sa jednim argumentom se poziva kada se objekat kreira u hip memoriji pomoću new operatora. Druga naredba deklariše Savings_Account pokazivač i postavlja ga da je jednak adresi od Savings_Account objekta sa ID brojem 3456, name - Smith, balance - $250.00, i rate - 0.062. Konstruktor sa 4 argumenta se poziva kada se objekat kreira na hipu koristeći new operator.

Pretpostavimo da želimo da napišemo program koji pita korisnika da kreira do 10 Savings_Account objekata. Jedan način da se reši problem je da se deklariše niz od 10 Savings_Account objekata, od kojih ni jedan nije inicijalizovan sa značajnim podacima. Kada korisnik odluči da kreira račun, program popunjava sledeći iskoristljivi account objekat podacima. Ovaj pristup vredi ali troši memoriju. Svaki od Savings_Account objekata zahteva najmanje 25 bajtova glavne memorije bez hip memorije potrebne za skladištenje imena vlasnika računa. Ako korisnik odluči da kreira samo tri računa, drugih sedam Savings_Account objekata okupiraće 175 bajtova, koji će ostati neiskorišćeni.

Umesto korišćenja niza od 10 Savings_Account objekata, možemo da koristimo niz od 10 pokazivača na Savings_Account objekte. Ovaj pristup koristi samo onoliko Savings_Account objekata koliko je potrebno, izbegavajući gubljenje prostora kao kod prvog pristupa.



Niz od 10 Savings_Account pokazivača

Sledi program koji ilustruje korišćenje niza pokazivača na objekte.

```
//Ovaj program ilustruje koriscenje dinamicki alociranih objekata.

#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   double rate;

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   ~Savings_Account();
   const char* Get_Id() const;
   const char* Get_Name() const;
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);   //kopira prvi argument id_no[]

 name = new char[strlen(n_p) + 1];  //kreira prostor za ime

 strcpy(name, n_p);  //kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;
}

Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
```

```
  delete [] name;
}

const char* Savings_Account::Get_Id() const
{
  return id_no;
}

const char* Savings_Account::Get_Name() const
{
  return name;
}

double Savings_Account::Get_Balance()
{
  return balance;
}

double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;
  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}
```

```
void Display_Account(Savings_Account&);     //prototipovi funkcija
Savings_Account* Create_Account();

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  Savings_Account* account_table[10];

  int count = 0;
  char response;

  cout << endl;
  cout << "Do you want to create an account?(Y/N): ";
  response = cin.get();
  cin.get();            //Brise ulazne podatke

  while (toupper(response) == 'Y' && count < 10)
  {
    account_table[count] = Create_Account();
    ++count;
    cout << endl;
    cout << "Do you want to create an account?(Y/N): ";
    response = cin.get();
    cin.get();          //Brise ulazne podatke
  }
  //Prikaz racuna

  for (int i = 0; i < count; ++i)
    Display_Account(*account_table[i]);

  //Brisanje

  for (i = 0; i < count; ++i)
    delete account_table[i];

  cout << endl;
  return 0;
}

void Display_Account(Savings_Account& acc)
{
  cout << endl << endl << endl;
  cout << "Data for Account# " << acc.Get_Id() << endl << endl;
```

```
  cout << "Owner's Name: " << acc.Get_Name() << endl << endl;

  cout << "Account Balance: " << acc.Get_Balance();
}
Savings_Account* Create_Account()
{
  char id[5];
  char buffer[81];
  double bal;
  double rt;

  Savings_Account* acc_ptr;
  cout << endl;
  cout << "Enter Account ID: ";
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

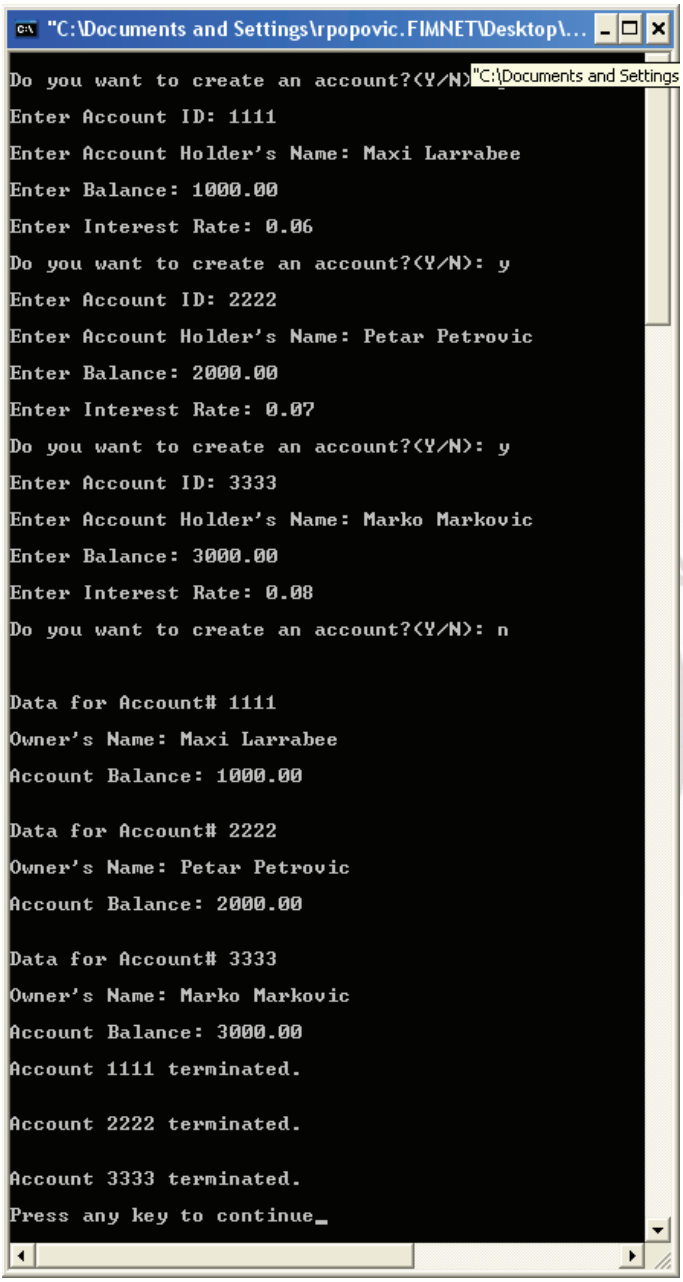
  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

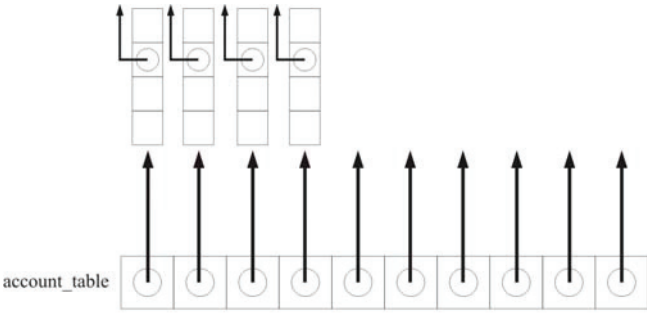
  cin.get();    //Brise newline karakter iz ulaznog buffer-a

  acc_ptr = new Savings_Account (id, buffer, bal, rt);

  return acc_ptr;
}
```



Identifikator account_table[i] je pokazivač na Savings_Account objekat. Funkcija Display_Account() zahteva ime Savings_Account objekta kao argument zato što je njegov argument Savings_Account referenca.



Svaki član account_class je pokazivač na Savings_Account objekat

9.6 Statički podaci članovi i funkcije

Statički podaci članovi

Static data member (statički podatak član) za datu klasu je nezavisna instanca, što znači da bez obzira koliko class objekata je kreirano u programu, postoji samo jedna kopija static data member. Zbog toga, njena vrednost je nezavisna od postojanja bilo kog objekta.

Ako deklarišemo član rate klase Savings_Account da je static, svi Savings_Account objekti će deliti ovu jednu vrednost. Ako promenimo vrednost rate, svi Savings_Account objekti će koristiti ovu novu vrednost automatski.

Kako statički podatak član postoji nezavisno od bilo kog objekta, kada se on kreira tako dobija vrednost. Kao sve static promenljive (bilo da su članovi klase ili ne), static podatak član klase se kreira za vreme kompajliranja, pre izvršenja main() funkcije i prima početnu vrednost nula. Kada deklarišemo podatak član kao static u deklaraciji klase, to je samo deklaracija, ne definicija. (Kod deklaracije klase ne može da se dodeli vrednost class data članu.) Zbog toga mora da definišemo član kao static član i, ako je neophodno, dati početnu vrednost izvan definicije klase. Sledi deklaracija klase Savings_Account sa rate data member koji je deklarisan kao static:

```
class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
```

```
  double balance;
  static double rate;  //rate se deklarise kao staticka promenljiva

 public:
  Savings_Account(char id[], char* n_p, double bal = 0.00);

  ~Savings_Account();

  const char* Get_Id() const;

  const char* Get_Name() const;

  double Calc_Interest();

  double Get_Balance();

  void Deposit(double);

  int Withdraw(double);
};

//Definicija tipa podatka clana klase kao static

double Savings_Account::rate = 0.040;
```

Treba obratiti pažnju da rate još uvek ostaje private. Takođe, kada definišemo rate izvan definicije klase, ne koristimo ključnu reč static ponovo i mora da okvalifikujemo ime imenom klase koristeći operator razrešenja dosega. Konačno, mora takođe da promenimo deklaraciju konstruktora. Više ne mora da se uključuje rate u konstruktor zato što će svi objekti sada da koriste zajednički static rate član.

Statički metodi

Za pristup statičkom podatku članu, treba da se koristi statički metod. Sledi druga deklaracija klase Savings_Account koja uključuje statički podatak član number_accounts, nove definicije konstruktora, destruktora i definicija statičkog metoda Total_Accounts(). Treba obratiti pažnju da se u definiciji Total_Accounts(), ne koristi rezervisana reč static.

```
class Savings_Account
{
 private:
```

```
    char   id_no[5];
    char*  name;
    double balance;
    static double rate;
    static int number_accounts;

  public:
    Savings_Account(char id[], char* n_p, double bal = 0.00);
    ~ Savings_Account();
    const char* Get_Id() const;
    const char* Get_Name() const;
    double Calc_Interest();
    double Get_Balance();
    void Deposit(double);
    bool Withdraw(double);

    static int Total_Accounts();   //static tip metoda
};

// Definicije tipa podatka clana klase kao static
double Savings_Account::rate = 0.040;
int Savings_Account::number_accounts = 0;

Savings_Account:: Savings_Account(char id[], char* n_p, double bal)
{
  strcpy(id_no, id);   //kopira prvi argument u id_no[]

  name = new char[strlen(n_p) + 1];   //kreira prostor za ime

  strcpy(name, n_p);  //kopira drugi argument u novi prostor

  balance = bal;
  ++number_accounts;
}

Savings_Account::~ Savings_Account()
{
  cout << endl << endl;
  cout << "Account " << id_no << " terminated." << endl;
  delete [] name;

  --number_accounts;

  cout << endl;
  cout << "Number Accounts Remaining: " << number_accounts << endl;
}
```

```
int Savings_Account::Total_Accounts()
{
 return number_accounts;
}
```

Statički metod, kao i statički podatak član, je nezavisna instanca. To znači da može da se izvrši nezavisno od bilo kog objekta u klasi. Da biste izvršili takav metod, morate da povežete ime metoda sa imenom klase i operatorom razrešenja dosega, kao u sledećem primeru.

```
cout << "Number Existing Accounts: "
    << Savings_Account::Total_Accounts() << endl;
```

Sledi program koji ilustruje korišćenje statičkih podataka članova i statičkih metoda.

```
//Program ilustruje koriscenje static data clanova i funkcija.
#include <iostream>
#include <iomanip>
#include <string>
#include <cctype>

using namespace std;

class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
   double balance;
   static double rate;          //static clan
   static int number_accounts;   //static clan

 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00);
   ~Savings_Account();
   const char* Get_Id() const;
   const char* Get_Name() const;
   double Calc_Interest();
   double Get_Balance();
   void Deposit(double);
   bool Withdraw(double);
   static int Total_Accounts();   //deklaracija metoda kao static
};
```

```
//Static data member definitions

double Savings_Account::rate = 0.040;
int Savings_Account::number_accounts = 0;

Savings_Account::Savings_Account(char id[], char* n_p, double bal)
{
 strcpy(id_no, id);  //kopira prvi argument u id_no[]

 name = new char[strlen(n_p) + 1];  //kreira prostor za ime

 strcpy(name, n_p);  //kopira drugi argument u novi prostor

 balance = bal;

 ++number_accounts;
}

Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
 delete [] name;

 --number_accounts;

 cout << endl;
 cout << "Number Accounts Remaining: " << number_accounts << endl;
}

const char* Savings_Account::Get_Id() const
{
 return id_no;
}

const char* Savings_Account::Get_Name() const
{
 return name;
}

double Savings_Account::Get_Balance()
{
 return balance;
}

double Savings_Account::Calc_Interest()
```

```
{
  double interest;
  interest = balance * rate;
  balance += interest;

  return interest;
}

void Savings_Account::Deposit(double amount)
{
  balance += amount;
}

bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance -= amount;
    result = true;
  }
  else
    result = false;

  return result;
}

int Savings_Account::Total_Accounts()
{
  return number_accounts;
}

void Display_Account(Savings_Account&);    //prototipovi funkcija
Savings_Account* Create_Account();

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);
  Savings_Account* account_table[10];

  int count = 0;
  char response;
  cout << endl;
```

```
  cout << "Do you want to create an account?(Y/N): ";
  response = cin.get();
  cin.get();          //Prazni ulazni bafer

  while (toupper(response) == 'Y' && count < 10)
  {
   account_table[count] = Create_Account();
   account_table[count] -> Calc_Interest();
   ++count;

   cout << endl;
   cout << "Do you want to create an account?(Y/N): ";
   response = cin.get();
   cin.get();          //Prazni ulazni bafer
}
  //Prikaz racuna
  for (int i = 0; i < count; ++i)
   Display_Account(*account_table[i]);

  //Brisanje

  for (i = 0; i < count; ++i)
   delete account_table[i];

  cout << endl;
  return 0;
}
void Display_Account(Savings_Account& acc)
{
  cout << endl << endl << endl;
  cout << "Data for Account# " << acc.Get_Id() << endl << endl;

  cout << "Owner's Name: " << acc.Get_Name() << endl << endl;

  cout << "Current Account Balance: " << acc.Get_Balance();
}
Savings_Account* Create_Account()
{
  char id[5];
  char buffer[81];
  double bal;

  Savings_Account* acc_ptr;

  cout << endl;
  cout << "Enter Account ID: ";
```

```
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);
  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

  cin.get();    //Brise ulazni bafer sa karakterom za novi red

  acc_ptr = new Savings_Account (id, buffer, bal);

  cout << endl << endl;
  cout << "Number Existing Accounts: "
            << Savings_Account::Total_Accounts() << endl;
  return acc_ptr;
}
```


"C:\Documents and Settings\rpopovic.FIMNET\Deskto...
Do you want to create an account?(Y/N): y
Enter Account ID: 1111
Enter Account Holder's Name: Petar Petrovic
Enter Balance: 100.00
Number Existing Accounts: 1
Do you want to create an account?(Y/N): y
Enter Account ID: 2222
Enter Account Holder's Name: Marko Markovic
Enter Balance: 200.00
Number Existing Accounts: 2
Do you want to create an account?(Y/N): y
Enter Account ID: 3333
Enter Account Holder's Name: Jovan Jovanovic
Enter Balance: 300.00
Number Existing Accounts: 3
Do you want to create an account?(Y/N): n
Data for Account# 1111
Owner's Name: Petar Petrovic
Current Account Balance: 104.00
Data for Account# 2222
Owner's Name: Marko Markovic
Current Account Balance: 208.00
Data for Account# 3333
Owner's Name: Jovan Jovanovic
Current Account Balance: 312.00
Account 1111 terminated.
Number Accounts Remaining: 2
Account 2222 terminated.
Number Accounts Remaining: 1
Account 3333 terminated.
Number Accounts Remaining: 0
Press any key to continue

10. Nasleđivanje

Jedan od najvažnijih koncepata u objektno orijentisanom programiranju je nasleđivanje. Nasleđivanje dozvoljava da definišemo klasu u odnosu na druge klase, što čini lakšim kreiranje i održavanje jedne aplikacije. Počinjemo sa nekoliko primera.

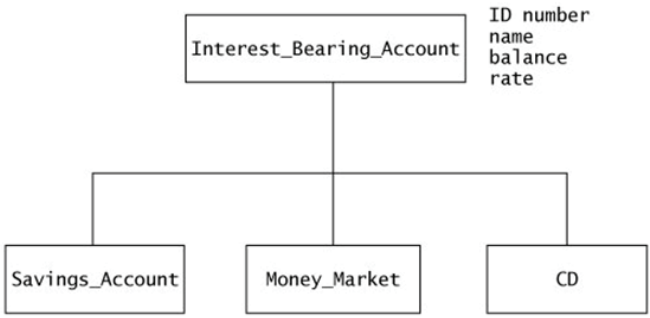
10.1 Primeri nasleđivanja i osnovna terminologija

Do sada smo razmatrali probleme koji sadrže samo jednu klasu. Uopšteno govoreći, problemi iz "realnog života" uključuju više klasa. Moguće je da neke klase u problemu nisu direktno povezane u smislu da ne dele bilo koji član podatak ili metod. Na primer, ako rešavamo problem crtanja geometrijskih slika, možemo da definišemo klase Rectangle i Circle na takav način da one nemaju bilo kakve zajedničke članove podatke ili zajedničke metode. Pravougaonik se opisuje svojom dužinom i širinom, a krug poluprečnikom. Svaka klasa ima svoje sopstvene metode za izračunavanje površine i obima: Area(), Perimeter(), ...

Međutim, postoje klase koje su povezane hijerarhijski od opštih klasa ka specifičnim. Takva hijerarhija primenjuje relaciju "je" u smislu da jedan objekat klase koja je ispod druge klase u hijerarhiji "je" tipa objekat u klasi iznad. Na primer, u bankarskim aplikacijama mogu postojati tri vrste kamatnih računa: savings accounts - štedni računi, money market accounts - računi na tržištu novca i certificates of deposit (CD) - sertifikati o ulaganju. Svaki tip računa "je" baziran na kamati, za razliku od tekućeg računa koji ne zavisi od kamate. Tri tipa računa imaju zajedničke atribute. Na primer, sva tri tipa računa imaju identifikacioni broj ID number, ime vlasnika name, stanje balance, i kamatnu stopu interest rate. Tri tipa računa takođe imaju iste zajedničke metode. Na primer, svi imaju metod za proračun kamate Calc_Interest(). Mada, svi imaju metod Withdraw(), svaka verzija klase za dati metod se ponaša drugačije. Na primer, preuranjeno podizanje novca sa CD-a zahteva da banka koristi zatezne kamate.

Tri tipa računa mogu da definišu kao odvojene klase, svaka sa svojim sopstvenim ID number, name, balance i interest rate, ali svaka je specijalna verzija opšteg tipa računa interest-bearing (račun baziran na kamati). Bolje je program dizajnirati da koristi klase hijerarhijski kao na slici. Veza između klasa u ovakvoj hijerarhiji, gde se klase definišu iz osnovne klase, od opštih ka posebnim klasama, poznata je kao nasleđivanje. Klasa

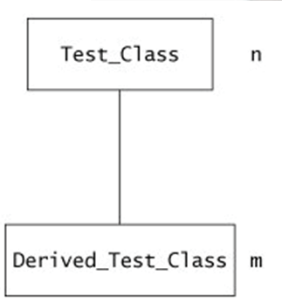
Interest_Bearing_Account je poznata kao osnovna klasa (roditelj klasa ili super klasa). Klase ispod osnovne u hijerarhiji su izvedene klase (dete klase ili podklase). Klase mogu biti direktno i indirektno izvedene.



Kada deklarišemo takvu hijerarhiju klasa u programu, Interest_Bearing_Account će uključiti ID number, name, balance, i rate članove (što je prikazano sa desne strane klase na donjoj slici). Izvedena klasa nasleđuje ove članove iz osnovne klase. Mi deklarišemo u izvedenoj klasi samo one članove koji su specifični za tu klasu.

Definisanje izvedenih klasa

Zbog jednostavnosti, prvo radimo sa klasom Test_Class iz prethodnog poglavlja. Želimo da klasa Test_Class služi kao osnovna klasa za izvedenu klasu Derived_Test_Class, koja takođe ima samo jedan podatak m.



Sledi definicija klase Test_Class iz prethodnog poglavlja. Uklonili smo metod Get_Value() da bi klasa bila što jednostavnija.

```
class Test_Class
{
 private:
   int n;

 public:
   Test_Class(int i = 0);              //Konstruktor sa jednim argumentom
   Test_Class(Test_Class&);            //Konstruktor kopije
   ~Test_Class();                      //Destruktor
};
```

Kao što smo napomenuli u prethodnoj sekciji, bilo koja klasa koju smo izveli iz Test_Class nasleđuje sve primerke varijabli klase Test_Class, uključujući i privatne članove i većinu metoda. Zbog toga, Derived_Test_Class objekat će imati n primerak varijable. Međutim, kako je n privatan član klase Test_Class, nije dostupan bilo kojoj funkciji izvan Test_Class. Sledi da mada Derived_Test_Class objekat ima primerak varijable n u smislu nasleđivanja, objekat nema pristup toj varijabli. Nepraktično je da objekat nema pristup jednom od svojih članova. Ovo možemo da ispravimo deklarisanjem nove kategorije pristupa protected za primerak varijable n klase Test_Class. Član u osnovnoj klasi, bez obzira da li je instanca promenljive ili metoda, koji ima kategoriju pristupa protected ponaša se kao da je private član za sve funkcije osim za metode iz klasa izvedenih iz osnovne klase. Metodi u izvedenoj klasi imaju pristup svim protected članovima u osnovnoj klasi.

Specifikator pristupa protected

Zaštićen primerak u osnovnoj klasi se ponaša kao da je privatan (private) za sve funkcije, osim za metode u klasama koje su izvedene iz osnovne klase.

Metodi u izvedenoj klasi imaju pristup svim zaštićenim (protected) članovima u osnovnoj klasi. Tabela prikazuje pristupe osnovnoj i izvedenim klasama za članove klase koji su public, protected i private.

	Ako je član osnovne klase		
	public	**protected**	**private**
Pristup preko metoda izvedene klase	DA	DA	NE
Pristup preko metoda koji ne pripada ni osnovnoj ni izvedenoj klasi	DA	NE	NE

Prema tome, ako želimo da metodi klase Derived_Test_Class imaju pristup primerku varijable n klase Test_Class, mora da promenimo kategoriju pristupa za n na protected.

```
class Test_Class
{
 protected:    //Podaci kojima se može pristupiti u izvedenim klasama
   int n;
 public:
   Test_Class(int i = 0);  //Konstruktor sa jednim argumentom
   Test_Class(Test_Class&);  //Konstruktor kopije
   ~Test_Class();
};
```

Kako da napišemo kod koji prikazuje činjenicu da je Derived_Test_Class izvedena iz Test_Class? Da biste definisali Derived_Test_Class kao izvedenu iz Test_Class, prvo pišemo ime izvedene klase, pa dvotačku, pa rezervisanu reč public i ime osnovne klase.

```
class Derived_Test_Class : public Test_Class
{
 protected:
     int m;

 public:
     Derived_Test_Class(int j = 0);
     Derived_Test_Class(Derived_Test_Class&);
     ~Derived_Test_Class();
};
```

U ovom slučaju ključna reč public znači da smo javno izveli ovu klasu iz osnovne klase. Javno izvođenje znači da se protected i public članovi osnovne klase posmatraju kao protected i public članovi izvedene klase. *Default* tip izvođenja je private, a ne public. Sledi, ako zanemarimo rezervisanu reč public, izvedena klasa biće privatno izvedena iz osnovne klase. Dobra je praksa da uključujemo reč public ili private a da ne koristimo *default*.

Koristimo public izvođenje ako želimo da korisnici izvedene klase imaju pristup public metodima osnovne klase. Koristimo private izvođenje, ako želimo da blokiramo korisnike izvedene klase od korišćenja public metoda osnovne klase. Da biste izveli klasu iz osnovne klase, u deklaraciji izvedene klase, nakon imena izvedene klase sledi zapeta pa onda ključna reč

public ili private, i ime osnovne klase. Tabela prikazuje mogućnost pristupa pri public i private izvođenju.

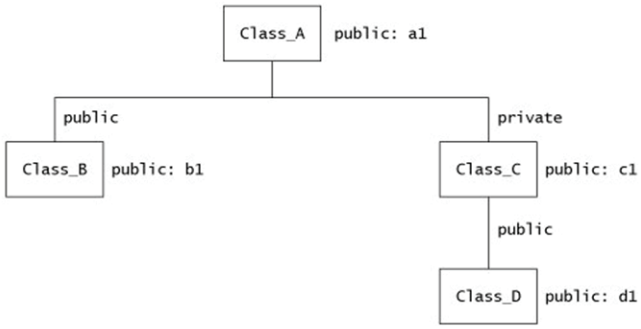
Kategorija pristupa članu osnovne klase	Javno izvedena klasa osnovne klase - Pristup članovima kod javnog nasleđivanja	Privatno izvedena klasa osnovne klase - Pristup članovima kod privatnog nasleđivanja
private	private	private
protected	protected	private
public	public	private

U srednjoj koloni (javno izvedena klasa), primerak promenljive ili metoda ima istu kategoriju pristupa kao i u osnovnoj klasi. U privatno izvedenoj klasi (desna kolona), svi primerci promenljivih i funkcija su private, bez obzira koja im je kategorija pristupa u osnovnoj klasi.

Na primer, posmatrajmo hijerarhiju klase sa donje slike. Promenljiva a1 je public primerak promenljive osnovne klase Class_A. Nasleđivanjem, primerak promenljive a1 je takođe primerak promenljive Class_B.

Šta je kategorija pristupa primerka promenljive a1 kao člana Class_B?

Prema prethodnoj tabeli, zato što je Class_B javno izvedena iz Class_A, promenljiva a1 je takođe public promenljiva klase Class_B. S druge strane, promenljiva a1, nasleđivanjem, je takođe primerak promenljive Class_C. Međutim, iz trećeg reda tabele sledi da je a1 private član Class_C zato što je Class_C privatno izvedena iz Class_A. Dva primerka varijable Class_C, c1 (koja je public u Class_C) i izvedena a1 (koja je private u Class_C) su nasleđene u Class_D. Zbog toga, c1 je public član i a1 je private član Class_D zato što je Class_D javno izvedena iz Class_C. Promenljivoj a1, kao privatnom članu Class_C, ne može se pristupiti metodima Class_D.

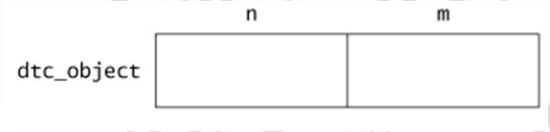


Treba obratiti pažnju da smo primerak promenljive m izvedene klase Derived_Test_Class definisali kao protected. Ovo omogućava da, ako smo izveli klasu iz Derived_Test_Class, članu m moći će da se pristupi kao protected članu svih javno izvedenih klasa.

Konstruktori u izvedenoj klasi

Izvedene klase nasleđuju sve metode osnovne klase osim:
-konstruktora i konstruktora kopije
-destruktora
-preklopljenih operatora
-prijateljskih funkcija
Sledi, mora da obezbedimo izvedene konstruktore, konstruktore kopija i destruktore date klase. Oni se ne nasleđuju.

Kada kreiramo objekat dtc_object, klase Derived_Test_Class, objekat ima dva primerka promenljivih. Ima n, član nasleđen iz Test_Class i m, član iz Derived_Test_Class.



Pretpostavimo da želimo da sledeća deklaracija kreira Derived_Test_Class objekat sa m primerkom promenljive čija je vrednost 7 i n primerkom promenljive sa vrednošću 8.

```
Derived_Test_Class dtc_object1(7);
```

Kada kreiramo objekat klase Derived_Test_Class, takođe kreiramo objekat klase Test_Class. Ovo je zato što Derived_Test_Class objekat "je" Test_Class objekat. Konstruktor za Derived_Test_Class se takođe izvršava. U stvari, konstruktor za Test_Class se izvršava pre konstruktora za Derived_Test_Class zato što objekat mora da dobije svoj identitet kao Test_Class objekat pre nego što dobije svoj identitet kao Derived_Test_Class objekat. Definicija konstruktora za Derived_Test_Class reflektuje ovo.

```
Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1)
{
  m = j;
```

```
  cout << endl;
  cout << "Derived_Test_Class Constructor Executed: "
     << "Data member initialized to "
     << m << endl;
}
```

Nakon zagrade zatvaranja u hederu definicije konstruktora je dvotačka nakon koje sledi eksplicitni poziv konstruktora sa jednim argumentom osnovne klase. Ovo se zove inicijalizaciona lista. Primerci na ovoj listi konstruktora se izvršavaju kako se objekat kreira. Tako, akt kreiranja Derived_Test_Class objekta sa početnom vrednošću j prouzrokuje da se konstruktor sa jednim argumentom osnovne klase izvršava sa početnom vrednošću od j+1. Ovo će prouzrokovati kreiranje Test_Class objekta sa n primerkom promenljive koja ima vrednost j+1. Ovo se dešava pre izvršenja tela Derived_Test_Class konstruktora.

Konstruktor za izvedenu klasu treba eksplicitno da pozove konstruktor osnovne klase. Poziv konstruktora osnovne klase je u inicijalizacionoj listi.

Inicijalizaciona lista nije ograničena na poziv konstruktora osnovne klase. Možemo je koristiti u bilo kom konstruktoru za inicijalizaciju podataka članova. Na primer, daćemo definiciju Test_Class konstruktora koji smo koristili ranije:

```
Test_Class::Test_Class(int i)
{
  n = i;

  cout << endl;
  cout << "Test_Class Constructor Executed: Data member initialized to "
       << n << endl;
}
```

Kada deklarišemo Test_Class objekat, objekat je kreiran sa jednim primerkom promenljive n koja ima nedefinisanu vrednost. Kada se konstruktor izvršava, naredba dodele u konstruktor smešta vrednost i (ili *default* vrednost 0) u n, kada n nema validnu vrednost. Sledeća definicija Test_Class konstruktora koristi inicijalizacionu listu da bi smestila vrednost i u n.

```
Test_Class::Test_Class(int i) : n(i)
{
```

```
  cout << endl;
  cout << "Test_Class Constructor Executed: Data member initialized to "
       << n << endl;
}
```

Napomena, operator dodele se ne koristi na inicijalizacionoj listi. Izraz n(i) inicijalizuje vrednost n na i. Sa takvom inicijalizacionom listom kada se objekat Test_Class kreira, primerak varijable n se kreira sa vrednošću i. Sledi, primerak varijable n uvek ima validnu vrednost. Ne postoji vreme za koje n ima nedefinisanu vrednost. Kako je ovo bezbednije i efikasnije za inicijalizaciju podatka člana u konstruktoru, uradićemo ovo u sledećim programima.

Ako primenimo ovo na Derived_Test_Class konstruktor, imaćemo:

```
Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1), m(j)
{
  cout << endl;
  cout << "Derived_Test_Class Constructor Executed: "
       << "Data member initialized to "
       << m << endl;
}
```

Inicijalizaciona lista za konstruktor sadrži dve naredbe odvojene zapetom. Prva naredba je eksplicitni poziv konstruktora osnovne klase, a druga je inicijalizacija m podatka člana na vrednost j.

Destruktor u izvedenoj klasi

Kada se jedan objekat izvedene klase (Derived_Test_Class) uništava, takođe uništavamo objekat osnovne klase (Test_Class) zato što, na osnovu nasleđivanja, objekat izvedene klase je takođe i objekat osnovne klase (Test_Class). Kada se destruktor za izvedenu klasu izvršava, destruktor za osnovnu klasu takođe treba da se izvrši. Međutim, destruktor za izvedenu klasu treba da se izvrši pre destruktora za osnovnu klasu. Za razliku od toga šta mora da uradimo za konstruktore, ne treba da uradimo ništa specijalno u kodu destruktora da bi sistem pozvao destruktor osnovne klase. C++ sistem radi ovo automatski, kao u sledećem primeru:

```
//Ovaj program ilustruje kako se konstruktori i destruktori izvršavaju
//u osnovnoj klasi.
```

```
#include <iostream>

using namespace std;

class Test_Class
{
 protected:
   int n;
 public:
   Test_Class(int i = 0); //Konstruktor sa jednim argumentom
   ~Test_Class();
};

Test_Class::Test_Class(int i) : n(i)

{
 cout << endl;
 cout << "Test_Class Constructor Executed: Data member initialized to "
      << n << endl;
}

Test_Class::~Test_Class()

{
 cout << endl;
 cout << "Test_Class Destructor Executed for object with data member "
      << n << endl;
}

class Derived_Test_Class : public Test_Class
{
 protected:
   int m;

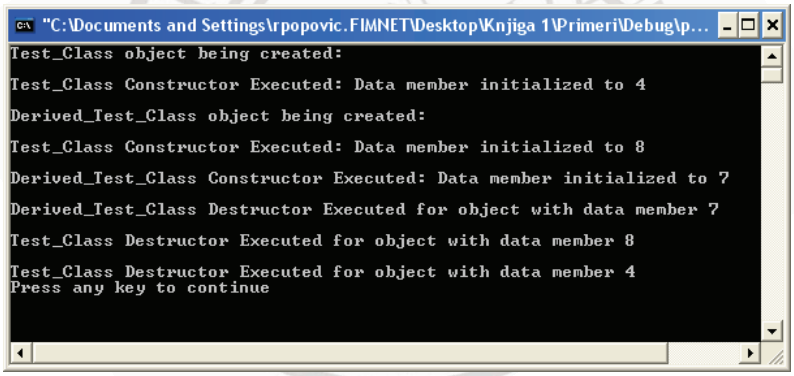
 public:
   Derived_Test_Class(int j = 0);
   ~Derived_Test_Class();
};

Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1), m(j)
{
 cout << endl;
 cout << "Derived_Test_Class Constructor Executed: "
      << "Data member initialized to "
      << m <<endl;
}
```

```
Derived_Test_Class::~Derived_Test_Class()
{
 cout << endl;
 cout << "Derived_Test_Class Destructor Executed "
      << "for object with data member "
      << m << endl;
}

int main()
{
 cout << "Test_Class object being created:" << endl;
 Test_Class tc_object1(4);
 cout << endl;
 cout << "Derived_Test_Class object being created:" << endl;
 Derived_Test_Class dtc_object1(7);

 return 0;
}
```



Treba obratiti pažnju na redosled izvršavanja konstruktora. Kada se kreira objekat tc_object1 klase Test_Class, konstruktor osnovne klase Test_Class se izvršava. Kada se kreira Derived_Test_Class objekat dtc_object1, Derived_Test_Class konstruktor se poziva. Međutim, pre nego što se bilo koja njegova naredba izvrši, poziva se konstruktor osnovne klase, Test_Class(). Nakon izvršenja i prikaza poruke, Test_Class konstruktor se završava. U toj tački telo konstruktora za Derived_Test_Class se izvršava i prikazuje svoju poruku.

Kada se program završava, objekti se uništavaju obrnutim redosledom u odnosu na kreiranje. Zbog toga, dtc_object1 se prvo uništava. Destruktor za Derived_Test_Class se izvršava, a zatim se destruktor za

Test_Class izvršava. Konačno, Test_Class objekat tc_object1 se uništava, Test_Class destruktor za ovaj objekat se izvršava.

Konstruktor kopije u izvedenoj klasi

Ako izvedena klasa ima konstruktor kopije, on treba da bude tretiran na isti način kao običan konstruktor. Zbog toga, definicija konstruktora kopije izvedene klase treba da pozove konstruktor kopije osnovne klase na inicijalizacionoj listi. Sledi primer:

```
//Ovaj program ilustruje konstruktor kopije u izvedenoj klasi.

#include <iostream>
using namespace std;

class Test_Class
{
 protected:
   int n;
 public:
   Test_Class(int i = 0);  //Konstruktor sa jednim argumentom
   Test_Class(const Test_Class &);  //Konstruktor kopije
   ~Test_Class();
};

Test_Class::Test_Class(int i) : n(i)
{
 cout << endl;
 cout << "Test_Class Constructor Executed: Data member initialized to "
      << n << endl;
}

Test_Class::Test_Class(const Test_Class & tc_r) : n(tc_r.n)
{
 cout << endl;
 cout << "Test_Class Copy constructor executed: "
      << "Data member initialized to " << n << endl;
}

Test_Class::~Test_Class()
{
 cout << endl;
 cout << "Test_Class Destructor Executed for object with data member "
      << n << endl;
}
```

```
class Derived_Test_Class : public Test_Class
{
 protected:
   int m;

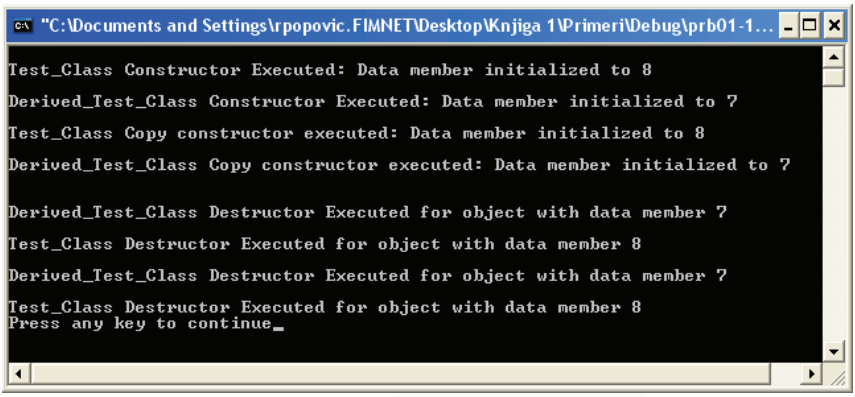
 public:
   Derived_Test_Class(int j = 0);
   Derived_Test_Class(const Derived_Test_Class &);
   ~Derived_Test_Class();
};

Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1), m(j)
{
 cout << endl;
 cout << "Derived_Test_Class Constructor Executed: "
      << "Data member initialized to "
      << m << endl;
}

Derived_Test_Class::Derived_Test_Class(const Derived_Test_Class & dtc_r)
                : Test_Class(dtc_r), m(dtc_r.m)
{
 cout << endl;
 cout << "Derived_Test_Class Copy constructor executed: "
      << "Data member initialized to " << m << endl;
}

Derived_Test_Class::~Derived_Test_Class()
{
 cout << endl;
 cout << "Derived_Test_Class Destructor Executed "
      << "for object with data member "
      << m << endl;
}

int main()
{
 Derived_Test_Class dtc_object1(7);
 Derived_Test_Class dtc_object2 = dtc_object1;
 cout << endl;
 return 0;
}
```



Kôd za konstruktor kopije sličan je kodu za konstruktor. Treba obratiti pažnju da u definiciji konstruktora kopije za Test_Class, koristimo izraz n(tc_r.n) u inicijalizacionoj listi. Ovo inicijalizuje n primerak promenljive objekta koja je inicijalizovana na vrednost n primerka promenljive objekta koji je na desnoj strani inicijalizacije. Definicija konstruktora kopije za Derived_Test_Class poziva konstruktor kopije, Test_Class(dtc_r), na svojoj inicijalizacionoj listi i inicijalizuje m primerak varijable objekta koji je bio incijalizovan na vrednost m primerka promenljive na desnoj strani inicijalizacije, m(dtc_r.m).

Važan je redosled izvršenja konstruktora i destruktora. Program prvo deklariše Derived_Test_Class objekat dtc_object1. Zatim se konstruktor za Test_Class izvršava. Nakon toga izvršava se konstruktor za Derived_Test_Class. Sledeće, program deklariše i inicijalizuje Derived_Test_Class objekat dtc_object2. Kako je objekat inicijalizovan, konstruktor kopije se koristi. Kao i kod običnog konstruktora, konstruktor kopije osnovne klase se izvršava prvo. Odnosno, konstruktor kopije za Test_Class, nakon čega sledi izvršenje konstruktora kopije za Derived_Test_Class.

Kada se program završi, destruktori se pozivaju po obrnutom redosledu od kreiranja objekata. Prvo se izvršava destruktor za dtc_object2, zatim se automatski izvršava destruktor za osnovnu klasu Test_Class.

Funkcije u hijerarhiji klasa

Sve funkcije se nasleđuju osim konstruktora, konstruktora kopije, destruktora, preklopljenih operatora i friend funkcija. Ova sekcija opisuje rad sa funkcijama koje se nasleđuju.

Nasleđivanje funkcija

Pretpostavimo da osnovna klasa Test_Class sadrži sledeći metod Get_Value().

```
int Test_Class::Get_Value()
{
  return n;
}
```

Metod Get_Value() nasleđuje izvedena klasa Derived_Test_Class. Sledi, možemo da primenimo Get_Value() na bilo koji objekat izvedene klase Derived_Test_Class kao što sledi:

```
Derived_Test_Class dtc_object(7);

cout << endl << endl;

cout << "The value returned by Get_Value() is "

    << dtc_object.Get_Value() << endl;
```

Kada se metod primeni na objekat odgovarajuće klase, kompajler proverava da vidi da li je metod, metod te klase. Ako je metod član klase, kompajler primenjuje metod na objekat. Ako nije, kompajler proverava da vidi da li je metod član osnovne klase (ako takav postoji) za tu klasu, ako jeste, onda on primenjuje taj metod na objekat, itd. U našem slučaju, kompajler vidi da metod nije član Derived_Test_Class i gleda u osnovnu klasu Test_Class. Zato što je Get_Value() metod osnovne klase Test_Class, kompajler primenjuje metod osnovne klase Get_Value() na dtc_object. Ima smisla uraditi ovo zato što, u značenju nasleđivanja, dtc_object je takođe Test_Class objekat. Objekat dtc_object ima m primerak promenljive vrednosti 7 i n primerak promenljive vrednosti 8. Tako da će prethodna cout naredba da prikaže:

```
The value returned by Get_Value() is 8
```

Ovo je prikazano u sledećem programu.

```
//Ovaj program ilustruje kako se funkcija nasledjuje.

#include <iostream>
```

```
using namespace std;

class Test_Class
{
 protected:
   int n;

 public:
   Test_Class(int i = 0);  //Konstruktor sa jednim argumentom
   ~Test_Class();
   int Get_Value();
};

Test_Class::Test_Class(int i) : n(i)
{
 cout << endl;
 cout << "Test_Class Constructor Executed: Data member initialized to "
      << n << endl;
}

Test_Class::~Test_Class()
{
 cout << endl;
 cout << "Test_Class Destructor Executed for object with data member "
      << n << endl;
}

int Test_Class::Get_Value()
{
 return n;
}

class Derived_Test_Class : public Test_Class
{
 protected:
   int m;

 public:
   Derived_Test_Class(int j = 0);
   ~Derived_Test_Class();
};

Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1), m(j)
{
 cout << endl;
 cout << "Derived_Test_Class Constructor Executed: "
```

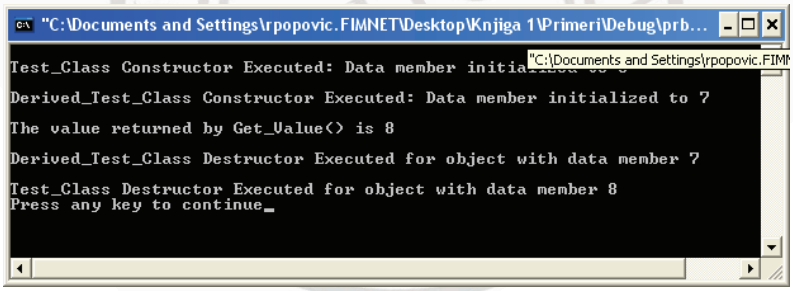
```
     << "Data member initialized to " << m << endl;
}

Derived_Test_Class::~Derived_Test_Class()
{
 cout << endl;
 cout << "Derived_Test_Class Destructor Executed "
     << "for object with data member " << m << endl;
}

int main()
{
 Derived_Test_Class dtc_object(7);

 cout << endl;
 cout << "The value returned by Get_Value() is "
     << dtc_object.Get_Value();

 cout << endl;
return 0;
}
```



Overriding (nadjačavanje) metoda

Šta se dešava ako i izvedena klasa Derived_Test_Class i osnovna klasa Test_Class imaju svoj sopstveni Get_Value() metod? Posmatrajmo sledeći program u kome je definisan metod Get_Value() i u osnovnoj klasi Test_Class i u izvedenoj klasi Derived_Test_Class.

```
//Ovaj program ilustruje nadjacavanje metoda - method overriding.

#include <iostream>

using namespace std;
```

```
class Test_Class
{
 protected:
   int n;

 public:
   Test_Class(int i = 0);  //Konstruktor sa jednim argumentom
   Test_Class(Test_Class &);  //Konstruktor kopije
   ~Test_Class();
   int Get_Value();
};

Test_Class::Test_Class(int i) : n(i)
{
 cout << endl;
 cout << "Test_Class Constructor Executed: Data member initialized to "
      << n << endl;
}

Test_Class::Test_Class(Test_Class & tc_r) : n(tc_r.n)
{
 cout << endl;
 cout << "Test_Class Copy constructor executed: "
      << "Data member initialized to " << n << endl;
}

Test_Class::~Test_Class()
{
 cout << endl;
 cout << "Test_Class Destructor Executed for object with data member "
      << n << endl;
}

int Test_Class::Get_Value()
{
 return n;
}

class Derived_Test_Class : public Test_Class
{
 protected:
   int m;

 public:
   Derived_Test_Class(int j = 0);
   Derived_Test_Class(Derived_Test_Class &);
```

```
   ~Derived_Test_Class();
   int Get_Value();
};

Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1), m(j)
{
 cout << endl;
 cout << "Derived_Test_Class Constructor Executed: "
      << "Data member initialized to " << m << endl;
}

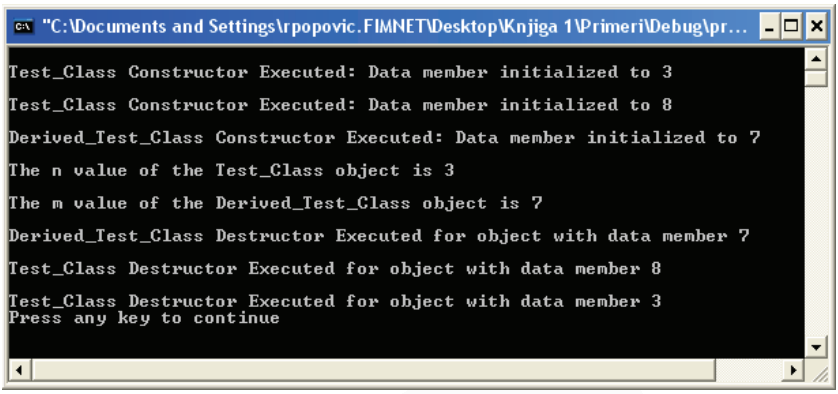
Derived_Test_Class::Derived_Test_Class(Derived_Test_Class & dtc_r)
                          : Test_Class(dtc_r), m(dtc_r.m)
{
 cout << endl;
 cout << "Derived_Test_Class Copy constructor executed: "
      << "Data member initialized to " << m << endl;
}

Derived_Test_Class::~Derived_Test_Class()
{
 cout << endl;
 cout << "Derived_Test_Class Destructor Executed "
      << "for object with data member " << m << endl;
}

int Derived_Test_Class::Get_Value()
{
 return m;
}

int main()
{
 Test_Class tc_object(3);
 Derived_Test_Class dtc_object(7);

 cout << endl;
 cout << "The n value of the Test_Class object is "
      << tc_object.Get_Value() << endl <<endl;
 cout << "The m value of the Derived_Test_Class object is "
      << dtc_object.Get_Value();
 cout << endl;
 return 0;
}
```



Primenjujemo Get_Value() na dva objekta u različitim klasama. C++ kompajler zna koji Get_Value() metod da koristi po objektu na koji je metod primenjen. U prvom slučaju, Get_Value() se primenjuje na Test_Class objekat. Iz tog razloga, Get_Value() metod iz Test_Class se koristi. U drugom slučaju, Get_Value() se primenjuje na Derived_Test_Class objekat. Zato, Get_Value() metod iz Derived_Test_Class se koristi. Ovo je method *overriding*; kada se metod primenjuje na objekat u izvedenoj klasi, metod u izvedenoj klasi overrides („nadjača-pokrije“) metod sa istim imenom i potpisom (tj. isti broj i tipovi argumenata) u osnovnoj klasi.

Ako su imena metoda ista i potpisi različiti, *overriding* se ne izvršava. Na primer, pretpostavimo da je metod Get_Value() izvedene klase Derive_Test_Class definisan kao što sledi:

```
int Get_Value(int i)
{
 return m + i;
}
```

Onda, poziv metoda dtc_object.Get_Value() koristi Get_Value() metod od Test_Class, a poziv metoda dtc_object.Get_Value(4) koristi Get_Value() metod od Derived_Test_Class.

Korišćenje operatora razrešenja dosega

Pretpostavimo opet da Test_Class i Derived_Test_Class imaju svoje sopstvene metode Get_Value() i da želimo da primenimo Get_Value() metod osnovne klase na Derived_Test_Class objekat da bi pristupili njenom n primerku promenljive.

Da biste primenili Get_Value() metod osnovne klase na objekat u izvedenoj klasi, mora da kvalifikujete ime metoda sa imenom klase nakon koje sledi operator razrešenja dosega "::". Tako, izraz

```
dtc_object.Test_Class::Get_Value()
```

primenjuje metod Get_Value() osnovne klase Test_Class na dtc_object. Operator razrešenja dosega može takođe da bude korišćen za pristup globalnoj varijabli kada postoji lokalna varijabla sa istim imenom.

Sledeći program prikazuje korišćenje operatora razrešenja dosega.

```
//Ovaj program ilustruje operator razresenja dosega

#include <iostream>

using namespace std;

class Test_Class
{
 protected:
   int n;

 public:
   Test_Class(int i = 0);  //Konstruktor sa jednim argumentomr
   Test_Class(const Test_Class &);  //Konstruktor kopije
   ~Test_Class();
   int Get_Value();
};

Test_Class::Test_Class(int i) : n(i)
{
 cout << endl;
 cout << "Test_Class Constructor Executed: Data member initialized to "
      << n << endl;
}

Test_Class::Test_Class(const Test_Class & tc_r) : n(tc_r.n)
{
 cout << endl;
 cout << "Test_Class Copy constructor executed: "
      << "Data member initialized to " << n << endl;
}

Test_Class::~Test_Class()
{
```

```
  cout << endl;
  cout << "Test_Class Destructor Executed for object with data member "
       << n << endl;
}

int Test_Class::Get_Value()
{
  return n;
}

class Derived_Test_Class : public Test_Class
{
  protected:
    int m;

  public:
    Derived_Test_Class(int j = 0);
    Derived_Test_Class(const Derived_Test_Class &);
    ~Derived_Test_Class();
    int Get_Value();
};

Derived_Test_Class::Derived_Test_Class(int j) : Test_Class(j+1), m(j)
{
  cout << endl;
  cout << "Derived_Test_Class Constructor Executed: "
       << "Data member initialized to " << m << endl;
}

Derived_Test_Class::Derived_Test_Class(const Derived_Test_Class & dtc_r)
                :Test_Class(dtc_r), m(dtc_r.m)
{
  cout << endl;
  cout << "Derived_Test_Class Copy constructor executed: "
       << "Data member initialized to " << m << endl;
}

Derived_Test_Class::~Derived_Test_Class()
{
  cout << endl;
  cout << "Derived_Test_Class Destructor Executed "
       << "for object with data member " << m << endl;
}

int Derived_Test_Class::Get_Value()
{
```

```
 return m;
}

int main()
{
 Test_Class tc_object(3);
 Derived_Test_Class dtc_object(7);

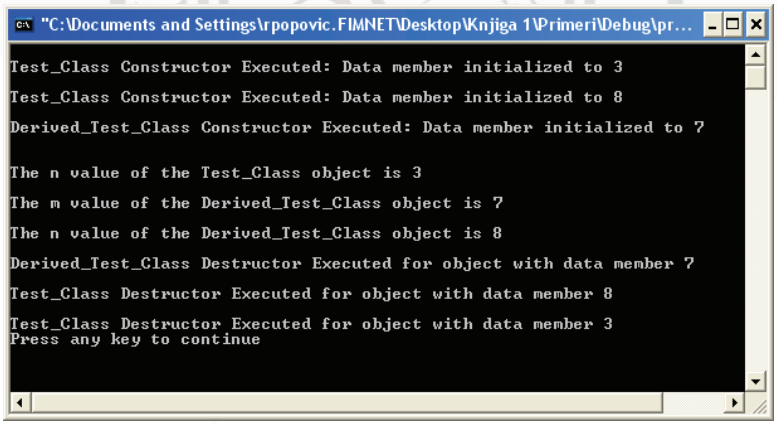
 cout << endl << endl;
 cout << "The n value of the Test_Class object is "
     << tc_object.Get_Value() << endl << endl;

 cout << "The m value of the Derived_Test_Class object is "
     << dtc_object.Get_Value() << endl << endl;

 cout << "The n value of the Derived_Test_Class object is "
     << dtc_object.Test_Class::Get_Value();

 cout << endl;

 return 0;
}
```



Prethodno definisani rezultati se mogu predstaviti na sledeći način:

-Metod osnovne klase nasleđuju sve klase izvedene iz nje ili direktno ili indirektno.

-Ako izvedena klasa sadrži metod sa istim imenom i karakteristikama kao metod osnovne klase, verzija izvedene klase metoda „nadjačava“ verziju osnovne klase.

-Da biste primenili metod osnovne klase na objekat izvedene

klase, kvalifikujte ime metoda sa imenom osnovne klase za kojim sledi operator razrešenja dosega :: .

10.2 Polimorfizam

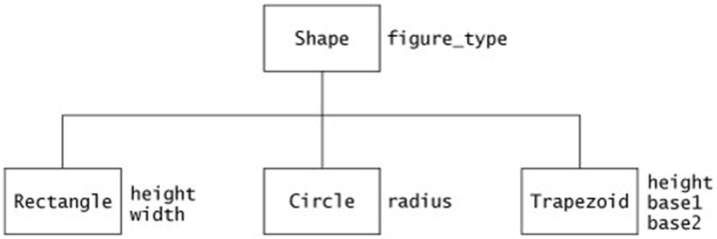
U hijerarhiji klasa, više klasa mogu da sadrže svoje sopstvene verzije metoda. Kada se koriste pokazivači za pristup objektima, polimorfizam dozvoljava da tip podatka objekta odredi koji je od ovih metoda primenjen.

Svojstvo da svaki objekat izvedene klase, čak i kada mu se pristupa kao objektu osnovne klase, izvršava metod tačno onako kako je to definisano u njegovoj izvedenoj klasi, naziva se polimorfizam.

Hijerarhija klasa

Pretpostavimo da razvijamo aplikaciju za crtanje koja treba da definiše i radi sa prostim geometrijskim slikama, odnosno da izračuna njihove površine. Slike su pravougaonici, krugovi i trapezoidi. Pošto su to geometrijski oblici, izvodimo ih iz klase Shape. Sa desne strane svake klase su privatne instance promenljivih date klase. Svaka klasa će imati konstruktor, destruktor i metod Area() koji izračunava površinu prema datoj formuli.

Koristićemo veoma prostu hijerarhiju klasa baziranu na familiji geometrijskih slika i njihovih površina. Pretpostavimo da razvijamo aplikaciju za crtanje koja treba da definiše i manipuliše sa prostim geometrijskim slikama i izračuna njihove površine. Slike su pravougaonici, krugovi i trapezi. Kako su ove slike geometrijski oblici (*shapes*), odlučujemo da ih izvedemo iz klase Shape. Hijerarhija klasa je prikazana na donjoj slici. Sa desne strane svake klase na slici su private primerci promenljivih klase. Svaka klasa ima konstruktor, destruktor i metod Area() koji izračunava površinu slike koristeći odgovarajuću formulu.



Problem: pretpostavimo da želimo da omogućimo korisniku da kreira seriju geometrijskih slika prema sopstvenom izboru. Na taj način, korisnik može prvo da odluči da kreira pravougaonik određenih dimenzija, zatim krug određenog poluprečnika itd. Nakon kreiranja geometrijskih slika, želimo da program prikaže površine svih geometrijskih slika. Kako da uradimo ovo? Možemo da kreiramo niz geometrijskih slika. Međutim svi elementi niza mora da budu istog tipa. Ne možemo da smestimo objekat pravougaonik i objekat krug u isti niz, zato što su različitog tipa. Odgovor je, kao što ćemo videti, u korišćenju niza pokazivača Shape i primena koncepta polimorfizma.

Definicija klasa

U nastavku su date deklaracije klasa Shape, Rectangle, Circle i Trapezoid. Zbog jednostavnosti, definišimo sve metode u deklaracijama klasa. Ne treba vam prefiks imena klase i operatora razrešenja dosega za ime metoda u definiciji metoda. Kako je metod definisan unutar deklaracije klase kompajler zna kojoj klasi metod pripada.

```
class Shape
{
 protected:
   string figure_type;
 public:
   Shape(string ft = "Default Figure") : figure_type(ft)
         {cout << endl << "Shape Constructor" << endl;}
   string Get_Type() {return figure_type;}
   ~Shape() {cout << endl << "Shape Destructor" << endl;}
   double Area() {return 0.0;}
};

class Rectangle : public Shape
{
 protected:
   double height, width;
 public:
   Rectangle(double h, double w): Shape("Rectangle"), height(h),width(w)
         {cout << endl << "Rectangle Constructor" << endl;}
   ~Rectangle() {cout << endl << "Rectangle Destructor" << endl;}
   double Area() {return height * width;}
};
```

```
class Circle : public Shape
{
 protected:
   double radius;
 public:
   Circle(double r) : Shape("Circle"), radius(r)
         {cout << endl << "Circle Constructor" << endl;}
   ~Circle() {cout << endl << "Circle Destructor" << endl;}
   double Area() {return 3.1416 * radius * radius;}
};

class Trapezoid : public Shape
{
 protected:
   double base1, base2, height;
 public:
   Trapezoid(double b1, double b2, double h)
          : Shape("Trapezoid"), base1(b1), base2(b2), height(h)
        {cout << endl << "Trapezoid Constructor" << endl;}
   ~Trapezoid() {cout << endl << "Trapezoid Destructor" << endl;}
   double Area() {return 0.5 * height * (base1 + base2);}
};
```

Osnovna klasa Shape ima jedan podatak član, string objekat figure_type, čija je vrednost ime tipa geometrijske slike (Rectangle, Circle, ili Trapezoid). Takođe uključujemo accessor metod Get_Type() koji vraća ime geomtrijske slike. Oba metoda figure_type i Get_Type() nasleđuju klase Rectangle, Circle, i Trapezoid. Konstruktori izvršavaju inicijalizaciju na svojim inicijalizacionim listama. Telo svakog konstruktora i destruktora prikazuje identifikacionu poruku.

Obratite pažnju da na inicijalizacionoj listi za konstruktor Shape je figure_type(ft). Ulaz na inicijalizacionu listu je ekvivalentan inicijalizaciji. Sledi, figure_type(ft) je ekvivalentno sledećem:

```
string figure_type = ft;
```

Definišimo Area() metod u Shape class-i koja vraća vrednost 0. Svaka od drugih klasa je izvedena javno iz klase Shape. Konstruktor za Rectangle, Circle, i Trapezoid poziva konstruktor za osnovnu klasu Shape i prosleđuje tom konstruktoru ime objekta koji se kreira. Area() metod za svaku klasu vraća površinu na osnovu matematičke formule za površinu geometrijske slike tog oblika.

Pokazivači u hijerarhiji klasa

Možemo da deklarišemo Rectangle pokazivač, ili Circle pokazivač, ili Trapezoid pokazivač, gde svaki pokazuje na objekat odgovarajućeg tipa. Na primer, možemo da napravimo sledeće deklaracije:

```
Rectangle r1(3, 5);
Circle c1(7);
Trapezoid t1(4, 2, 6);

Rectangle* r_p = &r1;
Circle* c_p = &c1;
Trapezoid* t_p = &t1;
```

Shape pokazivač može da pokazuje na Rectangle objekat, ili Circle objekat, ili Trapezoid objekat zato što, nasleđivanjem, svaki od njih je i Shape objekat. Tako, nastavljajući prethodne deklaracije, ispravno je:

```
Shape* s_p1;
Shape* s_p2;
Shape* s_p3;

s_p1 = &r1;
s_p2 = &c1;
s_p3 = &t1;
```

Mada su s_p1, s_p2, i s_p3 Shape pokazivači, dozvoljeno im je da pokazuju na objekte u bilo kojoj izvedenoj klasi. Obrnuto ne važi, Rectangle pokazivač (ili Circle pokazivač ili Trapezoid pokazivač) ne može da pokazuje na Shape objekat.

U hijerarhiji klasa, pokazivač osnovne klase može da pokazuje na objekat bilo koje izvedene klase (ili direktno ili indirektno) iz osnovne klase. Obrnuto nije moguće, pokazivač na izvedenu klasu ne može da pokazuje na objekat osnovne klase.

Koristeći prethodne deklaracije i dodele, pretpostavimo sledeće:

```
cout << "The area of the rectangle is " << s_p1 -> Area() << endl;
```

Pokazivač s_p1 je Shape pointer, ali on pokazuje na Rectangle objekat. Čiji Area() metod će ova naredba izvršiti: klase Rectangle ili metod Area() osnovne klase Shape? Odgovor daje izvršenje sledećeg programa:

```
//Ovaj program prikazuje koriscenje pokazivaca i metoda u hijerarhiji klasa.

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

class Shape
{
  protected:
    string figure_type;
  public:
    Shape(string ft = "Default Figure") : figure_type(ft)
          {cout << endl << "Shape Constructor" << endl;}
    string Get_Type() {return figure_type;}
    ~Shape() {cout << endl << "Shape Destructor" << endl;}
    double Area() {return 0.0;}
};

class Rectangle : public Shape
{
  protected:
    double height, width;
  public:
    Rectangle(double h, double w): Shape("Rectangle"), height(h),width(w)
         {cout << endl << "Rectangle Constructor" << endl;}
    ~Rectangle() {cout << endl << "Rectangle Destructor" << endl;}
    double Area() {return height * width;}
};

class Circle : public Shape
{
  protected:
    double radius;
  public:
    Circle(double r) : Shape("Circle"), radius(r)
         {cout << endl << "Circle Constructor" << endl;}
    ~Circle() {cout << endl << "Circle Destructor" << endl;}
    double Area() {return 3.1416 * radius * radius;}
};

class Trapezoid : public Shape
{
  protected:
    double base1, base2, height;
  public:
```

```
    Trapezoid(double b1, double b2, double h)
            : Shape("Trapezoid"), base1(b1), base2(b2), height(h)
          {cout << endl << "Trapezoid Constructor" << endl;}
    ~Trapezoid() {cout << endl << "Trapezoid Destructor" << endl;}
    double Area() {return 0.5 * height * (base1 + base2);}
};

int main()
{
  cout << setprecision(4)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);
  Shape* s_p1;
  Circle* c_p1;
  Circle c1(5);

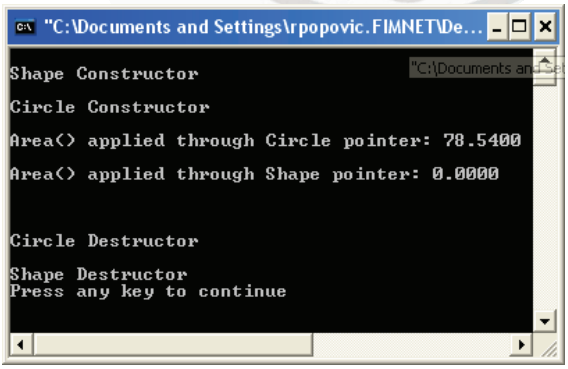
  s_p1 = &c1;          //Shape pointer pokazuje na objekat Circle

  c_p1 = &c1;          //Circle pointer pokazuje na objekat Circle

  cout << endl;
  cout << "Area() applied through Circle pointer: "
       << c_p1 -> Area() << endl << endl;

  cout << "Area() applied through Shape pointer: "
       << s_p1 -> Area() << endl << endl;

  cout << endl;
  return 0;
}
```



Iz dve linije koje prikazuje cout u main() funkciji, vidimo da Area() metod primenjen na Circle objekat kroz Circle pokazivač daje korektnu

površinu 78.5400. Međutim, Area() metod koji je primenjen na Circle objekat kroz Shape pokazivač je Area() metod iz Shape klase. Mada s_p1 pokazuje na Circle objekat, metod osnovne klase se poziva kada je on primenjen na objekat kroz pokazivač osnovne klase. Sledi, "nadjačavanje" metoda (*overriding*) ne radi u ovom slučaju.

Virtuelni metodi i polimorfizam

Koncept polimorfizma (bukvalno znači više oblika - *many forms*) proširuje *method overriding* na slučaj objekata u izvedenoj klasi kojima se pristupa kroz pokazivač osnovne klase. Da biste primenili polimorfizam, metod koji treba da bude nadjačan mora da bude deklarisan kao virtuelni. Rezervisana reč virtual mora da se pojavi u deklaraciji metoda u osnovnoj klasi. Sledeći program prikazuje razliku koja deklarisanjem metoda Area() da je virtuelni, čini izlaz iz prethodnog problema.

```
//Ovaj program prikazuje koriscenje virtuelne funkcije
//u hijararhiji klasa

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

class Shape
{
 protected:
   string figure_type;
 public:
   Shape(string ft = "Default Figure") : figure_type(ft)
     {cout << endl << "Shape Constructor" << endl;}
   string Get_Type() {return figure_type;}
   ~Shape() {cout << endl << "Shape Destructor" << endl;}
   virtual double Area() {return 0.0;}
};

class Rectangle : public Shape
{
 protected:
   double height, width;
 public:
   Rectangle(double h, double w): Shape("Rectangle"), height(h),width(w)
     {cout << endl << "Rectangle Constructor" << endl;}
```

```
   ~Rectangle() {cout << endl << "Rectangle Destructor" << endl;}
   double Area() {return height * width;}
};

class Circle : public Shape
{
 protected:
   double radius;
 public:
   Circle(double r) : Shape("Circle"), radius(r)
       {cout << endl << "Circle Constructor" << endl;}
   ~Circle() {cout << endl << "Circle Destructor" << endl;}
   double Area() {return 3.1416 * radius * radius;}
};

class Trapezoid : public Shape
{
 protected:
   double base1, base2, height;
 public:
   Trapezoid(double b1, double b2, double h)
          : Shape("Trapezoid"), base1(b1), base2(b2), height(h)
       {cout << endl << "Trapezoid Constructor" << endl;}
   ~Trapezoid() {cout << endl << "Trapezoid Destructor" << endl;}
   double Area() {return 0.5 * height * (base1 + base2);}
};

int main()
{
 cout << setprecision(4)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);

 Shape* s_p1;
 Circle* c_p1;

 Circle c1(5);

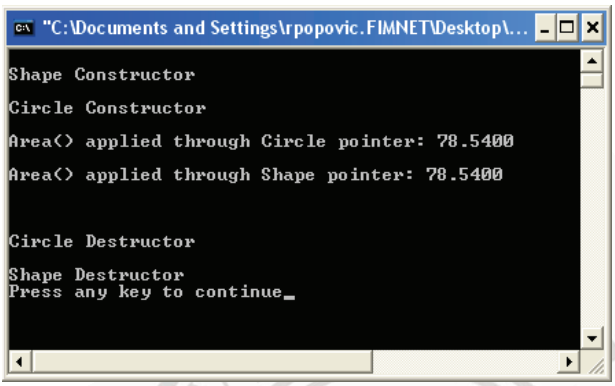
 s_p1 = &c1;          //Shape pointer pokazuje na objekat Circle

 c_p1 = &c1;          //Circle pointer pokazuje na objekat Circle

 cout << endl;
 cout << "Area() applied through Circle pointer: "
      << c_p1 -> Area() << endl << endl;
 cout << "Area() applied through Shape pointer: "
```

```
    << s_p1 -> Area() << endl << endl;

 cout << endl;
 return 0;
}
```



Ovoga puta, druga cout naredba izvršava Area() metod cilja Shape pointera s_p1. Ona radi ovo zato što smo deklarisali Area() metod u osnovnoj klasi Shape kao virtuelni metod.

Polimorfizam dozvoljava da izbor metoda zavisi od tipa podatka cilja pointera radije nego od tipa podatka pointera. Da bi polimorfizam radio, objektima u hijerarhiji klasa mora se pristupati kroz pokazivače. Samo metodi klase mogu biti virtuelni. Da bi metod u izvedenoj klasi nadjačao virtuelni metod u osnovnoj klasi, metod u izvedenoj klasi mora da ima isto ime i potpis (broj i tip argumenata) kao metod osnovne klase. Metod u izvedenoj klasi sa istim imenom i potpisom kao i virtuelni metod u osnovnoj klasi je takođe virtuelan, bez obzira da li se ili ne virtual ključna reč pojavljuje u njegovoj deklaraciji (Zato su, Area() metodi u izvedenim klasama iz prethodnog programa takođe virtuelni)

Funkcije članice osnovne klase koje se u izvedenim klasama mogu realizovati specifično za svaku izvedenu klasu, nazivaju se virtuelne funkcije.

Virtuelni destruktori

Sada razmatramo program u kome je zamenjen statički deklarisan objekat c1 iz prethodnog programa, dinamički alociranim objektom na koji pokazuje s_p1.

```
//Program prikazuje nekorektno koriscenje destruktora u hijerarhiji klasa

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Shape
{
 protected:
   string figure_type;
 public:
   Shape(string ft = "Default Figure") : figure_type(ft)
     {cout << endl << "Shape Constructor" << endl;}
   string Get_Type() {return figure_type;}
   ~Shape() {cout << endl << "Shape Destructor" << endl;}
   virtual double Area() {return 0.0;}
};

class Rectangle : public Shape
{
 protected:
   double height, width;
 public:
   Rectangle(double h, double w): Shape("Rectangle"), height(h),width(w)
     {cout << endl << "Rectangle Constructor" << endl;}
   ~Rectangle() {cout << endl << "Rectangle Destructor" << endl;}
   double Area() {return height * width;}
};

class Circle : public Shape
{
 protected:
   double radius;
 public:
   Circle(double r) : Shape("Circle"), radius(r)
     {cout << endl << "Circle Constructor" << endl;}
   ~Circle() {cout << endl << "Circle Destructor" << endl;}
   double Area() {return 3.1416 * radius * radius;}
};

class Trapezoid : public Shape
{
 protected:
   double base1, base2, height;
```

```
  public:
    Trapezoid(double b1, double b2, double h)
          : Shape("Trapezoid"), base1(b1), base2(b2), height(h)
      {cout << endl << "Trapezoid Constructor" << endl;}
    ~Trapezoid() {cout << endl << "Trapezoid Destructor" << endl;}
    double Area() {return 0.5 * height * (base1 + base2);}
};

int main()
{
  cout << setprecision(4)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

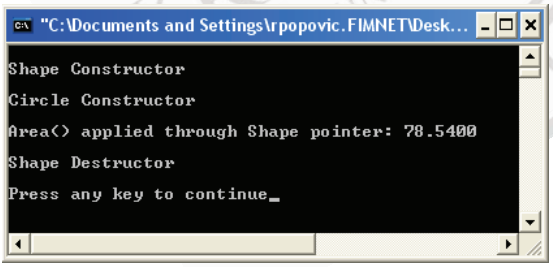
  Shape* s_p1 = new Circle (5);

  cout << endl;
  cout << "Area() applied through Shape pointer: "
       << s_p1 -> Area() << endl;

  delete s_p1;

  cout << endl;

  return 0;
}
```



Kada se delete naredba izvršava za s_p1, koji pokazuje na Circle objekat, samo se destruktor za osnovnu klasu izvršava. Ovo je zbog toga što se destruktor primenjuje na objekat kroz pokazivač. Kako destruktor osnovne klase nije deklarisan kao virtual, primena destruktora na cilj pointera s_p1 izvršava destruktor osnovne klase a ne destruktor izvedene klase.

Da biste propisno izvršili destruktore, tj. izvršili destruktor za izvedenu klasu, a zatim izvršili destruktor za osnovnu klasu, učinite da destruktor osnovne klase bude virtuelni metod.

Napomena: Pošto se konstruktor poziva pre nego što je objekat nastao, nema smisla da bude virtuelan, jezik C++ to ne dozvoljava.

Sledeći program koristi virtual destruktore.

```
//Program prikazuje korektno koriscenje destruktora u hijerarhiji klasa.

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Shape
{
 protected:
   string figure_type;
 public:
   Shape(string ft = "Default Figure") : figure_type(ft)
     {cout << endl << "Shape Constructor" << endl;}
   string Get_Type() {return figure_type;}
   virtual ~Shape() {cout << endl << "Shape Destructor" << endl;}
   virtual double Area() {return 0.0;}
};

class Rectangle : public Shape
{
 protected:
   double height, width;
 public:
   Rectangle(double h, double w): Shape("Rectangle"), height(h),width(w)
     {cout << endl << "Rectangle Constructor" << endl;}
   virtual ~Rectangle() {cout << endl << "Rectangle Destructor" << endl;}
   double Area() {return height * width;}
};

class Circle : public Shape
{
 protected:
   double radius;
 public:
   Circle(double r) : Shape("Circle"), radius(r)
     {cout << endl << "Circle Constructor" << endl;}
   virtual ~Circle() {cout << endl << "Circle Destructor" << endl;}
   double Area() {return 3.1416 * radius * radius;}
};
```

```
class Trapezoid : public Shape
{
 protected:
   double base1, base2, height;
 public:
   Trapezoid(double b1, double b2, double h)
        : Shape("Trapezoid"), base1(b1), base2(b2), height(h)
       {cout << endl << "Trapezoid Constructor" << endl;}
   virtual ~Trapezoid() {cout << endl << "Trapezoid Destructor" << endl;}
   double Area() {return 0.5 * height * (base1 + base2);}
};

int main()
{
 cout << setprecision(4)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);

 Shape* s_p1 = new Circle (5);

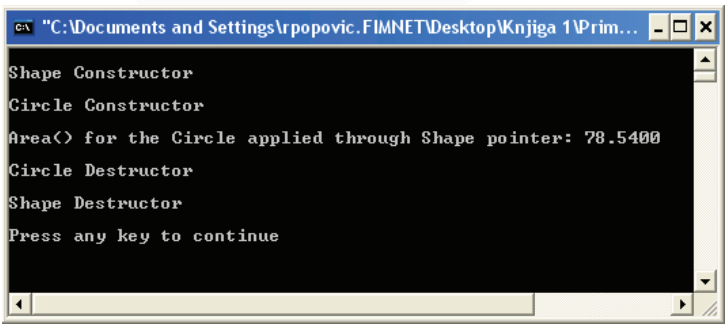
 cout << endl;

 cout << "Area() for the " << s_p1 -> Get_Type()
      << " applied through Shape pointer: "
      << s_p1 -> Area() << endl;

 delete s_p1;

 cout << endl;

 return 0;
}
```



Sada se destruktori za target pointer s_p1 izvršavaju korektno. Prvo se destruktor za izvedenu klasu izvršava a onda destruktor osnovne klase. Obratite pažnju da smo takođe koristili aksesor metod Get_Type(), koji je takođe primenjen kroz pokazivač.

Primer u kome se koristi polimorfizam

Sledeći program rešava problem koji smo postavili na početku ove sekcije: omogućiti korisniku da kreira niz oblika geometrijskih slika po izboru. Korisnik može prvo da kreira na primer pravougaonik određenih dimenzija, zatim krug određenog poluprečnika itd. Nakon toga želimo da program prikaže površine slika. Kako ćemo ovo da uradimo? Odgovor leži u kreiranju niza Shape pokazivača i korišćenju polimorfizma sa metodom Area(), kao u sledećem programu:

```
//Ovaj program prikazuje koriscenje polimorfizma

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

class Shape
{
 protected:
   string figure_type;
 public:
   Shape(string ft = "Default Figure") : figure_type(ft) {}
   string Get_Type() {return figure_type;}
   virtual ~Shape() {}
   virtual double Area() {return 0.0;}
};

class Rectangle : public Shape
{
 protected:
   double height, width;
 public:
   Rectangle(double h, double w): Shape("Rectangle"), height(h),width(w) {}
   virtual ~Rectangle() {}
   double Area() {return height * width;}
};
```

```
class Circle : public Shape
{
 protected:
   double radius;
 public:
   Circle(double r) : Shape("Circle"), radius(r) {}
   virtual ~Circle() {}
   double Area() {return 3.1416 * radius * radius;}
};

class Trapezoid : public Shape
{
 protected:
   double base1, base2, height;
 public:
   Trapezoid(double b1, double b2, double h)
        : Shape("Trapezoid"), base1(b1), base2(b2), height(h) {}
   virtual ~Trapezoid() {}
   double Area() {return 0.5 * height * (base1 + base2);}
};

Shape* Get_Shape();

int main()
{
 cout << setprecision(4)
      << setiosflags(ios::fixed)
      << setiosflags(ios::showpoint);

 const int num_shapes = 5;

 Shape* shapes[num_shapes];

 for (int i = 0; i < num_shapes; ++i)
   shapes[i] = Get_Shape();

 cout << endl;

 for (i = 0; i < num_shapes; ++i)
   cout << "\nThe area of the " << shapes[i] -> Get_Type() << " is "
        << shapes[i] -> Area();

 cout << endl;

 for (i = 0; i < num_shapes; ++i)
   delete shapes[i];
```

```
 cout << endl;

 return 0;
}

Shape* Get_Shape()
{
 Shape* s_p; //funkcija vraca pointer s_p

 double height,
        width,
        radius,
        base1,
        base2;

 cout << endl;
 cout << "Enter the number of the shape you want to create:"
      << endl << endl;
 cout << "For a Rectangle, enter 1" << endl;
 cout << "For a Circle, enter 2" << endl;
 cout << "For a Trapezoid, enter 3" << endl << endl;
 cout << "Please make your selection: ";

 int response;

 cin >> response;

 switch (response)
 {
  case 1:
    cout << endl << endl;
    cout << "Enter the height of the rectangle: ";
    cin >> height;
    cout << endl << endl;
    cout << "Enter the width of the rectangle: ";
    cin >> width;

    s_p = new Rectangle(height, width);

    break;

  case 2:

    cout << endl << endl;
    cout << "Enter the radius of the circle: ";
    cin >> radius;
```

```
    s_p = new Circle(radius);

    break;

  case 3:
    cout << endl << endl;
    cout << "Enter the first base of the trapezoid: ";
    cin >> base1;
    cout << endl << endl;
    cout << "Enter the second base of the trapezoid: ";
    cin >> base2;
    cout << endl << endl;
    cout << "Enter the height of the trapezoid: ";
    cin >> height;

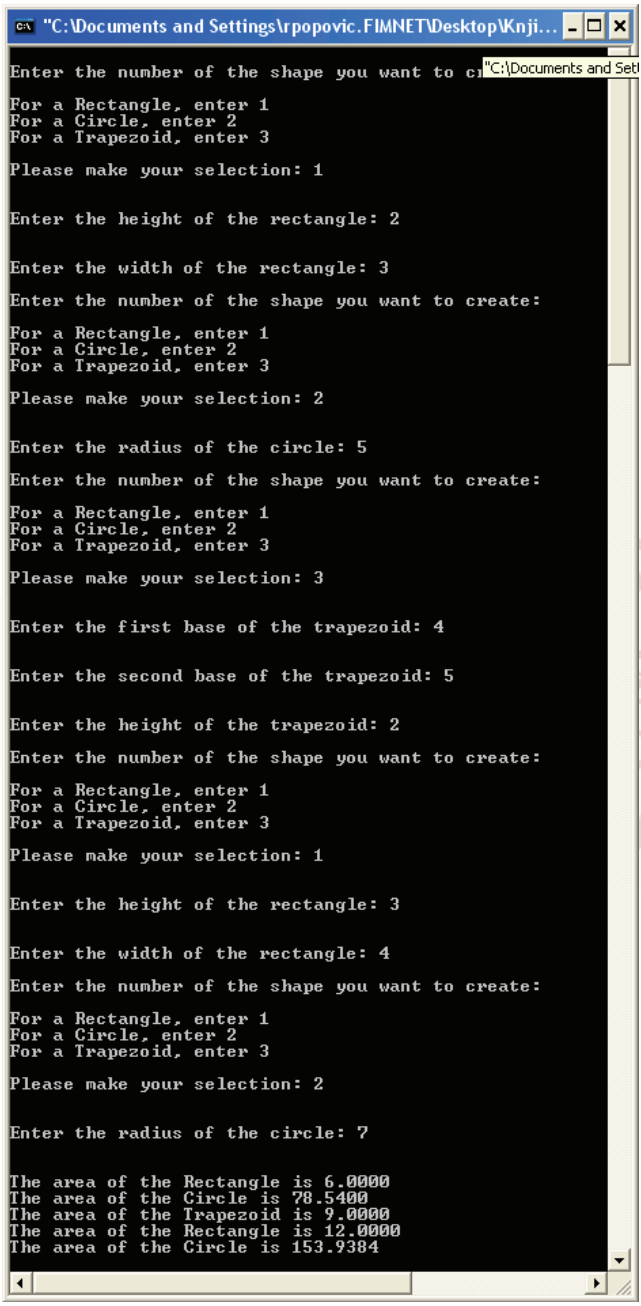
    s_p = new Trapezoid(base1, base2, height);

    break;

  default:
    cout << endl << endl;
    cout << "Invalid selection. Creating default object.";

    s_p = new Shape();

    break;
 }
 return s_p;
}
```



Svi destruktori su deklarisani kao virtual metodi. Takođe metod Area() u klasi Shape je deklarisan kao virtual. Ovo dozvoljava da se odgovarajući

metod Area() primeni kroz Shape pokazivač. U glavnoj funkciji main(), deklarisali smo niz shapes[] Shape pokazivača. Prva for petlja dodeljuje svakom elementu niza shapes[] Shape pokazivač koji vraća funkcija Get_Shape().

Kada se Get_Shape() izvršava, prikazuje se korisniku meni sa opcijama Rectangle, Circle, ili Trapezoid kreiranja geometrijskih slika. Naredba switch određuje akcije nakon korisnikove odluke. Ako, na primer, korisnik želi da kreira objekat pravougaonik Rectangle, funkcija pita za visinu i širinu pravougaonika. Zatim se kreira novi objekat Rectangle, dinamički korišćenjem operatora new. Mada ovo vraća pokazivač na objekat Rectangle, ispravno je dodeliti ga s_p, koji je Shape pokazivač. Ako korisnik ukuca broj koji nije 1, 2 ili 3, *default* slučaj iz naredbe switch kreira *default* Shape objekat.

Prva for petlja inicijalizuje Shape pointere u nizu shapes[] da pokazuju na neku kombinaciju Rectangle, Circle, Trapezoid i *default* objekata. Svi elementi niza mora da budu istog tipa, kao što su elementi niza shapes[] - oni su svi pointeri tipa Shape. Međutim, elementi niza shapes[] pokazuju na objekte različitog tipa. Zbog ovog razloga takvi nizovi se ponekad zovu heterogeni nizovi.

Druga for petlja koristi polimorfizam. U svakoj iteraciji ove for petlje, metod Area() se primenjuje kroz pointer shapes[i]. Metod Area() koji petlja koristi zavisi potpuno od tipa oblika shape na koji pokazivač pokazuje (polimorfizam). Suprotno, metod Get_Type() nije primenjen preko polimorfizma. Postoji samo jedan metod Get_Type(), koji je član klase Shape i koji druge klase nasleđuju.

Veoma je važno shvatiti da kada se program kompajlira, kompajler ne zna koji Area() metod će se u petlji koristi kada se program bude izvršavao. U stvari, u svakoj iteraciji, različit Area() metod se može primeniti. C++ odlučuje koji metod da koristi kada se petlja izvršava, a ne kada se program kompajlira. Ovo je kasno (*late* ili *dynamic binding*) ili dinamičko vezivanje: koji metod koristiti pri pozivu shapes[i] -> Area() se ne odlučuje sve dok se kod ne izvršava. Ovo je suprotno u odnosu na rano (*early* ili *static binding*) ili statičko vezivanje, *compiler* zna koja će funkcija biti izvršena u *compile time*.

Konačno, treća for petlja u glavnoj funkciji main() "čisti" memoriju brisanjem objekata koji su kreirani u prvoj for petlji.

10.3 Apstraktne osnovne klase

Hijerarhija klasa koju smo predstavili u prethodnoj sekciji ima osnovnu klasu Shape za tri preostale klase. Način na koji definišemo klasu Shape čini mogućim deklarisanje objekta Shape mada takav objekat nema dimenzije.

Čisti virtuelni metodi i apstraktne osnovne klase

U C++, možemo da definišemo klasu na takav način da je nemoguće da deklarišemo objekat toga tipa, čineći klasu da je apstraktna osnovna klasa - *abstract base class* ili ABC. Da biste napravili ABC klasu deklarišite bilo koji od njenih metoda kao čist virtuelni metod, treba da uradite sledeće:

-Smestite = 0 nakon liste argumenata metoda u deklaraciji metoda u klasi.

-Nemojte da definišete metod.

Kako čist virtuelni metod nema definiciju, nema smisla deklarisati objekat koji će da koristi takav metod. Zato se takva klasa zove apstraktna: ne možete kreirati bilo koji konkretan objekat tog tipa. Obezbeđujući definicije svih čistih virtuelnih metoda u izvedenoj klasi, specificiramo interfejs klase. Zato je moguće kreirati objekte u toj izvedenoj klasi.

Napravićemo Shape klasu kao apstraktnu osnovnu klasu na sledeći način:

```
class Shape
{
 protected:
     string figure_type;
 public:
     Shape(string ft = "Default Figure") : figure_type(ft) {}
     const char* Get_Type() {return figure_type;}
     virtual ~Shape() {}
     virtual double Area() = 0;
};
```

Ako je metod Area() čist virtuelni metod, sledi treba za izvedene klase obezbediti sopstvene definicije metoda Area(). Sledeće deklaracije u glavnoj funkciji main() sada su neispravne:

```
Shape s;            //neispravan objekat
```

```
Shape Func();          //neispravna povratna vrednost

double Func(Shape s);  //neispravna vrednost parametra
```

Kako je Shape apstraktna osnovna klasa, ne možete kreirati objekat tipa Shape. Zato je prva deklaracija neispravna. Druga deklaracija je neispravna zato što ona implicira da funkcija Func() sadrži return naredbu koja vraća Shape objekat. Ovo implicira da se Shape objekat kreira mehanizmom return, što je nesipravno. Treća naredba je neispravna zato što korišćenje funkcije zahteva prosleđivanje Shape objekta po vrednosti, što zahteva da je Shape objekat kreiran.

Međutim, zato što pokazivači i reference ne zahtevaju kreiranje objekta, sledeće deklaracije su ispravne.

```
Rectangle r(3, 5);

Shape& s_ref = r;

Shape* s_ptr = &r;

double Func(Shape*);
```

Podsetimo se da je Shape referenca drugo ime za Shape objekat. Kako Rectangle objekat nasleđuje Shape objekat, možemo koristiti Shape referencu na ovaj objekat.

Prisustvo samo jednog čistog virtuelnog metoda čini da klasa bude apstraktna osnovna klasa. Klasa može da sadrži druge čiste virtuelne metode, obične virtuelne metode, ili ne-virtuelne metode. Ako sada pokušate da deklarišete Shape objekat, kompajler će prikazati poruku sa greškom.

11. Fajlovi

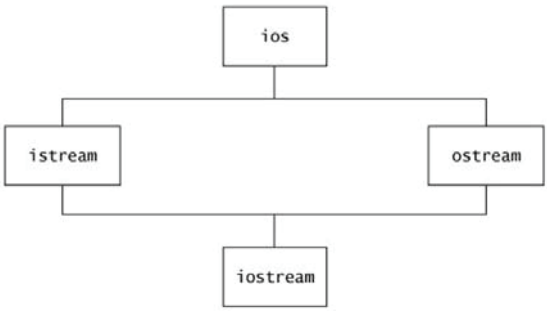
11.1 Ulazno/izlazni tokovi

Kao i jezik C, ni C++ ne sadrži ulazno/izlazne (I/O) operacije, već se one realizuju standardnim bibliotekama. Na raspolaganju su i stare C biblioteke sa funkcijama scanf i printf, ali njihovo korišćenje nije u duhu jezika C++.

U C++ fajl se tretira kao tok karaktera. Karakteri iz fajla su na raspolaganju programu u jednom trenutku jedan karakter u nizu (sekvencijalno). Ulaz i izlaz u C++ jeziku su poznati kao tokovi (*stream I/O*). C++ jezik ne pretpostavlja bilo koju drugu organizaciju fajla osim njegove strukture kao toka. Da bi dao fajlu dodatnu logičku strukturu, programer mora da napiše programske instrukcije koje to rade.

Od prvog poglavlja koristili smo standardni izlazni tok cout i standardni ulazni tok cin. Kasnije smo uveli tok standardne greške cerr. Po *default*, standardni ulazni tok je "prilepljen" za tastaturu. Tok cin prima svoj ulaz sa tastature. Standardni izlazni tokovi, cout i cerr, su "prilepljeni" displeju monitora. Bilo koji podatak ubačen u izlazne tokove cout i cerr pojavljuje se na monitoru. Standardni ulazni i izlazni tokovi automatski su na raspolaganju svakom C++ programu koji ima pretprocesorsku direktivu #include <iostream>. Sada ćemo objasniti prirodu cin, cout i cerr.

Ranije smo napomenuli da je cin objekat klase istream, a cout i cerr su objekti klase ostream. Ove dve klase su ustvari deo kompleksne hijerarhije klasa ulaza i izlaza koje su deo C++ jezika (donja slika).



Heder <iostream> sadrži definiciju klase iostream, koja je izvedena iz istream i ostream. Klasa kao što je iostream, koja je izvedena iz dve ili više

„roditelj“ klasa, je primer višestrukog nasleđivanja (*multiple inheritance*). Kod višestrukog nasleđivanja, izvedena klasa nasleđuje sve članove iz svih klasa iz kojih je izvedena. Na taj način klasa iostream nasleđuje sve članove iz istream, ostream i takođe, indirektno, iz klase ios, iz koje su i istream i ostream izvedene.

Klasa istream rukuje ulazom i uključuje definiciju operatora extraction >> (koji je preklopljen). Objekat cin je član klase istream. Metodi get() i getline() su članovi klase istream. Zbog toga se mogu primeniti na istream objekat kao što je cin. Na sledeći način smo koristili ove metode:

```
ch = cin.get();
cin.getline(buffer, 81);
```

Klasa ostream rukuje izlazom i uključuje definiciju *insertion* operatora << (koji je preklopljen). Objekat cout je član klase ostream. I/O manipulatori koje smo koristili da bismo oblikovali izlaz iz naših programa su ustvari metodi koji koriste ostream i ios metode za rad sa izlaznim tokom. Na primer, metod setprecision(), kada se pošalje na cout, određuje preciznost decimalnih brojeva koji se prikazuju od strane cout. Slično, setiosflags(ios::showpoint), kada se pošalje na cout, setuje jednobitni *flag* u primerku promenljive klase ios koji određuje da izlazni tok prikaže decimalnu tačku kada je decimalni broj izlaz. Operator razrešenja dosega :: je neophodan da bi se obezbedilo upućivanje na člana roditelj klase.

Tekstualni fajlovi

Fajlovi na diskovima mogu biti binarni (koriste se za kompaktno skladištenje podataka) i tekstualni (služe za skladištenje podataka u obliku čitljivom za čoveka). Za sada, pretpostavimo da radimo sa tekstualnim fajlovima. Tekstualni fajl se sastoji od slova alfabeta, brojeva i specijalnih znakova kao što su zapeta, crtica i tačka. Na kraju svake linije tekstualnog fajla je znak za novi red ('\n'). Tekst fajlove proizvode takozvani ASCII tekst editori ili samo tekst editori (Notepad, Windows i VI editor u UNIX-u, Visual C++ text editor). Bilo koji od ovih tekst editora se može koristiti za primere u ovom poglavlju. Napomena: fajlovi koje proizvode *word* procesori, kao npr. Microsoft Word, nisu tekstualni.

Korisnički deklarisani fajlovi

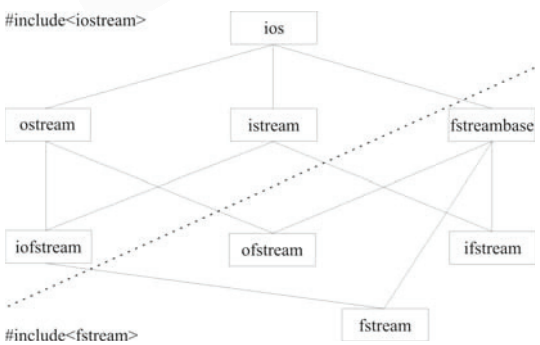
C++ dozvoljava programeru da deklariše I/O strimove, tj. fajlove drugačije od standardnih. Da biste odradili input ili output na imenovanim fajlovima zahteva se da napravite objekte u klasama koje su dizajnirane za izvođenje operacija nad fajlovima. Da biste radili sa fajlom u C++ mora da odradite sledeće korake:

-Deklarišite fajl objekat odgovarajućeg moda (npr. ulaz ili izlaz ili oba).
-Otvorite file object i pridružite tom objektu ime fajla iz vašeg računarskog sistema.
-Izvršite operacije na fajlu (npr. read data ili write data).
-Zatvorite fajl.

Deklarisanje fajl objekta

Pri deklarisanju objekta file, posmatraćemo klase koje podržavaju rad sa ulazno/izlaznim fajlovima.

-Koristite klasu ifstream za kreiranje ulaznih fajlova. Klasa ifstream je izvedena iz klase istream i klase zvane fstreambase. Klasa fstreambase, koja je izvedena iz klase ios, obezbeđuje mogućnosti za rad sa fajlovima sa imenima.
-Koristite klasu ofstream da biste kreirali izlazne fajlove. Ova klasa je izvedena iz ostream i fstreambase.
-Konačno, koristite klasu fstream da biste kreirali fajlove koji mogu biti i izlazni i ulazni fajlovi. Izvedena je iz iostream i fstreambase. Na slici, klase iznad isprekidane linije deklarisane su u heder fajlu <iostream> a klase ispod nje u heder fajlu <fstream>.



Da biste koristili fajl deklarišite objekat file odgovarajućeg tipa, kao u sledećim deklaracijama:

```
ofstream out_file;      //deklariše output file objekat nazvan out_file
ifstream in_file;       //deklariše input file objekat nazvan in_file
fstream io_file;        //deklariše file objekat io_file ili za izlaz ili za ulaz
```

Prva naredba deklariše out_file kao ofstream objekat, koji je tok izlaznog fajla. Ovo znači da naš program može da smesti podatke u out_file tok na skoro isti način kao što smeštamo podatke u standardni izlazni tok, cout.

Druga naredba deklariše in_file kao ifstream objekat. Zbog toga, naš program može da uzima podatke iz in_file toka na isti način kao što uzima podatke iz standardnog ulaznog toka, cin.

Treća naredba deklariše io_file kao objekat klase fstream. Naš program može da koristi io_file kao ulazni ili izlazni tok, zavisno od okolnosti u programu i kako želimo da otvorimo fajl.

Otvaranje fajla

Nakon deklarisanja fajl objekta, mora da otvorimo fajl i pridružimo fajl objekat stvarnom fizičkom fajlu koji se nalazi u računarskom sistemu. Prva od prethodnih deklaracija, na primer, deklariše out_file kao jedan ofstream objekat. Međutim, izlazni strim out_file još uvek nije pridružen ili dodeljen bilo kom stvarnom fajlu. Ovo je slično deklarisanju celobrojne varijable bez dodele početne vrednosti. Da bismo koristili varijablu svrsishodno mora da joj dodelimo vrednost. Sledi, jednom kada deklarišemo fajl objekat, mora da pridružimo taj objekat fajlu. Uradićemo ovo otvarajući fajl koristeći open() metod. Sledeća naredba otvara out_file strim i pridružuje ga fajlu

```
outdata.dat.
out_file.open("outdata.dat");
```

Od ove tačke u programu, sve operacije na fajl objektu out_file biće izvršene na fajlu outdata.dat. Treba obratiti pažnju da argument metoda open() mora biti string. Zbog toga, ime fajla mora biti uključeno sa navodnicima. U odsustvu direktorijumske putanje fajl će biti otvoren u katalogu gde se

program izvršava. Ime fajla može da uključi putanju do fajla. Na primer, sledeća naredba otvara fajl outdata.dat u root direktorijumu C: drajva.

```
out_file.open("C:\outdata.dat");
```

Drugi način za kreiranje i otvaranje fajla je korišćenje konstruktora sa jednim argumentom ofstream. Sledeća deklaracija kreira u jednom koraku ofstream objekat out_file i pridružuje ga fajlu outdata.dat.

```
ofstream out_file("outdata.dat");
```

Sledeći koraci prilikom otvaranja fajla su važni:

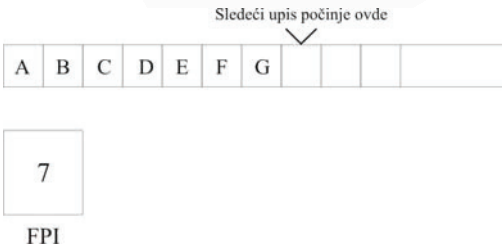
-Otvoriti fajl (koristeći open() metod ili konstruktor sa jednim argumentom kada deklarišete fajl objekat).

-Pridružiti stvaran fajl iz računarskog sistema fajl objektu koji se otvara.

-Otvoriti fajl prema modu koji odgovara tipu objekta.

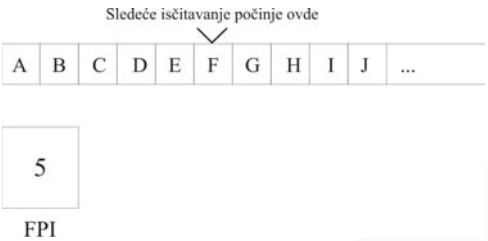
-Kreirati promenljivu koja je deo fajl objekta, nazvanu *file position indicator*, ili FPI, koja vodi računa o poziciji u fajlu. Vrednost FPI je programov *displacement* u fajlu, tj. broj bajtova od pozicije prvog bajta u fajlu.

Kada se otvori fajl objekat ofstream, ako fajl koji ste pridružili objektu ne postoji, sistem ga kreira automatski. Ako fajl već postoji otvara se za izlaz. U oba slučaja FPI se postavlja na 0, što je pozicija prvog bajta u fajlu. Prvi podatak koji program upiše u fajl je na toj poziciji. Ako prva operacija write smešta sedam znakova u fajl, sistem automatski povećava vrednost FPI za 7, kao na slici.



Kada otvorite fajl objekat ifstream, što je ulazni fajl, fajl koji ste pridružili objektu mora već da postoji. Ako ne postoji javlja se greška. Kako da rukujemo sa ovom greškom?

Pretpostavimo da fajl koji se otvara postoji, njegov FPI se postavlja na 0. Prvi podatak koji iščitavamo iz fajla je na poziciji 0. Ako pročitamo 5 karaktera iz fajla, FPI se poveća za 5. Sledi, sledeće čitanje podataka iz fajla počinje sa pomerajem 5.



Procesiranje fajla

Jednom kada je ofstream objekat kreiran i otvoren, možemo da koristimo bilo koji operator koji radi sa izlaznim fajlom, uključujući i *insertion* operator <<. Podsetimo se da je *insertion* operator deklarisan u klasi ostream, iz koje je izvedena klasa ofstream. Operatori su u stvari metodi (funkcije) i tako su tretirani u jeziku C++. Zbog toga, klasa ofstream nasleđuje *insertion* operator. Ovo dozvoljava da se uradi sledeće da bi se upisao string "Hello, World!" u fajl outdata.dat za kojim sledi karakter za novi red. Zapamtite da fajl objekat out_file predstavlja fajl outdata.dat.

```
out_file << "Hello, World!" << endl;
```

Zatvaranje fajla

Nakon završetka procesiranja fajla, treba da se zatvori fajl. C++ sistem automatski zatvara bilo koji otvoren fajl kada se program završi. Međutim, dobra je praksa zatvoriti fajlove koristeći metod close(). Na primer, sledeće naredba zatvara out_file

```
out_file.close();
```

Sledeći program ilustruje prosto korišćenje izlaznog fajla:

```
//Ovaj program prikazuje fajl kao obican izlaz

#include <fstream>
#include <iostream>
```

```
using namespace std;

int main()
{
  char buffer[81];

  ofstream out_file("outdata.dat");

  cout << "Enter your name: ";
  cin.getline(buffer, 81);

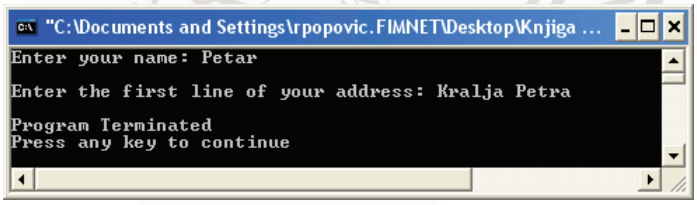
  out_file << buffer << endl;

  cout << endl;
  cout << "Enter the first line of your address: ";
  cin.getline(buffer, 81);
  out_file << buffer << endl;

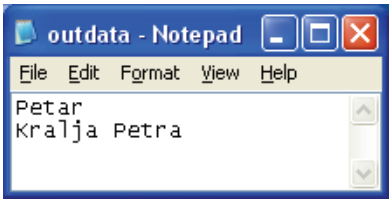
  out_file.close();

  cout << endl;
  cout << "Program Terminated" << endl;

  return 0;
}
```



Program ne prikazuje bilo kakav rezultat na displeju monitora zato što program smešta ime korisnika i adresu u fajl outdata.dat. Sledi listing izlaznog fajla.



Podsetimo se da je *insertion operator* nasleđen od ofstream klase. Znači, da biste primenili operator na ofstream objekat kao što je out_file, koristite *insertion operator* << da prosledite podatke u fajl. Smeštamo podatke u out_file tok na isti način kao što smeštamo podatke u standardni izlazni tok, cout.

Treba obratiti pažnju da koristimo C-stringove (tj. nizove karaktera) u ovom i u sledećim programima u ovom poglavlju, a ne string objekte. Možete u prethodnom primeru da deklarišete bafer kao string objekat i program će raditi na isti način. Međutim, uzdržaćemo se od korišćenja objekata sa cout zato što ne želimo da stvorimo utisak da možete da uradite tako sa proizvoljnim objektima. Na primer, ne možete da napišete kod za Savings_Account objekat acc1:

```
cout << acc1;
```

Kompajler prijavljuje grešku kao rezultat ove naredbe. Jedini način da se napravi da ovakva naredba radi je preklapanje operatora << koje smo definisali ranije za string objekte.

Provera grešaka kada se otvara fajl

Poslednji program ne proverava greške kada se fajl otvara. Ako se pojavi greška kada se fajl otvara, vrednost fajl objekta je 0.

Sledeći program prikazuje kako se proverava takva vrsta greške. Prvo izvršavanje programa daje izlaz u fajl, outdata.dat, koji je kreiran u folderu u kom se program izvršava. Drugi pokušaj izvršenja programa daje izlaz u fajl na disku M, koji ne postoji u sistemu gde se program izvršava.

```
//Ovaj program prikazuje fajl kao obican izlaz
//Program proverava da li je otvaranje fajla sa greskom ili bez

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
  char buffer[81];
  char file_name[81];
```

```
  cout << "Enter the name of the file you want to write to: ";
  cin.getline(file_name, 81);
  ofstream out_file(file_name);
  if (!out_file)
  {
    cerr << endl;
    cerr << "ERROR: File could not be opened." << endl;
    cerr << "Program Terminated" << endl;
    exit(1);
  }
  cout << endl;
  cout << "Enter your name: ";
  cin.getline(buffer, 81);

  out_file << buffer << endl;

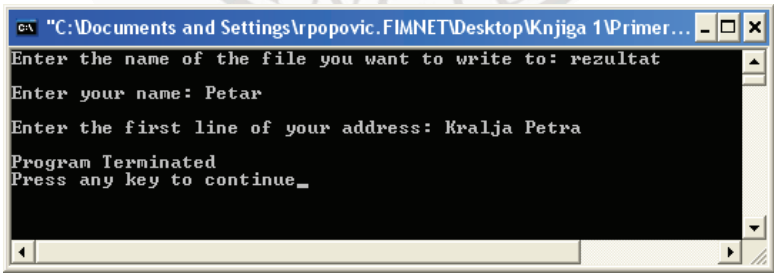
  cout << endl;
  cout << "Enter the first line of your address: ";
  cin.getline(buffer, 81);

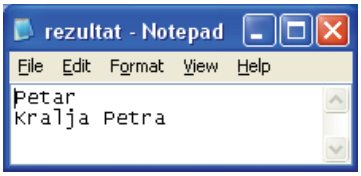
  out_file << buffer << endl;

  out_file.close();

  cout << endl;
  cout << "Program Terminated" << endl;

  return 0;
}
```





U ovom programu, dozvoljavamo korisniku da ukuca ime fajla u koji će ime i adresa da budu upisani. Program dobija ime fajla od korisnika i smešta ga u polje file_name[]. Kada je fajl objekat out_file deklarisan, koristimo kao argument konstruktora polje sa imenom file_name, koje je *character pointer*, ili string.

Kada se fajl otvara provera grešaka se izvodi na sledeći način:

```
if (!out_file)
{
  cerr << endl;
  cerr << "ERROR: File could not be opened.";
  cerr << "Program Terminated" << endl;
  exit(1);
}
```

Ako fajl nije otvoren propisno, vrednost out_file je 0. Zbog toga, testiranja !out_file je ustvari, "ako je out_file sa greškom pri otvaranju". U ovom slučaju, poruka o grešci se šalje na standardni tok error stream i program se završava exit(1) naredbom.

Kreiranje tekstualnog fajla sa zapisima

Mada C++ tretira sve fajlove kao tokove karaktera, nametnućemo logičku strukturu fajla. Zapis (*record*) je skup podataka, ili činjenica (fakta), koji se odnose na određeni entitet. Svaki fakt unutar zapisa je polje. Na primer, ako treba da napišemo program koji procesira inventar male kompanije, možemo da projektujemo part zapis. Svaki takav zapis može da se sastoji od tri polja: identifikacionog broja, količine i cene. Kompletan skup part zapisa je inventarski fajl.

Postoji nekoliko načina konstrukcije zapisa u C++. Kako za sada razmatramo samo tekstualne fajlove konstruisaćemo zapise na sledeći način. Svaki zapis u fajlu je linija teksta. Zbog toga, u toku karaktera koji čini fajl, znak za novi red odvaja zapise. Polja u svakom zapisu su odvojena sa jednim ili više razmaka (*spaces*). Pozicija unutar zapisa identifikuje polje. Inventarski fajl sa delovima sa tri zapisa, koristeći tekst editor izgleda ovako:

```
23 145 24.95
15 46   67.99
65 230 11.95
```

Prethodni fajl je primer sekvencijalnog fajla, koji je fajl čiji su zapisi snimljeni sekvencijalno jedan za drugim. Sledeći program kreira sekvencijalni tekstualni parts inventory fajl. On pita korisnika da unese podatke za zapise u fajl. Onda program upisuje zapis u delove fajla.

```
//Ovaj program prikazuje kreiranje fajla sa obicnim tekstualnim podacima
//inventara u kome su podaci odvojeni praznim prostorom.

#include <fstream>
#include <iostream>
#include <cstdlib>
#include <iomanip>

using namespace std;

int main()
{
 char file_name[81];

 int    id_no;
 int    qoh;
 double price;

 int response;

 cout << "Enter the name of the new inventory file: ";
 cin.getline(file_name, 81);

 ofstream inventory_file(file_name);

 if (!inventory_file)
 {
  cerr << endl;
  cerr << "ERROR: File could not be opened." << endl;
  cerr << "Program Terminated" << endl;
  exit(1);
 }
 inventory_file << setprecision(2)
                << setiosflags(ios::fixed)
                << setiosflags(ios::showpoint);

 cout << endl;
 cout << "Create an inventory record? (1 for Yes/0 for No): ";
 cin >> response;

 while (response)
```

```
  {
     // Uzima podatke za inventarski zapis

    cout << endl;
    cout << "ID Number: ";
    cin >> id_no;
    cout << endl;
    cout << "Quantity On Hand: ";
    cin >> qoh;

    cout << endl;
    cout << "Price: ";
    cin >> price;

       //Upisuje inventarski zapis

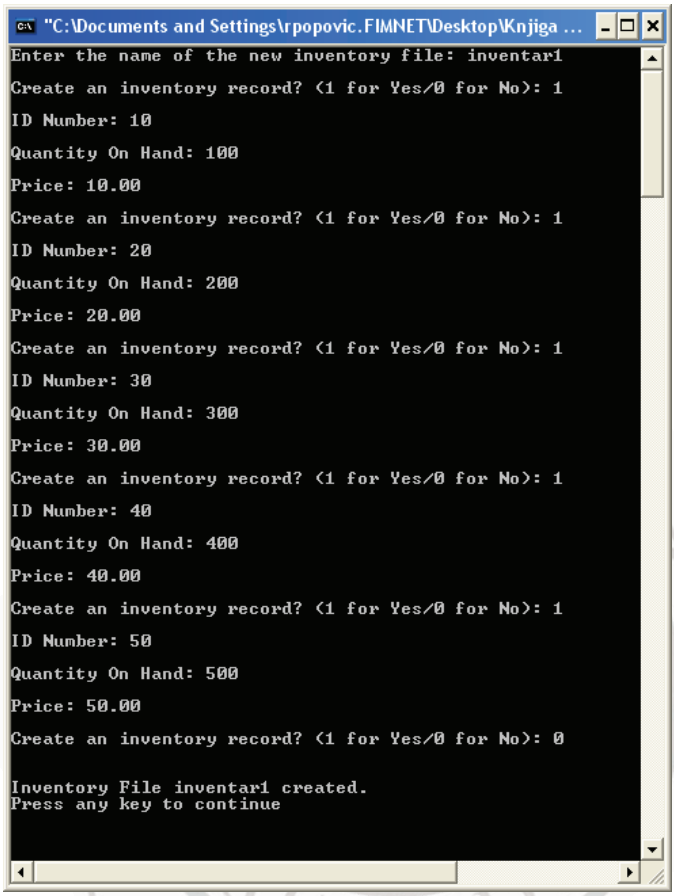
    inventory_file << id_no << " " << qoh << " " << price << endl;

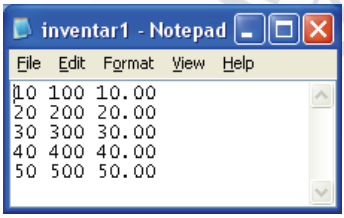
    cout << endl;
    cout << "Create an inventory record? (1 for Yes/0 for No): ";
    cin >> response;
  }

  cout << endl << endl;
  cout << "Inventory File " << file_name << " created." << endl;

  inventory_file.close();

  return 0;
}
```





Program prvo pita korisnika da ukuca ime inventar fajla. Nakon dobijanja imena, program otvara fajl kao ofstream fajl objekat inventory_file i testira ga da vidi da li je fajl otvoren propisno. Sledeće, program šalje u inventory_file setprecision() i setiosflags() naredbe koje obično šalje u cout. Ovo obezbeđuje da svaka vrednost za polje cena zadržava decimalnu tačku i ima dve cifre sa desne strane decimalne tačke kada se upisuje u output fajl.

Program zatim izvršava while petlju koja kreira *output* fajl. U telu petlje, program pita redom za identifikacioni broj, količinu i cenu svakog uzorka. Zatim se upisuje inventarski zapis u fajl.

```
inventory_file << id_no << " " << qoh << " " << price << endl;
```

Obratite pažnju da program smešta razmak između svakog polja, i završava zapis znakom za novi red. Zatim program pita korisnika da li želi da kreira novi zapis. Nakon napuštanja petlje, program prikazuje poruku i zatvara fajl.

Izlazni fajl

Kada završimo sa kreiranjem nekoliko fajlova, kako možemo da koristimo ove fajlove kao ulaz u program i iščitavanje njihovog sadržaja? Kao što je napomenuto prethodno, za kreiranje ulaznog fajla, deklarišemo ifstream fajl objekat i zatim otvaramo fajl. Na primer, da bismo otvorili fajl outdata.dat kao ulazni fajl, možemo da izvršimo sledeće naredbe:

```
ifstream in_file;
in_file.open("outdata.dat");
```

Alternativno, možemo da kreiramo i otvorimo fajl u isto vreme, koristeći konstruktor sa jednim argumentom na sledeći način:

```
ifstream in_file("outdata.dat");
```

Kada čitamo podatke iz fajla, mora uvek da proverimo uslov za kraj fajla (*end-of-file*). Uslov za kraj znači da program, pri procesiranju fajla, dolazi do kraja fajla. Sledi nema više podataka za čitanje iz fajla. Metod eof() vraća 1 (*true*) ako je FPI na kraju fajla, i vraća nulu ako nije. Znači, ako je za in_file objekat klase ifstream koji je otvoren, vrednost koju vraća in_file.eof() jednaka 1, uslov za kraj fajla je ispunjen.

Sledeći program ilustruje kako možemo da čitamo i prikažemo podatke koje program smešta u fajl outdata.dat.

```
//Ovaj program prikazuje rad sa prostim ulaznim fajlom

#include <fstream>
#include <iostream>
```

```
#include <cstdlib>

using namespace std;

int main()
{
 char buffer[81];
 char file_name[81];

 cout << "Enter the name of the file you want to display: ";
 cin.getline(file_name, 81);
 ifstream in_file(file_name);

 if (!in_file)
 {
  cerr << endl;
  cerr << "ERROR: File could not be opened." << endl;
  cerr << "Program Terminated" << endl;
  exit(1);
 }

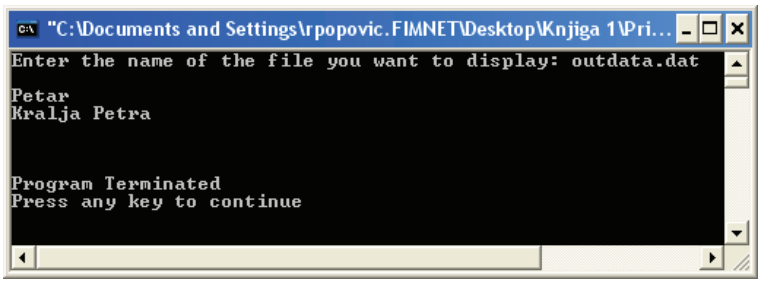
 cout << endl;

 while (!in_file.eof())
 {
  in_file.getline(buffer, 81);
  cout << buffer << endl;
 }

 in_file.close();

 cout << endl << endl;
 cout << "Program Terminated" << endl;

 return 0;
}
```



Proveru greške kada otvaramo fajl radimo na isti način kao što smo radili za izlazni fajl, proveravajući vrednost 0 za objekat klase ifstream, tj. in_file. Kopija i displej fajla su dati u while petlji. Uslov petlje proverava uslov za kraj ulaznog fajla. Ako nije kraj fajla, tj. ako je !in_file.eof() uslov ispunjen (*true*), program izvršava telo petlje. U telu petlje getline() metod čita liniju iz ulaznog fajla na potpuno isti način kao što čita linije ubačene sa tastature koristeći standardni ulazni strim cin. Podsetimo se da je metod getline() nasleđen iz klase istream. Naredba cout zatim prikazuje liniju na monitoru.

Procesiranje tekstualnih fajlova sa zapisima

Sledeći program prikazuje kako se procesira sekvencijalni tekstualni fajl koji radi sa zapisima. Program čita (prolazi) kroz fajl sa delovima inventara i treba da prikaže samo one zapise delova za koje ukupna vrednost dela (količina pomnožena sa cenom) prelazi 3000.00€.

```
//Ovaj program prikazuje citanje zapisa iz tekstualnog fajla
//ciji su podaci odvojeni praznim pozicijama.

#include <fstream>
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

int main()
{
 char file_name[81];

 int    id_no;
 int    qoh;
 double price;
 double total;

 cout << setprecision(2)
     << setiosflags(ios::fixed)
     << setiosflags(ios::showpoint);

 cout << "Enter the name of the inventory file: ";
 cin.getline(file_name, 81);

 ifstream inventory_file(file_name);
```

```
if (!inventory_file)
{
  cerr << endl;
  cerr << "ERROR: File could not be opened." << endl;
  cerr << "Program Terminated" << endl;
  exit(1);
}

cout << endl << endl;
cout << setw(8) << "ID No." << setw(10) << "Quantity"
     << setw(10) << "Price" << setw(12) << "Total"
     << endl << endl;

//Dobijanje prvog zapisa inventara
inventory_file >> id_no >> qoh >> price;

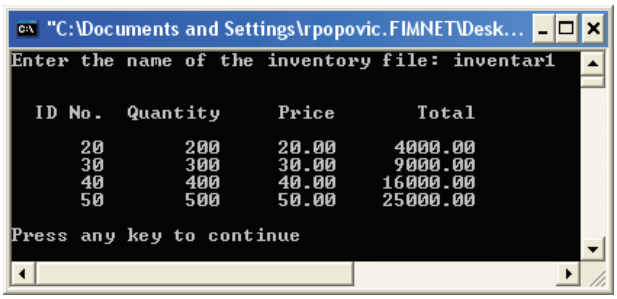
while (!inventory_file.eof())
{
  total = qoh * price;

  if (total > 3000.00)
  {
    cout << setw(8) << id_no << setw(10) << qoh
         << setw(10) << price << setw(12)<< total
         << endl;
  }

   //Dobijanje sledeceg zapisa inventara
  inventory_file >> id_no >> qoh >> price;
}

inventory_file.close();

cout << endl;
return 0;
}
```



Program počinje dobijanjem imena fajla sa inventarom od korisnika i otvaranjem fajla kao fajl objekta inventory_file klase ifstream. Zatim, program prikazuje liniju zaglavlja po kolonama.

Kada program čita iz fajla zapise sekvencijalno, tj. u jednom trenutku jedan karakter, mora da vodite računa da se procesira poslednji zapis u fajlu i da se zapis procesira samo jednom.

Sledi opis koraka za čitanje svih zapisa u sekvencijalnom ulaznom fajlu:

1. Učitaj prvi zapis fajla odmah nakon otvaranja fajla.
2. Učitaj ulazni fajl ponovi odmah nakon što program kompletira procesiranje ulaznih podataka.

Tako, tipičan program koji procesira takav fajl i koristi eof() da bi odredio kraj fajl treba da ima sledeći format napisan u pseudokodu:

```
otvoriti ulazni fajl input_file
iscitati prvi zapis
while (!input_file.eof())
        obradi zapis
        učitaj sledeći zapis
endwhile
```

Ovo je urađeno u programu, nakon otvaranja fajla za inventar, program čita prvi zapis izvršavajući naredbu

```
inventory_file >> id_no >> qoh >> price;
```

koja čita prva tri broja u fajlu, identifikacioni broj, količinu i cenu i smešta ih u tri varijable. Zatim, program ulazi u petlju za procesiranje ...

Ulazno/izlazni fajl

Možemo da koristimo fajl i kao ulaz i kao izlaz u istom programu. Da bismo uradili ovo, treba da imamo fajl objekat koji može da služi i kao ulazni i kao izlazni tok. Klasa fstream, nasleđena iz iostream, koja redom nasleđuje i ostream i istream, je upravo ono što nam treba. Zato što član klase fstream može da se koristi i kao ulaz i kao izlaz, kada se fajl otvara, mora da specificirate mod otvaranja kao drugi argument metoda open() ili konstruktorov drugi argument. (open() metod i fajl konstruktori su preklopljeni - oni mogu da uzmu ili jedan ili dva argumenta.) Indicirate mod

otvaranja koristeći konstante moda pristupa koje su definisane u klasi ios i zbog toga su na raspolaganju fstream, ifstream i ofstream.

Konstante moda pristupa:

ios::in	Otvara fajl za ulaz (ili čitanje). Ovaj mod može takođe da se koristi i sa ifstream fajl objektima.
ios::out	Otvara fajl za izlaz (ili upisivanje). Ovaj mod može takođe da se koristi i sa ofstream fajl objektima.
ios::app	Otvara fajl za izlaz i počinje na kraju fajla (dodavanje). Ovaj mod se može koristiti samo sa ofstream objektima.

Da bismo otvorili fajl outdata.dat za ulaz, treba da napišemo kôd:

```
fstream file("outdata.dat", ios::in);
```

Kada se otvara fajl u ios::in modu ili u ios::out modu, FPI se postavlja na 0. Kada se otvara izlazni fajl u ios::app, ili modu dodavanja, FPI se postavlja na broj bajtova u fajlu.

Sledeći program dozvoljava korisniku da ukuca ime i adresu i smesti ih u tekstualni fajl. Program zatim prikazuje sadržaj fajla.

```
//Ovaj program prikazuje jednostavan rad sa fajlom (ulaz i izlaz).
//Fajl je otvoren u ios::in modu i ios::out modu.

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
 char buffer[81];
 char file_name[81];

 cout << "Enter the name of the file you want to write to: ";
 cin.getline(file_name, 81);
 fstream file(file_name, ios::out);

 if (!file)
 {
  cerr << endl;
  cerr << "ERROR: File could not be opened." << endl;
  cerr << "Program terminated" << endl;
```

```
  exit(1);
 }
 cout << endl;
 cout << "Enter your name: ";
 cin.getline(buffer, 81);

 file << buffer << endl;

 cout << endl;
 cout << "Enter the first line of your address: ";
 cin.getline(buffer, 81);

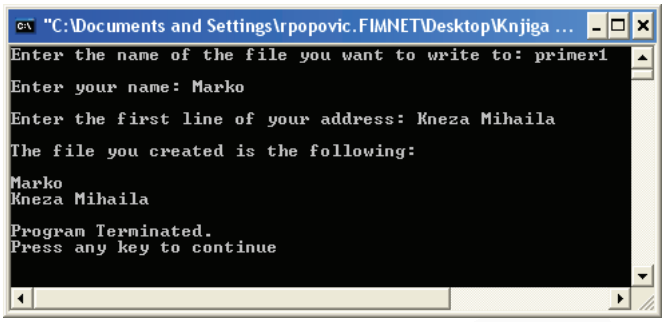
 file << buffer << endl;

 file.close();

 cout << endl;
 cout << "The file you created is the following:"
                << endl << endl;
 file.open(file_name, ios::in);
 if (!file)
 {
  cerr << endl;
  cerr << "ERROR: File could not be opened." << endl;
  cerr << "Program Terminated" << endl;
  exit(1);
 }
 while (!file.eof())
 {
  file.getline(buffer, 81);
  cout << buffer << endl;
 }
 file.close();

 cout << "Program Terminated." << endl;

 return 0;
}
```



Program prvo pita za ime fajla koji korisnik želi da kreira. Prvi put kada otvorimo fajl, kreiramo fstream fajl objekat u deklaraciji i otvaramo fajl u modu ios::out. Program zatim pita za korisnikovo ime i adresu, koje program upisuje u fajl. Program zatim zatvara fajl i otvara isti fajl u ios::in modu. Naredba while zatim prikazuje sadržaj fajla.

Sledeći program otvara fajl kreiran ranije i dodaje jednu liniju fajlu. Da bismo otvorili fajl u ios::app modu, fajl objekat koji predstavlja fajl file_app u programu, treba da bude ofstream objekat. Nakon dodavanja linije fajlu, program zatvara fajl a zatim ga otvara ponovo kao ifstream fajl, file_in u programu, tako da se može sadržaj fajla prikazati.

```
//Ovaj program prikazuje jednostavan fajl kao ulaz i izlaz.
//Fajl je otvoren u in, out i app modu.

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
  char buffer[81];
  char file_name[81];

  cout << "Enter the name of the file you want to append to: ";
  cin.getline(file_name, 81);

  cout << endl;
  cout << "Enter the city, state, and zip code in which you live: ";
  cin.getline(buffer, 81);

  cout << endl;
```

```
  cout << "The information will be appended to your file." << endl;

  ofstream file_app(file_name, ios::app);

  if (!file_app)
  {
    cerr << endl;
    cerr << "ERROR: File could not be opened in append mode." << endl;
    cerr << "Program terminated" << endl;
    exit(1);
  }
  file_app << buffer << endl;

  file_app.close();

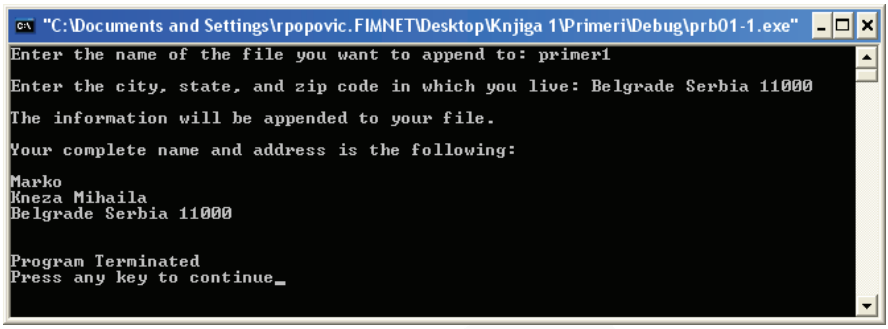
  cout << endl;
  cout << "Your complete name and address is the following:"
              << endl << endl;

  ifstream file_in(file_name);

  if (!file_in)
  {
    cerr << endl;
    cerr << "ERROR: File could not be opened." << endl;
    cerr << "Program terminated" << endl;
    exit(1);
  }

  while (!file_in.eof())
  {
    file_in.getline(buffer, 81);
    cout << buffer << endl;
  }
  file_in.close();
  cout << endl;
  cout << "Program Terminated" << endl;

  return 0;
}
```

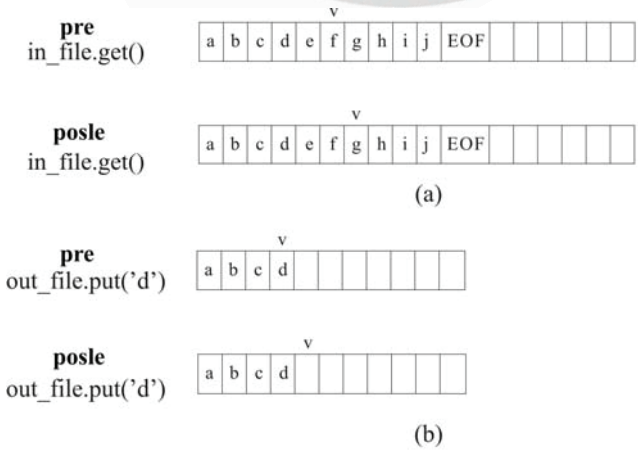


11.2 Procesiranje fajla karakter po karakter

U prethodnoj sekciji koristili smo *insertion* operator za smeštanje podataka u tekstualni fajl i *extraction* operator i metod getline() za uzimanje podataka iz tekstualnog fajla. U nekim slučajevima, ili je neophodno ili je efikasnije koristiti karakter po karakter za tekstualne fajlove.

Metodi get() i put()

Da biste učitali ili upisali jedan karakter u fajl, možete koristiti metode get() i put(). Ovi metodi su brži od korišćenja *insertion* i *extraction* operatora. Metod get() vraća jedan karakter iz ulaznog fajla. Ako je in_file otvoren ifstream objekat, in_file.get() vraća sledeći iskoristljiv objekat iz in_file toka. Metod get() uzima karakter na koji pokazuje FPI. Npr. ako je FPI 6, in_file.get() uzima karakter koji je 6 karaktera od početka fajla (tj. 7 karakter u fajlu). Zatim get() metod povećava FPI za 1 (7).



Uz to, get() metod uvek detektuje uslov za kraj fajla, dok *extraction* operator >> ne. Podsetimo se da smo koristili eof() metod za detekciju uslova za kraj fajla. Metod get() vraća EOF, konstanta definisana u iostream header fajlu, ako je program došao do kraja fajla u ulaznom fajlu.

Ako je out_file otvoren, onda out_file.put(ch) smešta kopiju karaktera uskladištenog u varijabli ch na poziciju koju određuje FPI, i tada put() metod povećava FPI za jedan. Sledeći program kopira jedan tekstualni fajl u drugi karakter po karakter.

```
//Ovaj program kopira jedan fajl u drugi koristeci get() i put()

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
 char file_name[81];
 char ch;

 cout << "Enter the name of the source file: ";
 cin.getline(file_name, 81);
 ifstream source_file(file_name);

 if (!source_file)
 {
  cout << endl;
  cout << "ERROR: File cannot be opened.";
  exit(1);
 }

 cout << endl;
 cout << "Enter the name of the target file: ";
 cin.getline(file_name, 81);

 ofstream target_file(file_name);

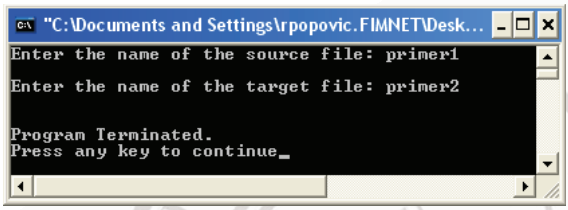
 if (!target_file)
 {
  cout << endl;
  cout << "ERROR: File cannot be opened.";
  exit(1);
 }
```

```
  while ( (ch = source_file.get()) != EOF )
    target_file.put(ch);

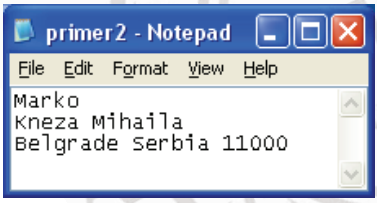
  source_file.close();
  target_file.close();

  cout << endl << endl;
  cout << "Program Terminated." << endl;

  return 0;
}
```



Na donjoj slici je dat listing fajla primer2.



Treba voditi računa o tome kako testiramo *end-of-file* u uslovu petlje while. Izraz

```
ch = source_file.get()
```

nije jednak EOF ako get() metod nije stigao do kraja fajla. Kada je stigao do kraja fajla izraz je jednak EOF i petlja se završava.

```
while ( (ch = source_file.get()) != EOF )
        target_file.put(ch);
```

Izlaz na printer

Ako znate ime pridruženo u vašem sistemu štampaču, npr. "prn" u Windows okruženju, možete da tretirate printer kao output stream. Da biste

otvorili printer za izlaz, deklarišite ofstream objekat i pridružite objekat printeru. Na primer, sledeća naredba deklariše ofstream objekat printer_file i dodeljuje ga printeru:

```
ofstream printer_file("prn");
```

Svi metodi koje smo koristili za ofstream objekte radiće i za printer_file, kao što bi i *insertion operator*. Zbog toga sledeća naredba će odštampati reč "Hello" na printeru.

```
printer_file << "Hello" << endl;
```

Sledeća naredba će odštampati karakter 'A' na printeru.

```
printer_file.put('A');
```

Sledeći program štampa sadržaje fajla kreiranog ranije sa dvostrukim proredom na printeru:

```
//Ovaj program stampa na stampacu sadrzaje fajla sa dvostrukim proredom

#include <fstream>
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
 char file_name[81];
 char ch;

 cout << "Enter the name of the source file: ";
 cin.getline(file_name, 81);

 ifstream source_file(file_name);
 if (!source_file)
 {
  cout << endl;
  cout << "ERROR: File cannot be opened.";
  exit(1);
 }

 ofstream printer_file("prn");
```

```
 if (!printer_file)
 {
  cout << endl;
  cout << "ERROR: Printer could not be opened.";
  exit(1);
 }
 printer_file << "Contents of the Source File - Double Spaced."
              << endl << endl;

 while ((ch = source_file.get()) != EOF)
 {
  printer_file.put(ch);
  if (ch == '\n')
    printer_file.put(ch);
 }

 source_file.close();
 printer_file.close();

 cout << endl << endl;
 cout << "Program Terminated." << endl;
 return 0;
}
```

Nakon otvaranja sors fajla i printer fajla, program insertuje poruku (Contents of the Source File - Double Spaced.) u printer_file strim. Zatim while petlja uspešno čita karaktere iz sors fajla i štampa ih na printeru. If naredba u petlji testira karakter koji čita get() da bi se videlo da li je znak za novi red, a ako jeste karakter se štampa ponovo, što stvara dvostruki prored.

11.3 Slučajni pristup fajlu

U prethodnim sekcijama procesirali smo fajlove sekvencijalno, jedan karakter ili zapis u jednom trenutku. Moguće je procesirati karaktere u fajlu i slučajno, odnosno program može da procesira fajl na bilo kojoj željenoj lokaciji unutar fajla.

Tekstualni i binarni fajlovi

Tekstualni fajl, na primer u C++ izvornom kodu, je niz karaktera koji povremeno uključuju znak za novi red. U nekim operativnim sistemima (Windows), kada program smešta *newline* karakter u fajl, *newline* karakter

se fizički skladišti kao dva karaktera: *carriage return* i *line feed* (CR+LF). Slično, kada program pročita CRLF par iz tekstualnog fajla, sistem ih konvertuje u jedan *newline* karakter. Mada program "vidi" samo jedan znak, *newline* znak, operativni sistem ustvari procesira CRLF par.

Kada procesiramo fajl na slučajan način, program mora da zna njegovu tačnu poziciju u fajlu. Ako pokušamo da procesiramo slučajno fajl koji je otvoren kao tekstualni fajl (što je po difoltu kada koristimo mod otvaranja) program može da izgubi poziciju u fajlu. Ovo može da nastane zbog konverzije *newline* karaktera u par CRLF. Zbog toga kada procesiramo fajl slučajno, ne treba da otvorimo fajl kao tekstualni.

Kada program otvori fajl kao binarni fajl, ni jedan karakter se ne menja kada se čita ili upisuje u fajl. Program će procesirati *newline* karakter koji je memorisan kao par CRLF, kao par karaktera umesto jednog karaktera.

Za otvaranje fajla binarno, mora da koristite mod otvaranja ios::binary zajedno sa jednim od modova koje smo ranije objasnili. Da biste ovo uradili zahteva se korišćenje operatora ili |, bitwise OR operatora. Nemojte da ga zamenite sa logičkim operatorom OR, ||. Bitwise OR radi na nivou bitova između operanada. Ako je jedan od dva bita 1, onda bitwise OR daje 1. Ako su oba bita 0, bitwise OR kao rezultat daje 0.

Na primer:

```
1010 1010      Prvi operand
0010 0111      Drugi operand
-------------
1010 1111      Bitwise OR
```

Bitwise OR se koristi da postavi (na 1) odgovarajuće bitove. Napomena, u prethodnom primeru, ako je bit prvog operanda 0, rezultat bitwise OR je kopiranje odgovarajućeg bita drugog operanda u rezultat. Sa druge strane, ako je bit prvog operanda 1, rezultat bitwise OR je 1 na odgovarajućoj poziciji rezultata, bez obzira koji je bit drugog operanda. Znači, da bi setovali bit na 1, treba primeniti bitwise OR sa 1. Primena bitwise OR sa 0 ostavlja bit nepromenjen.

Na primer, primenimo bitwise OR kada ne znamo drugi operand:

```
0010 0000      Prvi operand
????  ????      Drugi operand
------------
??1?  ????      Bitwise OR
```

Bez obzira koja je bit vrednost drugog operanda, treći bit sa leve strane mora biti 1 u rezultatu. Preostalih sedam bitova se kopira u rezultat.

Modovi otvaranja za fajlove, na primer ios::in i ios::binary, ustvari ukazuju na specifične bitove stanja varijable definisane u klasi ios.

Ilustracije radi, pretpostavimo da otvaranje fajla u ios:: in modu kao rezultat daje postavljanje drugog bita varijable na 1.

```
0100 0000
```

Takođe pretpostavimo da otvaranje fajla u ios::binary modu kao rezultat daje postavljanje petog bita na 1.

```
0000 1000
```

Sada, ako otvorimo fajl u ios::in | ios::binary modu, primena bitwise OR na dva moda daće:

```
0100 0000       ios::in
0000 1000       ios::binary
------------
0100 1000       ios::in | ios::binary
```

Fajl će biti otvoren i u ulaznom i u binarnom modu.

Sledeća naradba će zbog toga otvoriti fajl bindata.dat kao binarni ulazni fajl:

```
fstream in_file ("bindata.dat", ios::in | ios::binary);
```

Moguće je takođe otvoriti fajl koristeći tri moda otvaranja:

```
fstream iofile("bindata.dat", ios::in | ios::out | ios::binary);
```

Fajl bindata.dat je otvoren u binarnom modu i kao ulazni i kao izlazni fajl. Zbog toga, program može da čita i upisuje podatke u fajl bez potrebe da zatvori i otvori fajl između operacija.

Funkcije članice sa slučajnim pristupom

C++ I/O klase sadrže nekoliko metoda koji omogućavaju programu da pristupi fajlu slučajno. Svaki od ovih metoda radi sa fajlovim File Position

Indicator, FPI. Podsetimo se da metodi getc() i putc() određuju gde da pristupe fajlu koristeći FPI. Zbog toga, ako radimo sa FPI, možemo da čitamo ili upisujemo karaktere na bilo kojoj željenoj lokaciji u fajlu.

Sledeći metodi koji rade sa strim objektima su na raspolaganju za slučajan pristup fajlu:

```
seekg(offset, mode)     // za ulazne fajlove – pomeraj za vrednost offset-a kako
                        // je definisano modom.
seekp(offset, mode)     // za izlazne fajlove - pomeraj za vrednost offset-a kako
                        // je definisano modom.
tellg()          // za ulazne fajlove – vraca tekucu vrednost za FPI (long).
tellp()          // za izlazne fajlove - vraca tekucu vrednost za FPI (long).
```

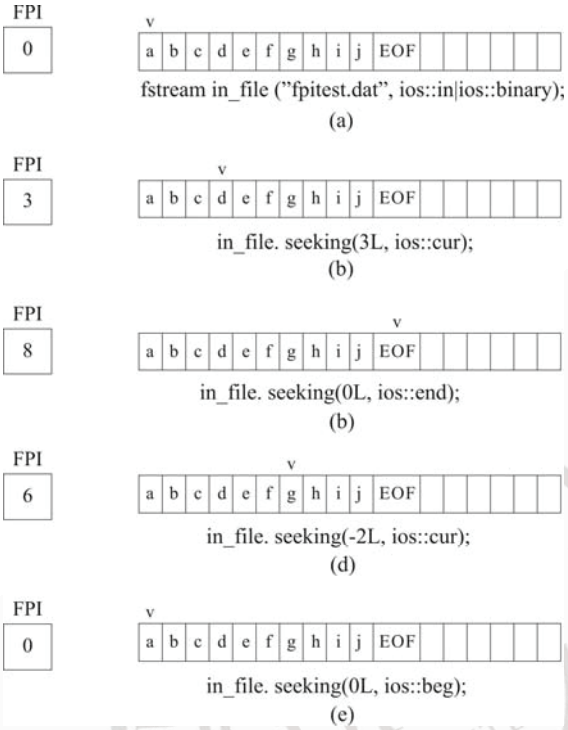
Sufiks "p" je od "put" a sufiks "g" od "get."

Pomeraj-*offset* mora biti long integer (pozitivan ili negativan). Mod može biti:

```
ios::beg        //smesta FPI na pocetak fajla.
ios::cur        // smesta FPI na tekucu poziciju u fajlu.
ios::end        // smesta FPI na kraj fajla.
```

Nekoliko primera ilustruju korišćenje ovih metoda. Slika prikazuje fajl sa osam karaktera "fpitest.dat", koji je otvoren kao binarni fajl:

```
fstream in_file("fpitest.dat", ios::in | ios::binary);
```



Otvaranje fajla u ios::in modu postavlja njegov FPI na 0 (slika a). Karet je iznad znaka na koji FPI pokazuje.

Slika b prikazuje FPI nakon izvršenja in_file.seekg(3L, ios::cur). Sufiks L prouzrokuje da računar uskladišti integer 3 kao long umesto *default* tip int. Kako je drugi argument seekg() - ios::cur, metod inicijalno ne menja vrednost FPI. Pošto je prvi argument 3L, seekg() dodaje 3 vrednosti FPI. Tako da je vrednost FPI = 3 i karakter na koji pokazuje je "d".

Slika c prikazuje kako pomeriti FPI na kraj fajla izvršavajući in_file.seekg(0L, ios::end).

Slika d prikazuje da prvi argument u seekg() može biti negativan. Naredba in_file.seekg(-2L, ios::cur) oduzima 2 od trenutne vrednosti FPI. Zato, naredba "vraća" FPI na karakter 'g'.

Konačno, slika e, naredba in_file.seekg (0L, ios::beg) postavlja FPI nazad na 0, što čini da FPI pokazuje na prvi karakter u fajlu.

Možete da koristite metod tellg() da nađete vrednost FPI u bilo kojoj tački programa. Na primer, sledeće naredbe izračunavaju veličinu fajla u bajtovima:

```
long file_size;
in_file.seekg(0L, ios::end);
file_size = in_file.tellg();
```

Sledeći program ilustruje ove koncepte. Program pita korisnika za ime fajla. Program otvara fajl, menja u velika slova njegove karaktere i zatvara fajl.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <cctype>
#include <iomanip>

using namespace std;

int main()
{
 char file_name [51];
 char ch;

 cout << "Enter the file you want to uppercase: ";
 cin.getline(file_name, 51);

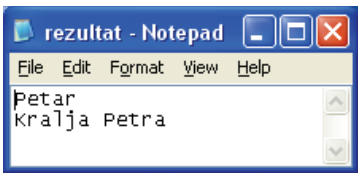
 fstream file(file_name, ios::in | ios::out | ios::binary);

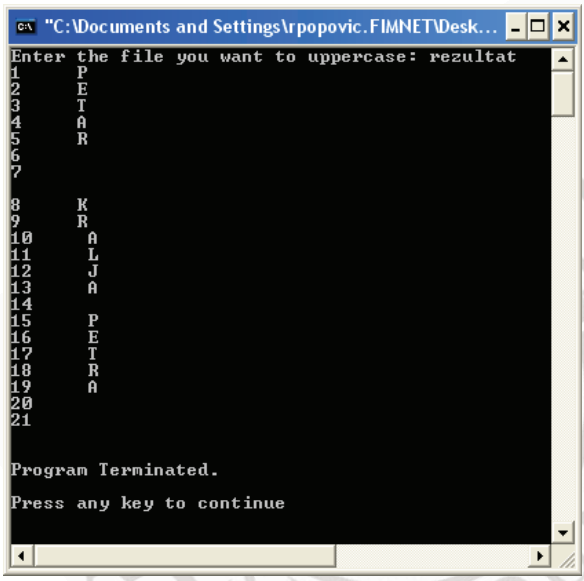
 if (!file)
 {
  cout << endl;
  cout << "ERROR: Could not open file.";
  exit(1);
 }
 while ( (ch = file.get()) != EOF)
 {
  ch = toupper(ch);
  file.seekg(-1L, ios::cur);
  file.put(ch);
  cout << file.tellg() << setw(6) << ch << endl;
 }

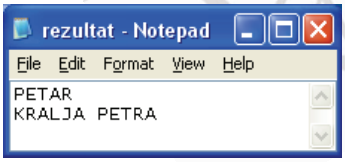
 cout << endl;
 cout << "Program Terminated." << endl;
 cout << endl;

 file.close();
```

```
 return 0;
}
```







Program počinje porukom koja je upućena korisniku da ukuca ime fajla na koji se primenjuje promena malih u velika slova. Program skladišti ime fajla u niz file_name[]. Program zatim kreira fstream objekat nazvan file, koji je otvoren kao binarni fajl i za ulaz i za izlaz. Nakon provere da li je fajl otvoren korektno, program ulazi u while petlju. Petlja while testira (gets) karakter iz fajla i testira uslov za kraj fajla. Ako se karakter učita, prva naredba u telu petlje skladišti *uppercase* verziju karaktera u varijablu ch. Zato što file.get() povećava FPI za jedan, sledeća naredba u telu petlje oduzima jedan od FPI. Ovo omogućuje da sledeća fajl operacija, file.put(ch),

smešta karakter nazad na njegovu originalnu poziciju u fajlu. Napomena, izvršenje file.put(ch) ponovo povećava FPI za jedan, tako sledeći file.get() će vratiti sledeći karakter u fajl. Poslednja naredba u telu petlje prikazuje trenutnu vrednost FPI koristeći file.tellg() i trenutnu vrednost ch.

11.4 Procesiranje binarnog fajla sekvencijalno

U jeziku C++ možemo da skladištimo i procesiramo sve podatke u zapis kao jedinicu koristeći strukturu. Na primer, možemo koristiti sledeću strukturu part record fajla za inventar, gde se svaki zapis u fajlu sastoji od tri polja: identifikacionog broja, količine i cene. Članovi strukture su polja zapisa:

```
struct Part_Record
{
  char   id_no[3];
  int    qoh;
  double price;
};
```

Part_Record struktura je u binarnoj formi.

Zato što je svako polje u strukturi Part_Record fiksne veličine (3 bajta za id_no, 4 bajta za qoh, i 8 bajtova za price), nema potrebe za odvajanjem polja razmakom. Takođe, kada upisujemo zapise u binarni sekvencijalni fajl, *newline* karakter neće da odvaja jedan zapis od drugog. Umesto toga zapisi se razlikuju po dužini zapisa. Ovo je moguće zato što su svi zapisi u fajlu iste veličine.

U ovoj sekciji prodiskutovaćemo kako da kreiramo i procesiramo binarne sekvencijalne fajlove koji su sastavljeni od niza struktura, odnosno kako da kreiramo i procesiramo sekvencijalni binarni fajl zapisa.

Kreiranje binarnog sekvencijalnog fajla zapisa

Prvo posmatrajmo binarne fajlove koji su kreirani i kojima se pristupa sekvencijalno. Pretpostavimo da imamo fajl sa podacima koji su zapisi inventara koji su kreirani ranije. Svaki zapis je linija u polju. Svaka linija se sastoji od broja dela iz dva bita, od količine u formatu celog broja i jedinične cene predstavljene u formatu decimalne vrednosti. Blanko prostor odvaja jedno polje od drugog. Sledi primer fajla sa popisom inventara.

```
23   145    24.95
15   46     67.99
65   230    11.95
56   37     73.95
59   1074    3.95
12   26     104.99
84   19     55.95
03   81     18.95
44   39     31.95
```

Želimo da napišemo program koji čita zapise iz ovog fajla i upisuje ih u binarni sekvencijalni fajl. Pretpostavimo da su zapisi iz binarnog sekvencijalnog fajla Part_Record strukture, kao što je opisano ranije. Jedan način da iščitamo zapise iz ulaznog tekst fajla je korišćenje operatora ekstrakcije <<. Kako treba da dobijemo tri polja iz fajla za svaki zapis, koristimo funkciju Read_Part_Record() da bismo pročitali zapis iz fajla. Definicija funkcije je sledeća:

```
void Read_Part_Record(ifstream& ifs, Part_Record& pr)
 {
  ifs >> pr.id_no;
  ifs >> pr.qoh;
  ifs >> pr.price;
 }
```

Funkcija ima dva argumenta. Prvi je referenca na ifstream objekat. Zato, možemo da prosledimo input file object name kao prvi argument. Drugi argument je referenca na strukturu Part_Record. Funkcija čita tri uzorka podataka iz ifstream objekta, ifs, i smešta ih u strukturu sa tri člana. Program može zatim da radi sa part strukturom.

Pretpostavimo sledeće deklaracije:

```
Part_Record part;
ifstream input_file(input_file_name);
```

Da bismo čitali podatke iz ulaznog fajla, možemo da izvršimo sledeću naredbu, koja čita zapis iz input_file i smešta kopiju zapisa u strukturu part.

```
Read_Part_Record(input_file, part);
```

Sada kada možemo da čitamo podatke iz ulaznog fajla, želimo da ih upišemo u binarni fajl kao strukturu. Da bismo ovo uradili zahteva se metod write().

Kastujemo pokazivač na strukturu kao (char*) iz sledećeg razloga. Programeri koji su pisali kod za metod write() nisu mogli da znaju tip podatka prvog argumenta. Prvi argument može da bude pokazivač na bilo koju strukturu koju izaberete. Da biste dozvolili da metod write() bude korišćen sa bilo kojim pokazivačem, metod pretpostavlja da njegov prvi argument bude karakter pointer. Onda, da bi koristio metod, programer prosto kastuje prvi argument kao karakter pointer.

Operator sizeof() (nije funkcija) vraća broj bajtova koji su potrebni za skladištenje promenljive tipa koji je njegov argument. Koristite operator sizeof() na isti način kao što koristite funkciju. Smestite ime tipa podatka (kao jedan od ugrađenih C++ tipova ili korisnički definisanih tipova) u zagrade nakon sizeof(). Vrednost koju vraća operator je broj bajtova potrebnih za skladištenje varijable tog tipa. Na primer, sizeof(int) je broj bajtova potrebnih da uskladišti ceo broj u sistem u kome je funkcija određena. Na Windows ili UNIX/Linux sistemu, sizeof(int) je 4.

Metod write(), koji može da se primeni na bilo koji objekat izlaznog fajla otvoren u binarnom modu, ima sledeću sintaksu:

```
write( (char*) ptr, item_size)
```

U prethodnoj sintaksi, ptr je pokazivač na podatke koji treba da budu upisani. Pokazivač mora da se kastuje kao karakter pointer zato što će u opštem slučaju on pokazivati na istu vrstu strukture. Drugi argument item_size je broj bajtova koji treba da se upišu, koji je obično veličina strukture koja se upisuje u fajl. Koristite operator sizeof().

Da bi se upisala kopija promenljive part strukture u fajl input_file, treba da se izvrši sledeća naredba:

```
input_file.write( (char*) &part, sizeof(Part_Record) );
```

Prvi argument, &part je adresa prostora u memoriji koji treba da se kopira u fajl. Kao što se traži od metoda, (char*) kastuje Part_Record pokazivač kao karakter pointer. Drugi argument je broj bajtova koji se kopiraju.

Sledeći program čita zapise iz tekstualnog fajla i upisuje strukture u binarni sekvencijalni fajl zapisa.

```
//Ovaj program kreira sekvencijalni fajl od struktura iz tekstualnog fajla podataka
//On koristi operator >> za dobijanje podataka iz tekstualnog fajla i koristi write()
//metod za upis struktura u izlazni fajl.

#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

struct Part_Record
{
 char   id_no[3];
 int    qoh;
 double price;
};
void Read_Part_Record(ifstream&, Part_Record&);

int main()
{
 int record_count = 0;

 char input_file_name[51];
 char output_file_name[51];
 Part_Record part;

 cout << "Enter the name of the text file: ";
 cin.getline(input_file_name, 51);

 ifstream input_file(input_file_name);

 if (!input_file)
 {
  cout << endl;
  cout << "ERROR: Could not open text file.";
  exit(1);
 }

 cout << endl;
 cout << "Enter the name of the sequential file: ";
 cin.getline(output_file_name, 51);

 ofstream output_file(output_file_name, ios::binary);

 if (!output_file)
 {
```

```
  cout << endl;
  cout << "ERROR: Could not open sequential file: ";
  exit(1);
 }

 //Cita prvi zapis

 Read_Part_Record(input_file, part);

 while (!input_file.eof())
 {
  output_file.write( (char *)&part, sizeof(Part_Record) );
  ++record_count;

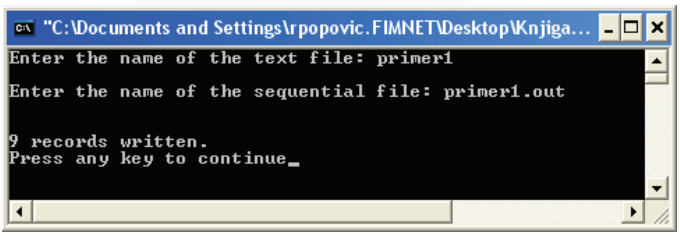
  Read_Part_Record(input_file, part);
 }
 cout << endl << endl << record_count << " records written.";

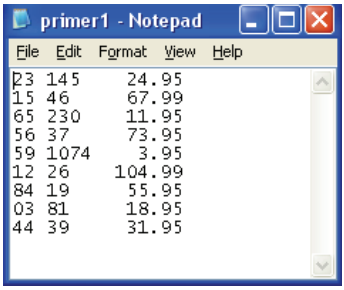
 input_file.close();
 output_file.close();

 cout << endl;

 return 0;
}

void Read_Part_Record(ifstream& ifs, Part_Record& pr)
{
 ifs >> pr.id_no;
 ifs >> pr.qoh;
 ifs >> pr.price;
}
```





U programu prvo deklarišemo brojač record_count, koji broji zapise koje program procesira. Dva niza karaktera memorišu sistemska imena ulaznog i izlaznog fajla. Deo part strukture Part_Record je deo gde program smešta podatke. Program zatim dobija imena ulaznog i izlaznog fajla i otvara pridružene fajl objekte. Obratite pažnju da se ulazni fajl otvara, po *default*-u, kao tekstualni fajl, a izlazni se otvara kao binarni fajl.

Kada se čita fajl sekvencijalno, morate da vodite računa da vaš program procesira poslednji ulazni zapis, i da ga procesira samo jednom.

Programski kôd prethodnog primera sledi ove napomene. Nakon otvaranja ulaznog fajla, program čita prvi zapis koristeći funkciju Read_Part_Record(). Program prosleđuje kao reference argumente funkciji ime fajla i deo part strukture Part_Record. U telu funkcije, operator ekstrakcije dobija tri polja i smešta svako u odgovarajući podatak član strukture. Petlja while zatim testira uslov za kraj fajla. Ako nije kraj fajla, telo petlje izvršava i procesira zapis. Procesiranje je prosto upis kopije zapisa u sekvencijalni izlazni fajl i povećanje brojača zapisa. Zatim se fajl čita ponovo. Program napušta petlju kada se ispuni uslov za kraj fajla. Program na kraju prikazuje broj zapisa koji su procesirani i zatvara fajlove.

12. Specijalne teme: Prijateljske funkcije, preklapanje operatora, makroi i inline funkcije

12.1 Prijateljske (friend) funkcije

Do sada smo videli da su metodi klasa jedine funkcije koje imaju pristup privatnim instancama promenljivih te klase. Međutim, ponekad je poželjno da funkcija koja nije metod te klase ima pristup privatnim instancama promenljivih date klase.

Definisanje prijateljske funkcije

Pretpostavimo da u sistemu koji koristi Savings_Account klasu želimo da primenimo funkciju koja prebacuje novac sa jednog štednog računa na drugi. Moguće rešenje bi bilo da se napiše Savings_Account metod, Transfer_Method. Metod bi imao dva argumenta: prvi bi bio račun sa koga prebacujemo određenu količinu, a drugi argument bi bio količina para za transfer. Račun na koji primenjujemo metod bi bio račun na koji prebacujemo sredstva. Na primer, ako su acc1 i acc2 štedni računi, sledeća naredba bi izvršila transfer 100.00 sa acc2 na acc1

```
acc1.Transfer_Method(acc2, 100.00);
```

Drugi pristup bio bi definisanje funkcije koja nije član Transfer1() koja ima tri argumenta: prvi je prijemna sredstva, drugi sredstva koja se predaju i treći sredstva za transfer. Tako bismo za transfer 100.00 sa acc2 na acc1, napisali sledeću naredbu:

```
Transfer1(acc1, acc2, 100.00);
```

Koristićemo metode klase Savings_Account da bismo napisali kod za takvu funkciju. Funkcija vraća true ako je transfer uspešan i false ako nije.

```
bool Transfer1(Savings_Account acc1, Savings_Account acc2, double amount)
{
  if (!acc2.Withdraw(amount))
    return false;

  acc1.Deposit(amount);
```

```
  return true;
}
```

Podsetimo se da metod Withdraw() vraća true ako je podizanje sa računa moguće. Naredba if u funkciji transfer1() vraća false ako podizanje nije izvršeno. Smetnja za primenu funkcije za transfer je da funkcija zahteva izvršenje dve druge funkcije Withdraw() i Deposit(). Funkcija bi bila efikasnija kada bi mogla direktno da pristupi instancama varijabli parametara acc1 i acc2.

Rešenje je kreirati „prijateljsku“ friend funkciju. Prijateljska funkcija klase ima ista prava pristupa kao metodi klase, ali sama nije metod. Sledi, prijateljska funkcija klase ima pristup svim privatnim članovima klase kao da je metod te klase. Važno je znati da se u klasi mora deklarisati funkcija da bude prijateljska. Ovo narušava princip skrivanja podataka ako bilo koja funkcija može da deklariše sebe kao friend funkciju te klase!

Da bismo deklarisali funkciju da bude prijateljska funkcija klase treba uraditi sledeće:

-Deklarisati funkciju unutar deklaracije klase, gde prethodi ključna reč friend povratnom tipu funkcije. Nije važno u kojoj kategoriji pristupa je definisana funkcija (public ili private).

-Kada se definiše funkcija izvan klase, ne treba ponoviti rezervisanu reč friend. Rezervisana reč friend se može pojaviti u deklaraciji klase.

U deklaraciji klase Savings_Account napisaćemo sledeći kod:

```
class Savings_Account
{
  friend bool Transfer(Savings_Account&, Savings_Account&, double);

  private:
  .
  .
  .
  public:
  .
  .
  .
};
```

Treba obratiti pažnju da smeštamo deklaraciju friend funkcije pre svakog specifikatora pristupa. Prosleđujemo dva štedna računa funkciji po referenci zato što funkcija mora da promeni oba Account objekta koji su prosleđeni kao argumenti. Ako prosledimo Savings_Account objekte po vrednosti, promene koje funkcija napravi biće urađene na parametrima funkcije (to su kopije argumenata) a ne na samim argumentima. Sledi kod za funkciju:

```
bool Transfer(Savings_Account& acc_r1, Savings_Account& acc_r2, double amount)
{
  //Ako iznos prelazi stanje na racunu „acc_r2“, ne radi nista i vrati „false“

  if (amount > acc_r2.balance)
    return false;

  //U suprotnom, izvrsi transfer i vrati  "true"

  acc_r2.balance = amount;  //smanji „acc_r2 balance“ prema iznosu
  acc_r1.balance += amount;  //povecaj „acc_r1 balance“ prema iznosu

  return true;
}
```

Ova funkcija je nešto duža od funkcije transfer1(), ali je efikasnija jer ne mora da izvršava funkcije kako bi izvršila svoj zadatak. Napomena, možemo direktno da pristupimo private podacima članovima sa oba računa zato što je transfer() friend funkcija klase Savings_Account. Mora da koristimo operator tačka (*dot* .) kako bi pristupili članovima podacima oba parametra zato što Transfer() nije metod klase.

Sledi kompletan program koji koristi transfer() funkciju. Da bismo uprostili program koji smo ranije napravili, treba da uklonimo Calc_Interest(), Deposit(), i Withdraw() metode.

```
//Ovaj program prikazuje rad sa friend funkcijama.

//Da bismo uprostili program uklanjamo funkcijeCalc_Interest(), Deposit() i Withdraw().

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

```
using namespace std;

class Savings_Account
{
 friend bool Transfer(Savings_Account&, Savings_Account&, double);

 private:
   char    id_no[5];
   char*   name;
   double  balance;
   double  rate;
 public:
   Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
   ~Savings_Account();
   const char* Get_Id() const;
   const char* Get_Name() const;
   double Get_Balance();
};

Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{
 strcpy(id_no, id);  //kopira prvi argument u id_no[]

 name = new char[strlen(n_p) + 1]; //kreira prostor za name

 strcpy(name, n_p);  //kopira drugi argument u novi prostor

 balance = bal;
 rate    = rt;
}
Savings_Account::~Savings_Account()
{
 cout << endl << endl;
 cout << "Account " << id_no << " terminated." << endl;
 delete [] name;
}
const char* Savings_Account::Get_Id() const
{
 return id_no;
}
const char* Savings_Account::Get_Name() const
{
 return name;
}
double Savings_Account::Get_Balance()
{
```

```
  return balance;
 }
 bool Transfer(Savings_Account& acc_r1, Savings_Account& acc_r2, double
amount)
 {
  //Ako iznos prelazi stanje na racunu acc_r2, ne radi nista i vrati 0

  if (amount > acc_r2.balance)
   return false;

  //U suprotnom, prenesi sredstva i vrati 1

  acc_r2.balance = amount;  //umanjuje stanje acc_r2 za odredjeni iznos
  acc_r1.balance += amount;  //uvećava stanje na acc_r1 za odredjenu kolicinu

  return true;
 }

 void Display_Account(Savings_Account&);    //Prototip funkcije

 int main()
 {
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);

  double amount;

  Savings_Account acc1("1111", "Adam Zapple", 100.00, 0.08);
  Savings_Account acc2("2222", "Poly Ester", 200.00, 0.06);
  Display_Account(acc1);
  Display_Account(acc2);

  cout << endl;
  cout << "Enter an amount to transfer from the second account "
       << "to the first: ";
  cin >> amount;
  if (Transfer(acc1, acc2, amount))
  {
   cout << endl;
   cout << "Transfer made. Updated Account Information:" << endl;
   Display_Account(acc1);
   Display_Account(acc2);
  }
  else
  {
```

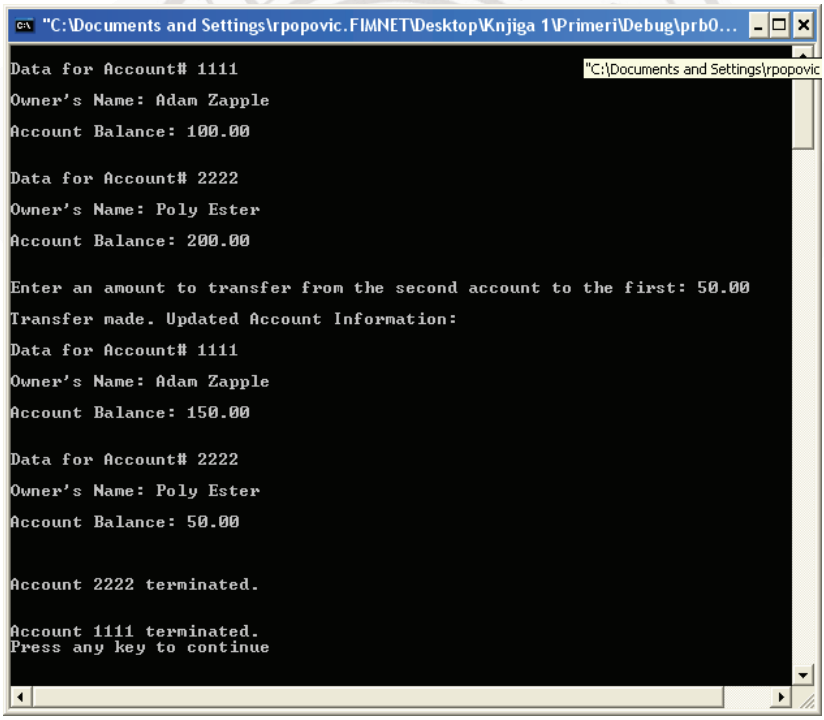
```
   cout << endl;
   cout << "Transfer could not be made." << endl;
  }

 return 0;
}

void Display_Account(Savings_Account& acc)
{
 cout << endl;
 cout << "Data for Account# " << acc.Get_Id() << endl << endl;

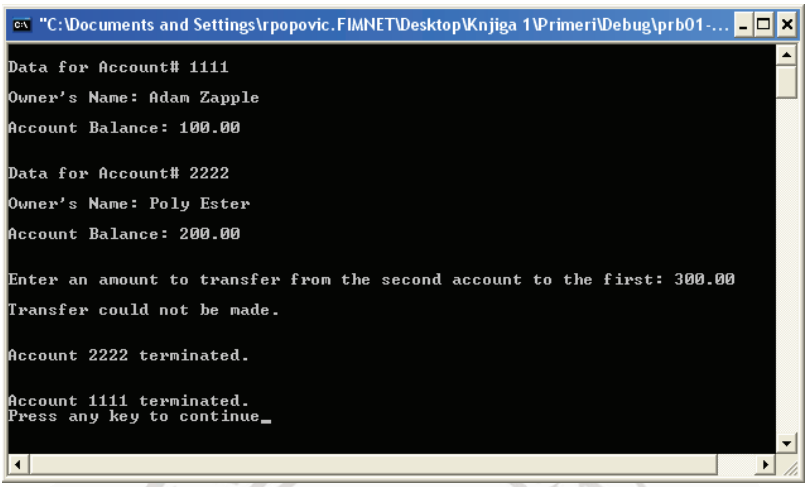
 cout << "Owner's Name: " << acc.Get_Name() << endl << endl;

 cout << "Account Balance: " << acc.Get_Balance() << endl << endl;
}
```



Obratite pažnju na način kako glavna funkcija main() koristi funkciju Transfer(). Naredba if testira uslov da li je transfer bio uspešan. Ako jeste program prikazuje na displeju poruku i nove informacije o štednom računu. Ako transfer nije uspešan, program prikazuje odgovarajuću poruku. Sledi

izvršenje programa u kome tražimo transfer više novca sa prvog računa nego što je na računu.



Nasleđivanje i prijateljske funkcije

Prijateljske funkcije se ne nasleđuju u hijerarhiji klasa. Ako je funkcija "prijatelj" osnovne klase u hijerarhiji klasa, ona nije "prijatelj" bilo kojoj izvedenoj klasi. Ako želite da takva funkcija bude prijatelj izvedenoj klasi, mora da je deklarišete kao takvu u izvedenoj klasi.

12.2 Preklapanje osnovnih aritmetičkih operatora

Videli smo u poglavlju da C++ dozvoljava preklapanje konstruktora. C++ takođe dozvoljava programeru preklapanje operatora. Možemo da proširimo značenje operatora sabiranja +, operatora insertovanja <<, i većine drugih C++ operatora. Prikazaćemo sada kako se preklapaju osnovni aritmetički operatori, drugi operatori dodele, insertovanja i ekstrakcije, kao i relacioni operatori.

Sabiranje dva objekta

Razmotrimo klasu Test_Class, koju smo koristili u prethodnom poglavlju.

```
class Test_Class
{
 private:
   int n;

 public:
   Test_Class(int i = 0);       //Konstruktor sa jednim argumentom
   Test_Class(Test_Class&);     //Konstruktor kopije
   ~Test_Class();
   int Get_Value();
};

Test_Class::Test_Class(int i)
{
 n = i;

 cout << endl;
 cout << "Constructor Executed: Data member initialized to "
      << n << endl;
}

Test_Class::Test_Class(Test_Class& tc_r)
{
 n = tc_r.n;
 cout << endl << endl;
 cout << "Copy constructor executed: "
      << "Data member initialized to " << n << endl;
}

Test_Class::~Test_Class()
{
 cout << endl << endl;
 cout << "Destructor Executed for object with data member "
      << n << endl;
}

int Test_Class::Get_Value()
{
 return n;
}
```

Pretpostavimo da smo napravili sledeće deklaracije u glavnoj funkciji main().

```
Test_Class tc_object1(4);
Test_Class tc_object2(7);
```

```
Test_Class tc_object3;
```

Test_Class objekat ima samo jedan podatak član, ako želimo da napravimo sledeću dodelu:

```
tc_object3 = tc_object1 + tc_object2;
```

operacija dodele treba da se proširi tako da uključuje i dodelu dva objekta tipa Test_Class drugom objektu tipa Test_Class. Trebalo bi da operator dodeli vrednost 11 (što je suma vrednosti podataka članova objekata tc_object1 i tc_object2) podatku članu tc_object3. Možemo usvari proširiti značenje sabiranja kao u ovom primeru preklapanjem + operatora.

Operator može biti preklopljen kao metod klase ili kao friend funkcija.

Jedan preklopljen operator koji menja jedan od svojih operanada treba da bude definisan kao metod klase. Svi drugi preklopljeni operatori treba da budu definisani kao prijateljske funkcije. Međutim, sledeći operatori mora da budu preklopljeni kao metodi klase:

dodela (=), funkcijski poziv (()), indeksi nizova ([]), operator indirekcije člana (->).

Sledi, operatori kao što su: sabiranje, oduzimanje, proizvod i deljenje, kao i operatori insertovanja i brisanja, treba da budu deklarisani kao prijateljske funkcije. Operatori kao što su složena dodela (+=, - =, i tako dalje) treba da budu deklarisani kao metodi klase.

Počećemo od operatora koji su definisani kao prijateljske funkcije. Takvi operatori uključuju aritmetičke operatore i *insertion* i *extraction* operatore. Da biste deklarisali preklopljen friend operator, koristite sledeću sintaksu:

```
friend povratni-type operator simbol-operatora (lista_argumenata);
```

Ključna reč operator je obavezna i mora da bude pre simbola operatora. Na primer, ovako izgleda deklaracija preklopljenog operatora sabiranja za klasu Test_Class, koja mora da se pojavi u deklaraciji klase:

```
friend Test_Class operator + (const Test_Class&, const Test_Class&);
```

Dva argumenta u funkciji predstavljaju dva operanda (respektivno) u binarnoj operaciji sabiranja. Kod preklapanja binarnog operatora kao friend funkcije, prvi argument preklapanja uvek predstavlja levi operand a drugi argument uvek predstavlja desni operand.

Sledi kôd preklopljenog operatora sabiranja:

```
Test_Class operator + (const Test_Class& left_op, const Test_Class& right_op)
{
  Test_Class temp_obj;     // Uzima „default“ vrednost 0 za n

  temp_obj.n = left_op.n + right_op.n;

  return temp_obj;
}
```

koristimo imena left_op i right_op za prvi i drugi argument zato što oni predstavljaju levi i desni operand operatora. Ako pišemo kôd za izraz tc_object1 + tc_object2, tc_object1 je vrednost left_op zato što je on levo od operatora, i tc_object2 je vrednost right_op zato što je on desno od operatora.

Preklopljen operator sabiranja vraća Test_Class objekat. Zato, prva naredba u funkciji deklariše privremeni objekat Test_Class, temp_obj, čiju vrednost funkcija vraća pozivajućem programu. Sledeća naredba u funkciji dodeljuje vrednosti instance varijable n privremenog objekta temp_obj, zbir instanci varijable n left_op i right_op. Konačno, funkcija vraća kopiju temp_obj pozivajućem programu.

Sabiranje objekta i celog broja

Posmatrajmo primer sabiranja obične celobrojne konstante i Test_Class objekta:

```
Test_Class tc_object1(4);
Test_Class tc_object2;

tc_object2 = tc_object1 + 15;
```

Bilo bi razumno da se dodeli 19 (zbir n instance varijable objekta tc_object1 i celobrojne konstante 15) instanci promenljive n objekta tc_object2. Sledi deklaracija i kod preklopljenog operatora sabiranja:

```
friend Test_Class operator + (const Test_Class&, const int);
```

```
Test_Class operator + (const Test_Class&  left_op, const int right_op)
{
 Test_Class temp_obj;

 temp_obj.n = left_op.n + right_op;

 return temp_obj;
}
```

Na osnovu prethodnog koda nije ispravno pisati:

```
tc_object2 = 7 + tc_object1;
```

Sledi kompletan program koji koristi dva preklapanja operatora sabiranja. Treba obratiti pažnju da smo dali komentare cout naredbe u konstruktoru, konstruktoru kopije i destruktoru kako bi izlaz bio jednostavniji.

```
//Ovaj program ilustruje preklapanje operatora sabiranja

#include <iostream>

using namespace std;

class Test_Class
{

 friend Test_Class operator + (const Test_Class&, const Test_Class&);
 friend Test_Class operator + (const Test_Class&, const int);

 private:
  int n;

 public:
  Test_Class(int i = 0); //Konstruktor sa jednim argumentom
  Test_Class(Test_Class&); //Konstruktor kopije
  ~Test_Class();
  int Get_Value();
};

Test_Class::Test_Class(int i)
{
 n = i;
```

```
  //cout << endl;
  //cout << "Constructor Executed: Data member initialized to "
  //     << n << endl;
}

Test_Class::Test_Class(Test_Class& tc_r)
{
  n = tc_r.n;

  //cout << endl << endl;
  //cout << "Copy constructor executed: "
  //     << "Data member initialized to " << n << endl;
}

Test_Class::~Test_Class()
{
  //cout << endl << endl;
  //cout << "Destructor Executed for object with data member "
  //     << n <<endl;
}

int Test_Class::Get_Value()
{
  return n;
}

Test_Class operator + (const Test_Class& left_op, const Test_Class& right_op)
{
  Test_Class temp_obj;

  temp_obj.n = left_op.n + right_op.n;

  return temp_obj;
}

Test_Class operator + (const Test_Class& left_op, const int right_op)
{
  Test_Class temp_obj;

  temp_obj.n = left_op.n + right_op;

  return temp_obj;
}

int main()
{
```

```
    Test_Class tc_object1(4);
    Test_Class tc_object2(7);
    Test_Class tc_object;

    cout << endl << endl;
    cout << "Prior to adding the objects, the value of tc_object is "
        << tc_object.Get_Value() << endl;

    tc_object = tc_object1 + tc_object2;

    cout << endl << endl;
    cout << "After adding the objects, the value of tc_object is "
        << tc_object.Get_Value() << endl;

    tc_object = tc_object1 + 15;

    cout << endl << endl;
    cout << "After adding 15 to tc_object1, the value of tc_object is "
        << tc_object.Get_Value() << endl;

    return 0;
}
```

Program Output

```
Prior to adding the objects, the value of  tc_object is 0

After adding the objects, the value of tc_object is 11

After adding 15 to tc_object1, the value of  tc_object is 19
```

Objekat tc_object počinje time da instanca promenljive n ima *default* vrednost 0. Nakon sabiranja objekata tc_object1 (vrednost instance promenljive je 4) i tc_object2 (vrednost instance promenljive je 7) i dodele rezultata objektu tc_object, vrednost instance promenljive objekta tc_object je 11. Konačno, nakon dodavanja celobrojne vrednosti 15 objektu tc_object1 (čija je vrednost instance promenljive 4) i dodele rezultata objektu tc_object, instanca varijable objekta tc_object ima vrednost 19.

Treba obratiti pažnju da preklapanje operatora sabiranja kao rezultat daje da su naredbe kao što je sledeća moguće:

```
tc_object = tc_object1 + tc_object2 + tc_object3 + 6;
```

Prvo sabiranje kao rezultat daje privremeni objekat Test_Class, koji se onda dodaje objektu tc_object3, gde se opet stvara objekat Test_Class. Treće sabiranje onda dodaje ovaj privremeni objekat celom broju 6 i kao rezultat daje novi privremeni objekat, koji se dodeljuje objektu tc_object.

Pravila preklapanja operatora

Sledeća pravila mora da se uzmu u obzir pri preklapanju operatora.

- Svi operatori se mogu preklopiti osim sledećih:
  .pristup članu (dot operator)
  :: razrešenje dosega
  ?: *condicional*
  sizeof operator veličine
  new dinamička alokacija
  delete dealokacija
  .* pristup članu
- Sledeći operatori moraju biti metodi klasa:
  = dodela
  () funkcijski poziv
  [] članovi niza
  -> indirektan pristup članu
- Ne možete da menjate unarnu/binarnu prirodu operatora. Sledi, deljenje mora da ostane binarni operator. Ako je jedan operator i unarni i binarni (na primer +), onda on mora da bude preklopljen ili na jedan ili na drugi ili na oba načina.
- Ne možete da preklopite unapred definisana pravila prioriteta.

Preklapanje operatora dodele i this pokazivača

U ovoj sekciji, objasnićemo kako da preklopite operator dodele. Da bismo preklopili operator dodele, na isti način kao i dodelu ugrađenih tipova, mora da koristimo specijalan pokazivač this, koji je na raspolaganju svakom metodu klase.

Preklapanje operatora dodele

Spomenuli smo više puta da možemo da dodelimo jedan objekat drugom objektu iste klase. Pri tome se dodela izvodi član-po-član (*member-*

by-member). Ovaj tip dodele je odgovarajući za objekte Test_Class, koji imaju jednu celobrojnu instancu promenljive. Međutim member-by-member kopiranje nije odgovarajuće za klasu koja ima jedan ili više pokazivača kao instance promenljivih. Na primer, neka su acc1 i acc2 objekti klase Savings_Account, koji imaju name instancu promenljive koja pokazuje na ime osobe koja je vlasnik štednog računa. Pretpostavimo da je dodela acc1 = acc2. Kako je kopija načinjena na osnovu *member-by-member*, ime članova oba objekta pokazuju sada na ime pridruženo objektu acc2. Na taj način oba pokazuju na isti fizički prostor u hip memoriji. Sledi da promena imena ili štednog računa će prouzrokovati da se ta promena pojavi u oba. Da bismo izbegli ovo mora da preklopimo operator dodele.

Treba da napišete kod za preklapanje operatora na takav način da se preklopljeni operator ponaša kao odgovarajući tipovi podataka. Za preklopljen operator dodele, ovo znači da sledeće dodele imaju smisla za objekte klase Savings_Account.

```
acc1 = acc2 = acc3;
```

Podsetimo se da u C++, svaki izraz ima vrednost. Vrednost dodele je leva strana operatora dodele. Kako se dodela izvršava sa desne strane na levu vrednost prve dodele acc2 = acc3, je objekat acc2. Zatim druga dodela pridružuje objekat acc2 objektu acc1. Kako bismo načinili da se preklopljeni operator dodele ponaša kao običan operator dodele, želimo da i sledeći izraz radi korektno:

```
(acc1 = acc2) = acc3;
```

U ovom slučaju leva dodela se prvo izvršava. Rezultat je objekat acc1, kome se onda dodeljuje vrednost objekta acc3.

Da bismo učinili da višestruke dodele rade na način koji je nama pogodan, naša preklapanja mora da vrate objekat na koji je operator primenjen. Zato, operator mora da vrati reference na prvi argument, a ne kopiju prvog argumenta.

Dodela je binarni operator, ima levi i desni operand. Za razliku od slučaja kada preklapamo operator sabiranja, operacija dodele menja levi operand. Zato, najbolje je razmišljati o dodeli kao operaciji na levom operandu.

Kada preklapamo binarni operator kao metod klase, objekat na koji se funkcija primenjuje predstavlja levi operand i jedan argument operatora predstavlja desni operand.

Sledi deklaracija operatora. Jedan argument preklopljenog operatora je desna strana operatora dodele. Leva strana operatora dodele je objekat na koji je dodela primenjena.

```
Savings_Account& operator = (const Savings_Account&);
```

Pokazivač this

Svaki objekat ima svoje sopstvene kopije instanci promenljivih date klase, ali postoji samo jedna kopija svakog metoda klase za sve objekte koje program može da kreira. Metod zna na koji objekat se primenjuje u smislu specijalnog pokazivača koji sistem automatski obezbeđuje i koji možete da koristite kako vama odgovara. Unutar koda bilo kog metoda klase, pokazivač this, koji je rezervisana reč, uvek upućuje na objekat na koji je metod primenjen. Kada C++ kompajler prevodi kôd metoda, this -> prethodi svakoj referenci na instancu varijable. Mada nije neophodno, možete da smestite izraz this -> pre svake instance promenljive name u metodu i program će se kompajlirati i izvršiti korektno. Na primer, za konstruktor klase Test_Class:

```
Test_Class::Test_Class(int i)
{
  n = i;

  //cout << "\nConstructor Executed: Data member initialized to " << n;
}
```

možemo da zamenimo prethodni kôd sledećim ekvivalentnim kodom:

```
Test_Class::Test_Class(int i)
{
  this -> n = i;

  //cout << "\nConstructor Executed: Data member initialized to " << n;
}
```

Unutar koda metoda klase, *this predstavlja objekat na koji se metod primenjuje.

Sledi kôd za preklopljeni operator dodele.

```
Savings_Account& Savings_Account::operator = (const Savings_Account&
right_op)
{
 if (&right_op == this)
   ;                //ne radi nista
 else
 {
   strcpy(id_no, right_op.id_no); //kopira  id desnog operanda right_op u id_no[]
   delete [] name;

   name = new char[strlen(right_op.name) + 1]; //kreira prostor za  name

   strcpy(name, right_op.name);  //kopira drugi argument u novi prostor
   balance = right_op.balance;
   rate    = right_op.rate;
 }
 return *this;
}
```

Uslov u if naredbi testira da li je desni operand u dodeli jednak prvom operandu. Na taj način, proveravamo takozvano samo dodeljivanje kao što je acc1 = acc1. Ako je ovaj uslov true, ništa se ne radi, pa se null naredba izvršava. Inače, članovi sa desne strane operanda su kopije leve strane operanda na isti način kao što je dato u kodu konstruktora i konstruktora kopije.

Treba da obratite pažnju da je neophodno da dealocirate prostor na koji pokazuje ime name instance promenljive levog operanda. Ako ovo ne uradite ostaje alociran prostor koji nije na raspolaganju programu. Program koji sledi testira preklapanje operatora dodele.

```
//Ovaj program prikazuje overloading operatora dodele.

#include <iostream>
#include <iomanip>
#include <string>

using namespace std;
class Savings_Account
{
 private:
   char   id_no[5];
   char*  name;
```

```
    double balance;
    double rate;
  public:
    Savings_Account(char id[], char* n_p, double bal = 0.00, double rt = 0.04);
    Savings_Account(const Savings_Account &);
    ~Savings_Account();
    double Calc_Interest();
    double Get_Balance();
    const char* Get_Id() const;
    const char* Get_Name() const;
    void Deposit(double);
    bool Withdraw(double);

    Savings_Account& operator = (const Savings_Account&);
};
Savings_Account::Savings_Account(char id[], char* n_p, double bal, double rt)
{

  strcpy(id_no, id);  //kopira prvi argument u id_no[]

  name = new char[strlen(n_p) + 1]; //kreira prostor za  name

  strcpy(name, n_p);  //kopira drugi argument u novi prostor

  balance = bal;
  rate    = rt;
}
Savings_Account::Savings_Account(const Savings_Account& acc_r)
{
  strcpy(id_no, acc_r.id_no);  //kopira prvi argument u id_no[]

  name = new char[strlen(acc_r.name) + 1]; //kreira prostor za name

  strcpy(name, acc_r.name);  //kopira drugi argument u novi prostor

  balance = acc_r.balance;
  rate    = acc_r.rate;
}
Savings_Account::~Savings_Account()
{
  delete [] name;
}
double Savings_Account::Get_Balance()
{
  return balance;
}
```

```
double Savings_Account::Calc_Interest()
{
  double interest;

  interest = balance * rate;
  balance += interest;

  return interest;
}
const char* Savings_Account::Get_Name() const
{
  return name;
}
const char* Savings_Account::Get_Id() const
{
  return id_no;
}
void Savings_Account::Deposit(double amount)
{
  balance += amount;
}
bool Savings_Account::Withdraw(double amount)
{
  bool result;

  if (amount <= balance)
  {
    balance = amount;
    result = true;
  }
  else
    result = false;

  return result;
}
Savings_Account & Savings_Account::operator = (const Savings_Account &
                                                    right_op)
{
  if (&right_op == this)
        ;                  //ne radi nista
  else
  {
    strcpy(id_no, right_op.id_no);  //kopira  id desnog operanda right_op u id_no[]

    delete [] name;
    name = new char[strlen(right_op.name) + 1]; //kreira prostor za name
```

```
   strcpy(name, right_op.name);  //kopira drugi argument u novi prostor

   balance = right_op.balance;
   rate    = right_op.rate;
  }
  return *this;
}
Savings_Account Get_Account();    //Prototip za funkciju Get_Account()

void Display_Account(Savings_Account&);      //Prototip za funkciju
                                             //Display_Account()

int main()
{
  cout << setprecision(2)
       << setiosflags(ios::fixed)
       << setiosflags(ios::showpoint);
  Savings_Account acc1 = Get_Account();

  cout << endl;

  Display_Account(acc1);

  Savings_Account acc2 = Get_Account();

  cout << endl;

  Display_Account(acc2);

  acc1 = acc2;

  cout << endl << endl;
  cout << "After the assignment:" << endl << endl;

  Display_Account(acc1);
  Display_Account(acc2);

  return 0;

}
Savings_Account Get_Account()
{
  char id[5];
  char buffer[81];
  double bal;
  double rt;
```

```
  cout << endl;
  cout << "Enter Account ID: ";
  cin.getline(id, 5);

  cout << endl;
  cout << "Enter Account Holder's Name: ";
  cin.getline(buffer, 81);

  cout << endl;
  cout << "Enter Balance: ";
  cin >> bal;

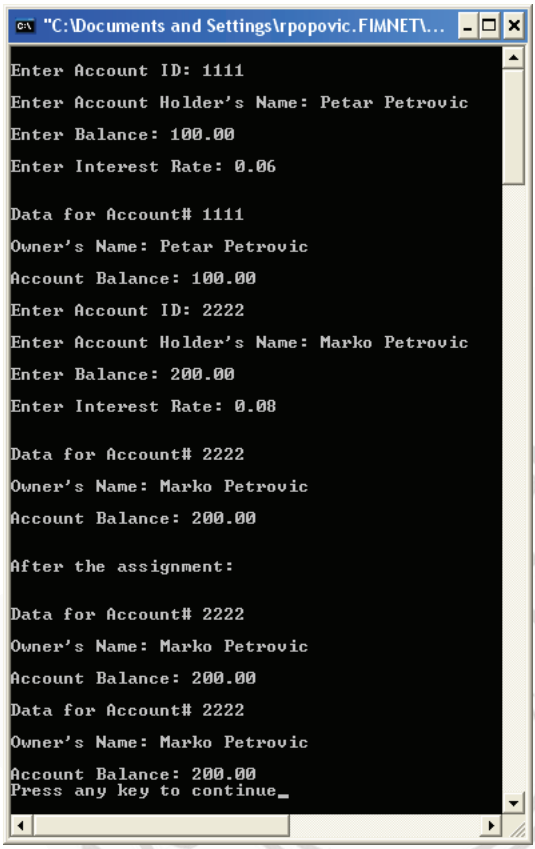
  cout << endl;
  cout << "Enter Interest Rate: ";
  cin >> rt;

  cin.get();   //brise karakter za novi red iz bafera

  Savings_Account acc(id, buffer, bal, rt);

  return acc;
}
void Display_Account(Savings_Account& acc_r)
{
  cout << endl;
  cout << "Data for Account# " << acc_r.Get_Id() << endl << endl;
  cout << "Owner's Name: " << acc_r.Get_Name() << endl << endl;

  cout << "Account Balance: " << acc_r.Get_Balance() << endl;
}
```



Program kreira dva štedna računa koristeći funkciju Get_Account() i prikazuje sadržaje oba računa koristeći Display_Account(). Zatim program dodeljuje objekat acc2 objektu acc1 i prikazuje sadržaje oba tekuća računa da bi pokazao da su tekući računi zaista isti.

Nasleđivanje i preklopljeni operatori

Kao i prijateljske funkcije, preklopljeni operatori se ne nasleđuju u hijerarhiji klasa. Zbog toga, ako se preklopi operator u osnovnoj klasi, mora da ga preklopite ponovo i u izvedenoj klasi.

12.3 Makroi i inline funkcije

Do sada smo definisali simboličke konstante kao const varijable. Takođe, moguće je definisati simboličke konstante koristeći pretprocesorsku

direktivu #define. Možete koristiti pretprocesorsku direktivu #define da biste definisali makroe, koji mogu ponekad da zamene funkcije.

Simboličke konstante koristeći pretprocesorsku direktivu #define

Podsetimo se da C++ pretprocesor koristi mehanizam zamene teksta. Pretprocesorska direktiva #define dozvoljava da zamenimo bilo koji izraz definisanim imenom. Na primer deklaraciju

```
const double COMMISSION_RATE = 0.050;
```

možemo da zamenimo po definiciji sa

```
#define COMMISSION_RATE 0.050
```

Ova direktiva definiše da simbolička konstanta COMMISSION_RATE ima vrednost 0.050. Kada pretprocesor skenira kod datog programa on zamenjuje svako pojavljivanje simbola COMMISSION_RATE vrednošću 0.050. Sledi, kada pretprocesor skenira liniju

```
straight_commission = sales * COMMISSION_RATE;
```

on zamenjuje simbol COMMISSION_RATE njegovom unapred definisanom vrednošću 0.050.

Pretprocesorske direktive se ne završavaju sa ; , zato što to može da prouzrokuje da tačka zarez bude deo definicije, odnosno to prouzrokuje sintaksnu ili logičku grešku.

Mada možete da koristite #define za definisanje simboličkih konstanti u bilo kom C++ programu više se koristi definicija simboličke konstante koristeći const.

Definicija makroa

Direktiva #define se takođe može koristiti da definiše makroe sa parametrima. Makro je ime za naredbe i izraze. Na primer:

```
#define SQ(x) x * x
```

Makro koji se definiše je SQ(x), a njegova definicija je x * x. Kada C++ pretrocesor vidi ovakav #define, on pretražuje program da bi pronašao sva pojavljivanja makroa sa imenom SQ. Pretpostavimo, na primer, da pretprocesor pronađe SQ(count) u programu. Pretprocesor tada zamenjuje makro SQ(count) izrazom dobijenim zamenom parametra x u definiciji makroa sa count. Tako će pretprocesor zameniti izraz

```
s = SQ(count);
```

sa

```
s = count * count;
```

Zbog toga što pretprocesor koristi mehanizam zamene teksta, on ne proverava korektnost u sintaksi ili značenju. Na primer, ne možete da stavite razmak nakon imena makroa i pre leve zagrade. Ako ovo uradite pretprocesor će sve što sledi nakon imena makroa da interpretira kao deo definicije, čak i zagrade. Na taj način, nekorektno definisanje SQ kao

```
#define SQ (x) x * x
```

kao rezultat pretprocesiranja naredbe

```
s = SQ(count);
```

daje izraz bez značenja

```
s = (x) x * x(count);
```

Data definicija SQ takođe može da prouzrokuje neželjene rezultate (takozvane bočne efekte) odnosno promenu u značenju. Pretpostavimo da imamo sledeće definicije i dodelu:

```
int i = 2,
    j = 4,
    k;

k = SQ(i + j);
```

Operator dodele ne dodeljuje vrednost 36 promenljivoj k kako prvo može da se pomisli. Pretprocesor ekspanduje makro u dodeli, zamenjujući svaku pojavu parametra x stringom i + j, na sledeći način.

```
k = i + j * i + j;
```

Kako računar izvršava operaciju množenja pre operacije sabiranja, desna strana operacije dodele ima vrednost 14.

Da biste izbegli ovakav bočni efekat razvoja makroa stavite u male zagrade parametre u definiciji makroa na sledeći način:

```
#define SQ(x) (x) * (x)
```

sledi:

```
k = (i + j) * (i + j);
```

Daćemo sada pregled pravila za pisanje makroa:

- Ne stavljajte u kôd razmak između imena makroa i leve zagrade koje slede nakon imena.
- Ne završavajte definiciju makroa sa ;.
- Stavite svaki parametar u definiciji makroa u zagrade.
- Stavite u definiciji makroa ceo makro u zagrade.

Makroi mogu imati više parametara. Makroi koji su definisani u programu, mogu da se koriste u definiciji drugih makroa.

Poređenje funkcija i makroa

Postoji više prednosti korišćenja makroa umesto funkcija. Prvo, kako korišćenje makroa ne uključuje poziv funkcije, ili prosleđivanje parametara, makro se izvršava brže od funkcije. Drugo, parametri u makrou nemaju tip.

Postoje i loše osobine korišćenja makroa umesto funkcija. Prvo, makroi mogu imati bočne efekte koji dovode do pogrešnih rezultata. Kod funkcija nema ovakvih bočnih efekata. Drugo, mada postoje opšti troškovi - *overhead* pri pozivu funkcije i prosleđivanju parametara, mehanizam funkcije je sigurniji za korišćenje. Zapamtite da kompajler koristi prototip funkcije da bi obezbedio korektno korišćenje funkcijskih parametara i povratnog tipa. Kompajler nema takve provere za makroe.

Inline funkcije

U C++, postoji način da se iskombinuju efikasnost makroa sa bezbednošću funkcija koristeći inline funkcije (neposredno ugrađivanje u kod). Ako se funkcija insertuje sa inline, kompajler smešta kopiju koda te funkcije u svakoj tački gde se poziva funkcija. Zato, kratke funkcije (jedna ili dve linije koda) su idealne za korišćenje inline. Zato što kompajler zamenjuje pozive inline funkcije kodom funkcije, sledi da kompajler mora da zna definiciju funkcije pre bilo kog poziva u programskom kodu. Da bi se ubacila funkcija, treba da se smesti rezervisana reč inline pre imena funkcije i definiše funkcija pre bilo kog poziva funkcije. Ovo znači da treba da smestite definiciju funkcije gde obično smeštate prototip funkcije. U našim programima ovo znači da su definicije inline funkcija smeštene pre funkcije main().

Na primer, za sledeći makro:

```
#define MAX(x, y) ( ((x) > (y))? (x): (y) )
```

ekvivalentna inline funkcija je:

```
inline double Max(double x, double y) {return (x > y)? x : y;}
```

Treba obratiti pažnju da ne treba da koristimo previše zagrada, kao što je bilo neophodno u definiciji makroa zbog izbegavanja bočnih efekata. Pretpostavimo da funkcija main() sadrži sledeće deklaracije i funkcijske pozive:

```
double d1 = 4.8,
   d2 = 9.6
   d;

d = Max(d1, d2);
```

Kada kompajler realizuje funkcijski poziv Max(), on zamenjuje funkcijski poziv kodom inline funkcije koristeći vrednosti argumenata kao vrednosti parametara. Zato, dodela je ekvivalentna sledećem

```
d = (d1 > d2)? d1 : d2;
```

Inline funkcije takođe igraju važnu ulogu kada su metodi definisani u deklaraciji klase. Razlika između definisanja metoda unutar ili izvan deklaracije klase je da je metod klase koji je definisan unutar deklaracije klase, po *default*, inline. Ako definišete metod izvan deklaracije klase, tada inline metod, mora da smestite da prethodi definiciji odnosno ključnoj reči inline.

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

Dodatak A:
Integrisano razvojno okruženje Microsoft Visual C++

1.1 Uvod

U ovom delu biće dat kratak uvod u terminologiju okruženja *Microsoft Visual* C++, odnosno paketa *Developer Studio*, koji predstavlja zajedničko integrisano razvojno okruženje grupe *Microsoft*-ovih prevodilaca različitih programskih jezika.



Slika 1.1.1: *Microsoft Visual Studio* logo

1.2 Pojmovi

U okruženju *Microsoft Visual* C++ (u daljem tekstu MSVC) razvija se jedan programski paket pomoću radnog prostora (eng. *workspace)* koji sadrži jedan ili više projekata. Projekat (eng. *project)* se sastoji iz:

- izvornih (eng. *source)* fajlova
- zaglavlja (eng. *header)* fajlova
- resursnih (engl. *resource)* fajlova
- fajlova koji sadrže informacije o podešavanjima i konfiguraciji

Programski kôd definisan unutar jednog projekta prevodi se u jedan nezavisni izvršni program ili biblioteku. Unutar jednog radnog prostora može postojati više projekata, recimo za različite aplikacije unutar jednog programskog paketa ili za različite uslužne programe koji čine paket, za različite verzije programa, biblioteke itd.

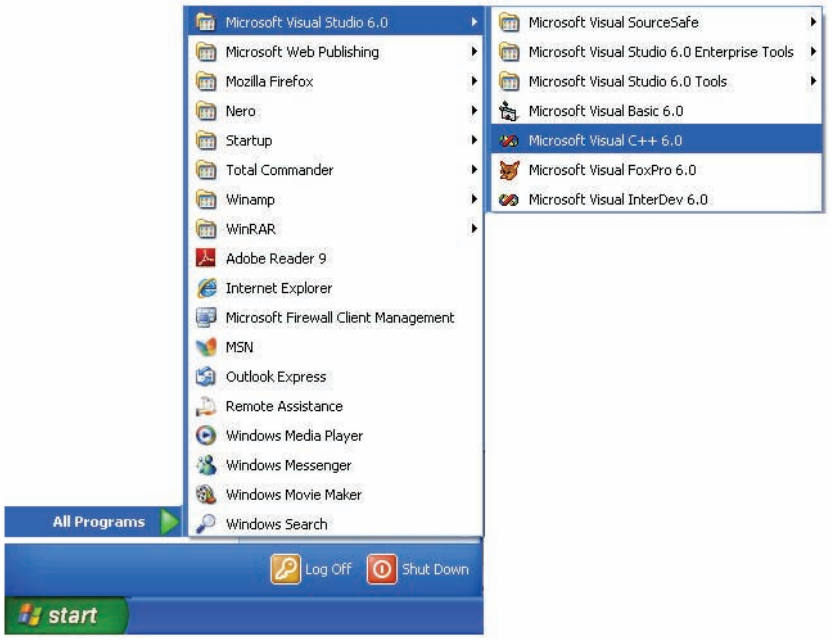
Fajl zaglavlje (eng. *header file)* predstavlja interfejs jedne programske celine (modula) koji sadrži deklaracije elemenata definisanih u tom modulu koje koriste drugi moduli (deklaracije klasa, funkcija, objekata, tipova, konstanti itd.).

Izvorni fajl (eng. *source file)* sadrži implementaciju programskog modula.

Resursni fajl (eng. *resource file)* sadrži definicije elemenata grafičkog interfejsa koje koristi *Windows* kada izvršava dati program (bit mape, meniji, dijalozi itd.). Pravljenje novog programa počinje stvaranjem novog radnog prostora i projekta. Otvaranje prethodno kreiranog projekta (nastavak rada sa započetim projektom) omogućeno je otvaranjem fajla sa nastavkom *dsw*.

1.3 Pokretanje *Visual* C++ okruženja

Okruženje *Microsoft Visual C++* se može otvoriti klikom na taster *Start* i izborom stavke *Microsoft Visual Studio 6.0/Microsoft Visual C++ 6.0* iz liste programa (grupa *All Programs* ili *Programs* kod starijih verzija MS *Windows* operativnog sistema). Na slici 1.3.1 je ilustrovan ovaj postupak. Nakon startovanja otvoriće se novi prozor prikazan na slici 1.4.1.

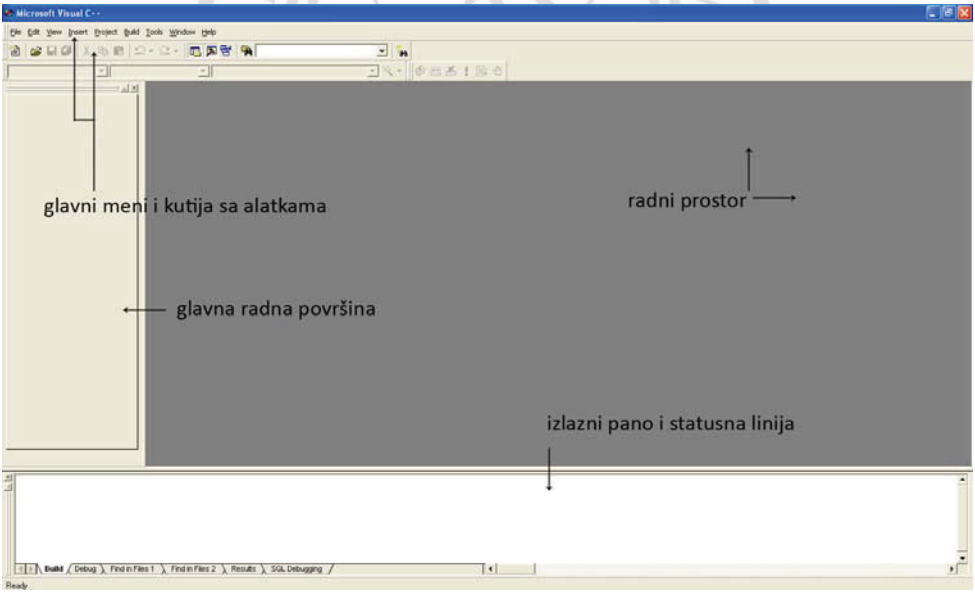


Slika 1.3.1: Pokretanje *Microsoft Visual* C++

1.4 Radno okruženje

Radno okruženje *Microsoft Visual* C++ sastoji se iz sledećeg (slika 1.4.1):

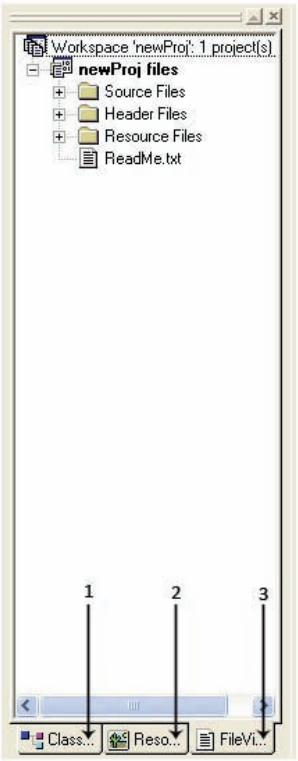
- Na vrhu su glavni meni i kutija sa alatkama.
- Sa leve strane nalazi se prozor radnog prostora (engl. *workspace).* Ovaj deo je ključ za kretanje kroz razne delove projekta.
- Sa desne strane je glavna radna površina na kojoj se uređuju fajlovi.
- Na dnu su izlazni pano (engl. *output panel)* i statusna linija (engl. *status bar).* Izlazni pano možda neće biti vidljiv kada se program startuje prvi put, ali se pojavljuje pri prvom prevođenju programa. Na izlaznom panou vide se poruke koje prevodilac šalje (npr. greške) pri prevođenju i povezivanju.



Slika 1.4.1: Radno okruženje *Microsoft Visual* C++

1.4.1 Prozor radnog prostora

Prozor radnog prostora omogućava kretanje kroz delove projekta na tri različita načina (slika 1.4.1.1):



- *Class View* (1) omogućava kretanje i manipulaciju izvornim C++ kodom na nivou klasa. On koristi različite ikone za predstavljanje privatnih, zaštićenih i javnih funkcija članica i atributa. Ako se dvaput pritisne mišem bilo koja stavka, otvara se odgovarajući fajl pri čemu je kursor automatski postavljen na odgovarajuće mesto. Npr. dvostruko pritiskanje mišem imena klase postavlja kursor na njenu deklaraciju, a dvostruko pritiskanje imena funkcije postavlja kursor na početak njene definicije. Dvostruko pritiskanje imena atributa postavlja kursor na mesto gde je on deklarisan. Pritiskanje imena klase desnim tasterom miša otvara kontekstni meni koji omogućava brzo nalaženje mesta gde je klasa definisana, dodavanje funkcije ili podataka toj klasi, pregled liste svih mesta gde je klasa referisana, njene odnose sa drugim klasama itd. Slično tome, pritiskanje imena funkcije desnim tasterom miša otvara kontekstni meni koji omogućava da nalaženje mesta gde je funkcija deklarisana odnosno definisana, brisanje, postavljanje tačke prekida (eng. *breakpoint*), pregled liste svih mesta u kodu odakle se funkcija poziva itd. Pritiskanje imena atributa desnim tasterom otvara manji meni sličan opisanima.

- *Resource View* (2) omogućava pretraživanje i uređivanje raznih resursa aplikacije, kao što su dijalozi, ikone i meniji.

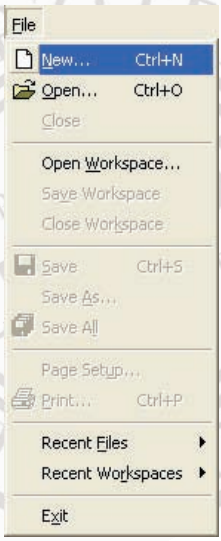
- *File View* (3) omogućava kretanje kroz fajlove od kojih se sastoji aplikacija i njihovo otvaranje. Željeni fajl se otvara dvostrukim pritiskom njegovog imena.

2. OPŠTE MANIPULACIJE PROJEKTIMA

2.1 Kreiranje novog projekta

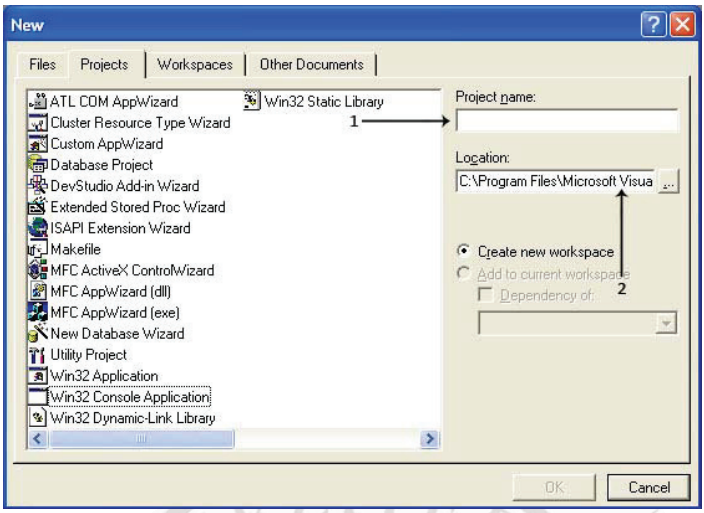
Novi projekat (slika 2.1.1) u *Microsoft Visual* C++ kreira se na sledeći način. Odabirom opcije *File->New* otvara se dijalog sa sledećim stranama (slika 2.1.2):

- *Files*
- *Projects*
- *Workspaces*
- *Other Documents*



Slika 2.1.1: Kreiranje novog projekta

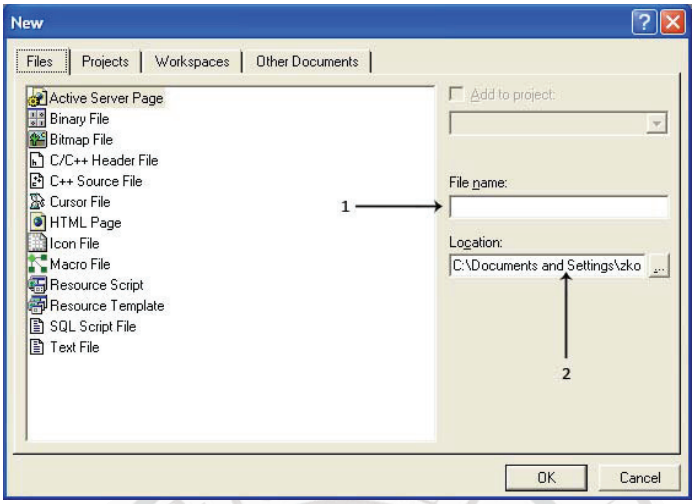
Na strani *Projects* bira se vrsta projekta koji se želi napraviti. U ovom trenutku značajni su *Win32 Console Application, Win32 Dynamic-Link Library* i *Win32 Static Library.* Unesite ime projekta u polje *Project name.* U polje *Location* upišite putanju do direktorijuma na disku gde želite da bude upisan novi projekat. Na strani *Workspaces* u početku moguće je pravljenje samo praznog radnog prostora. Ako se ne napravi radni prostor pri započinjanju projekta, MSVC će ponuditi da napravi podrazumevani radni prostor za vas.



Slika 2.1.2: *Project*:
1. *Project name*
2. *Location*

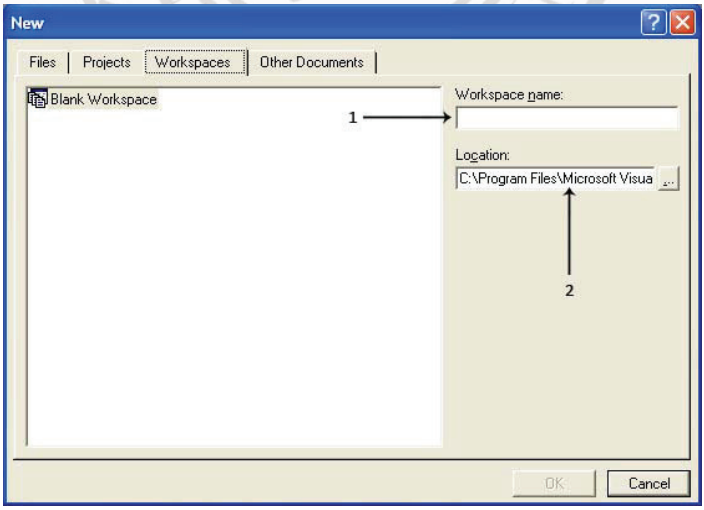
Kada se pritisne OK pojavljuje se dijalog u kome se može odabrati vrsta aplikacije koja se želi napraviti:

- Prazan projekat znači upravo to - biće napravljen projekat koji neće sadržati ni jedan fajl niti klasu.
- Ako se odabere jednostavna aplikacija, biće napravljen projekat koji sadrži neke početne fajlove, odnosno kostur aplikacije u koji se dodaje kod.
- Ako se odabere tip aplikacije „Hello, world!", MSVC će napraviti program koji, kada se startuje, štampa „Hello, world!".



Slika 2.1.3: *Files*:
1. *Project name*
2. *Location*

Kada se odabere tip aplikacije i pritisne *Finish,* prikazaće se prozor koji sadrži informacije o projektu koji će biti napravljen. Potrebno je izabrati OK i zapaziti šta je MSVC napravio.



Slika 2.1.4: *Workspaces*

Kreiranje novog projekta (korak-po-korak) vrši se na sledeći način:

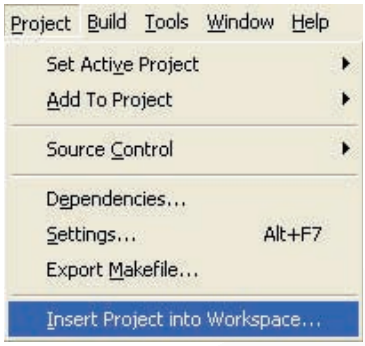
- Odabrati *File->New*
- Odabrati stranu *Projects*
- Odabrati *Win 32 Dynamic-Link Library*
- U polje *Projects* upisati *Figure*
- Pritisnuti dugme pored polja *Location* da be se izabralo gde će se smestiti novi projekat ili se lokacija može direktno upisati u polje predviđeno za to
- Pritisnuti OK
- U dijalogu koji se pojavi odabrati *An empty DLL project* i pritisnuti *Finish*
- Pritisnuti OK u dijalogu koji prikazuje informacije o projektu koji će biti napravljen

2.2 Rad sa više projekata

MSVC nije ograničen na rad samo sa jednim projektom. Biranjem komande *Insert Project into Workspace* menija *Project* može se dodati postojeći projekat u trenutno otvoren radni prostor. Projekti se mogu dodati kao projekti najvišeg nivoa (eng. *top-level*) ili kao potprojekti (eng. *subprojects*). Glavna razlika je u tome što potprojekat učestvuje u procesu izgradnje (eng. *build*) svog natprojekta (njemu nadređenog) - kada se gradi *top-level* projekat, prvo se izgrade njegovi potprojekti.

Rad sa više projekata (korak-po-korak) vrši se na sledeći način:

- Odabrati *File->Close Workspace* ako je prethodno formirani radni prostor još otvoren
- Odabrati *File->New*
- U dijalogu *New* odabrati stranu *Project*, a potom opciju *Win32 Console Application*
- U polje *Project* upisati „MojProgram“
- Odabrati lokaciju i pritisnuti OK
- U dijalogu koji se pojavi odabrati opciju *A simple application* i pritisnuti *Finish*
- U sledećem dijalogu pritisnuti OK
- Odabrati *Project->Insert Project into Workspace* (slika 2.2.1) i pronaći prethodni projekat (fajl *Figure*, dsp)

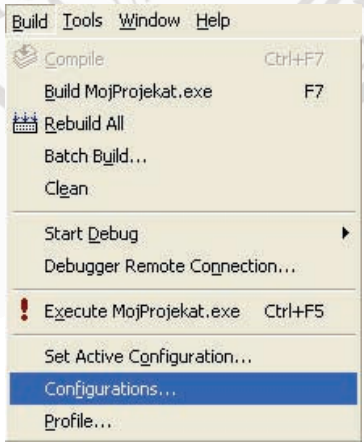


Slika 2.2.1: Ubacivanje novog projekta

2.3 Osnovna podešavanja

Konfiguracija projekta definiše kako će izvršni program ili biblioteka koje projekat predstavlja biti sagrađeni (eng. *build).* Kada se pravi projekat pomoću nekog od čarobnjaka (eng. *wizard),* njemu će biti dodeljen skup podrazumevanih konfiguracija. Podrazumevane konfiguracije su:

- *Debug:* služi za pravljenje programa u radnoj fazi.
- *Release:* služi za konačno pravljenje programa (kada se smatra da je spreman za isporuku).



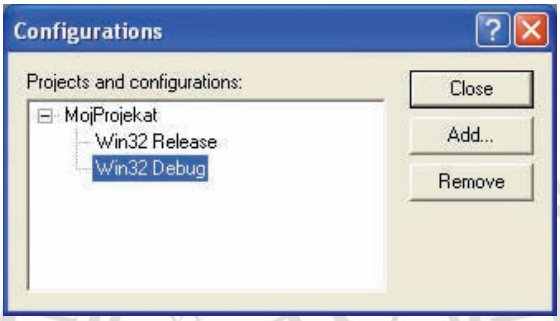
Slika 2.3.1: Putanja za podešavanje konfiguracija

Konfiguracije se mogu dodati ili obrisati biranjem komande *Configurations* iz menija *Build.* Određena konfiguracija se podešava biranjem komande *Settings* iz menija *Project.*

Podešavanje konfiguracija vrši se na sledeći način:

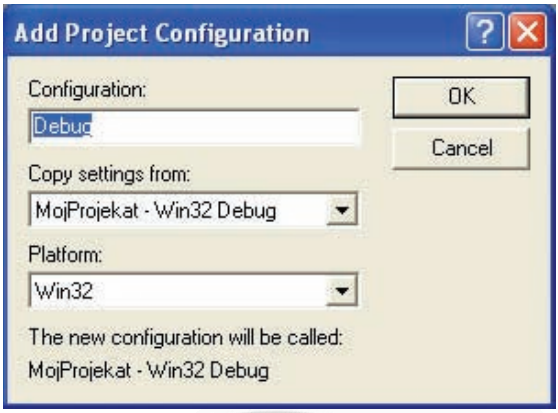
- Odabrati *Build->Cofigurations* (slika 2.3.1)

- Izabrati konfiguraciju (slika 2.3.2) *Win32 Debug* projekta „MojProgram“ i pritisnuti *Remove*



Slika 2.3.2: Odabir *Win32 Debug*

- Potvrditi uklanjanje konfiguracije
- Pritisnuti ime projekta („MojProgram“)
- Pritisnuti *Add*
- U polje *Configuration* uneti *Debug.* Naravno, da bi konfiguracija služila za debagovanje ona ne mora da se zove *Debug* – može se nazvati po želji (slika 2.3.3)
- U listi *Copy settings from* izabrati „MojProgram“ - *Win32 Release* (koji je jedini u listi, ali u opštem slučaju će postojati prilika da se izabere konfiguracija čija će se podešavanja kopirati). To znači da se ne kreće iz početka kada se pravi konfiguracija.
- U opštem slučaju, u listi *Platform* nalaziće se samo *Win32*
- Pritisnuti OK u dijalogu *Add Project Configuration,* a zatim *Close* u dijalogu *Configurations*



Slika 2.3.3: *Add Project Configuration*

3. UREĐIVANJE PROGRAMA

3.1 Dodavanje fajlova u projekat

Dodavanje postojećeg fajla u aktuelan projekat se može uraditi biranjem *Projects -> Add* to *Project -> Files*. Novi fajl se dodaje biranjem *Project -> Add to Project -> New* i biranjem vrste fajla.

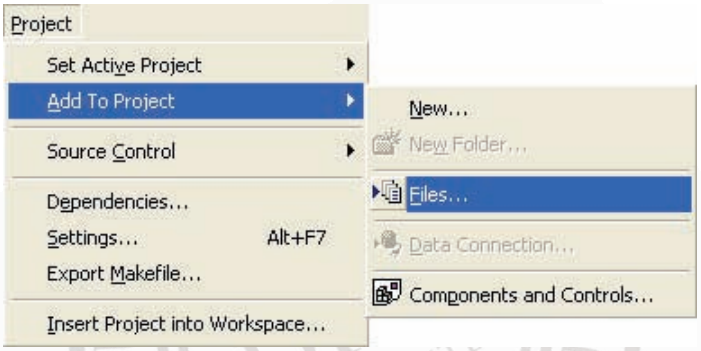
Fajlovi se mogu hijerarhijski organizovati unutar projekta, prema logičkoj organizaciji programa. Novi *header* fajl se dodaje na sledeći način:

- Zatvoriti trenutno otvoren radni prostor (ako je otvoren) biranjem *File->Close Workspace*
- Odabrati *File->New*
- Odabrati stranu *Files* i C/C++ *Header File*
- U polje *File name* uneti „Pravougaonik“
- U polje *Location* upisati putanju projekta *Figure* ili pritisnuti odgovarajuće dugme i pronaći direktorijum *Figure*
- Pritisnuti OK
- Uneti sledeći kôd u fajl „Pravougaonik.h“:

```
typedef unsigned int UINT;
class CPravougaonik
 {
public:
```

```
CPravougaonik();
virtual -CPravougaonik();
private:
UINT visina;
UINT sirina;
};
```

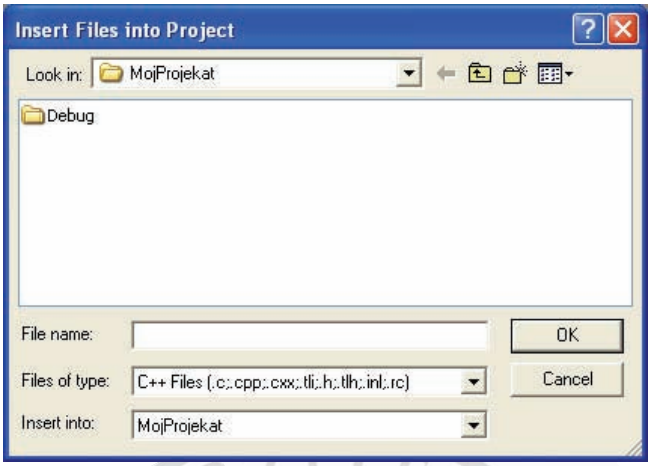
- Odabrati *File->Save* i zatvoriti fajl biranjem *File->Close*
- Otvoriti radni prostor „MojProgram“ biranjem *File->Recent Workspaces*
- Odabrati *Project->Add to Project->Files* (slika 3.1.1)



Slika 3.1.1: Putanja za dodavanje fajlova u projekat

- U dijalogu *Insert Files into Project* (slika 3.1.2) pronaći fajl „Pravougaonik.h“ i pritisnuti OK. Ako radni prostor sadrži više projekata, izabrani fajl će biti dodat u trenutno aktivan projekat (njegovo ime je prikazano crnim slovima u prozoru radnog prostora). Ako se želi da drugi projekat bude aktivan, pritisne se desnim tasterom miša njegovo ime i u kontekstnom meniju odabere *Set as Active Project.*

Da bi projekat bio dodat u nov fajl upotrebiti putanju *File->New*.



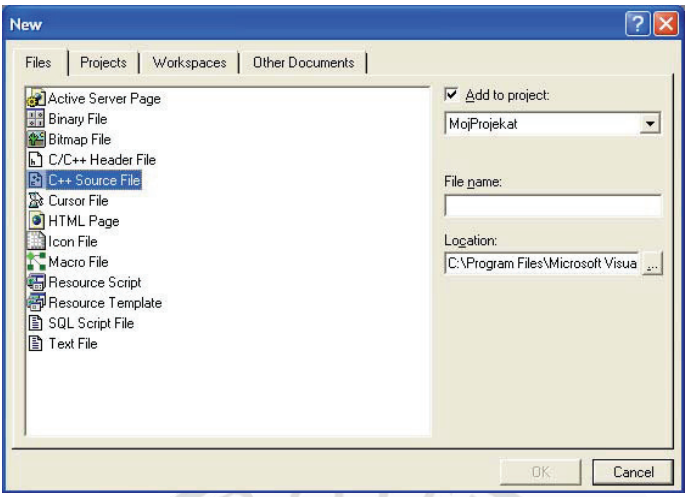
Slika 3.1.2: Ubacivanje fajlova u projekat

Novi *source* fajl se dodaje na sledeći način:

- Odabrati *File->New*
- Odabrati stranu *Files* i C+ + *Source File* (slika 3.1.3)
- U polje *File name* uneti „Pravougaonik“ i odabrati istu lokaciju kao za „Pravougaonik.h“
- Potvrditi *Add to project* i u listi ispod, odabrati projekat *Figure*
- Uneti sledeći kod u „Pravougaonik.cpp“:

```
#include "Pravougaonik.h"

CPravougaonik::CPravougaonik()
{
sirina=8;
visina=5;
}
CPravougaonik:: ~CPravougaonik()
{
sirina=0;
visina=0;
}
```



Slika 3.1.3: Dijalog za dodavanje novog fajla

3.2 Osnovni alati

ClassWizard predstavlja jednu od najmoćnjih alata MSVC okruženja. Njegove najvažnije funkcije tiču se rada sa MFC projektima, ali i u radu sa konzolnim aplikacijama biće od koristi (mada neuporedivo manje). *WizardBar* omogućava brz pristup mnogim funkcijama *ClassWizard* - a. Iz prve liste možete odabrati bilo koju klasu iz aktuelnog projekta, iz druge birate filter, a iz treće funkciju klase koja se želi modifikovati. Dugme *WizardBar Actions* omogućava brz pristup brojnim komandama (spisak tih komandi zavisi od tipa projekta, kao i od toga da li je trenutno odabrana klasa ili globalna funkcija itd.). *SourceBrowser* omogućava brzo nalaženje definicije i reference simbola i traženje veza među simbolima. Ovaj alat zahteva prisustvo fajla koji sadrži potrebne informacije. Za pristup ovom fajlu potrebno je ponovno podešavanje projekta. Najlakši način je da, kada se signalizira da taj fajl ne postoji (dešava se kada se odabere neka od komandi koja pokreće *SourceBrowser,* npr. *References* iz kontekstnog menija podatka člana), dozvoli MSVC-u da ga automatski napravi. Drugi način je ručno podešavanje uz pomoć odabira komande *Settings* iz menija *Project,* pa odabrati stranu *Browse Info* i potvrditi *Build browse info file.*

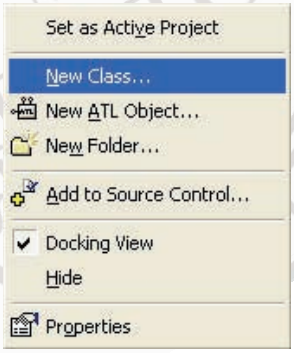
3.2.1 *Class Wizard*

U osnovi, *ClassWizard* je alat za automatsko generisanje klasa. Pokreće se biranjem komande *New Class* bilo iz *WizardBar Actions* ili iz menija *Insert.* Takođe se može pokrenuti biranjem komande *ClassWizard* iz menija *View* (ako je komanda omogućena).

Kada se otvori dijalog *New Class,* potrebno je uneti ime klase i odabrati njenu osnovnu klasu iz spiska. Ukoliko se klasa iz koje se želi izvesti nova klasa ne nalazi na spisku, to će se rešiti ručnim menjanjem koda. Kada se pritisne OK*, ClassWizard* će generisati željenu klasu i automatski dodati konstruktor i destruktor.

Prethodno opisano, korak-po-korak, izgledalo bi na sledeći način:

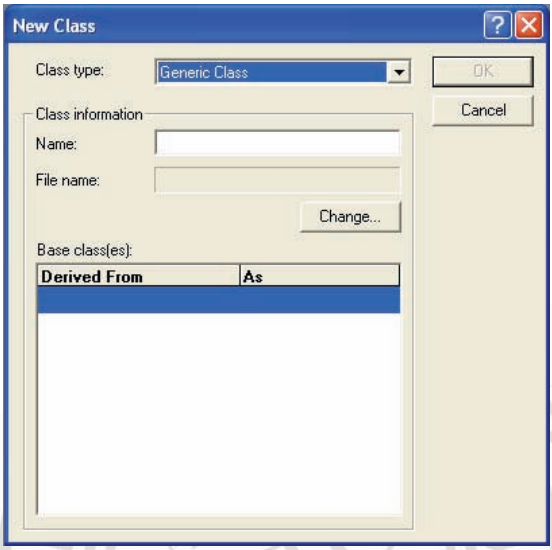
- U prikazu *Class View* pritisnuti desnim tasterom miša projekat *Figure*
- U kontekstnom meniju odabrati *New Class* (slika 3.2.1.1)



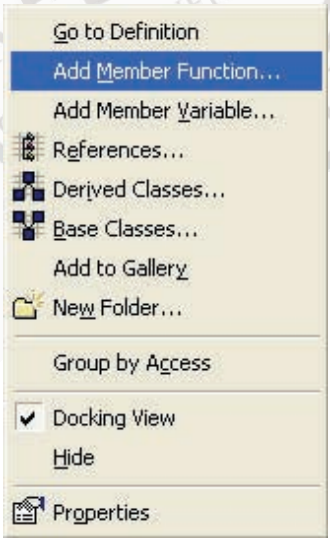
Slika 3.2.1.1: Dodavanje nove klase

- U dijalogu *New Class,* u polje *Name* uneti „CKvadrat“
- Pritisnuti mišem ispod *Derived From* i uneti „CPravougaonik“ (slika 3.2.1.2)
- Ispod *As* će se pojaviti reč *public,* što je podrazumevan način izvođenja. Neka ostane tako
- Pritisnuti OK
- U prikazu *Class View* pritisnuti desnim tasterom miša ime klase „CPravougaonik“ i odabrati *Add Member Function* (slika 3.2.1.3)

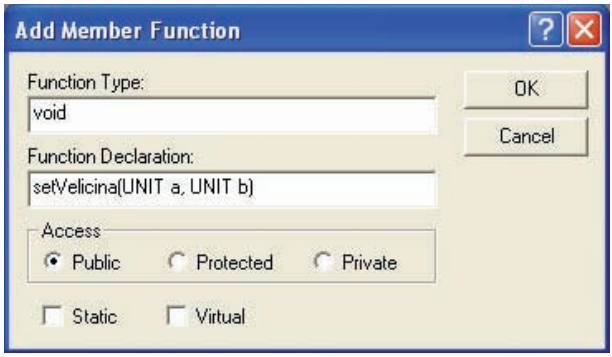
- U dijalogu *Add Member Function* (slika 3.2.1.4) uneti void u polje *Function Type,* a u polje *Function Declaration* uneti: setVelicina(UINT a, UINT b)



Slika 3.2.1.2: Navođenje odakle je klasa izvedena

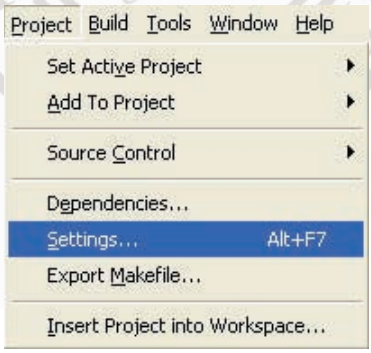


Slika 3.2.1.3: Dodavanje funkcije



Slika 3.2.1.4: Tip i deklaracija funkcije

- U opcijama *Access* odabrati *public*
- Pritisnuti OK
- Ako projekat *Figure* nije trenutno aktivan, aktivirati ga
- Iz prve liste *WizardBar*-a odabrati klasu „CPravougaonik“
- Iz druge liste odabrati *All class members*
- Iz treće liste odabrati „setVelicina“. Otvoriće se odgovarajući fajl („Pravougaonik.cpp“), sa kursorom postavljenim na početak definicije funkcije i obeleženim potpisom (eng. *signature)* funkcije
- Odabrati komandu *Project->Settings* (slika 3.2.1.5)

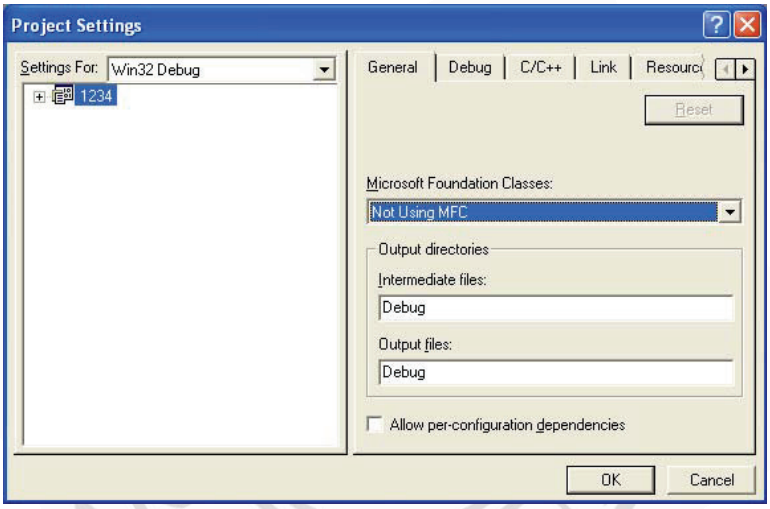


Slika 3.2.1.5: Putanja za podešavanje parametara

- Otvoriće se dijalog *Project Settings* sa izabrana oba projekta sa leve strane (slika 3.2.1.6)
- Iz liste sa leve strane odabrati *Win32 Debug.* Kako su oba projekta izabrana i oba imaju konfiguraciju koja se zove *Win32 Debug,* sva

podešavanja koja budu promenjena sa desne strane odraziće se na oba projekta:

- Odabrati stranu C/C++, zatim *General* iz liste *Category* i potvrditi *Generate browse info*
- Odabrati stranu *Browse Info* i potvrditi *Build browse info file*
- Da be se mogao koristiti *SourceBrowser,* mora se odabrati komanda *Build->Rebuild All*, koja će iz početka prevesti sve fajlove i povezati projekte sa novim podešavanjima



Slika 3.2.1.6: *Project Settings*

- U prikazu *ClassView* pritisnuti desnim tasterom miša ime funkcije članice setSize klase CKvadrat i odabrati opciju *References, Calls* ili *Called by* iz kontekstnog menija. Pri tom treba imati u vidu da se *SourceBrowser* može koristiti samo za trenutno aktivan projekat
- Prozor koji se pojavio sadrži sa leve strane informacije o tome odakle se poziva funkcija CKvadrat::setVelicina ili o funkcijama koje ona poziva, u zavisnosti od toga koja je komanda menija izabrana. Sa desne strane nalazi se ime fajla u kome je funkcija definisana, a ispod toga spisak fajlova u kojima se ona spominje. Pritisnuti dvaput bilo koji fajl sa spiska (sa desne strane) ili bilo koju funkciju (sa leve strane)

3.2.2 Uređivanje koda

Unošenje, izmena, brisanje, kopiranje teksta itd. isti su kao u drugim editorima teksta, npr. *Microsoft Word*-u. Međutim, kako je MSVC okruženje za programiranje, ima i neke specifične mogućnosti. Ako se odabere desnim tasterom miša bilo gde u okviru fajla koji se menja, otvoriće se kontekstni meni koji sadrži sledeće komande:

- *Cut* iseca odabrani tekst i kopira ga na *Clipboard*

-*Copy* kopira odabrani tekst na *Clipboard*

-*Paste* zamenjuje odabrani tekst tekstom koji je trenutno na *Clipboard*-u. Ako nije odabran nikakav tekst, ubacuje sadržaj *Clipboard*-a na mesto gde se trenutno nalazi kursor

-*Insert File Into Project* dodaje fajl koji se menja u trenutno otvoren projekat

-*Open* otvara fajl iznad čijeg imena se nalazi kursor miša. Ovo je posebno korisno za otvaranje zaglavlja

-*List Members* prikazuje listu podataka i funkcija članica objekta iznad čijeg imena se nalazi kursor miša

-*Type Info* prikazuje mali prozor koji podseća na tip podatka iznad čijeg imena se nalazi kursor miša

-*Parameter Info* podseća na parametre koje prima funkcija iznad čijeg imena se nalazi kursor miša

-*Complete Word* pomaže u vezi s imenom promenljive ili funkcije koje je delimično otkucano

-*Go To Definition* otvara fajl u kome je definisano ime iznad kojeg se nalazi kursor

-*Go To Reference* postavlja kursor na mesto gde se sledeći put referiše ime iznad kojeg se nalazi kursor

-*Insert/Remove Breakpoints* postavlja tačku prekida na mesto gde se nalazi kursor ili je uklanja ako ona već postoji

-*Enable Breakpoint* omogućava tačku prekida

-*ClassWizard* otvara dijalog *ClassWizard*

-*Properties* otvara listu svojstava

Sve ove komande imaju ekvivalente u meniju i kutiji sa alatkama.

3.2.3 Sintaksno bojenje

MSVC označava elemente koda sintaksnim bojenjem. Podrazumevane boje su: crna za kod koji se prevodi, zelena za komentare i plava za rezervisane reči. Stringovima, brojevima itd. može se dodeliti druga boja tako što se odabere komanda *Options* iz menija *Tools*, a zatim odabere strana *Format.*

3.2.4 Vežba

Koristeći što više *ClassWizard,* uraditi sve što treba da bi klase CPravougaonik i CKvadrat i funkcija *main* projekta „MojProgram“ izgledale ovako:

```
//Pravougaonik.h

typedel unsigned int UINT;
class CPravougaonik {

public:
CPravougaonik{};
virtual -CPravougaonik();
UINT getObim();
UINT getPovrsina();
void setVelicina(UINT a, UINT b);

private:
UIXT visina;
UTNT sirina;
};

//Pravougaonik.cpp
((include "Pravougaonik. h"

CPravougaonik::CPravougaonik()
{
sirina = 8 ;
visina = 5;
}
CPravougaonik::~CPravougaonik ()
{
sirina = 0;
visina = 0;
}
```

```
void CPravougaonik::setVelicina(UINT a, UINT b)
{
sirina = a;
visina = b;
}

UINT CPravougaonik::getPovisina()
{
return sirina*visina;
}

UINT CPravougaonik::getObim()
{
return 2*(sirina+visina);
}

// Kvadrat.h

class CKvadrat:public CPravougaonik
{
public:
CKvadrat();
virtual CKvadrat0;
void setVelicina(UINT a);
};

// Kvadrat.cpp
#include "Kvadrat.h"
CKvadrat::CKvadrat()
setVelicina(5);

CKvadrat::~CKvadrat()
{
};
void CKvadrat::setVelicina(UINT a);
CPravougaonik::setVelicina(a,a);

// MojProgram.cpp

int main(int argc, char* argv[])
{
CPravougaonik pravougaonik;
CKvadrat kvadrat;
UINT a;
UINT b;
cout << endl;
```

```
cout << "Obim pravougaonika: "<<pravougaonik.getObim()<<endl;
cout << "Povrsina pravougaonika: "<<pravougaonik.getPovrsina()<<endl;
cout << endl;
cout << "Obim kvadrata: "<<kvadrat.getObim()<<endl;
cout << "Povrsina kvadrata: 11 < <kvadrat. getPovrsina ()< <endl;

a=12 ;
b = 9 ;

pravougaonik.setVelicina(a,b);
kvadrat.setVelicina(a);

cout << endl;
cout << "Obim pravougaonika: "<< pravougaonik.getObim() << endl;
cout << "Povrsina pravougaonika:" << pravougaonik.getPovrsina()<< endl;
cout << endl;
cout << "Obim kvadrata: "<< kvadrat.getObim() << endl;
cout << "Povrsina kvadrata: 11 << kvadrat. getPovrsina() << endl ;

return 0;
```

4. PREVOĐENJE PROGRAMA

Svaki izvorni fajl u projektu mora biti najpre preveden (eng. *compiled*) da bi se potom moglo preći na povezivanje (eng. *linking).* Kada se radi na većem projektu postaje zamorno da, svaki put kada se promeni kod, da se sve prevodi iz početka. Najbrži način je da se ponovo prevedu samo fajlovi u kojima su unete promene.

Fajl će se prevesti biranjem komande *Compile* "nekiFajl.cpp" iz menija *Build.* Ovde "nekiFajl.cpp" predstavlja fajl koji je trenutno otvoren u editoru.

U toku prevođenja, MSVC šalje poruke o greškama i upozorenjima na već pomenuti izlazni pano (eng. *output panel).* Pod greškom (eng. *error*) se podrazumeva nešto što nije moglo da se prevede, jer nije ispravno sa stanovišta sintakse ili semantike jezika C++. Upozorenje (eng. *varning*) se odnosi na nešto što nije greška sa stanovišta samog jezika, tj. može da se prevede, ali MSVC smatra da tu postoji mogućnost semantičke greške u programu. Na primer, deklarisana je promenljiva, ali nakon toga nije korišćena - to nije greška, ali MSVC na to upozorava, jer sumnja da se možda nešto zaboravilo. Nakon izlistavanja postojećih grešaka i upozorenja (ukoliko broj grešaka ne prelazi određenu granicu posle koje prevodilac

prekida prevođenje), na izlaznom panou se pojavljuje broj grešaka i upozorenja. Ako je program sintaksno ispravan, a MSVC ga smatra i semantički korektnim, javiće: 0 grešaka, 0 upozorenja.

Program se prevodi na sledeći način:

- Pritisnuti desnim tasterom miša ime projekta *Figure* i odabrati *Set as Actilve Project*
- Otvoriti fajl "Pravougaonik.cpp" pritiskanjem dvaput njegovog imena u prikazu *File View*
- Odabrati *Build->Compile* "Pravougaonik.cpp".

4.1 Vežba

Prevesti na isti način i "Kvadrat.cpp" i "MojProgram.cpp". Zašto MSVC prijavljuje greške? Šta nedostaje?

4.2 Pravljenje projekta

Pravljenje projekta znači prevođenje i povezivanje svih njegovih komponenti da bi se proizvela ciljna verzija izvršnog programa ili biblioteke. Svaki projekat ima svoj *make* fajl, koji sadrži podatke o zavisnostima između pojedinih fajlova. Kada se neki fajl promeni, onda svi fajlovi koji zavise od njega moraju ponovo da se prevedu i povežu.

U procesu kreiranja projekta, potrebno je odabrati komandu *Build* "nekiProjekat" iz menija *Build,* gde "nekiProjekat" predstavlja trenutno aktivan projekat. Biranjem ove komande, MSVC će prevesti samo one fajlove koji su promenjeni posle prethodnog prevođenja, zatim fajlove koji su od njih zavisni i uradiće povezivanje.

Ako je potrebno da se svi fajlovi bezuslovno prevedu i povežu, potrebno je odabrti komandu *Rebuild All* iz menija *Build.* Ovo je potrebno uraditi kada se promene neke opcije prevođenja od kojih zavise svi delovi programa, pa ne bi bilo dobro da neki fajlovi budu prevedeni sa starim, a neki sa novim podešavanjima.

4.3 Pomoć u radu

Dok je aktivan prozor editora, postaviti kursor na mesto gde se nalazi željena reč, a zatim pritisnuti taster F1. Automatski će se pokrenuti MSDN (*Microsoft Developer Network,* kolekcija članaka koji se tiču MSVC) sa spiskom članaka u kojima se spominje odabrana reč.

Drugi način da se dobije pomoć jeste tzv. *Info View,* koji je deo prozora *Workspace.* Ako se odabere prikaz *Info View,* u ovom prozoru će se prikazati kolekcija članaka vezanih za MSVC i *Developer Studio.*

4.4 Podešavanja

MSVC obezbeđuje podešavanje mnoštva opcija prevođenja i debagovanja. Odabrati kamandu *Settings* menija *Project.*

Na levoj strani dijaloga *Project Settings* videće se lista svih konfiguracija projekta. Na desnoj strani može se izabrati određena stranica i tako pristupiti mnogim opcijama projekta.

Podešavanje konfiguracije se može izabrati sa leve strane. Ako se želi definicija podešavanja za određeni faji projekta, potrebno je odabrati ga na levoj strani tako što će se raširiti stablo pritiskom znaka plus pored imena konfiguracije.

Prva strana je strana *General.* Ovde se može definisati da Ii program koristi MFC i kako. Takođe, na ovoj strani mogu se izabrati direktorijumi u kojima će biti smešteni tzv. među-fajlovi kao i krajnji fajlovi projekta.

Druga strana, *Debug*, definiše opcije debagovanja. Biranjem kategorije *General* ili *Additional DLLs* ustanoviće se da ova strana ima dve podstrane. Od značaja je polje *Executable for debug session.* Ako je projekat standardni izvršni program, onda nema razloga da se sadržaj ovog polja razlikuje od imena programa. Ali, na primer, ako se radi na DLL projektu, definisanjem izvršnog programa, postojaće mogućnost da se i on debaguje.

Treća strana, C/C++, omogućava podešavanja opcija prevodioca. Ova strana ima više podstrana koje se menjaju biranjem *Category.*

Četvrta strana, *Link*, omogućava pristup raznim opcijama povezivanja.

Strana *Browse info* se koristi kada se želi upotreba mogućnosti pretraživanja koda (eng. *browse*).

5. DEBAGOVANJE

MSVC debager se aktivira kada se izvršava program u režimu debagovanja (eng. *debug mode*). To se postiže biranjem neke od komandi podmenija *Start debug* koji se nalazi u meniju *Build.* Komande su: *Go, Step Into, Run to Cursor.*

Ali, pre nego što se pokrene debager, aplikacija mora biti prevedena tako da sadrži informacije potrebne za debagovanje (eng. *debug info).*

5.1 Pripremanje aplikacije za debagovanje

Ako je projekat napravljen pomoću nekog od čarobnjaka, verovatno neće biti neophodne promene konfiguracija *Debug* je jedna od podrazumevanih. Ali, ako postoji potreba za pravljenjem ove konfiguracije, to će se uraditi podešavanjem u dijalogu *Project settings.* Ovaj dijalog se otvara biranjem komande *Settings* menija *Project.*

Kada je pokrenut dijalog, potebno je odabrati stranu C/C++. Izabrati kategoriju *General* i sa leve strane konfiguraciju za koju se želi da bude *debug* konfiguracija. Zatim, uraditi sledeće:

- U polju *Debug info* odabrati *Program Database*
- U polju *Optimizations* odabrati *Disable (Debug).*

Još je bitno da projekat bude povezan sa *debug* verzijom biblioteke C *Run-time Library.* Odabrati kategoriju *Code Generation*, a zatim željenu *debug* biblioteku.

Treba još podesiti povezivanje. Odabrati stranu *Link* i u polju *Category* odabrati *General.* Potvrditi *Generate debug info.*

Korak po korak:

- Odabrati komandu *Project->Settings*
- Pošto je odabrana samo konfiguracija *Debug* za projekat "MojProgram", izabrati na desnoj strani samo taj projekat.
- Odabrati stranu C/C++ i kategoriju *General.* Iz liste *Optimizations* odabrati *Disable (Debug),* a iz liste *Debug info* stavku *Program Database for Edit and Continu*, a zatim kategoriju *Code Generation* i iz liste *Use run-time library* stavku *Debug Single-Threaded.*

- Odabrati stranu *Link* i potvditi *Generate debug info* i *Link incrementally*
- Pritisnuti OK

U cilju osiguranja da će sledeći put biti izgrađena *Debug* konfiguracija projekta, odabrati komandu *Build->Set Active Configuration*. U dijalogu *Set Active Project Configuration* odabrati konfiguraciju "MojProgram" - *Win32 Debug* i pritisnuti OK.

Da bi projekat bio kompletno spreman za debagovanje, treba ga ponovo izgraditi sa novim podešavanjima. Ako projekat "MojProgram" nije trenutno aktivan, potrebno je aktivirati ga i odabrati komandu *Build->Rebuild All*.

5.2 Pokretanje aplikacije u režimu debagovanja

Pošto se ponovo prevede cela aplikacija nakon svih podešavanja, može se pokrenuti biranjem bilo koje komande iz podmenija *Start Debug* menija *Build.*

Kada se započne debagovanje, pojaviće se neki od prozora debagera. Takođe će nestati neki prozori koji su inače vidljivi. Glavni meni će se promeniti i meni *Build* će biti zamenjen menijem *Debug.*

Aplikacija koja se pokreće u režimu debagovanja izvršava se dok ne stigne do tačke prekida ili se ne prekine u izvršavanju biranjem *Stop debugging* ili *Break.*

Tokom debagovanja, MSVC prikazuje informacije u nizu prozora. Ako neki od njih nisu vidljivi, mogu se prikazati biranjem odgovarajuće komande podmenija *Debug Windows* menija *View.*

5.3 Prozori debagera i pregled objekata

Prozor *Variables* prikazuje vrednosti objekata u tekućoj funkciji. Ovaj prozor ima tri strane:

- Strana *Auto* prikazuje objekte koji se koriste u tekućem i prethodnom redu programskog koda.
- Strana *Locals* prikazuje objekte koji su lokalni u tekućoj funkciji, uključujući i argumente funkcije.

- Strana *This* prikazuje podatke članove objekta na koji pokazuje *this* pokazivač tekuće funkcije članice.

Prozor *Variables* takođe može da prikaže objekte u oblasti važenja funkcija koje su pozvale tekuću funkciju.

Ovaj prozor može da se koristi i za modifikovanje vrednosti prostih objekata ugrađenog tipa. Da bi se promenila takva vrednost, potrebno je pritisnuti je dvaput u prozoru *Variables.* Ako je modifikacija moguća, pojaviće se kursor tastature.

Prozor *Watch* se koristi za proveru vrednosti izraza. Izraz uneti u polje *Name.* Ovaj prozor ima četiri iste strane, tako da u isto vreme se mogu pratiti vrednosti četiri različita izraza. Isto kao i prozor *Variables* i on se može koristiti za modifikovanje vrednosti.

Postoje još dva načina pregleda promenljivih: *QuickWatch* i *DataTips.*

DataTips liči na *ToolTips.* Za vreme debagovanja, postaviti i zadržati par sekundi kursor miša iznad imena objekta, i njegova vrednost će se prikazati (ako ta vrednost može da se odredi). Za izračunavanje vrednosti izraza koristiti dijalog *QuickWatch* koji će se otvoriti komandom *QuickWatch* menija *Debug.* Ako se kursor trenutno nalazi iznad imena objekta, njegova vrednost će se automatski pojaviti u dijalogu *QuickWatch.* Ako je obelezen izraz, pojaviće se njegova vrednost.

Prozor *Registers* prikazuje trenutne vrednosti u registrima procesora.

Prozor *Memory* ornogućava praćenje sadržaja memorije u adresnorn prostoru procesa koji se trenutno debaguje.

Prozor *Call Stack* prikazuje istoriju poziva funkcija koja je dovela do poziva tekuće funkcije. Višestruko pritiskanje funkcije u ovorn prozoru promeniće sadržaj ostalih *debug* prozora, tako da odgovaraju toj funkciji.

Prozor *Disassembly* obezbeđuje pogled na generisani asemblerski kod. Dok ovaj prozor ima fokus, izvršavanje korak po korak drugačije funkcioniše: umesto da prolazi kroz redove C++ koda, prolazi kroz zasebne asemblerske instrukcije.

5.4 Tačke prekida

Najjednostavniji način da se postavi tačka prekida jeste da se postavi kursor na željeno mesto u kodu i pritisne F9. Pojaviće se kruzić sa leve strane reda.

Mnogo bolja kontrola postiže se biranjern *Edit->Breakpoints*. Biranjem ove komande pojavljuje se dijalog *Breakpoints,* gde se mogu postaviti tri različite vrste tačaka prekida:

- *Location breakpoints* prekidaju izvršavanje programa na određenoj lokaciji. Ovo su tačke prekida koje se postavljaju sa F9. Može se dodati i provera uslova na tački prekida, tako da se izvršavanje programa ne prekida svaki put, već samo kada je ispunjen taj uslov.
- *Data breakpoints* prekidaju izvršavanje programa kada se promeni vrednost određenog izraza.
- *Message breakpoints* prekidaju izvršavanje programa kada neka od procedura dobije određenu poruku.

Komande za izvršavanje su sledeće:

- Komanda *Step Into* izvršava sledeći red koda ili sledeću instrukciju u prozoru *Disassembly.* Ako je u tom redu poziv funkcije, ovom komandom se ulazi u telo te funkcije.
- Komanda *Step Into Specific Function* koristi se radi kontrole u koju se funkciju ulazi u slučaju ugneždenih poziva funkcija u jednom redu koda.
- Komanda *Step Over* je slična komandi *Step into,* osim što se ne ulazi u telo funkcije ako je u tom redu koda poziv funkcije, već se samo izvrši poziv kao jedinstvena akcija.
- Komanda *Step Out* nastavlja izvršavanje programa do kraja tekuće funkcije, izlazi iz nje i prekida izvršavanje iza reda u kome je ta funkcija pozvana.
- Kornanda *Run to Cursor* nastavlja izvršavanje programa do reda gde se trenutno nalazi kursor tastature ili dok se ne naiđe na tačku prekida.
- Dok izvršavanje programa stoji, može se pomoću komande *Set Next Statement* zadati naredba gde izvršavanje treba da se nastavi.

Korak po korak:

- Otvoriti fajI “MojProgram.cpp”.
- Postaviti kursor tastature na prvi red koda u funkciji *main*.
- Pritisnuti F9.
- Odabrati komandu *Build->Start Debug->Go*.
- Kada se program pokrene u režimu debagovanja i stane kod tačke prekida koja je postavljena, pritisne se F11 (ili odabere *Debug->Step Into*). Potrebno je pritiskati F11 i pratiti kuda prolazi izvršavanje programa. Kada se želi sa nastavkom izvršavanja u telu funkcije koja se poziva u tekućem redu koda, pritisnuti F10. Dok teče izvršavanje programa, pratiti šta se dešava u vidljivim prozorima. Postaviti kursor miša iznad imena objekta za čiju trenutnu vrednost postoji interesovanje.

- Kada izvršavanje programa stigne do drugog reda koda funkcije *main*, pa se nakon lančanih poziva funkcija nađe u funkciji članici CPravougaonik::setVelicina(), odabrati komandu *View->Debug Windows->Call Stack*. Proučiti sadržaj tog prozora i ukloniti ga. Zatim pritisnuti SHIFT + F11 da bi se izvršavanje programa nastavilo izvan tekuće funkcije.

- Kada izvršavanje programa stane nakon postavljanja promenljivih a i b u funkciji *main*, proveriti vrednost izraza a*a-2*a*b+b*b koristeći prozor *Watch.*
- Kada izvršavanje programa stane na kvadrat.setVelicina(a), potrebno je promeniti vrednost promenljive a, tako što će se pritisnuti polje *Value* u prozoru *Variables* i uneti željena vrednost. Potrebno je imati u vidu da je a neoznačen ceo broj, pa i vrednost koja se zadaje mora biti odgovarajućeg tipa.

Dodatak B: Integrisano razvojno okruženje Microsoft Visual C++ 2008



Slika 1.1.1: *Microsoft Visual Studio* 2008 logo

1. Uvod

Prvo izdanje programa *Visual Studio* pojavilo se na tržištu 1997. godine i sadržao je odvojene IDE-eve (eng. *Integrated Development Environment*) za programe *Visual* C++, *Visual Basic*, J++ i alatku poznatu kao *InterDev*. Program *Visual Studio* 6.0 imao je drastična poboljšanja koja su označila rođenje programa *Visual Basic* 6 i koja su oslikavala ideju skupa jedinstvenih usluga za sve jezike.

Visual Studio .NET 2002 i *Visual Studio* .NET 2003 su tu ideju sproveli u delo sa aplikacijom .NET *Framework*. Po prvi put programer je mogao da napiše aplikaciju na jeziku po svom izboru korišćenjem mogućnosti opšteg skupa alatki uključujući dizajnere, kontrole *Drag&Drop* i *IntelliSense* (*intelliSense* je *Microsoft*-ov mehanizam automatskog dopunjavanja kôda). Zajedno sa porastom produktivnosti, došlo je i do porasta u veličini i kompleksnosti razvojnih projekata i timova.

Visual Studio 2008 je napravio revoluciju u načinu na koji se rukuje podacima. Uz novi alat *Language Integrated Query* (LINQ), može se raditi sa svim vrstama podataka sa konzistentnim pristupom i pristupati podacima sa novim dizajnerom za podatke. LINQ omogućava programerima lakši pristup podacima kroz skup oznaka tipa datoteke u programu C# i *Visual Basic* programskim jezicima, kao i u aplikaciji *Microsoft* .NET *Framework*. Ove oznake tipa datoteke omogućavaju integrisane upite za objekte, baze podataka i XML podatke. Koristeći LINQ, mogu se napisati upiti u svom jeziku u programu C# ili programu *Visual Basic* bez korišćenja posebnih jezika, kao što su SQL i *Xpath*.

Pored površina za dizajn, u programu *Visual Studio*, dizajneri korisničkih interfejsa mogu koristiti poznate alatke kao što su *Microsoft Expression Suite* kako bi upravljali prikazima, kontrolama i povezivanjem

podataka. Datoteke sa rešenjima koje su generisane alatom *Expression* mogu se otvoriti i uređivati u programu *Visual Studio*. Dizajneri i programeri su u mogućnosti da izgrade biblioteke opšteg dizajna korisničkog interfejsa, formata i elemenata kojima se može lako upravljati i koji se mogu ponovo koristiti.

Visual Studio 2008 je jedinstven skup alata koji omogućava pravljenje kvalitetnih aplikacija na *Microsoft*-ovim platformama. Podrška za razvoj operativnog sistema *Windows Vista* i sistema *Microsoft Office* pomaže kreiranju efektne i bogate aplikacije. Uz uključenu podršku za ASP.NET AJAX i programski dodatak *Silverlight* za *Visual Studio* 2008, moguća je izgradnja velikog broja bogatih interaktivnih aplikacija za *Web*.

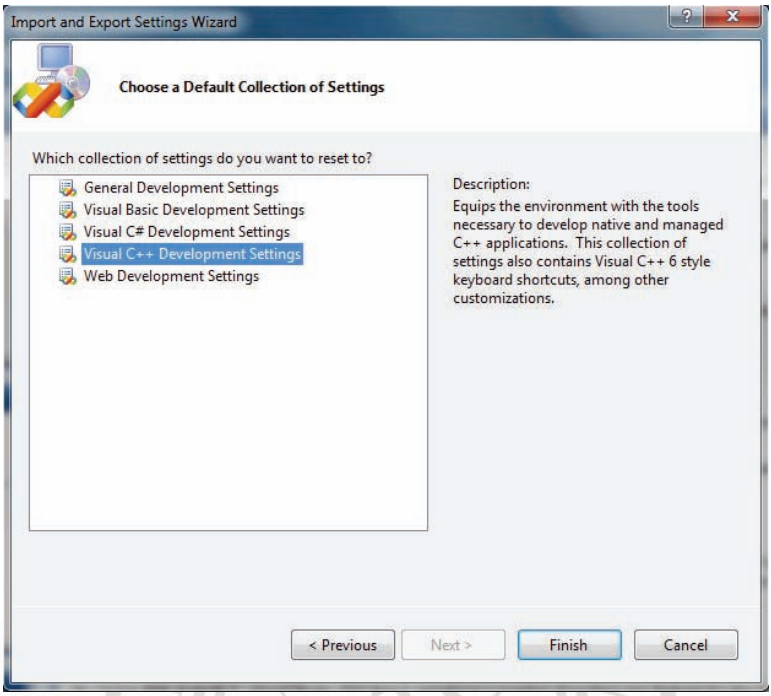
Visual Studio omogućava razvoj WPF aplikacija (za programiranje u C#) u cilju direktnog uređivanja XAML (sa *IntelliSense* podrškom) ili kreiranja korisničkog interfejsa kroz nove vizuelne dizajnere. Promena koja je načinjena u prikazu aplikacije jednom od ovih alatki odmah se oslikava na drugoj. Pored toga, program *Visual Studio* nudi podršku za više od 8.000 novih osnovnih API-ja u operativnom sistemu *Windows Vista*.

Lako korišćenje znači da *Visual Studio* u velikoj meri asistira korisniku i umesto njega radi mnoštvo poslova koji su dobro definisani. U prevodu, umesto korisnika pravi kostur aplikacije i time oslobađa korisnika od tog posla, kao i od učenja kako se to radi.

1.1 Razvojno okruženje

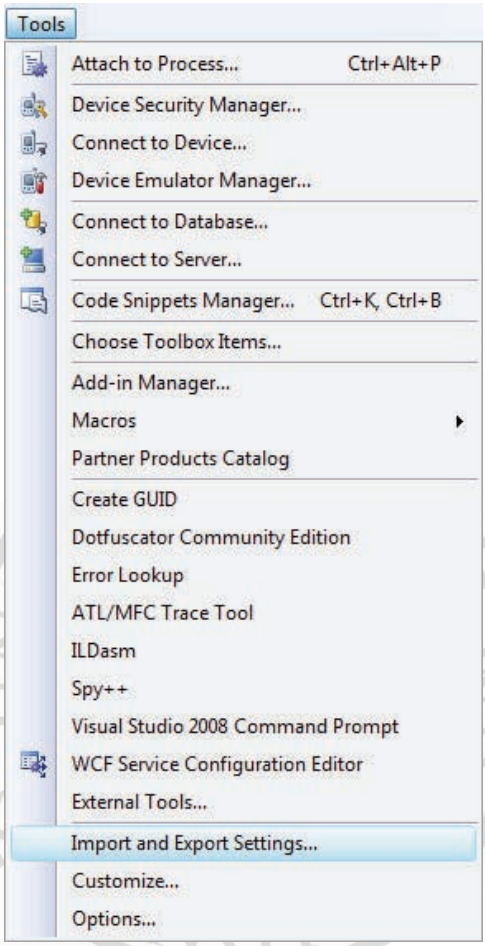
Visual Studio 2008 razvijan je više od tri godine pod kodnim imenom Orcas. Zamišljen je kao razvojno okruženje koje bi trebalo da omogući programerima da u potpunosti iskoriste mogućnosti operativnog sistema *Windows Vista*.

Kod prvog pokretanja *Visual Studio* 2008, uočiće se da je izgled radnog okruženja identičan okruženju verzije 2005. Na početku, korisniku se nudi mogućnost da izabere u kom programskom jeziku želi da radi kako bi u skladu s'tim bilo podešeno radno okruženje prilagođeno njegovim potrebama (slika 1.1.1).



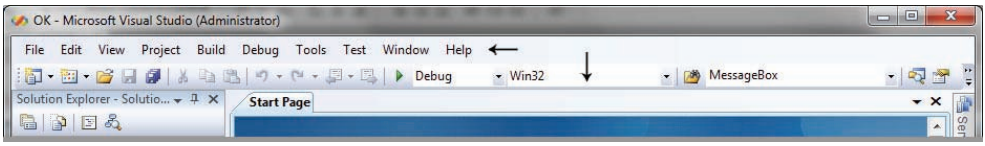
Slika 1.1.1 Odabir programskog jezika

Naravno, ukoliko se naknadno javi potreba za radom u drugom okruženju, aktivno okruženje se veoma lako menja korišćenjem opcije *Import and Exports settings* koja se nalazi u meniju *Tools* (slika 1.1.2).



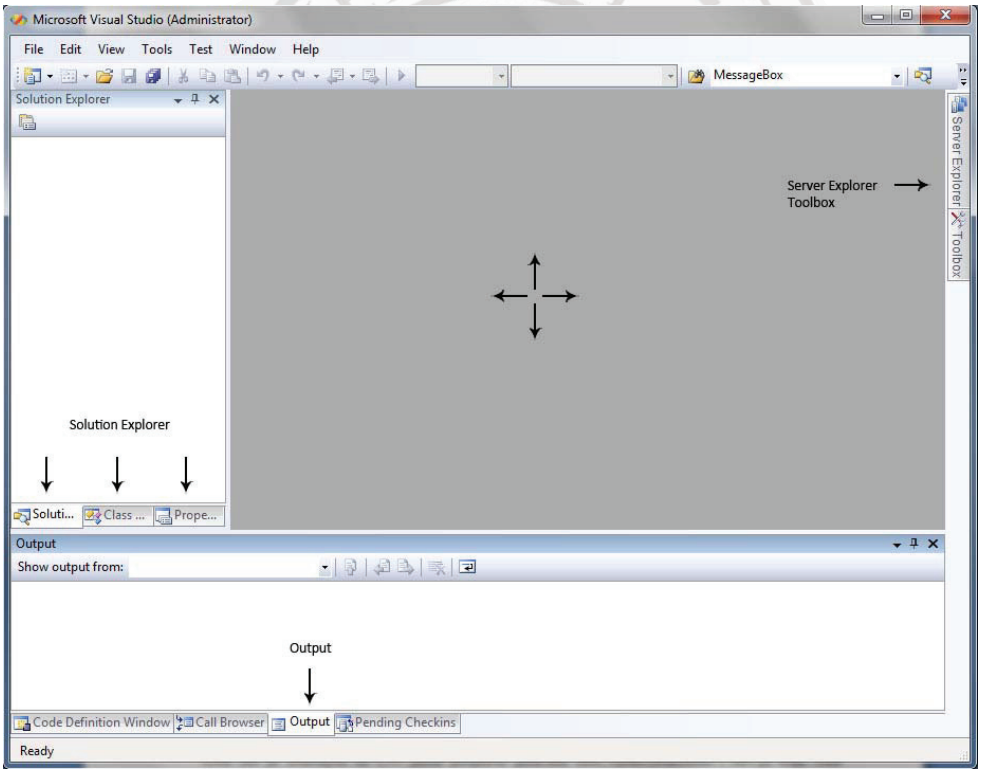
Slika 1.1.2 *Import* i *Export Settings*

Ispod glavnog menija, nalazi se *toolbar* sa ikonama za manipulaciju projektima i fajlovima (slika 1.1.3), izbor aktivnih prozora koji se prikazuju u okruženju, rad sa tekstom i manipulisanje aplikacijom na kojoj se radi i još mnogo drugih opcija.



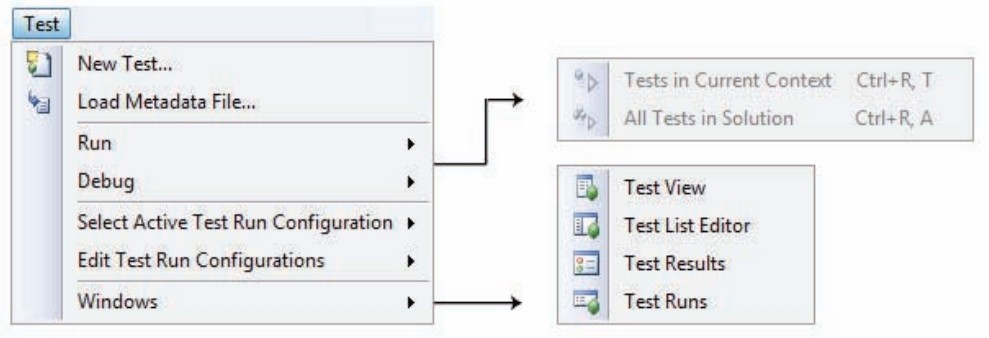
Slika 1.1.3 *Import* i *Export Settings*

Sa leve strane prozora je inicijalno prikazan *Solution Explorer* (pregled fajlova u aktivnom projektu), *Class View* (prikaz klasa u aktivnom projektu) i *Resource Explorer* (prikaz resursa aktivnog projekta). Centralni deo aplikacije zauzima prozor u kome se modifikuje programski kod ili resursi koji se u programu koriste, uz desnu ivicu prozora su prikačeni *Server Explorer i Toolbox* koji se otvaraju na klik korisnika. Dno prozora je rezervisano za sekciju *Output* (slika 1.1.4).



Slika 1.1.4 Okruženje

Naravno, ovu inicijalnu konfiguraciju radnog okruženja moguće je proizvoljno menjati prema željama korisnika. Iako je izgled radnog okruženja identičan, unapređeni IDE navigator omogućava lakše kretanje između delova okruženja, *doking* prozora u okruženju je unapređen i olakšan, kao i mogućnost podešavanja *default* fonta za okruženje.



Slika 1.1.5 Test meni

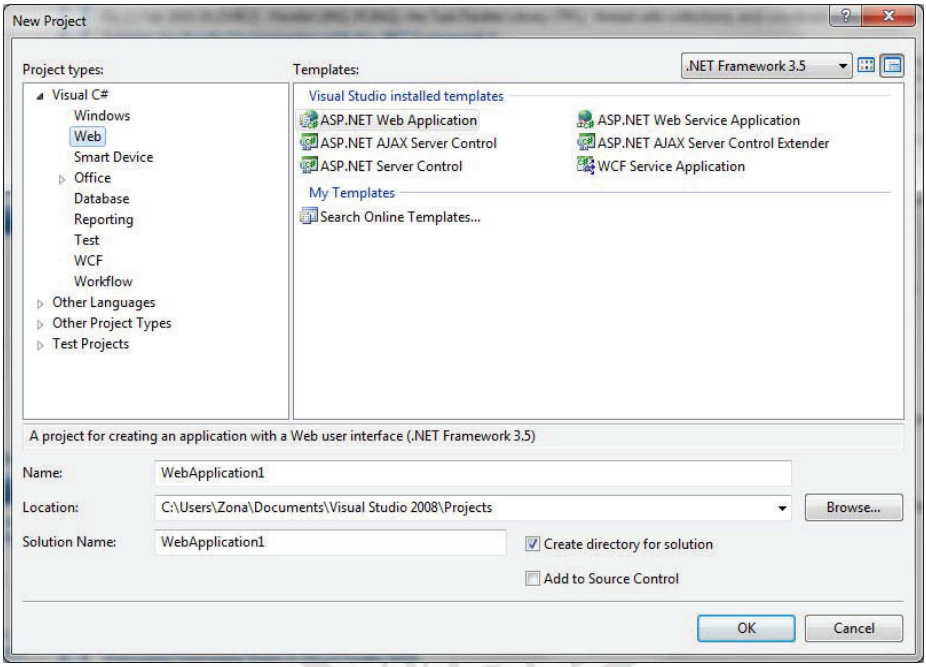
U promene spada novododati *Test* meni, kao i odsustvo menija *Community,* koji je postojao u prethodnoj verziji (*Visual Studio* 2005)*,* a čije su preostale opcije premeštene u *Help* meni. Opcije *Test* menija (slika 1.1.5) nude programerima mogućnost lakšeg kreiranja *unit* testova za aplikacije koje razvijaju, kao i bolju kontrolu nad njihovim izvršavanjem, što doprinosi razvoju kvalitetnijih, aplikacija koje su bolje testirane.

U odnosu na prethodnu verziju *Visual Studio*-a, kod verzije 2008, značajnih noviteta po pitanju korisničkog okruženja - nema. Noviteti *Visual Studio*-a 2008, u odnosu na ranije verzije su sledeće:

- Novi *template*-i za *Visual Basic* i *Visual* C#
- Kod C++ javlja se podrška svih funkcionalnosti i API-ja koje nudi *Windows Vista* (više od 150 funkcija i 18 klasa, preko 8000 API-ja)
- *Class Designer*, alat koji je u prethodnoj verziji *Visual Studio*-a omogućavao vizuelno dizajniranje i dokumentovanje klasa samo za *Visual Basic* i *Visual* C#, ovoga puta postoji i za *Visual* C++

U *Visual Studio* 2008, posebna pažnja posvećena je *Web* aplikacijama. Tako je među *template*-ove za *Visual Basic* i *Visual* C# dodata posebna potkategorija *template*-ove nazvana *Web* u kojoj su ponuđeni tipični kosturi nekoliko vrsta *Web* aplikacija (slika 1.1.6). Jedna od mogućnosti jeste ugrađena podrška za *JavaScript intelliSense*. Takođe,

unapređeno je debagovanje – postavljanje *breakpoint*-a u *JavaScript* pre izvršavanja ASP.NET stranica, reorganizovanje *breakpoint*-a dok se stranice još izvršavaju, poboljšan prikaz sadržaja objekata i još mnogo detalja koji u svakodnevnom radu puno znače. Kada je rad sa podacima u pitanju, *Visual Basic* i *Visual* C# nude mogućnost dobijanje podataka iz baza, XML fajlova ili čak objekata, korišćenjem LINQ, specijalnih *query* operatora koji su integrisani u ove jezike.

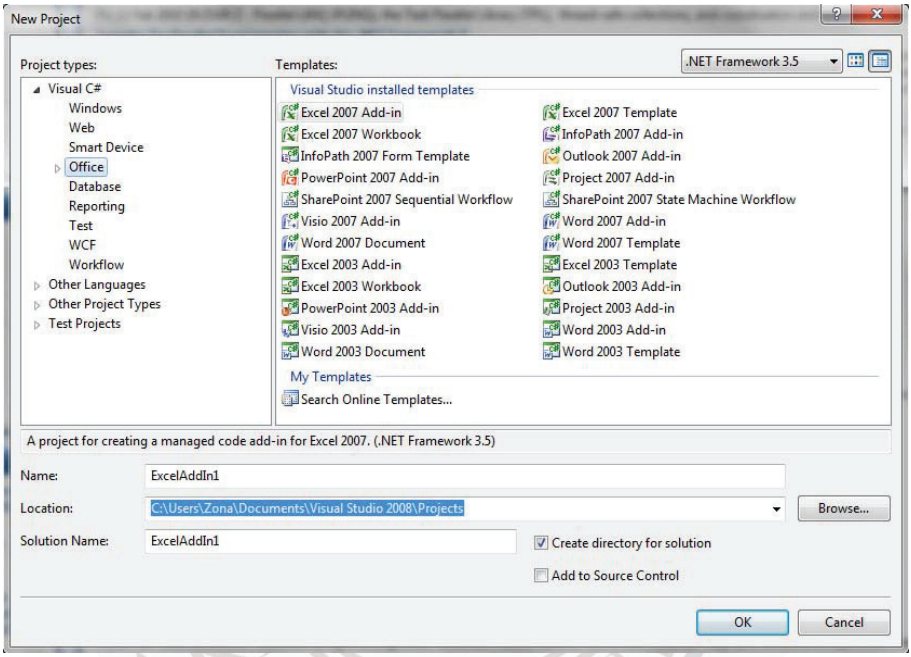


Slika 1.1.6 *Web templates*

Još jedna nova kategorija *template*-ova koja je dodata skupu VB i C# *template*-ova je *Office,* koja nudi kreiranje *Office* 2003/2007 projekata, od kojih su (slika 1.1.7):

- *Excell Add-in*
- *Workbook*
- *Word Add-in*
- *Document*
- *Visio Add-in*

Tehnologija koja stoji iza ovih *template*-ova je *Visual Studio Tools for Office* (VSTO) i ona omogućava korišćenje otvorenih funkcija bilo koje od *Office* aplikacija za kreiranje samostalnih aplikacija ili dokumenata zasnovanih na *Office* aplikacijama.



Slika 1.1.7 *Office templates*

1.2 *IntelliSence* ikone

IntelliSense je *Microsoft*-ov mehanizam automatskog dopunjavanja kôda. Kada se ukuca tačka nakon imena klase, *Visual Studio* će ponuditi listu svih članova te klase. Ovo smanjuje broj grešaka nastalih prilikom unošenja imena promenljivih. Pored svakog imena sa liste članova, nalazi se ikonica koja asocira na tip (da li je metod, klasa, struktura i sl.). Pomenute ikonice su prikazane na slici 1.2.1.

IntelliSense ikonice nisu vezane samo za klase. Postoji još mnogo drugih koji će se javljati u različitim delovima programa.

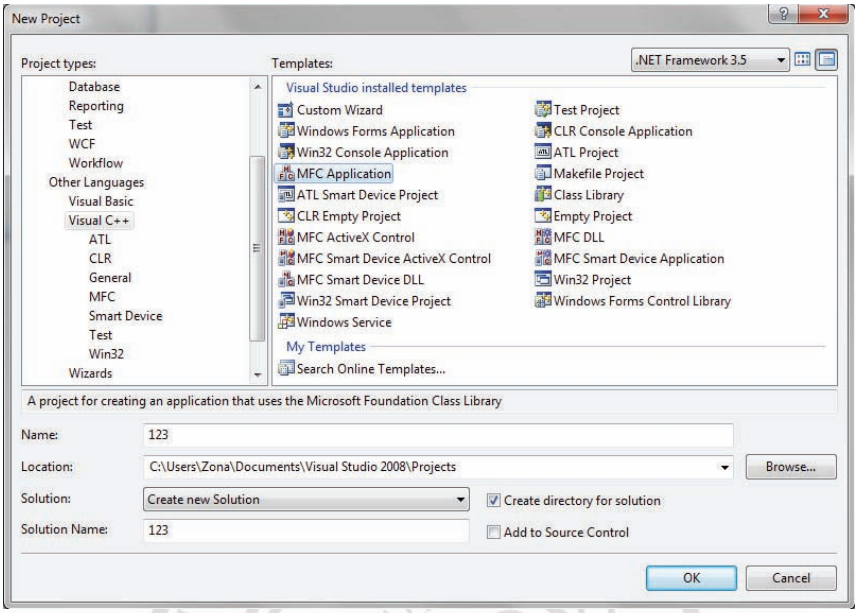
Ikonica	Značenje
	Metod
	Svojstvo
	Klasa
	Struktura
	Enum
	Okruženje

Slika 1.2.1 *IntelliSence* ikonice

2. Kreiranje grafičkih aplikacija

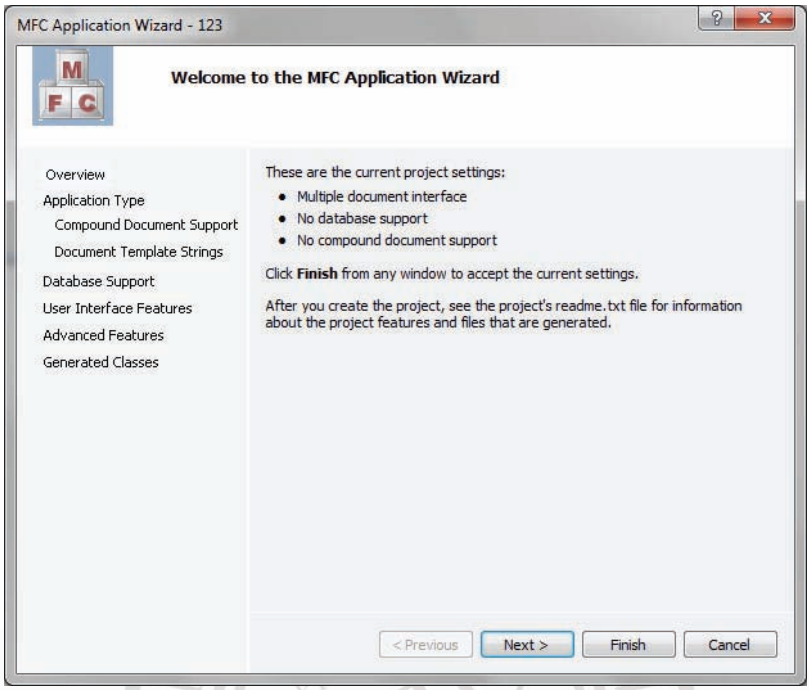
Korišćenjem *Visual Studio* 2008 veoma jednostavno se mogu kreirati grafičke aplikacije koje se baziraju na formi. *Visual Studio* 2008 omogućava kreiranje dva različita tipa grafičkih aplikacija kroz dve različite vrste projekata – *Windows forms applications* i MFC *application*. Pomenuto se isključivo odnosi na programiranje u *Visual* C++ (jezik C# za kreiranje grafičkih aplikacija koristi WPF *application*). Izborom jednog ili drugog tipa projekta *Visual Studio* kreira potreban kostur aplikacije i omogućava interaktivno kreiranje grafičkog korisničkog okruženja.

Ispratiti putanju *File -> New Project* za kreiranje novog projekta. Kreiranje kompletnog kostura moguće je odabirom *template*-a MFC *Application* (slika 2.1).



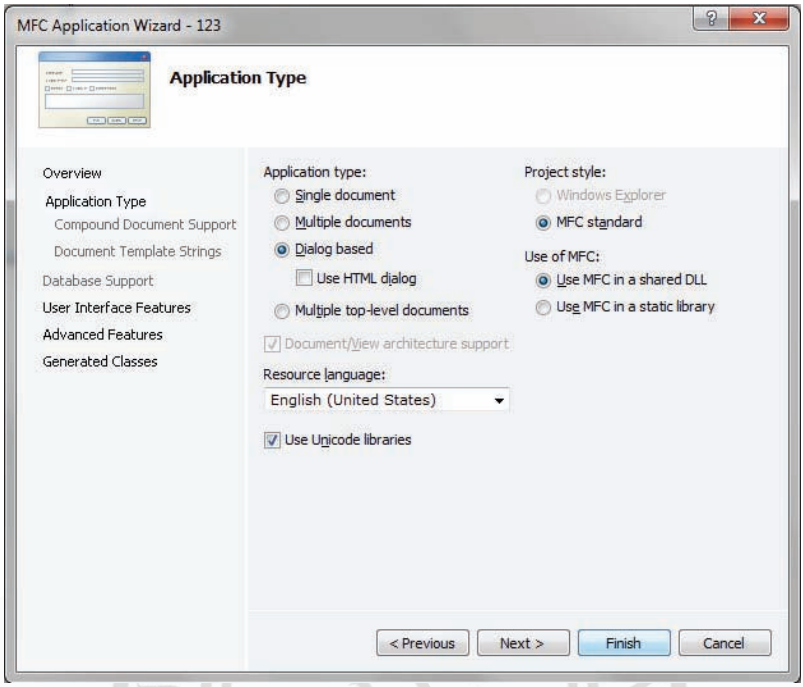
Slika 2.1 MFC *Application*

Nakon unosa imena i lokacije na kojoj će projekat biti sačuvan, kliknuti OK. Pokrenuće se MFC *Application Wizard* koji upućuje na trenutno podešene parametre, koji će izgraditi kostur samo klikom na dugme *Finish*. Ukoliko su potrebna dalja podešavanja, potrebno je kliknuti na dugme *Next* (slika 2.2).



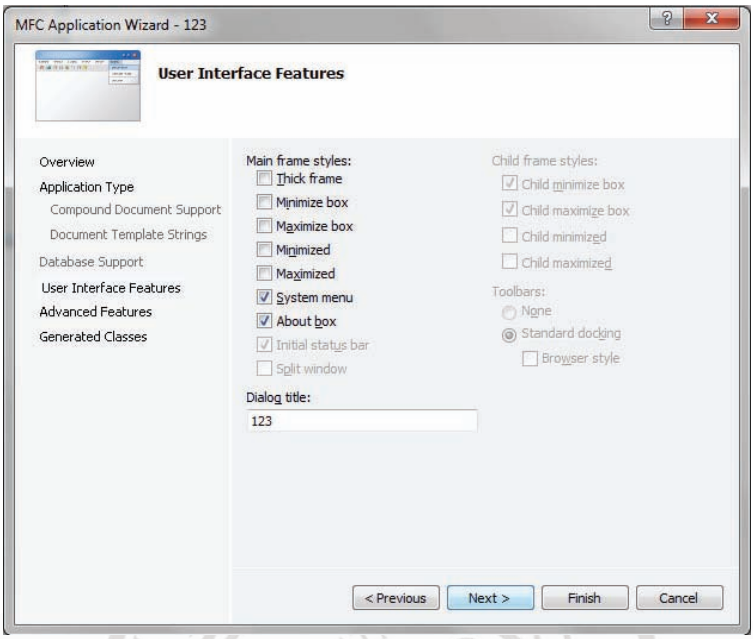
Slika 2.2 MFC *Application Wizard*

Pritiskom na dugme *Next*, otvara se prozor za odabir tipa aplikacije. Savet je, na početku, da se odabere *Dialog based applcation* (slika 2.3).

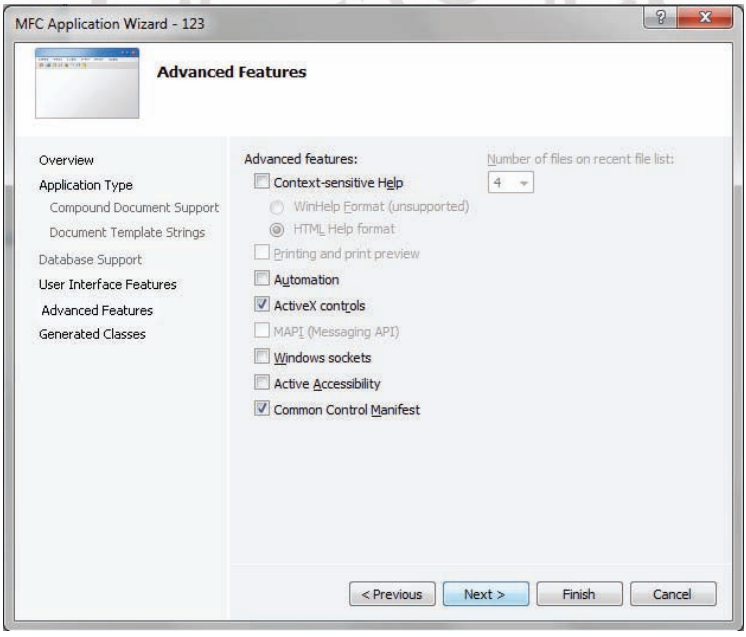


Slika 2.3 *Application Type*

Sledeća dva prozora nude mogućnosti dodavanja delova funkcionalnosti prozora aplikacije, raznih dugmića novog dijaloga, *ActiveX* kontrola, kao i unošenje naslova koji će se nalaziti na vrhu novog prozora (slike 2.4 i 2.5).

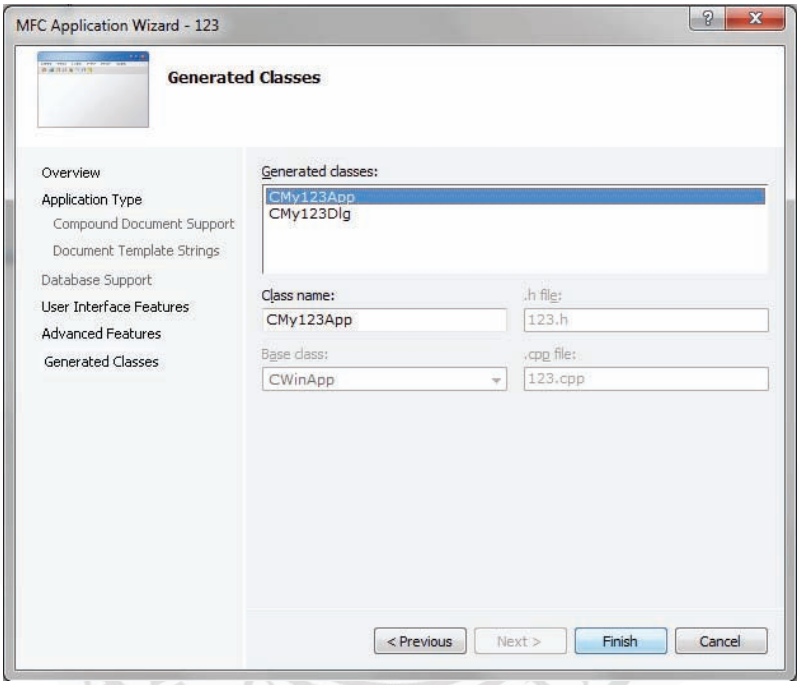


Slika 2.4 *User Interface Features*

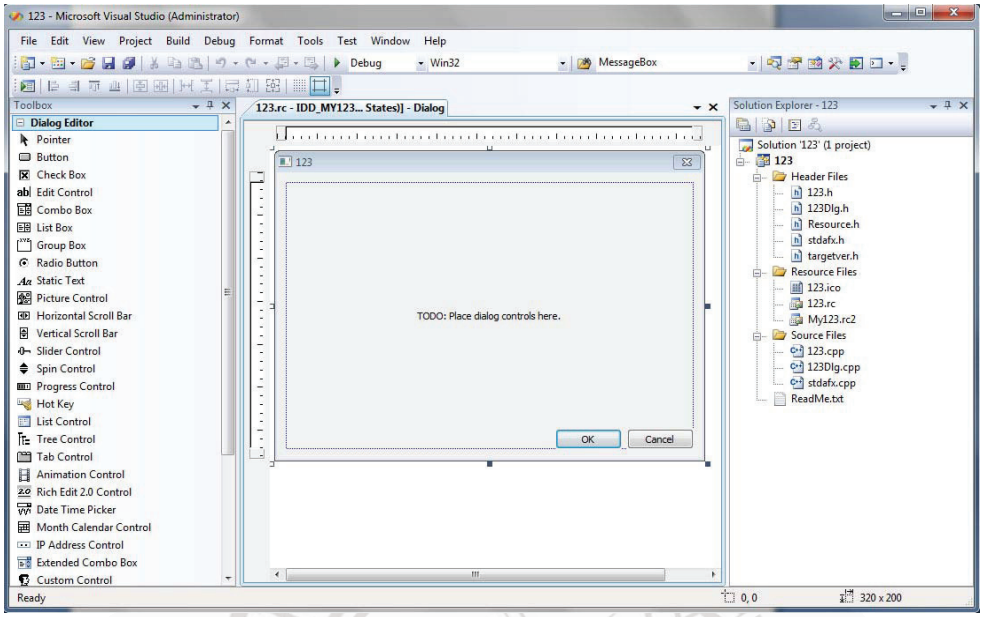


Slika 2.5 *Advanced Features*

Na kraju *MFC Application Wizard* generiše klase i daje prikaz njihovih imena (slika 2.6). Klikom na *Finish* otvara se novo radno okruženje sa podešenim opcijama i praznim dijalogom za dalje kreiranje aplikacije (slika 2.7).



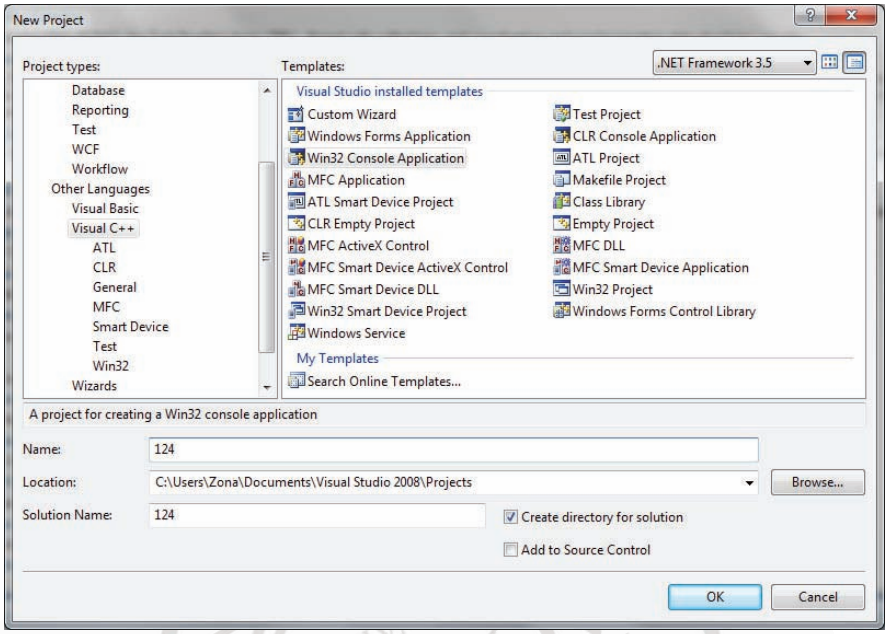
Slika 2.6 *Generated Classes*



Slika 2.7 Radno okruženje nakon kreiranja kostura grafičke aplikacije

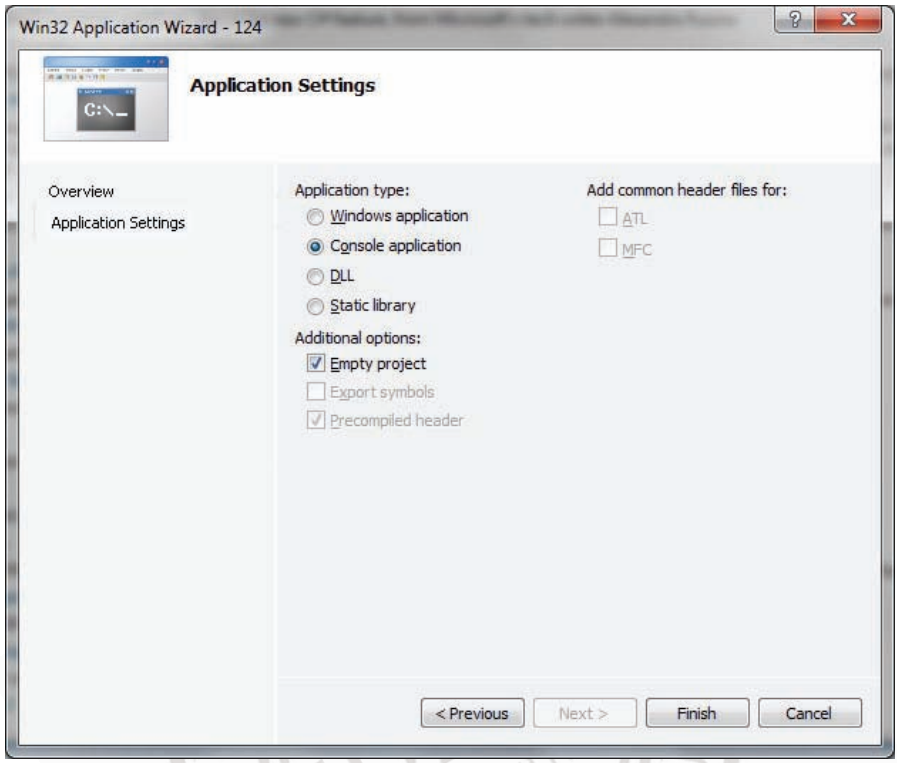
3. Kreiranje konzolnih aplikacija

Ispratiti putanju *File -> New Project* za kreiranje novog projekta. Kreiranje konzolih aplikacija počinje odabirom Win32 *Console Application*, iz liste *template*-ova (slika 3.1). Nakon unošenja imena i lokacije na kojoj će se aplikacija nalaziti, kliknuti na dugme OK.



Slika 3.1 Kreiranje kostura konzolne aplikacije

Za razliku od procesa kreiranja grafičkih aplikacija, kostur konzolne aplikacije se definiše daleko jednostavnije. Već sledeći prozor nudi opciju kreiranja praznog projekta ili projekta sa prekompajliranim zaglavljem. Potrebno je selektovati opciju *Empty project* (slika 3.2) kako bi se kreirao prazan projekat, čiju bi izgradnju korisnik mogao da kontroliše.



Slika 3.2 Kreiranje praznog projekta

3.1 Dodatak – MSDN *Subscriptions Comparison Chart*

MSDN Subscriptions Comparison Chart

	Visual Studio Team System 2008 Team Suite with MSDN Premium Subscription	Visual Studio Team System 2008 Development Edition with MSDN Premium	Visual Studio Team System 2008 Architecture Edition with MSDN Premium	Visual Studio Team System 2008 Test Edition with MSDN Premium	Visual Studio Team System 2008 Database Edition with MSDN Premium	Visual Studio 2008 Professional with MSDN Premium	Visual Studio 2008 Professional with MSDN Professional	Visual Studio 2008 Professional Edition	Visual Studio 2008 Standard Edition
MULTI-LANGUAGE DEVELOPMENT ENVIRONMENT FOR THE PROFESSIONAL DEVELOPERS									
Database project with source control integration of all database objects and support for offline representation of a database schema	●				●				
Database Unit Testing, Data Generation, Database Refactoring, Schema Compare, Data Compare, Database Schema Deployment Tools	●				●				
Advanced Database and T-SQL Tools	●				●				
Unit Testing	●	●		●		●	●	●	
Code Coverage	●	●		●					
Code Metrics, Static Code Analyzer, Code Profiler, Dynamic Code Analyzer	●	●							
Application Designer, System Designer, Logical Infrastructure Designer, Deployment Designer	●		●						
Load Testing, Manual Testing, Test Case Management, and Web Testing	●			●					
Code Profiler integration with Web and Load Test tools	●								
IntelliSense	●	●	●	●	●	●	●	●	●
Code Editor, Code Snippets	●	●	●	●	●	●	●	●	●
Reusable component creation	●	●	●	●	●	●	●	●	
Server Development Tools	●	●	●	●	●	●	●	●	
Advanced debugging tools including cross-machine debugging	●	●	●	●	●	●	●	●	
Smartphone, Pocket PC, and Windows CE Development, Web application precompilation	●	●	●	●	●	●	●	●	
Visual Basic® and Visual C#® Stored Procedure Authoring and Deployment	●	●	●	●	●	●	●	●	
Visual Database Tools, Oracle and DB2 database access, XSD Editor, XSLT Editor and Debugger	●	●	●	●	●	●	●	●	
Web application project, JavaScript IntelliSense, JavaScript debugging, Web data controls, ASP.NET AJAX project templates	●	●	●	●	●	●	●	●	●
LINQ support	●	●	●	●	●	●	●	●	●
Multi-targeting support	●	●	●	●	●	●	●	●	●
Visual Studio Tools for Office	●	●	●	●	●	●	●	●	
Visual Basic, Visual C#, Visual C++®, Visual Web Developer	●	●	●	●	●	●	●	●	●
Action Pane and Smart Tag programmability	●	●	●	●	●	●	●	●	
DEVELOPER TOOLS									
Visual Studio Team System 2008 Team Suite	●								
Visual Studio Team System 2008 Development Edition	●	●							
Visual Studio Team System 2008 Architecture Edition	●		●						
Visual Studio Team System 2008 Test Edition	●			●					
Visual Studio Team System 2008 Database Edition	●				●				
Visual Studio 2008 Professional Edition	●	●	●	●	●	●	●	●	
Visual Studio Team System 2008 Team Foundation Server Workgroup Edition [1] & Client Access License	●	●	●	●	●				
DESIGNER TOOLS									
Expression Web & Expression Blend	●	●	●	●	●	●			
Expression Studio	●								
OPERATING SYSTEMS									
Windows Vista Ultimate/Enterprise/Business/Home Premium/Home Basic	●	●	●	●	●	●	●		
Windows XP Professional/Home/Media Center Edition/Tablet PC Edition	●	●	●	●	●	●	●		
Windows Server	●	●	●	●	●	●	●		
Compute Cluster, Windows SharePoint® Services, Windows Services for UNIX	●	●	●	●	●	●	●		
SERVERS									
SQL Server	●	●	●	●	●	●	●		
BizTalk® Server, Exchange Server, Commerce Server, Content Management Server, ISA Server	●	●	●	●	●	●			
Host Integration Server, Identity Integration Server, Connected Services Framework, Customer Care Framework	●	●	●	●	●	●			
Live Communications Server, Forms Server, Groove Server, PerformancePoint™ Server, Speech Server, SharePoint Server, Windows SharePoint Services	●	●	●	●	●	●			
Microsoft Operations Manager, Microsoft System Center Capacity Planner & Data Protection Manager, Systems Management Server	●	●	●	●	●	●			
BUSINESS PRODUCTIVITY APPLICATIONS									
Office Ultimate/Enterprise/Professional 2007 [2]	●	●	●	●	●	●			
Office Word, Office Excel, Office PowerPoint, Office Outlook with Business Contact Manager, Office Access [2]	●	●	●	●	●	●			
Office Publisher, Office InfoPath, Office OneNote®, Office Communicator, Office Groove®, Office SharePoint Designer, Office Visio, Office Project Standard [2]	●	●	●	●	●	●			
Accounting, Business Scorecard Manager, MapPoint®, FrontPage®	●	●	●	●	●	●			
Office Project Professional	●	●	●	●	●	●			
Office Project Server, Office Project Portfolio Server	●	●	●	●	●	●			
MICROSOFT DYNAMICS (FORMERLY KNOWN AS MICROSOFT BUSINESS SOLUTIONS)									
Microsoft Dynamics™ AX, GP, NAV, SL, Microsoft Forecaster & FRx	●	●	●	●	●	●			
Microsoft Dynamics CRM, Point of Sale, Microsoft Small Business Accounting & Financials	●	●	●	●	●	●			
OTHER TOOLS, SDKS AND DDKS									
Virtual PC, Virtual Server	●	●	●	●	●	●	●		
Access Developer Extensions	●	●	●	●	●	●	●		
.NET Framework, .NET Compact Framework, .NET Micro Framework	●	●	●	●	●	●	●		
Windows SDK, Platform SDK, DirectX® SDK	●	●	●	●	●	●	●		
Microsoft Baseline Security Analyzer v2.0, Application Compatibility Tool Kit	●	●	●	●	●	●	●		
Windows Automated Installation Kit, Windows Installer	●	●	●	●	●	●	●		
Windows Driver Kit, Windows Hardware Compatibility Test	●	●	●	●	●	●	●		
MSDN LIBRARY	●	●	●	●	●	●	●		
TECHNICAL SUPPORT INCIDENTS	4	4	4	4	4	4	2		
MANAGED NEWSGROUP SUPPORT	●	●	●	●	●	●	●		
ONLINE CONCIERGE	●	●	●	●	●	●	●		

Reference:

1. Laslo Kraus, Programski jezik C++ sa rešenim zadacima, Akademska misao, Beograd, 2004.

2. John C. Molluzzo, "C++ for Business Programmers", Prentice Hall, 2005.

3. Dragan Milićev, Objektno orijentisano programiranje na jeziku C++, skripta sa praktikumom, Mikro knjiga, 2001.

4. Stanley B. Lippman, Josee Lajoie, C++ izvornik, prevod trećeg izdanja, CET, 2000.

5. Tony Gaddis, Starting out with C++, Scott/Jones, 2001.

6. Cay Horstmann, Timothy Budd, Big C++, John Wiley, 2005.

7. William Roetzheim, Borland, C++ 4.5, programiranje za Windows, Mikro knjiga, 1995

8. Julian Templeman, Andy Olsen, Microsoft, Visual C++ .NET, CET, 2002.

Odlukom Senata Univerziteta "Singidunum", Beograd, broj 636/08 od 12.06.2008, ovaj udžbenik je odobren kao osnovno nastavno sredstvo na studijskim programima koji se realizuju na integrisanim studijama Univerziteta "Singidunum".

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.432.2C++

ПОПОВИЋ, Ранко, 1956-
Programski jezik C++ : sa rešenim zadacima / Ranko Popović, Zona Kostić. - Beograd : Univerzitet Singidunum, 2010 (Loznica : Mladost grup). - 412 str. : ilustr. ; 25 cm

Tiraž 250. - Bibliografija: str. 412.

ISBN 978-86-7912-286-5
1. Костић, Зона, 1983- [аутор]
а) Програмски језик "C++"

COBISS.SR-ID 178203916

© 2010.

Sva prava zadržana. Ni jedan deo ove publikacije ne može biti reprodukovan u bilo kom vidu i putem bilo kog medija, u delovima ili celini bez prethodne pismene saglasnosti izdavača.