VISOKA ŠKOLA ELEKTROTEHNIKE I RAČUNARSTVA
STRUKOVNIH STUDIJA

Dr Perica S. Štrbac, dipl. ing.

OBJEKTNO PROGRAMIRANJE 1

Prvo izdanje

Beograd, 2019.

OBJEKTNO PROGRAMIRANJE 1

Autor: dr Perica S. Štrbac, dipl. ing.

Recenzenti: dr Miroslav Lutovac, dipl. ing.
mr Miloš Pejanović, dipl. ing.

Izdavač: Visoka škola elektrotehnike i računarstva strukovnih studija, Beograd, Vojvode Stepe 283, www.viser.edu.rs

Za izdavača: dr Vera Petrović, dipl. ing.

Tehnička obrada: dr Perica Štrbac, dipl. ing.

Dizajn korica: dr Perica Štrbac, dipl. ing.

Godina izdanja: 2019.

Tiraž: 30 primeraka

Štampa: Razvojno-istraživački centar grafičkog inženjerstva TMF, Beograd.

ISBN 978-86-7982-311-3

Odlukom Nastavno-stručnog veća Visoke škole elektrotehnike i računarstva strukovnih studija iz Beograda, na sednici održanoj dana 04.07.2019. godine, odobreno je izdavanje i primena ovog udžbenika u nastavi.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42.045(075.8)

ШТРБАЦ, Перица С., 1968-
Objektno programiranje 1 / Perica S. Štrbac. - 1. izd. - Beograd : Visoka škola elektrotehnike i računarstva strukovnih studija, 2019 (Beograd : Razvojno-istraživački centar grafičkog inženjerstva TMF). - 229 str. : ilustr. ; 26 cm

Tiraž 30. - Bibliografija: str. 223. - Registar.

ISBN 978-86-7982-311-3

a) Објектно оријентисано програмирање

COBISS.SR-ID 282097420

POSVETA

Anđelki

Predgovor

Knjiga je napisana prvenstveno studentima druge godine Visoke škole elektrotehnike i računarstva strukovnih studija u Beogradu za predmet Objektno programiranje 1 koji obrađuje objektno orijentisano programiranje u C++ tehnologiji.

Knjiga kreće od samog početka objektno orijentisanog programiranja u programskom jeziku C++ tako da je podesna i za studente koji su apsolutni početnici za objektno orijentisano programiranje.

Uvodno poglavlje daje smernice za C++ tehnologiju i preporuke za integrisano razvojno okruženje.

Poglavlje Programski jezik C++ obuhvata: elemente koji nisu objektno orijentisani, tipove podataka, oblasti važenja, životni vek objekta, memorijske oblasti i konverzije tipova.

U poglavlju Objektno orijentisani pristup obrađeni su: klase i objekti, konstruktor, konstruktor kopije, move konstruktor, destruktor, inspektori, mutatori, statički članovi, prava pristupa, prijatelji klase, preklapanje operatora, nasleđivanje (javno, zaštićeno, privatno), kreiranje i destrukcija objekata izvedene klase, višestruko izvođenje, virtuelne klase, polimorfizam, apstraktne klase i virtuelni destruktor.

Poglavlje Šabloni obuhvata prikaz upotrebe šablona klasa, specijalizacije šablona i šablona funkcija.

U poglavlju Izuzeci obrađeni su: try-catch blok, korisnički definisani izuzeci, propagacija izuzetka, postavljanje ograničenja funkcije u smislu postavljanja izuzetka, funkcija terminate, funkcija unexpected, funkcija abort, makro assert te korišćenje klase std::exception .

Poglavlje Uvod u standardnu biblioteku (C++ i STL) obuhvata neke često korišćene klase i funkcije: string, array, list, vector, pair, map, stack, queue, deque i sort.

U poglavlju Niti obrađeno je korišćenje klase std::thread u smislu kreiranja, izvršavanja i sinhronizacije lakih paralelnih procesa.

Poglavlje Lambda obrađuje lambda funkcije gde su objašnjeni: korišćenje vanjskih promenljivih, postavljanje povratnog tipa, argumenti lambde, telo lambde i primeri upotrebe.

Zahvaljujem se svima koji su na bilo koji način doprineli da ova knjiga bude pred Vama, a posebnu zahvalnost dugujem kolegama prof. dr Miroslavu Lutovcu, dipl. ing. i mr. Milošu Pejanoviću, dipl. ing, za korisne sugestije koje su uticale na pisanje ove knjige.

U ovom izdanju udžbenika mogući su propusti, tako da su dobrodošle sve dobronamerne sugestije u cilju poboljšanja sledećeg izdanja.

Knjiga je podržana projektom Ministarstva prosvete, nauke i tehnološkog razvoja Republike Srbije MGKOP – Modernizacija kurikuluma grupe predmeta Objektno programiranje 1 i Objektno programiranje 2 na studijskom programu Računarska tehnika.

U Beogradu, juna 2019. godine.

Autor

Sadržaj

1. UVODNO POGLAVLJE

U knjizi je obrađeno gradivo koje se sluša na drugoj godini osnovnih strukovnih studija smera Računarska tehnika u okviru predmeta Objektno programiranje 1 na Visokoj školi elektrotehnike i računarstva strukovnih studija u Beogradu. Tehnologija koja se koristi u knjizi je programski jezik C++.

C++ je hibridni jezik opšte namene koji obuhvata i tradicionalno programiranje i objektno orijentisano programiranje od sistemskih programa do aplikativnih programa. Tradicionalno programiranje je usmereno ka razmišljanju o rešavanju realnog problema u koracima na način kako se i kada nešto radi. O objektno orijentisanom programiranju trebalo bi pristupiti kao drugačijem načinu razmišljanja programera gde se realan problem posmatra po delovima gde je u prvom planu šta oni rade, kako su povezani i kako se njima pristupa i šta se vidi spolja, dok je u drugom planu kako to rade. OOP pristup kontroliše menjanje stanja ovih delova sistema čime je olakšano nalaženje grešaka. Dobro napisan kod za problem ili deo problema može se ponovo koristiti bez izmene ili uz minimalnu izmenu za drugi sličan problem ili deo problema, čime je postignuta ponovna upotreba napisanih delova softvera (*software reuse*).

Softver predstavlja model realnog problema pri čemu je cilj da ovaj model bude što bliži realnom problemu, a to se može postići objektno orijentisanim načinom razmišljanja.

Ideja je da se student usvajanjem objektno orijentisanog načina razmišljanja osposobi da se uključi u programerski tim koji kreira ogroman program od samog početka ili u programerski tim u kom bi radio na daljem razvoju (proširenju) programa koji već ima stotine hiljada linija programskog koda.

Preporuka autora je da student pri proučavanju predstavljenog materijala maksimalno programira u integrisanom razvojnom okruženju (preporučujem MS Visual Studio ili JetBrains CLion) kako bi stekao znanje i veštinu koja se traži na IT tržištu.

Čitalac se kroz programske primere postepeno uvodi u materiju da bi praktično ovladao izloženom materijom. Preporuka je da se dati primeri prouče i samostalno urade u razvojnom okruženju (preporučujem *Microsoft Visual Studio* ili *JetBrains CLion* razvojna okruženja) kako bi čitalac ovladao i znanjem i veštinom u korišćenju odabranog integrisanog razvojnog okruženja koja je neophodna za dobijanje posla na IT tržištu.

Knjiga je pisana od nivoa izlaganja pravila objektno orijentisanog razmišljanja do izrade objektno orijentisanih programa da bi u konačnom ishodu student bio u stanju da samostalno rešava zadatke na objektno orijentisan način koristeći C++ tehnologiju.

2. Programski jezik C++

Cilj poglavlja je da student ovlada: elementima koji nisu objektno orijentisani, tipovima podataka, oblastima važenja, životnom veku objekata, memorijskim oblastima i konverzijom tipova.

Programski jezik C++ stvorio je Bjarne Stroustrup 1983. godine dodavanjem novih elemenata u programski jezik C, pri čemu je stvorio hibridni programski jezik u kome se moglo programirati i tradicionalno i objektno (prvobitni naziv je bio "C sa klasama"). Ono što odlikuje C++ je brzina i lako održavanje programa, tako da je idealan za ogromne programe gde su performanse izvršavanja i održavanje programa prioritet (npr. operativni sistem Windows 10 ima oko 50 miliona linija koda). Jezik C++ je standardizovan od strane organizacija ANSI (American National Standards Institute) i IEC (International Electrotechnical Commission) kao što sledi: 1998. godine standard ISO/IEC 14882:1998, 2003. godine standard ISO/IEC 14882:2003, 2011. i 2014. godine standard ISO/IEC 14882:2011/2014 te u ovom trenutku standard ISO/IEC 14882:2017.

C++ je hibridni programski jezik srednjeg nivoa (*middle-level language*) u smislu podrške za programiranje na niskom nivou i visokom nivou koji podržava: proceduralno, objektno orijentisano i generičko programiranje.

Sledi popis nekih značajnih karakteristika programskog jezika C++:

- radi se o hibridnom programskom jeziku: može se programirati tradicionalno i objektno orijentisano za opšte namene;
- nije prenosiv u smislu da je izvršna verzija ciljno vezana za dati operativni sistem;
- lak je za upravljanje arhitekturom mašine na kojoj će se izvršavati;
- brzina izvršavanja i mogućnost održavanja programa su ogromne;
- podesan je za pisanje programa od komunikacije do operativnih sistema.

Referentna dokumentacija za standard programskog jezika C++ je data na sajtu https://isocpp.org/std/the-standard, dok sajt http://www.cplusplus.com/ daje opis najvažnijih biblioteka.

U poglavljima koja slede biće dati i koncepti jezika C++ koji nisu objektno orijentisani i nasleđeni su iz jezika C. Funkcija main je zadržana u C++ tako da izvršavanje programa počinje od ove funkcije. Ova funkcija ne pripada ni jednoj klasi, već je recidiv tradicionalnog programiranja. U tom smislu bi je trebalo i posmatrati samo kao početnu tačku izvršavanja programa.

2.1. Elementi koji nisu objektno orijentisani

U ovom poglavlju biće reči o elementima koji nisu objektno orijentisani, a značajni su za programiranje u C++. Poglavlje obrađuje makroprocesor i tokene.

2.1.1. Makroprocesor

Prevodilac vrši sintaksnu analizu čiji rezultat je tekst rastavljen na tokene koji predstavlja ulaz za makroprocesor.

Makroprocesor ili pretprocesor (*preprocessor*) transformiše svoj ulaz prema direktivama za pretprocesor koje su date u samom izvornom kodu, a onda svoj rezultat prosleđuje prevodiocu za sintaksnu analizu.

Makroprocesor se koristi kao što sledi:

- za makrozamene;
- za uključivanje fajlova;
- za uslovno prevođenje.

2.1.1.1. Makrozamene

Direktiva makroprocesoru počinje znakom # i može biti bilo gde u programu, pri čemu vredi sve do kraja programa ili do mesta ukidanja važenja direktive. Ako se navede prazna direktiva (samo znak #) ona neće uticati na makroprocesor. Direktiva za makrozamenu (macroexpansion) počinje sa #define, razmak, identifikator makroa (macro), otvorena mala zagrada ili razmak, a onda sledi navođenje zamene:

```
#define PI 3.14
#define RECENICA Malo u ovom redu, a malo \
                 u sledecem redu.
```

Makropocesor razlikuje velika i mala slova (case-sensitive). Paziti na sledeće (sve što se napiše kao zamena biće uzeto u obzir):

```
#define ZBIR a+b;
```

Menja tekst ZBIR u tekst a+b; (sa znakom tačka-zarez na kraju) tako da se može desiti sledeće:

```
d = c + ZBIR;  // c+a+b;;  ne bi smetalo (prihvatljivo)
d = ZBIR + c;  // a+b;+c;  NOK
```

Definicija makroa ukida se direktivom #undef i od tog mesta više ne postoji taj makro.

```
#define ZBIR a+b
c1 = ZBIR;
#undef ZBIR
```

```
c2 = ZBIR;
```

izlaz pretprocesiranja je:

```
c1 = a+b;
c2 = ZBIR;
```

Ako se želi parametarska makrozamena, onda odmah iza makro identifikatora mora da se navede otvorena mala zagrada, zatim se navodi lista formalnih argumenata makrodefinicije, zatvorena mala zagrada razmak i tekst zamene u kome se uračunavaju navedeni parametri:

```
#define ZBIR(a,b) (a+b)
```

Ako tekst makro zamene sadrži neki drugi makro identifikator isti će biti zamenjen zamenskim tekstom. Za ovo koristiti funkcije ugrađene u kod (*inline functions*).

Iako se može koristiti mogućnost da se definišu makro konstante, u C++ se koriste konstante deklarisane ključnom reči const:

```
#define MIN 2
const int MAX = 99;
```

Ako se želi da se argument prosleđen makrou pretprocesira kao istovetan tekst, onda se koristi operator # kao što sledi:

```
#define tekst(bilosta) #bilosta
#define Joe Dzo

tekst(Joe)  // izlaz Joe
tekst("Hm") // izlaz "Hm"
```

Ako se navede kao što sledi:

```
#define tekst2(bilosta) "bilosta"
```

bilo koji poziv makroa tekst2 daće kao izlaz `"bilosta"`.

Za konkatenaciju zamenskog teksta i formalnog argumenta se koristi operator ## kao što sledi:

```
#define C(i) C##i
C(1) = C(2) * C(3);   // izlaz C1 = C2 * C3;
```

2.1.1.2. Uključivanje fajlova

Uključivanje teksta drugih fajlova u dati fajl postiže se direktivama:

```
#include <ime_fajla>
```

ili

```
#include "ime_fajla"
```

pri čemu se prva sintaksa koristi ako je reč o fajlu iz standardnog direktorijuma, a druga se korsiti ako je reč o fajlu iz tekućeg direktorijuma, pri čemu, ako se ne nađe u tekućem direktorijumu, onda se traži u standardnom (obićno se navede u projektu lista direktorijuma). Ova direktiva se može ugnežđavati.

Da bi se neki fajl samo jednom uključio u program koristi se direktiva:

```
#pragma once
```

koja se navodi na početku tog fajla.

2.1.1.3. Uslovno prevođenje

Uslovne direktive omogućuju biranje konačnog teksta koji će biti uključen u prevođenje . Posmatra se direktiva #if #elif #else #endif :

```
#if if_izraz_1
    if_blok
#elif elif_izraz_2
    elif_blok
#else
    else_blok
#endif
```

Navedeni konstantni izrazi se računaju u vreme prevođenja. Ako je izraz koji sledi iza #if različit od 0 onda se za prevođenje uračunava if_blok tekst, inače se ide redom dok se ne nađe prvi elif_izraz (mogu bitiređani jedan iza drugog) i tada se za prevođenje uračunava taj elif_blok tekst, odnosno, ako se ne nađe niti jedan logički tačan izraz dolazi se do else blok teksta koji se tada uračunava za prevođenje.

Direktive #if defined ili #ifdef, odnosno, #if !defined ili #ifndef ispituju da li je identifikator koji sledi iza njih definisan, odnosno, nije definisan.

```
#define _DEBUG
#if defined _DEBUG          // ili #ifdef _DEBUG
    StampajNaTermickomStampacu();
#endif
```

Moguće je iskoristiti direktive #ifndef #define i #endif da se obezbedi da se neki fajl samo jednom uključi u program:

```
#ifndef _Tacka2D
#define _Tacka2D
  class Tacka2D
  {
      double x;
      double y;
    public:
```

```
            Tacka2D();
        void PostaviKoordinate(double xx, double yy);
        double UdaljenostDoIshodista();
        double UdaljenostDoTacke(Tacka2D drugatacka);
        void IspisiPodatke();
    };
  #endif
```

Prethodno znači sledeće: ako nije definisan makro _Tacka2D, onda će on biti definisan i zaglavlje klase Tacka2D biće uključeno u program. Ako se sledeći put proba uključenje ovog fajla, onda će makro _Tacka2D biti definisan i zaglavlje klase Tacka2D neće biti ponovo uključeno u program.

Za uslovno prevođenje koristi se i direktiva:

```
__has_include ( "imefajla" )
__has_include ( <imefajla> )
```

koja vraća 1 ako je fajla "imefajla" uključeno, odnosno, 0 ako nije.

Makro NODISCARD se odnosi na generalizovani atribut nodiscard (koristi se u obliku [[nodiscard]]) i specificira da povratnu vrednost ne bi trebalo odbaciti (trebalo bi je koristiti):

```
int [[nodiscard]] f()  // ili int NODISCARD f()
{
 /* ... */
}

f();  // warning (opomena) jer se ne koristi povratna int vrednost
      // deklarisane funkcije f
      // ako je uključen i generalizovani atribut
      // warn_unused_result

class NODISCARD Automobil { /* ... */ };

Automobil g();

g();  // warning: ignorisana je povratna vrednost funkcije g
```

Makro FALLTHROUGH se odnosi na generalizovani atribut fallthrough (trebalo bi uključiti i -Wimplicit-fallthrough warning) i koristi se kada se u switch naredbi na kraju datog ulaza ne nalazi break, a potrebno je da se to dozvoli nul-naredbom FALLTHROUGH (null-statement - sama naredba u jednom redu) koja se stavlja na kraj naredbe case.

```
enum class option { A, B, C };

void choice(option value) {
  switch (value) {
    case option::A:
      // ...
```

```
    case option::B: // warning: nenotiran prolaz ka ovom ulazu
      // ...
      [[fallthrough]];  // null-statement
    case option::C: // OK, notiran je prolaz iz prethodnog ulaza
      // ...
      break;
  }
}
```

Makro MAYBE_UNUSED odnosi se na generalizovani atribut maybe_unused i koristi se kada se želi izbeći da se daje opomena (warning) za promenljivu koja se potencijalno neće koristiti.

```
MAYBE_UNUSED int value = 42;

assert(value == 42);
```

Nekada se koristilo (void)value; da bi se postigao ovaj efekat.

2.1.1.4. Predefinisani makroi

Makroi koji su predefinisani se ne mogu ni redefinisati ni ukinuti. Sledi spisak predefinisanih makroa:

`__LINE__` predstavlja tekst tekuće linije programa;

`__FILE__` predstavlja ime fajla koji se prevodi;

`__DATE__` predstavlja datum prevođenja ("*mm dd gggg*");

`__TIME__` predstavlja vreme prevođenja ("*čč*:*mm*:*ss*");

`__cplusplus` je definisan ako se prevodi C++ program.

Direktiva:

```
#line intkonstanta "novo_ime_fajla"
```

- makrou `__LINE__` dodeljuje navedenu konstantu koja "govori" prevodiocu da je ta konstanta sledeći redni broj linije programa;
- makrou `__FILE__` dodeljuje novo_ime_fajla (inače bi bio podrazumevano jednak stvarnom imenu fajla)

Za poruku o grešci koristi se direktiva #error

```
#error tekst_poruke
```

Direktiva #pragma ima više značenja (pogledati dokumentaciju prevodioca), npr:

```
#pragma comment( lib, "opengl32.lib")
```

uključuje biblioteku opengl32.lib u projekat;

```
#pragma once
```

ukljucuje fajl samo jednom u program;

```
#pragma comment(linker,"/subsystem:windows /entry:mainCRTStartup")
```

ne prikazuje konzolni prozor već samo grafički prozor (freeglut).

2.1.2. Tokeni

Prevodilac vrši leksičku analizu rastavljajući tekst programa na elementarne gramatičke jedinice koje se nazivaju tokeni (token). C++ poseduje sledeće grupe tokena:

- ključne reči; (npr. if, while,...)
- separatore; (npr. beline,...)
- identifikatori; (npr. x, z, y,...)
- operatori; (npr. +,-,...)
- literali. (npr. 2.71, 3.14,"Hm",...)

2.1.2.1. Ključne reči

Ključne reči jezika C++ su rezervisane i ne mogu se koristiti za bilo šta drugo osim za ono što su namenjene (zauzete se u smislu korišćenja za imena identifikatora). U tabeli 2.11. dat je spisak ključnih reči programskog jezika C++.

Tabela 2.1.1. Spisak ključnih reči C++ jezika

alignas (C++11)	delete C++11	register (C++17)
alignof (C++11)	do	reinterpret_cast
and	double	requires (C++20)
and_eq	dynamic_cast	return
asm	else	short
atomic_cancel	enum	signed
atomic_commit	explicit	sizeof C++11
atomic_noexcept	export (C++11)(C++20)	static
audit (C++20)	extern (C++11)	static_assert (C++11)
auto (C++11)	false	static_cast
axiom (C++20)	final (C++11)	structC++11
bitand	float	switch
bitor	for	synchronized
bool	friend	template
break	goto	this
case	if	thread_local (C++11)
catch	inline (C++11)	throw
char	int	true
char8_t (C++20)	import (C++20)	try
char16_t (C++11)	long	typedef
char32_t (C++11)	module (C++20)	typeid
class (C++11)	mutable (C++11)	typename
compl	namespace	union
concept (C++20)	new	unsigned
const	noexcept (C++11)	usingC++11
consteval (C++20)	not	virtual

constexpr (C++11) const_cast continue co_await (C++20) co_return (C++20) co_yield (C++20) decltype (C++11) default(C++11)	not_eq nullptr (C++11) operator or or_eq override (C++11) private protected public reflexpr	void volatile wchar_t while xor xor_eq

U zagradama, u tebeli 2.1.1, je stavljeno kod kog C++ standarda je došlo do uvođenja ili menjanja ključne reči.

2.1.2.2. Separatori

Separatore (beline - *white space*) čine: komentari, blanko znaci, tabulatori (horizontalni, vertikalni), znaci prelaska u novi red i znaci prelaska na novu stranicu. Separatori se u smislu razdvajanja odnose na tokene.

Prevodilac pokušava da nađe najduži niz znakova koji čine token (npr. x - - - y predstavlja x-- jer postoji kao token, oduzimanje i umanjilac je y).

2.1.2.3. Identifikatori

Identifikator je niz slova i cifara, pri čemu bi prvi znak trebalo da bude slovo engleske abecede (obratiti pažnju da je C++ case-sensitive). Obično su identifikatori koji počinju sa donjom crtom ili dve donje crte predefinisani ili se posebno koriste. C++ razlikuje velika i mala slova.

Napomena: Dve donje crte ne koristiti za početak identifikatora.

U kodovima koji su navedeni u ovom materijalu dato je par stilova pisanja identifikatora.

2.1.2.4. Operatori

Operatori u C++ su grupisani kao što sledi:

- aritmetički operatori:

`+ - * /`	sabiranje, oduzimanje, množenje, deljenje
`+ -`	unarni operatori znaka broja
`%`	ostatak pri celobrojnom deljenju
`+= -= *= /=`	operacija sabiranja, oduzimanja, množenja i deljenja uz dodelu, npr. `a+=5;` je isto što i `a=a+5;`

`++ --`	autoinkrement, autodekrement, npr. postfiskni: aa++; aa se koristi u izrazu a onda uveća za 1; aa--; aa se koristi u izrazu a onda umanji za 1; prefiksni (su *lvalue*, mogu biti sa leve strane =): ++aa; uveća aa za 1 a onda ga koristi u izrazu; --aa; umanji aa za 1 a onda ga koristi u izrazu;

- relacioni operatori:

`==`	jednako
`<`	manje
`>`	veće
`!=`	različito
`>=`	veće ili jednako
`<=`	manje ili jednako

- logički operatori (&&, ||, !);

`&&`	logičko i
`||`	logičko ili
`!`	logička negacija

- bit-operatori (*bitwise* operatori) koji operišu na nivou bita:

`&`	operator i
`|`	ili
`~`	komplement
`^`	ekskluzivno ili
`>>`	pomeranje u desno
`<<`	pomeranje u levo
`&= |= ~= ^=` `>>= <<=`	kao prethodno samo uz dodelu, npr. `a&=5;` je isto što i `a=a&5;`

- operator dodele (=), grupiše zdesna ulevo, vrši dodelu i vraća vrednost;

a=b; // a dobija vrednost b

- trojni operator uslovne dodele (`?:`). npr.
  `c = (a<b)? true: false;`
  ispituje se da li je `a<b`, ako jeste `c` postaje true inače postaje false.
  `d = (aa<ba)? aa: bb;`
  ispituje se da li je `aa<bb`, ako jeste `d` dobija (dodeljuje se) vrednost aa inače dobija vrednost bb.
- operatori razrešavanja imena: tačka (`.`), minus-veće (`->`) i dve dvotačke (`::`)
- operatori uzimanja adrese &
- operator dereferenciranja *
- operator sekvence ,

2.1.2.5. Literali

Literali su izrazi koji se koriste kao što su napisani:

- kao celi brojevi — npr. 100
- kao realni brojevi — npr. 100.0, 100.0f
- kao karakteri — npr. 'A'
- kao nizovi karaktera — npr. "Niz"
- kao logičke vrednosti — npr. true

2.1.3. Deklaracije

Deklaracija predstavlja uvođenje novog identifikatora datog tipa i ponašanja u program.

Deklaracija je definicija ako je reč o kreiranju objekta ili je dato telo metode ili funkcije.

Postoji pravilo jedinstvene definicije objekta, funkcije, klase i nabrajanja. Ovo ne treba posmatrati prema imenu, jer se u različitim područjima važenja može pojaviti isto ime identifikatora, a koji se odnosi na različite npr. objekte.

Prema prethodnom, ako u programu ponovite isti kod i prevodilac ne dojavljuje grešku, reč je o deklaracijama koje nisu definicije.

```
class C;
class C;
```

Prethodno znači da je C ime klase i ništa više.

Ako se ne sme ponoviti kod deklaracije, onda ta deklaracija predstavlja definiciju:

```
double pi = 2.99;
```

2.1.4. Organizacija koda

Postupak prevođenja koda za C++ je takav da se programski fajlovi prevedu u objektni kod (*object code*) nakon čega se vrši linkovanje (povezivanje - *linking*) u izvršni kod (*executable code*). Prevode se svi fajlovi projekta, ali jedan po jedan.

Vredi pravilo jedinstvene definicije: jedna definicija, a deklaracije se stavljaju svuda gde se koristi definisano ime.

U objektnom kodu su informacije o identifikatorima koji su u tom kodu definisani i o identifikatorima koji su potrebni u tom kodu, ali su definisani na drugom mestu.

Linker (povezivač - *linker*) obrađuje sve prikupljene informacije o korišćenju imena, a onda povezuje fajlove u jedinstven izvršni program.

Organizacija fajlova je takva da se zaglavlje klase postavlja u pripadni .h fajl, dok se implementacija postavlja u pripadni .cpp fajl.

Korišćenjem direktive za uključenje fajlova, uključuju se fajlovi zaglavlja (.h) tamo gde je potrebno u drugim fajlovima (kao i u pripadnom .cpp fajlu). Na ovaj način postoji jedan fajl zaglavlja gde će se raditi potrebne promene, naravno uz pripadnu implementaciju u odgovarajućem .cpp fajlu.

2.1.5. Kontrola toka programa

Kontrole toka su nasleđene iz jezika C i realizovane su sledećim naredbama (instrukcijama) koje predstavljaju iskaze koji proizvode dejstvo, ali ne vraćaju vrednost:

- uslovne naredbe,
  ako je uslov `uslov1` zadovoljen izvršava se blok naredbi `naredba/e` (blok je ograničen parom velikih zagrada koji predstavlja grupisanje naredbi u složenu naredbu (*compound statement*)):

  ```
  if (uslov1) {naredba/e}
  ```

  ako je uslov `uslov2` zadovoljen izvršava je blok naredbi `naredbe2` inače se izvršava blok naredbi `naredbe3`:

  ```
  if(uslov2){naredbe2}
  else      {naredbe3}
  ```

  ako je vrednost izraza `izraz1` u `switch`-u jednaka nekoj vrednosti iza ključne reči `case` izvršiće se blok naredbi koji sledi dati `case`, a u slučaju da niti jedna `case` vrednost ne odgovara izrazu `izraz1` u `switch`-u, izvršiće se blok naredbi iza ključne reči `default` (`naredbe6`):

  ```
  switch (izraz1) {
  ```

```
        case vrednost1: naredbe4; break;
        case vrednost2: naredbe5; break;
        default: naredbe6;
    }
```

- iterativne,
  programska petlja pre koje se izvrši inicijalizacija `inic` i koja izvršava blok naredbi `naredbe8` sve dok je zadovoljen uslov `uslov3`, pri čemu se na kraju svake iteracije izvrši `naredbe7`, a onda ponovo ispituje `uslov3` za novu iteraciju (ako se ne navede, podrazumevano je `uslov3` zadovoljen):

  ```
  for(inic; uslov3; naredbe7){naredbe8}
  ```

  programska petlja koja izvršava blok naredbi `naredbe9` sve dok je zadovoljen uslov `uslov4` što znači da je `uslov4` različit od `0`:

  ```
  while(uslov4) {naredbe9}
  ```

  programska petlja koja jednom izvršava blok naredbi `naredbe10`, a onda proverava da li je zadovoljen uslov `uslov5` (znači da je `uslov5` različit od `0`)za novu iteraciju:

  ```
  do {naredbe10} while(uslov5);
  ```

- naredbe skoka,

  ako se želi izlazak iz prvog okružujućeg bloka (izlazak iz petlje):

  ```
  break;
  ```

  ako se želi skok na kraj tekuće iteracije:

  ```
  continue;
  ```

Za lakšu čitljivost programa koriste se komentari kao što sledi:

```
// jednolinijski komentar

/* višelinijski komentar, početak komentara u ovom redu
nastavak komentara
završetak komentara u ovom redu do znaka */
```

2.2. Tipovi podataka

U programskom jeziku C++ koriste se ugrađeni tipovi podataka (*built-in types*) i korisnički tipovi podataka (*user-defined types*). Ugrađeni tipovi podataka nemaju metode i mogu se koristiti za kreiranje složenih struktura podataka unutar korisničkih tipova podataka kao elementi tradicionalnog programiranja. Ugrađeni tipovi podataka obuhvataju osnovne tipove (*fundamental types*) i jedan broj izvedenih tipova podataka (*derived types*): nizovi, funkcije, reference, pokazivači, pokazivači na članove klasa, konstante. Korisnički definisani tipovi

su: klase, strukture, unije, enumeracije i tipovi definisani specifikatorom typedef .

2.2.1. Osnovni tipovi

Osnovni tipovi podataka se koriste kao elementi tradicionalnog programiranja. Ovakvo programiranje se u opštem slučaju koristi kao sastavni deo realizacije metoda klase. Osnovni tipovi dati su u tabeli 2.2.1. sa podacima o veličini tipa u bajtovima i značenju tipa na 64-bitnoj mašini. Ovi tipovi su model za predstavljanje elementarnih podataka na računaru.

Prefiks signed tretira bit najveće težine datog objekta kao znak celog broja koji se predstavlja tim objektom (pozitivan celi broj, nula ili negativan broj), dok prefiks unsigned znači da se radi o pozitivnoj (N_0) predstavi celog broja. Prefiksi za tip int: short , long i long long određuju koliki se memorijski prostor rezerviše za smeštanje takvog tipa int. Prefiks long za tip double određuje 80-bitnu preciznost broja u pokretnom zarezu.

Tabela 2.2.1. Osnovni tipovi podataka

Primitivni tip	Veličina u bajtovima za model podataka LLP64/LP64	Značenje
bool	1/1	logička vrednost, true (različito od 0), false (ako je 0)
char signed char	1/1	znak opseg od -128 do 127 (za x86 i x64) zavisi od prevodioca
unsigned char	1/1	znak opseg od 0 do 255
wchar_t	2/2	široki znak, na Windowsu je 16-bitni
short short int signed short signed short int	2/2	ceo broj opseg od -2^{15} do $2^{15}-1$
unsigned short unsigned short int	2/2	ceo broj opseg od 0 do $2^{16}-1$
int signed signed int	4/4	ceo broj opseg od-2^{31} do $2^{31}-1$

unsigned unsigned int	4/4	ceo broj opseg od 0 do 2^{32}-1
long long int signed long signed long int	4/8	ceo broj opseg od -2^{31} do 2^{31}-1 / opseg od -2^{63} do 2^{63}-1
unsigned long unsigned long int	4/8	ceo broj opseg od 0 do 2^{32}-1 / opseg od 0 do 2^{64}-1
long long long long int signed long long signed long long int	8/8	ceo broj opseg od -2^{63} do 2^{63}-1
unsigned long long unsigned long long int	8/8	ceo broj opseg od 0 do 2^{64}-1
float	4	realan broj (standard IEEE-754) opseg od ±1.18e-038 do ±3,4e+038 tačnost oko 7 cifara
double	8	realan broj dvostruke preciznosti (standard IEEE-754) opseg od ±2,23e-308 do ±1,8e+308 tačnost oko 15 cifara
long double	10 ili 16	realan broj proširene preciznosti (standard IEEE-754) za 10-bajtni, opseg od $\pm 3.65\times10^{-4951}$ do $\pm 1.18\times10^{4932}$ tačnost oko 19 cifara
void	-	bez vrednosti

Model podataka na mašini LLP64 je Win64 API format, dok je LP64 model podataka na mašini Linux ili Mac OS X.

Tip bool koristi se za pohranjivanje logičke vrednosti (true, false). Vrednost nula je false.

```
bool bIstina = true;
```

Za pohranjivanje znaka iz skupa znakova na datoj mašini koristi se tip char. Ovaj tip zauzima 1 bajt. Da li će biti označen ili neoznačen zavisi od prevodioca. Znakovi se predstavljaju uokvireni apostrofima npr:

```
char znakA = 'A';
```

```
char znakB;//Linux dodeljuje ''; Windows ostavlja neodređenu vrednost
```

Vrednost ovakvog objekta odgovara kodu datog znaka prema kodovanju na toj mašini. Za ASCII (*American Standard Code For Information Interchange*) kodovanje neke vrednosti su kao što sledi:

```
char c1='0', // vrednost 48
     c2='A', // vrednost 65
     c3=' '; // vrednost 32
char c='\xff';              // vrednost -1
signed char sc='\xff';      // vrednost -1
unsigned char uc='\xff';    // vrednost 255
```

Objekti tipa char koriste se gde i objekti tipa int jer postoji standardna konverzija char u int, npr. sledi kod koji redom ispisuje znakove za cifre:

```
char c1='0';
char c2;
for (int i=0; i<=10; i++)
{
  c2=c1+i;        // kodovi cifara su redom postavljeni
  cout<<c2<<endl; // u ASCII tabelu
}
```

Tip wchar_t se koristi za predstavljanje znakova iz skupa svih karakter setova i zauzima 2 bajta ili 4 bajta.

```
#include <fcntl.h>
#include <io.h>
#include <iostream>
int main()
{
      _setmode(_fileno(stdout), _O_U8TEXT); // UTF-8
      wchar_t wch = L'Ш';
      const wchar_t m[] = L"трбац\n";
      std::wcout << wch << m ;
      return 0;
}
```

Slika 2.2.1. Primer ispisa ćirilicom

Na slici 2.2.1. dat je jedan način ispisa ćiriličnog teksta. U zaglavlju fcntl.h je simbolička konstanta _O_U8TEXT za UTF-8, a u io.h je definisana funkcija _setmode tipa int _cdecl (int, int) koja kao parametre prihvata broj fajl deskriptor i mod ispisa. Funkcija _fileno je tipa int (FILE*) i za dati parametar vraća njegov fajl deskriptor. Objekat wcout ispisuje wchar_t znakove. Prefiks L je oznaka za wide character/string literal.

Celobrojne vrednosti su predstavljene tipom int sa modifikatorima: unsigned, signed, short, long i long long. U tabeli 2.2.1. dati su podaci o veličini i opsegu celih brojeva koje mogu da obuhvate.

Obratite pažnju na sledeće:

```
unsigned short ui = 0;
int i = -1;
ui = ui + i;                // ui = 65535
```

- na desnoj strani poslednje linije koda vrši se svođenje tipa unsigned short i tipa int na zajednički tip int kako bi se obavila operacija sabiranja;
- rezultat ove operacije sabiranja je broj -1 čija je binarna predstava sve jedinice 11111111 11111111 11111111 11111111;
- binarna predstava prvih 16 bita sa desne strane se smešta u 16 bita ui, tako da se dobija 11111111 11111111, a ovo predstavlja vrednost 65535 za zapis tipa unsigned short.

Pri deklarisanju je dozvoljeno izostaviti reč int iza short, long i long long. Primer:

```
short siVrednost1;              // tip je short int
short int siVrednost2;          // tip je short int
unsigned short usiVrednost3;    // tip je unsigned short int
unsigned short int usiVrednost4;// tip je unsigned short int
```

Daje se primer prefiksne oznake literala u oktalnom, heksadecimalnom i binarnom zapisu za npr. decimalnu vrednost 42:

```
int oktalni         = 052;      //prefiks je 0
int heksadecimalni  = 0x2a;     //prefiks je 0x
int heksadecimalni2 = 0X2A;     //prefiks je 0X
int binarni         = 0b101010; //prefiks je 0b
```

Sufiksne oznake int literala su kao što sledi:

- za `unsigned int` je `u`
  ```
  unsigned int uiC = 100u;
  ```
  za long je `L` ili `l`
  ```
  long liVr1 = 100L;
  long liVr2 = 100l;
  ```
- za unsigned long je `UL` ili `Ul` ili `uL` ili `ul`
  ```
  unsigned long ulVr1 = 100UL; // 100Ul, 100uL, 100ul
  long double ld = 1.2L;
  ```
- za long long je LL ili `ll` kome se dodaje na početak `U` ili `u` ako je tipa `unsigned`
  ```
  unsigned long long l1 = 18446744073709550592ull;
  unsigned long long l2 = 18'446'744'073'709'550'592llu;
  unsigned long long l3 = 1844'6744'0737'0955'0592uLL;
  unsigned long long l4 = 184467'440737'0'95505'92LLU;
  ```

Integralni (*integral types*) tipovi su: bool, char, signed char, unsigned char, char8_t, char16_t, char32_t, wchar_t, short, int, long, long long, unsigned short, unsigned int, unsigned long, unsigned long long.

Za predstavljanje realnih brojeva koriste se tipovi float, double i double float koji zauzimaju respektivno 4, 8, 10 bajtova. Za predstavljanje realnih brojeva u pokretnom zarezu koristi se format IEEE-754 kao što sledi:

- za float, $(-1)^{s} 2^{K-127} (1.m)$, (po bitovima 1-8-23)
- za double, $(-1)^{s} 2^{K-1023} (1.m)$, (po bitovima, 1-11-52)
- za long double $(-1)^{s} 2^{K-16383} (1.m)$, (po bitovima, 1-15-64)

Vrednosti za opseg i tačnost navedenih tipova dati su u standardnom zaglavlju <cfloat>. U udžbeniku će kasnije, u okviru kastovanja, biti dat primer za ispis float formata.

Sufiksna oznaka float literala je f:

```
float pi = 3.14f;
```

Sufiksna oznaka long double literala je L:

```
long double pi = 3.14159265358979323846264338327950288419716939 9L;
```

Aritmetički tipovi (*arithmetic types*) su celobrojni i racionalni tipovi

Tip void znači da nema vrednosti i koristi se za funkcije i metode koje ne vraćaju vrednost (napomena: konstruktor, destruktor i konverzija nemaju povratni tip). Ne postoji objekat tipa void, već se koristi kao izvedeni tip, kao pokazivač na void*, čime pokazuje na sirovu memoriju (na bilo koji objekat).

U nekim sistemima, deklaracijom (uvođenje imena u program) promenljive koja je i definicija (dodeljuje se i memorijski prostor za datu promenljivu) bez navođenja inicijalne vrednosti postavlja se da inicijalna vrednost bude "nula" za dati tip ili se ostavlja da to bude vrednost koja se "zatekne" u memoriji koja je dodeljena promenljivoj.

Preporuka je da se inicijalizacija uradi eksplicitno, a ne da se oslanjate na podrazumevane vrednosti.

Uvođenje imena promenljivih je kao što sledi:

```
int iVrednost;              // deklaracija jedne promenljive
int iVrednost1, iVrednost2;// deklaracija vise promenljivih
float fVrednost = 3.14f;    // postavljanje pocetne vrednosti
int iBroj(100);             // postavljanje pocetne vrednosti
```

2.2.2. Izvedeni tipovi

U izvedene tipove spadaju: nizovi, funkcije, pokazivači, reference i konstante.

2.2.2.1. Nizovi

Nizovi predstavljaju izvedeni tip. Niz elemenata nekog tipa predstavlja vektor koji sadrži elemente tog tipa. Oznaka je kao što sledi:

```
T identifikatorniza[kci]
```

što se čita kao niz `identifikatorniza` koji sadrži `kci` elemenata tipa T. Izraz `kci` predstavlja konstantni celobrojni izraz veći od nule koji se računa u vreme prevođenja programa. Indeks elemenata niza počinje od nule i ide do broja koji je za 1 manji od broja elemenata. Prevodilac ima informacije o početnoj adresi niza, veličini niza i tipu elemenata niza.

Elementi niza mogu da budu:

- objekti nekog od osnovnih tipova (osim tipa void);
- pokazivači;
- pokazivači na članove klase;
- objekti korisnički definisanih tipova (struktura, unija, klasa, nabrajanja);
- drugi nizovi.

Nizovi nisu promenljive lvrednosti (nešto što je sa leve strane znaka jednakosti), tako da ne može da se piše dodela niza=nizb već se mora vršiti kopiranje element po element .

Deklaracija tipa:

```
int mojniz[10];
```

je i definicija jer prevodilac izdvaja potreban memorijski prostor za smeštanje 10 elemenata tipa int za ovaj niz. Indeksi elemenata su redom od 0 do 9.

NAPOMENA: U ovom udžbeniku koristi se da se element niza imenuje prema pripadnom indeksu (da ne bi bilo zabune), dakle, nulti element, prvi element, ...

Sledi primer pristupanja datom elementu:

```
mojniz[0]=300; //postavljanje vrednosti za nulti element
mojniz[1]=mojniz[0]+20; //prvi element ima vrednost za 20
                        //veću od nultog elementa
```

Višedimenzioni nizovi predstavljaju nizove čiji su elementi nizovi (nizovi nizova).

```
int mojniz[4][3]; // deklaracija i definicija matrice 4x3
```

Bilo bi dobro ovo čitati kao niz od 4 elementa tipa niz od 3 elementa tipa int. Pristupanje kod ovakvih nizova je kao što sledi:

```
mojniz[1][2]=10;
```

ovde se pristupa prvom elementu niza, koji sadrži niz od tri elementa, a onda se uzima drugi element ovog niza i dodeljuje mu se vrednost 10 (napomena, dogovor je da element imenujemo prema pripadnom indeksu).

Višedimenzioni niz se smešta u memoriju kao vektor (jednodimenzioni niz) tako da se najbrže menja desni indeks višedimenzionog niza. Dozvoljeno je izostavljanje prve dimenzije višedimenzionog niza ako se radi o deklaracija niza koja nije definicija. Posmatra se primer prenosa niza kao argumenta funkcije:

```
void f(int arr[][3][4]) {
  //...
  arr[1][1][2]+=10; //dodavanje vrednosti 10 članu 1,1,2
}
void main() {
  int a[2][3][4];       // 3D niz veličine 2x3x4
  //...
  f(a);
}
```

Stvarni argument `a` pri pozivu funkcije `f` predstavlja adresu početka niza ***a*** i ona se kopira u formalni argument `arr` funkcije `f`. Niz *a* ima elemente koji su tipa `int[3][4]`. Adresa člana niza `arr[1,1,2]` u funkciji `f` se računa kao što sledi:

- `arr[0]` počinje na adresi `arr + (0 * veličina int[3][4])`
- `arr[1]` počinje na adresi `arr + (1 * veličina int[3][4])`
- `arr[1][1]` počinje na adresi koja se dobija tako što se na adresu `arr[1]` doda `1 * veličina int[4]`
- `arr[1][1][2]` počinje na adresi koja se dobije kada se na adresu `arr[1][1]` doda `2 * veličina int.`

2.2.2.2. Konstante

Konstantni tip je tip izveden iz nekog drugog tipa. Ovaj tip se označava stavljanjem reči const ispred specifikatora tipa u deklaraciji objekta. Prevodilac ne dozvoljava promenu vrednosti konstantnog objekta. Konstantni objekt datog tipa zadržava sve ostale osobine tog tipa.

```
const double pi = 3.14;
// nije dozvoljeno sada napisati npr. pi=2.71;
```

Celobrojni konstantni objekat koji je inicijalizovan nekim konstantim izrazom može se upotrebiti u konstantom izrazu. Prevodilac obično direktno ugrađuje vrednost konstantnog objekta u sam kod.

```
const int n=10;
int arr[n*n];
```

Element konstantog niza je konstantan. Nestatički podatak član konstantog objekta neke klase je konstantan (jer pripada objektu, a ne klasi).
String literalima se definišu konstante tipa niz char elemenata. Radi se o zero stringu koji se završava znakom '\0' (nula).
Primer:

```
char *pc="Hello!"; char ch[7]="Hello!";
```

Za smeštanje niza "Hello!" potrebno je 7 elemenata tipa char jer je na kraju i nula. Ne treba pokušavati izmenu string literala.

Postoje 4 vrste literala:

- Znakovne konstante (*character constants*), npr. 'A'
- Celobrojne konstante (*integer constants*),
    - decimalni broj (baza je 10, cifre 0,1,2,3,4,5,6,7,8,9), npr. 20
    - binarni broj (baza je 2, cifre 0,1), počinje sa 0b npr. 0b10100
    - oktalni broj, (baza je 8, cifre 0,1,2,3,4,5,6,7) počinje nulom, npr. $20_{10} = 024_8$
    - heksadecimalni broj (baza 16, znakovi 0..9, A...F ili 0..9, a..f), počinje sa 0x ili 0X npr. $20_{10} = 0x14_{16}$
- Racionalne konstante (*floating constants*) npr. 3.14, .2, 2., 2E5, 2e-2
- String-literali (*string literals*), npr. "Porsche"

String literali mogu sadržavati i specijalne znake kojima prethodi znak \. I tip char može sadržati specijalni znak. U tabeli 2.2.2. dat je spisak specijalnih znakova.

Tabela 2.2.2. Specijalni znaci

Opis znaka	Znak	Predstava	Primer
Novi red (*new line*)	NL(LF)	\n	'\n'
Horizontalni tabulator (*horizontal tab*)	HT	\t	'\t'
Vertikalni tabulator (*vertical tab*)	VT	\v	'\v'
Backspace	BS	\b	'\b'
Carriage return	CR	\r	'\r'
Nova strana (*form feed*)	FF	\f	'\f'
Zvono (*alert*)	BEL	\a	'\a'
Obrnuta kosa crta (*backslash*)	\	\\	'\\'
Znak pitanja (*question mark*)	?	\?	'\?'
Apostrof (*single quote*)	'	\'	'\''
Znaci navoda (*double quotes*)	"	\"	'\"'
Oktalni broj		*\ooo*	'\137'
Heksadecimalni broj		*\xhhh*	'\x2FC'

2.2.2.3. Funkcije

Funkcije su izvedeni tip koji pokazuje koji je povratni tip te koliko i koje tipove formalnih argumenata prihvata pri svom pozivu. Pri pozivu funkcije navodi se lista stvarnih argumenata kojima se inicijalizuju formalni argumenti funkcije uz potrebne standardne i korisničke konverzije. Funkcija je potprogram koji se poziva iz programa i koji vraća rezultat. Formalni argumenti su lokalni automatski elementi. Funkcija može imati podrazumevane vrednosti argumenata. Za poziv funkcije koristi se operator (). Povratna vrednost funkcije se navodi iza naredbe return (ako nije navedena vraća se tip void). Ovom vrednošću inicijalizovaće se privremeni objekat koji je na mestu poziva funkcije.

Ako je formalni argument funkcije referenca ili pokazivač onda se preko ovakvog formalnog argumenta može uticati na original (stvarni argument).

Izvršavanje programa počinje pozivom funkcije main koja je najčešće tipa int (void) ili int (int, char**) (neki kompajleri će dozvoliti i tip void (void)), pri čemu se povratna vrednost funkcije main vraća operativnom sistemu (u Windowsu možete videti u %ERRORLEVEL%, a u Linuxu sa echo $?). Povratna vrednost nula predstavlja regularan završetak programa. Kada se pozove izvršni C++ program, onda operativni sistem poziva funkciju main i prosledi joj parametre koji su dati u komandnoj liniji. Za ovakav poziv funkcija main ima sintaksu kao što sledi:

```
int main (int argc, char* argv[])  // ili int argc, char** argv
{
  // kod
  // na kraju ide naredba return nekiint; za izlazak iz programa
}
```

gde je *argc* broj argumenata, a *argv* je niz pokazivača na znakove.

Niz argv sadrži:

- na indeksu 0 ime programa (ili "");
- na svim indeksima osim poslednjeg su reči navedene iza imena programa (parametri);
- na indeksu argc je 0.

Primer poziva main funkcije:

```
  ispisi Elvis Arron Presley R&R
```

biće kao što sledi:

```
  argv[0] = "ispisi"
  argv[1] = "Elvis"
  argv[2] = "Arron"
  argv[3] = "Presley"
```

```
argv[4] = "R&R"
argv[5] = 0
```

Napomena: U programu nije dozvoljeno pozivanje funkcije main() niti uzimanje njene adrese.

Za izlazak iz programa koriste se:

- funkcija exit (zaglavlje <cstdlib>):

  ```
  void exit(int status);  //EXIT_SUCCESS 0, EXIT_FAILURE 1
  ```

  koja vraća operativnom sistemu vrednost status.

- funkcija atexit (zaglavlje <cstdlib>):

  ```
  int atexit ( void (*function) (void) );
  ```

  koja postavlja koja će se funkcija tipa void (void) pozvati pre izlaska iz programa (pokazivači na funkcije slede kasnije);

- funkcija abort (zaglavlje <cstdlib>):,

  ```
  void abort();
  ```

  koja odmah (nasilno) završava program.

Funkcija koja može da prihvati neodređen broj argumenata ima na kraju liste formalnih argumenata tri tačke koje mogu (ne moraju) biti odvojene zarezom. Primer funkcije koje prihvata dva obavezna argumenta tipa int, a onda i proizvoljan broj argumenata proizvoljnog tipa:

```
void f1(int,int,...);
void f2(int,int...);
```

Proizvoljan broj argumenata znači da ih može biti i nula. Pozivi navedenih funkcija su navođenjem dva stvarna argumenta, a onda proizvoljan broj argumenata proizovljnog tipa:

```
f1(1,2);
f1(1,2, 10%3);
f1(1,2, 3,4,'a');
f1(1,2, 3.14,"Zdravo!",2.71);
```

Za pristupanje proizvoljnim argumentima funkcije koriste se deklaracije iz zaglavlja <cstdarg> kao što sledi:

- `va_list` je tip liste neimenovanih argumenata;
- `void va_start(va_list valist, posledjni_arg);`
  makro va_start se poziva na početku da bi se inicijalizovao valist da gleda na prvi neimenovani argument iza poslednjeg obaveznog argumenta u listi formalnih argumenata funkcije;
- `tip va_arg(va_list valist, tip);`

makro va_arg se poziva posle poziva va_start i ovaj makro vraća vrednost tipa *tip* na koga gleda va_list, nakon čega se va_list postavlja da gleda na sledeći neimenovani argument;

- `void va_end(va_list valist);`
  makro va_end se poziva pre povratka iz funkcije i nakon pristupanja neimenovanim argumentima.

Na slici 2.2.2. dat je primer funkcije sa neodređenim brojem argumenata.

```
#include <cstdarg>
double zbir (char *format ...){
  va_list valist;  // pokazuje na tekuci neimenovani argument
  va_start(valist,format); //valist gleda na prvi neim. arg.
  double suma=0.0;
  int i;
  double d;
  for (char *p=format; *p; p++)    // prolaz kroz niz znakova
    switch (*p) {                  // u zavisnosti od tipa
      case 'i':
      case 'I':
      i = va_arg(valist,int);       // neim. arg. tipa int
      suma+=i; break;
      case 'd':
      case 'D':
      d = va_arg(valist,double);    // neim. arg. tipa double
      suma+=d; break;
    }
  va_end(valist);
  return suma;
}
#include <iostream>
int main(){
  double d =zbir("idid",int(4),double(5.6),int(-5),double(6));
  std::cout<<"d="<<d<<std::endl;
  system("pause");
  return 0;
}
```

Slika 2.2.2. Računanje dužine niza znakova

Funkcija može da ima podrazumevane (default) argumente kojima se u deklaraciji funkcije dodeljuju podrazumevane vrednosti.
Primer:

```
void f (int,int=5,double=10.14);
```

Funkcija f u prethodnom primeru može biti pozvana na sledeće načine:

```
f(1);
f(1,2);
f(1,2,3.14);
```

Prethodni pozivi biće prevedeni u sledeće pozive:

```
f(1,5,10.14);
f(1,2,10.14);
f(1,2, 3.14);
```

Stvarni argumenti koji inicijalizuju obavezne argumente se navode pri pozivu funkcije, dok se stvarni argumenti koji se odnose na podrazumevane argumente ne navode ili navode u redosledu gledano sa leva na desno, pri čemu, ako se navede jedan podrazumevani argument, onda se moraju navesti i svi podrazumevani argumenti koji slede iza njega. Ovo znači da od prvog podrazumevanog formalnog elementa u funkciji i svi ostali koji ga slede moraju biti podrazumevani gledano do na sukcesivne deklaracije. Nije dozvoljena redefinicija podrazumevanog argumenta.

Primeri:

```
void f1(int,int=2,int); // NOK, ne bi prosao poziv tipa f1(1,,3)

void f(int, int  , int  ); //OK
void f(int, int  , int=3); //OK, poslednji argument;
void f(int, int=2, int  ); //OK, sada je int,int=2,int=3

void g(int, int  , int);
void g(int, int=2, int); // NOK: desno je je obavezni argument

void h(int, int  , int=3);
void h(int, int=2, int=3);//NOK: redefinicija podrazumevanog
                          //argumenta.
```

Ako je podrazumevanom argumentu funkcije dodeljen izraz, onda se pri pozivu te funkcije vrši izračunavanje vrednosti izraza. Tipovi i imena kao i oblast važenja moraju biti zadovoljeni u takvom izrazu.

Primer:

```
int hm = 1;
int f(int);
int g(int = f(hm));
void h()
{
  int hm = 2;
  g(); //OK, podrazumevani argument je f(::hm)
}
```

Nije dozvoljeno kao što sledi:

- da se lokalna promenljiva koristi u izrazu čija se vrednost dodeljuje podrazumevanom argumentu:

```
void f()
{
  int iloc;
  extern void g(int x = iloc);
  ...
}
```

- da se formalni argumenti koriste u izrazu čija vrednost se dodeljuje podrazumevanom argumentu.
  ```
  void f(int a, int b = a);
  ```
- da se kod preklapanja operatora stavi podrazumevana vrednost formalnog argumenta (poglavlje o preklapanju operatora sledi kasnije).

Napomene:

- Podrazumevana vrednost formalnog argumenta ne utiče na tip funkcije. Primer:
  ```
  int* f(int=10);   // tip int*(int)
  ```
  Prethodno bi trebalo imati na umu pri deklarisanju pokazivača na funkciju, gde se mora napisati potpuno poklapanje tipa funkcije na koju će da ukazuje pokazivač. Za prethodnu funkciju f:
  ```
  int*(*p1f)(int) = f;
  int*(*p2f)() = f;    // NOK
  ```
- Paziti na to da deklaracija funkcije u unutrašnjoj oblasti važenja sakriva deklaraciju istoimene funkcije u spoljašnjoj oblasti važenja.

2.2.2.4. Pokazivači

Pokazivač na tip je izveden iz tog tipa i namenjen je za čuvanje adrese objekta datog tipa na koji ukazuje. Sintaksa je kao što sledi:

```
TIP *pt;
```

Prethodno se čita kao `pt` je pokazivač na objekte tipa `TIP`, ili na niz objekat tipa `TIP` pri čemu vredi pointerska aritmetika koja vrši promenu pokazivača za veličinu tipa na koji ukazuje. Navedeni tip ne može biti referenca ili bit polje. Ako se želi pristup objektu na koji ukazuje `pt`, onda se koristi dereferenciranje operatorom *(zvezdica). Uzimanje adrese objekta datog tipa se vrši operatorom ampersand (&) nad tim objektom, a onda se ta adresa može dodeliti pokazivaču na takav tip. Pokazivač može da ukazuje i na izvedeni i korisnički tip. Pokazivač koji ne ukazuje ni na kakav objekat ima vrednost `0` i ta vrednost mu se može direktno dodeliti.
Primer:

```
int i=0;
int *pi=&i;    // uzimanje adrese, pi ukazuje na i
```

```
*pi=10;         // dereferenciranje i dodela, i = 10
int **ppi=&pi; // ppi ukazuje na pi
**ppi=20;       // dvostruko dereferenciranje kojim se
                // dolazi do pi, a onda do i,
                // tako da je i=20
pi=0;           // ne ukazuje ni na šta, pravilnije je
                // pi=nullptr;

Tacka2D A(0,0);    // objekat tipa Tacka2D sa
                   // koordinatama 0,0
Tacka2D *pT2D=&A; // uzimanje adrese,
                   // pT2D ukazuje na tačku A
```

Pravilno je koristiti vrednost nullptr, jer se izbegava dvosmislenost u sledećem smislu:

```
void func(int n);
void func(char *s);
func( NULL ); //#define NULL 0, zove prvu funkciju
```

rešenje za poziv druge funkcije:

```
func( nullptr );
```

Poseban tip pokazivača je pokazivač na ugrađeni tip `void` i čita se kao pokazivač na nešto u memoriji, tako da može da uzme adresu bilo kog objekta ili vrednost bilo kog pokazivača. Suprotna dodela, gde se vrednost pokazivača na `void` dodeljuje nekom pokazivaču nije dozvoljena bez eksplicitne konverzije.

```
int i=0;
int *pi=&i;     // uzimanje adrese, pi ukazuje na i
void *pvoid=pi; // pvoid ukazuje na i
pi=(int*)pvoid; // ili pi=static_cast<int*>(pvoid)
pi=pvoid;       // NOK
```

C++ podržava pointersku aritmetiku, tako da se na pokazivač koji ukazuje na niz objekata može primeniti npr. operacija sabiranja sa celobrojnom veličinom. Rezultat je pokazivač koji ukazuje na isti tip kao i pokazivač koji je operand sabiranja. Pomeranje pokazivača je prema veličini objekta na koji ukazuje. Npr. ako pokazivač ukazuje na objekat npr. tipa `Avion`, onda će se inkrementiranjem pokazivača dobiti vrednost pokazivača koja u memoriji gleda na sledeći `Avion` u nizu. Rezultat sabiranja pokazivača sa celim brojem ukazuje na element niza koji je u memoriji udaljen za proizvod tog broja i veličine objekta na koji taj pokazivač ukazuje. Prethodno vredi i za oduzimanje, samo što je pozicija u memoriji udaljena ispred mesta na koje ukazuje pokazivač za vrednost navedenog proizvoda. Identifikator niza se u izrazu implicitno konvertuje u pokazivač na tip elemenata tog niza koji ukazuje na nulti element niza (osim kod upotrebe operatora sizeof i & ili pri inicijalizaciji referenci).

Primer:

```
int a[10];      // niz a od 10 elemenata tipa int
int *pi=&a[0];  // pi je pokazivač na int koji ukazuje na a[0]
*(pi+2)=10;     // pi+2 je pokazivač koji ukazuje na a[2],
                // dereferenciranje sa *, tako da a[2]=10
pi++;           // pi ukazuje na sledeci element, na a[1]
pi--;           // pi ukazuje na prethodni element, na a[0]
```

Rezultat ovakvog sabiranja ili oduzimanja se zna u vreme izvršavanja programa. Dozvoljeno je da ovaj rezultat premaši granicu niza za 1 mesto u oba smera, čime su omogućene petlje u kojima se sekvencijalno pristupa elementima niza. Ako pokazivač premaši navedenu granicu nije definisano šta će se dobiti dereferenciranjem ovakvog pokazivača.
Primer:

```
void f(int a[], int n){
  // obrađuje se n elemenata niza a, gde je n velicina niza
 int *p=&a[0];
  for(int i=0; i<n; i++){
    // ...
    p++;// u poslednjoj iteraciji p ce premasiti granicu niza
  }
}
```

Oduzimanje dva pokazivača istog tipa ima smisla ako oba pokazuju na elemente istog niza. Razlika govori koliko je elemenata tog tipa između mesta u memoriji na koja ukaziju ovi pokazivači.
Primer:

```
int a[10];
int *pi1=&a[0];
int *pi2=&a[3];
int i=pi2-pi1;  // i = 3
pi2-=i;         // pi2=pi2-3, pi2 gleda na a[0]
```

Pokazivač na funkcije odgovarajućeg tipa mora imati potpuno slaganje sa tipom funkcije, dakle, isti povratni tip i isti broj i tipovi argumenata.
Primer:

```
int f(int);    // funkcija tipa int(int)
int g();       // funkcija tipa int(void)
void h(int)    // funkcija tipa void(int)
int (*pf)(int);// pokazivač na funkciju tipa int(int)
pf=&f;         // pf ukazuje na funkciju f
pf=&g;  // pogresno bi bilo jer g nije tipa int(int)
pf=&h;  // pogresno bi bilo jer h nije tipa int(int)
```

Deklaracija pokazivača na funkciju `pf` koja je data kao :

```
int(*pf)(int);
```

ima pripadnu dodelu kao:

```
pf=&f; // ili pf=f
```

Prethodnom dodelom pokazivač ukazuje na funkciju `f` koja ima potpuno slaganje sa tipom pokazivača `pf`. Kod druge i treće dodele, rezultat operacije `&` nad funkcijom g, odnosno, h je tipa pokazivača na funkciju bez argumenata koja vraća tip int, odnosno, tip void int, tako da bi ove dodele bile pogrešne.
Dereferenciranjem pokazivača na funkciju (operatorom *) je kao da je naveden identifikator funkcije na koju pokazivač ukazuje, tako da je moguć poziv date funkcije daljim navođenjem liste stvarnih argumenata. Povratna vrednost izvršavanja navedene funkcije odgovara povratnom tipu navedene funkcije. Dalje se barata sa povratnom vrednosti.
Primer:

```
int f(int);         // tip int(int)
int g(int);         // tip int(int)
int (*pf)(int);     // pf je pokazivač na funkciju int(int);
pf=&f;              // pf ukazuje na f, ili pf=f
int i=(*pf)(1);     // poziva se funkcija na koju ukazuje pf
                    // sa argumentom 1 i rezultat smešta u i;
pf=&g;              // pf sada ukazuje na g, ili pf=g
i=(*pf)(2);         // poziva se funkcija na koju ukazuje pf
                    // sa argumentom 2;
```

Deklaracija pokazivača na konstantu vrši se stavljanjem reči const ispred specifikatora tipa na koji ukazuje pokazivač.

```
const char *pk="asdfgh";        // pokazivač na konstantu
pk="qwerty";                    // OK
pk[3]='a';                      // NOK
```

Deklaracija konstantnog pokazivača vrši se stavljanjem reči const neposredno ispred identifikatora pokazivača.

```
char c[]="asdfgh";
char *const kp=c;               // konstantni pokazivač
kp[3]='a';                      // OK
kp="qwerty";                    // NOK
```

Deklaracija konstantnog pokazivača na konstantu vrši se stavljanjem dve reči const, pri čemu je prva ispred specifikatora tipa na koji ukazuje pokazivač, a druga neposredno ispred identifikatora pokazivača.

```
const char *const kpk="asdfgh";// konstantni pokazivač na
                               // konstantu
kpk[3]='a';                    // NOK
```

```
kpk="qwerty";                    // NOK
```

Konstantni objekat nije promenljiva lvrednost, što znači da rezultat dereferenciranja pokazivača na konstantni objekat nije promenljiva lvrednost.

Često se koristi da je formalni argument funkcije pokazivač na konstantni objekt. Ovim je sprečeno da se taj objekat menja u telu funkcije.

```
int strlen (const char *s);
```

Ako funkcija vraća konstantni tip, onda će privremeni objekat koji prihvata povratnu vrednost funkcije biti konstantnog tipa.

Npr. funkcija f vraća pokazivač na konstantni int, tada nije dozvoljeno da se uradi dodela tom int-u:

```
const int* f();      // f vraća pokazivač na const int
*f()=10;             // NOK
```

Dozvoljeno je da se pokazivač tipa const T* inicijalizuje pokazivačem tipa T* (obrnuto ne može).
Primer:

```
int f1(const char*); // f1 je tipa int const char*
char *p1="qwerty";   // p1 je tipa char*(pravilnije const char*)
int i1=f1(p1);        // ispravno
int f2(char*);            // f2 je tipa int char*
const char *p2="qwerty";  // p2 je tipa const char*
int i2=f2(p2);             // NOK
```

Funkcija f1 govori da se ono na šta ukazuje prosleđeni pokazivač neće menjati. Ovu kontrolu obavlja prevodilac. Moguće je upotrebiti eksplicitnu konverziju static_cast kojom bi se omogućilo da se pokazivač na nekonstantni objekat inicijalizuje pokazivačem na konstantni objekat.

Pri indeksiranju nizova vredi komutacija identifikatora niza i indeksa: niz[indeks] je isto što i indeks[niz]. Gledano prema pokazivačima prethodni izrazi su isto što i *(niz+indeks).
Primer:

```
int ar[10];
int i=0;
ar[i]=6;        // isto kao *(ar+i)=6;
(ar+1)[i]=10;   // isto kao *((ar+1)+i)=3, ar[i+1]=10;
int *pi=ar;     // ar se konvertuje u pokazivač na int,
                // pi ukazuje na ar[0];
pi[i+2]=11;     // isto kao *(pi+(i+2))=11, ar[i+2]=11;
(i+3)[ar]=12;   // isto kao *((i+3)+ar)=12, ar[i+3]=12
```

Bilo bi dobro da se niz ar koji je deklarisan npr. kao:

```
int ar[3][5];
```

čita kao pokazivač na nulti element niza koga čine 3 elementa koji su tipa niz od 5 elementa tipa int.

Sled konverzija za izraz ar[x][y] za prethodno deklarisani ar je kao što sledi:
- ar se konvertuje u pokazivač na nulti element niza ar;
- na prethodni pokazivač se dodaje x;
- vrši se dereferenciranje prethodnog pokazivača tako da se dobije element tipa niz od 5 elemenata tipa int;
- sada se za ovaj niz konvertuje pokazivač na nulti element ovog niza;
- na prethodni pokazivač se dodaje y;
- vrši se dereferenciranje prethodnog pokazivača čime se dobija traženi element int.

Program koji računa dužinu niza znakova dat je na slici 2.2.3.

```
int strlen (char *pstr)
{
  int iduzina=0;
   for (; *pstr++; iduzina++);
   return iduzina;
}
```

Slika 2.2.3. Računanje dužine niza znakova

Funkcija strlen ima formalni argument `pstr` tipa pokazivač na znak (niz znakova). Stvarni argument može biti objekat istog tipa ili tipa niz znakova. U drugom slučaju se vrši potrebna implicitna konverzija. Nakon ovoga `pstr` će biti "kopija" koja gleda na original. Indeks `iduzina` se postavlja na nulu i njegova vrednost će predstavljati broj znakova u nizu. Izrazu `*pstr++` znači da se prvo radi dereferenciranje (uzimanje znaka na koga pokazuje `pstr`), a onda se vrši pointerska artitmetika sabiranja (ovde inkrementiranja) `sptr+1`. Ako je uzeti znak različit od nule, koja je poslednji znak niza, onda se u petlji inkrementira indeks `iduzina` i ide na sledeću iteraciju. Ako je uzeti znak nula, petlja se prekida i ide se na vraćanje indeksa `iduzina` pozivaocu funkcije, a `pstr` ukazuje iza poslednjeg znaka u nizu. Primeri poziva funkcije strlen:

```
const char *s="Zdravo!";
int len1=strlen("Kako se zoves?");
int len2=strlen(s);
```

Program koji vraća prvi indeks traženog znaka u datom nizu karaktera dat je na slici 2.2.4. U datoj for petlji se postavlja da je indeks traženog znaka `ind` jednak `0` te se ispituje na šta ukazuje pokazivač `pstr`. Ako `pstr` ne gleda na kraj niza (nije `0`), onda se u `if` naredbi proverava da li je taj znak na koga gleda `pstr` jednak znaku `znak`, uz istovremeno inkrementiranje pokazivača `pstr`. Ako je to traženi znak, onda funkcija vraća indeks prvog takvog znaka u prosleđenom nizu karaktera. U suprotnom slučaju će se inkrementirati indeks `ind` i ići u

sledeću itreraciju gde se opet vrši isti postupak. Ako se dođe do kraja niza, *pstr ima vrednost `0` te for petlja završava, nakon čega se izvršava naredba return -1, što znači da nije nađen prosleđeni znak u prosleđenom nizu znakova.

```
int strsrch (char *pstr, char znak)
{
  for (int ind=0; *pstr; ind++)
    if (*pstr++ == znak) return ind;
  return -1;
}
```

Slika 2.2.4. Vraćanje indeksa traženog znaka u datom nizu

Sledi par primera gde treba biti pažljiv pri razumevanju koda u kome se navode pokazivači:

```
int(*x[])(int);      // niz pokazivača na funkcije tipa int(int)
int(*(*y)(int))(int);// pokazivač na funkciju koja ima argument
                     // tipa int, a vraca pokazivač na funkciju
                     // tipa int(int)

int *(*pf)(int), *pi, *ar[10];
// pf     je pokazivač na funkciju tipa int*(int)
// pi     je pokazivač na int
// ar     niz od 10 elemenata tipa int*
```

Pokazivač na objekat `bmw` tipa `Automobil` može se inicijalizovati sintaksom kao što sledi:

```
Automobil bmw("BMW740", 75000.00); //naziv modela i cena
Automobil *pautomobil = &bmw;    // pautomobil ukazuje na bmw
(*pautomobil).IspisiPodatke();   // poziv metode klase Automobil
pautomobil->IspisiPodatke();     // moze i ovako
```

2.2.2.5. Pokazivači na članove klase

Pokazivači na članove (*class member pointers*) klase ukazuju na nestatičke članove klase, a ne na člana objekta. Radi se pokazivaču na nestatičkog člana klase datog tipa, tako da taj pokazivač može da ukazuje na bilo koji nestatički član klase tog tipa. Napomena: Pokazivači na statičke članove klase su standardni pokazivači.

Sintaksa pokazivača na članove klase koji su nestatički atributi je kao što sledi:

```
TIP KLASA::* pclkltip;
```

Prethodno se čita kao `pclkltip` je pokazivač na nestatičke članove klase `KLASA` tipa `TIP`.

Neka je data klasa:

```
class C
{
  public:
```

```
    int aa;
    int bb;
    int ff();
    int gg();
};
int C::*pci; // pokazivač na int članove klase C
pci=&C::aa;  // pci ukazuje na aa koji je int član klase C
```

Primenom operatora uzimanja adrese (ampersand `&`) na identifikator nestatičkog člana klase tipa `T` vrši se dodela kojom pokazivač na člana klase tipa `T` ukazuje na taj član klase. Pristup članu objekta obavlja se navođenjem identifikatora objekta i operatora tačka zvezdica (`.*`) iza koga sledi pokazivač na članove klase. Pristup članu objekta preko pokazivača na objekat klase izvodi se korišćenjem operatora minus-veće-zvezdica (`->*`) pri čemu je levi operand pokazivač na objekat date klase dok je desni operand pokazivač na članove klase datog tipa.

Nastavak prethodnog primera:

```
C cc;
cc.*pci=10;   // pristupanje članu cc.aa (=10)
C *pc=&cc;    // pc ukazuje na cc
pc->*pci=20;  // isto kao pc->cc (cc=20)
pci=&C::bb;   // pci ukazuje na članove bb klase C tipa int
cc.*pci=30;   // pristupanje članu cc.bb (=30)
```

Ako se radi o pokazivaču na članove klase koji su nestatičke metode onda je sintaksa kao što sledi:

```
TIP (KLASA::* pcktip)(lista_formalnih_argumenata_metode);
```

Nastavak prethodnog primera:

```
int (C::*pcf)() = &C::ff; // pokazivač na nestatičku metodu
                          // klase C tipa int()
int i = (cc.*pcf)();        // isto kao cc.ff()
```

Pokazivač na člana klase može biti i const i volatile.

```
int C::*const pxf;
int C::*volatile pxf;
int (C::*const pxf)();
int (C::*volatile pxf)();
```

Tip pokazivača na člana klase uključuje klasu u kojoj je nestatički član deklarisan, a ne na klasu izvedenu iz te klase.
Primer:

```
class Osnovna {
  public:
```

```
    int osn;
};
class Izvedena : public Osnovna
{
  public:
    int izv;
};
int main()
{
  int Osnovna::*po1 = &Osnovna::osn;
  int Osnovna::*po2 = &Izvedena::osn;
  Izvedena iz;
  iz.*po2 = 10; iz.*po1 += 1;  // osn = 10, pa osn = 11
  return 0;
}
```

Jedan tip pokazivača na članove date klase može se eksplicitno konvertovati u drugi tip pokazivača na članove te klase.

Pokazivač na atribute klase čuva relativni pomeraj atributa u strukturi podataka objekta, pri čemu se startuje od 1 (0 bi značilo da pokazivač ne gleda ni na šta). Npr. ako bi klasa imala float i char atribute onda bi ofset bio redom 1 i 5.

2.2.2.6. Reference

Referenca je izvedeni tip koji se označava kao T& što znači referenca na tip T. Referenca je kao nadimak (sinonim) za objekat. Pri definiciji reference tipa T&, ista se mora i inicijalizovati objektom tipa T sa kojim je neraskidivo vezana (kao konstantni pokazivač). Referenca nije objekat i ne zauzima prostor u memoriji, tako da se ne može ni kreirati kao dinamički objekat. Zamislite je kao konstantnu adresu objekta na koji ona upućuje. Referenci, za razliku od pokazivača, ne treba operator dereferenciranja za posredan pristup. Dozvoljeno je vezati referencu za privremeni objekat.

Nije dozvoljeno deklarisati kao što sledi:

- niz referenci,
- pokazivač na referencu,
- referencu na void,
- referencu na bit-polje,
- referencu na referencu.

Na slici 2.2.5. dat je primer poziva funkcije u kome se argument prosleđuje kao referenca tako da se dešavanje sa argumentom reflektuju na original na koji upućuje ta referenca. Funkcija f prihvata 3 argumenta, int, referencu na int i pokazivač na int. U telu funkcije f se prvi argument postavlja na 1, drugi (na

koga upućuje referenca) se postavlja na 2 i treći (na koga ukazuje pokazivač) se postavlja na 3. U funkciji main kreirane su tri promenljive tipa int koje su inicijalizovane na 0, a onda sledi poziv funkcije f gde su stvarni argumenti redom: identifikator prvog argumenta (prenos po vrednosti), identifikator drugog argumenta (prenos po referenci) i adresa trećeg argumenta (prenos po adresi). Iz poziva funkcije ne može se znati da li argument prenosi po vrednosti ili po referenci jer je ista sintaksa, ali se to zna po formalnim argumentima date funkcije.

```
void f(int i, int& j, int* k)
{
  i=1;  // ne utiče na original, stvi=0
  j=2;  //    utiče na original, stvj=2
  *k=3; //    utiče na original  stvk=3
}
int main ()
{
  int stvi=0, stvj=0, stvk=0;
  f(stvi,stvj,&stvk);
  return 0;
}
```

Slika 2.2.5. Referenca kao formalni argument

Deklaracija reference ne mora imati inicijalizaciju reference kao što sledi:

- pri deklaraciji formalnog argumenta funkcije (prethodni primer);
- pri deklaraciji povratne vrednosti funkcije;
- pri deklaraciji u kojoj je naveden specifikator extern;
- pri deklaraciji klase gde je referenca član klase.

Obraćanje referenci je obraćanje objektu na koga ista upućuje.
Primer:

```
int ocena=5;
int &ri=ocena;  // referenca koja upućuje na prom. ocena
ri+=5;          // menjanje originala: ocena je 10
int *pi=&ri;    // pi ukazuje na prom. ocena
int kopijar=ri; // pristup preko reference do prom. ocena
int kopijap=*pi;// pristup preko pokazivača do prom. ocena
```

Može postojati referenca na funkciju datog tipa:

```
int (&rf)()=f;  // referenca na funkciju f tipa int(void)
int& f2();      // funkcija tipa int&(void)
int* pi;        // pokazivač na int
int*& rpi = pi; // referenca na pokazivač na int
```

Referenca deklarisana kao const ili volatile nema značenje:

```
int& const cri = i,     // const referenca na int
&volatile vri = i;      // volatile referenca na int
```

Referenca može da referencira const ili volatile objekat:

```
const int& rci = i;     // referenca na const int
volatile int& rvi = j; // referenca na volatile int
```

Napomena:
Biti pažljiv pri korišćenju operanda &, odnosno, paziti da li se radi o tipu reference ili o uzimanje adrese.

Referenca na dati tip mora se inicijalizovati objektom tog tipa ili objektom koji se može konvertovati u taj tip.
Primer:

```
int i = 0;
int &ri1 = i;
int &ri2 = ri1; // OK, ri1 je referenca na tip int
```

Ako se želi da referenca upućuje na konstantu, onda se mora u deklaraciji reference navesti da se radi o referenci na konstantan objekat, tako da se onda od konstante kreira privremeni objekat na koga upućuje referenca.
Primer:

```
const int &rci = 10;
```

U tabeli 2.2.3. data su pravila inicijalizovanja referenci na tip T prema specifikatorima const, volatile i nasleđivanju.

Tabela 2.2.3. Inicijalizacija referenci za nasleđivanje, const i volatile slučajeve

Referenca na objekat tipa	Može se inicijalizovati objektom tipa
T	T
const T	T const T
volatile T	T volatile T
OsnovnaKlasa	KlasaJavnoIzvedenaIzOsnovneKlase

Ako funkcija vraća referencu, onda poziv takve funkcije može biti left-value (stoji sa leve strane znaka jednakosti). Primer gde funkcija vraća referencu dat je na slici 2.2.6. U funkciji `main` inicijalizovan je objekat `ocena` tipa `int` na vrednost `5`, a onda je pozvana funckija `f` kojoj je kao stvarni argument prosleđena referenca na objekat `ocena`. Funkcija f vraća referencu na tip int, a ovde je to referenca na promenljivu `ocena`.

```
int& f(int &ri)
{
  return ri;      // ri upućuje na stvarni argument
```

```
}
int main ()
{
  int ocena(5);
  f(ocena) = 10;  // f(ocena) vraća referencu na prom. ocena
                  // tako da vredi dodela te je ocena=10
  return 0;
}
```

Slika 2.2.6. Funkcija koja vraća referencu

Operatori koji se primenjuju na referencu daju efekat kao da su primenjeni na ono na šta referenca upućuje. Pošto je referenca za objekat datog tipa njegov sinonim onda i veličina tipa i veličina reference na isti tip moraju biti jednake (`sizeof(int)` je isto kao i `sizeof(int&)`).

Vezivanje reference za dinamički kreirani objekat je kao što sledi:

```
int &ri = *new int(10);
```

Ovde je operatorom dereferenciranja izvršena inicijalizacija reference na tip `int` sa dinamičkim objektom koji je kreiran operatorom `new` i inicijalizatorom `10`. Na ovaj način je dinamički kreiran bezimeni objekat tipa `int` dobio sinonim `ri`. Sve sledeće operatorske radnje nad referencom `ri` su radnje nad tipom `int`, odnosno, objektom koji je u dinamičkoj memoriji (npr. inkrement `ri++`, ili dodela pokazivača na taj objekat, `int *pi = &ri;`).

Ukidanje dinamičkog objekta na koga upućuje referenca je kao što sledi:

```
delete &ri;            // potrebna je adresa objekta
```

Inicijalizacija reference nekim objektom zahteva da taj objekat živi jednako ili duže od reference.
Primer:

```
int& f()
{
  int& ri=*new int(10);
  // NOK, ako bi se ovde uradilo delete &ri
  return ri;
}
int& g(int &ri)
{
  return ri;
}
int& f1(){
  int alo=1;
  return alo;// NOK, referenca na automatski lokalni objekt
}
int& f2(int i)
```

```
{
  return i; // NOK, referenca na formalni argument kao lokalni
            // automatski objekat
}
int& f3(int &ri)
{
  int r=*new int(ri); //NOK, automatskom lokalnom objektu je
                      //  dodeljena vrednost koju ima
                      //  dinamički kreirani objekat tipa
                      //  int koji je inicijalizovan
                      //  vrednošću objekta na koji upućuje ri
  return r;
}
```

Curenje memorije (*memory leak*) će se desiti ako se uradi da se samo uzme vrednost dinamičkog objekta, a on sam ostavi nereferenciran:

```
int ri=*new int(10);
```

objekat koji je stvoren sa new int(10) ostaje da živi sve do kraja programa te ako bi se ovakvo nešto dešavalo više puta u toku rada programa videli biste u task manager opcijama da Vam program zauzima sve više i više memorije.

2.2.3. Korisnički tipovi

Korisničke tipove podataka kreira korisnik prema pravilima objektno orijentisanog programiranja. Korisnički tipovi podataka su: nabrajanja, strukture, unije i klase.

2.2.3.1. Nabrajanja

Nabrajanje (*enumeration*) predstavlja skup diskretnih, unapred definisanih vrednosti (simboličkih konstanti). Nabrajanje omogućuje da korisnik definiše apstraktni, diskretni skup vrednosti koje imenuje simboličkim konstantama. Često je u programima potrebno da neke funkcije vraćaju kôd koji predstavlja status izvršenja odgovarajuće operacije. Operacija se može završiti uspešno, ali i neuspešno, kada funkcija treba da vrati kôd greške.
Ako se definiše poseban tip za statusni kôd funkcije (ne koristimo neki od ugrađenih tipova) kao skup tačno definisanih diskretnih vrednosti povećaće se čitljivost koda.

```
enum Status {OK,NOK,NUMBER_OF_ENUMERATORS};
```

Ključnom reči `enum` počinje definicija nabrajanja, sledi identifikator tipa nabrajanja (može se upotrebiti kao ime tipa pri deklarisanju objekata tipa tog nabrajanja) i sledi skup diskretnih vrednosti koje predstavljaju dato nabrajanje.

Objekti tipa nabrajanja deklarišu se na uobičajen način:

```
Status statusA = NOK;   // promenljiva statusA je tipa Status
```

```
retValue = f();          // f je tipa Status(void)
```

Diskretne vrednosti iz skupa koji je naveden u deklaraciji nabrajanja predstavljaju konstante koje imaju vrednosti celih brojeva počevši od 0.

Ova dodela vrednosti može se promeniti eksplicitnim definisanjem celobrojne vrednosti neke konstante iz skupa. Tada je naredna konstanta za jedan veća ako nije bilo dodele toj konstanti.

Primer:
Nabrajanje automobila

```
enum Cars { FIAT,     // vrednost  0
            AUDI=9,   // vrednost  9
            MERCEDES, // vrednost 10
            BMW=AUDI, // vrednost  9
            VOLVO=8}; // vrednost  8
```

Dve konstante iz istog nabrajanja mogu imati iste vrednosti (enum Hm {A=0, B=A, C};).
Postoji standardna konverzija iz tipa nabrajanja u tip int.

```
int i=MERCEDES+1;     // vrednost 11
```

Nije dozvoljena konverzija iz int u enum.

```
Cars car = 10;  // NOK
```

Konstantama je moguće dodeliti izraz koji sadrži pobrojane konstante:

```
enum Cars {FIAT, NISSAN, MERCEDES=FIAT+5};
```

Problem sa siromašnom enumeracijom je sledeći:

```
enum Voce{Jabuka, Banana};
enum NebeskaTela {Merkur, Venera};
int i = Voce::Jabuke;
int j = NebeskaTela::Merkur;
if(i==j) cout<<"Ovo bi proslo"<<endl;
```

Bogatija deklaracija nabrajanja (*Strongly Typed Enumerations*) je kao što sledi:

```
enum class Cars {FIAT, NISSAN, MERCEDES=FIAT+5};
Cars car = Cars::FIAT;
if (car == Cars::MERCEDES)
{
  cout<<"The best !"<<endl;
}
//ne bi bilo dozvoljeno int i = Cars::MERCEDES;
```

Ovim je izbegnuto da se izvrši nekorektno poređenje. Moguće je pisati i još preciznije opisujući i koji je tip mapiranja vrednosti simboličkih konstanti nabrajanja:

```
enum class Cars: char {FIAT, NISSAN, MERCEDES=FIAT+5};
```

2.2.3.2. Strukture

Struktura je specijalni slučaj klase kod koje su svi članovi podrazumevano javni, a podrazumevano nasleđivanje takođe javno. Deklaracija strukture počinje ključnom rečju struct, a ostalo je isto kao i kod klase. Struktura (kao i klasa) može imati konstruktore, atribute, metode i destruktore. Članovi strukture mogu biti po pravu pristupa javni (*public*), zaštićeni (*protected*) i privatni (*private*).

Ime strukture je (kao i kod klase) ime korisničkog tipa. Koriste se za realizaciju jednostavnih struktura podataka.
Primer:

```
struct Tacka {
  double x;         // javni atributi x i y
  double y;
  Tacka(double,double);  // javni konstruktor
};

Tacka::Tacka (double fx, double fy)
{
  x = fx;
  y = fy;
}  // definicija konstruktora

int main()
{
   Tacka A(1.0,1.0), B(2.0,2.0);
   Tacka C(A.x+(B.x-A.x)/2, A.y+(B.y-A.y)/2); // srednja tacka
   ...
   return 0;
}
```

Struktura Tacka omogućuje kreiranje korisničkih objekata tipa 2D tačaka. Javni članovi su: x, y i konstruktor. U funkciji main su kreirane dve tačke (A i B) te tačka C kao srednja tačka duži AB. Kreiranje tačke se izvodi pozivom konstruktora koji inicijalizuje atribute prosleđenim stvarnim argumentima.

2.2.3.3. Bit polja

Bit polje se deklariše kao celobrojni atribut strukture ili klase gde se navodi dužina tog polja u bitima. Sintaksa je kao što sledi:

```
int bp : 16;  // bit polje bp dužine 16 bita
```

Bit polja se realizuju tako da se smeštaju jedno iza drugog u memoriji, pri čemu se može forsirati poravnanje bit polja na memorijsku ćeliju računara.

Primer:
Struktura S sadrži redom bit polja bp1, bp2, bp3 dužine 6, 3 i 7 bita, respektivno. Neka su značenja ovih bit polja redom: operacioni kod, mod adresiranja, ofset.

```
struct S
{
  int bp1 : 6;     // bit polje širine 6 bita
  int bp2 : 3;     // bit polje širine 3 bita
  int bp3 : 7;     // bit polje širine 7 bita
};

void SetFields(S *ps, int opcode, int mode, int offset)
{
  ps->bp1 = opcode;
  ps->bp2 = mode;
  pc->bp3 = offset;
}
```

Bit polja mogu biti smeštana ulevo ili udesno što zavisi od mašine. Forsiranje poravnanja sledećeg bit polja na adresabilnu jedinicu memorije se postiže bezimenim bit poljem dužine 0, kao što sledi:

```
int :0;
```

Ovo bezimeno bit polje nije član strukture (klase) i ne može se inicijalizovati. Ranije je rečeno da se ne može uzeti adresa bit polja. Pošto je namena da se prvenstveno korsite za najniži nivo sistemskih programa date mašine, onda je jasno da je prenosivost koda vezana za karakteristike same mašine.

2.2.3.4. Unije

Unija je korisnički tip podataka gde svi atributi unije počinju od iste adrese. Pošto se atributi preklapaju, onda unija u jednom trenutku sadrži samo jedan od svojih deklarisanih atributa. Unija, pored atributa, može imati konstruktore, destruktore i metode. Nije dozvoljeno ugrađivanje gde atibut ima konstruktor, destruktor ili preklopljene operatore dodele. Unija ne može imati članove na nivou unije niti vurtuelne metode. Unija ne može biti natklasa niti potklasa.

Za deklaraciju unije koristi se ključna reč union.

```
union Unija {
  short s;
  double d;
};
int main ()
{
  Unija unija;
  unija.s=6;     // sada unija sadrži atribut s
  unija.s+=4;
  unija.d=3.14;  // sada unija sadrži atribut d
  int a=unija.s; // NOK, jer trenutno živi samo atribut d
```

```
    return 0;
  }
```

Unija koja nema ime jeste anonimna unija. Deklaracijom anonimne unije ne definiše se korisnički tip već se definiše bezimeni objekat tipa te unije. Članovima anonimne unije (moraju biti javni) se pristupa direktno i moraju se po imenima razlikovati od ostalih identifikatora u istoj oblasti važenja. Bezimena unija nema metode. Ako je anonimna unija globalna, onda se mora deklarisati kao static.

Primer:

```
  void f() {
    union
    {
      int i;
      double d;
    };
    i=10;     // direktana pristup članu bezimene unije
    d=3.14;
  }
```

Ako se za uniju kojoj nije navedeni ime definišu objekti tipa te unije ili pokazivači na objekte tipa te unije, onda se ne radi o anonimnoj uniji.

Primer:

```
  union
  {
    int i;
    double d;
  }
  obj,        // objekat tipa navedene unije i pokazivač na uniju
  *pu=&obj;   // pokazivač na objekat navedene unije
  i=1;        // NOK, nije anonimna unija
  obj.i=10;   // OK
  pu->i=3;    // OK
```

Namena unija je da se uštedi memorijski prostor u sistemskim programima pri definisanju korisničkih tipova niskog nivoa pri čemu se u jednom trenutku čuva maksimalno jedan atribut unije. Objekti tipa unije obično su ugrađeni u strukture ili klase.

Primer:

```
  struct NumOrStr {
    char tag;     // broj 'N', string 'S'
    union {
      short ibroj;
      char* pchar;
    } brstr;
  };
```

Struktura `NumOrStr` ima atribut `tag` koji određuje da li je u pitanju broj ili string. Programer će, ako atribut `tag` ima vrednost `'N'` koristiti unijin član `ibroj`, odnosno, ako atribut `tag` ima vrednost `'S'` koristiće se unijin član `pchar`.

2.2.3.5. Klase

Klasu opisuju članovi klase koje čine: atributi (podaci članovi) i metode (funkcije članice). Objekat predstavlja primerak klase čije je stanje određeno vrednošću njegovih atributa, dok je ponašanje opisano metodama. Metode u opštem slučaju mogu da vrše izmenu stanja objekta, ispitivanje stanja objekta i da komuniciraju sa drugim klasama. Atributi mogu biti ugrađeni ili korisnički tipovi (npr. atributi su objekti druge klase ili pokazivači na objekte iste ili druge klase). Detaljnije u poglavlju klase i objekti.

Primer koji opisuje dvodimenzionu tačku, kao jednostavni korisnički tip, dat je na slici 2.2.7.

```
// fajl Tacka2D.h -------------------------------------------
#pragma once
class Tacka2D
{
    double x;
    double y;
  public:
        Tacka2D();
    void PostaviKoordinate(double xx, double yy);
    double UdaljenostDoIshodista();
    double UdaljenostDoTacke(Tacka2D drugatacka);
    void IspisiPodatke();
};
// fajl Tacka2D.cpp -----------------------------------------
#include "Tacka2D.h"
#include <cmath>
#include <iostream>
Tacka2D::Tacka2D()
{
  x = y = 0; //ishodiste
}
void Tacka2D::PostaviKoordinate(double xx, double yy)
{
  x = xx;
  y = yy;
}
double Tacka2D::UdaljenostDoIshodista()
{
  return hypot(x,y);
}
```

```
  double Tacka2D::UdaljenostDoTacke(Tacka2D drugatacka)
  {
    return hypot(x-drugatacka.x,y-drugatacka.y);
  }
  void Tacka2D::IspisiPodatke()
  {
     std::cout<< "("<< x <<","<< y <<")";
  }
  // fajl gde je main ----------------------------------------
  #include <iostream>
  #include "Tacka2D.h"
  #define _ISPISI_PODATKE
  int main()
  {
    Tacka2D A;
    A.PostaviKoordinate(3.0,4.0);
    std::cout<<"Udaljenost tacke A do koordinatnog pocetka je "
             <<A.UdaljenostDoIshodista()<<std::endl;
    Tacka2D B; B.PostaviKoordinate (9.0,12.0);
    std::cout<<"Udaljenost tacke A do tacke B je "
             <<A.UdaljenostDoTacke(B)<<std::endl;
#if defined _ISPISI_PODATKE
    std::cout<<std::endl;
    std::cout<<"Tacka B";B.IspisiPodatke();
    std::cout<<std::endl;
    std::cout<<"Linija koda je "<<__LINE__<<std::endl;
#endif
    return 0;
  }
```

Slika 2.5. Primer klase Tacka2D

Na slici 2.2.7. dat je sadržaj fajlova zaglavlja (`Tacka2D.h`), implementacije (`Tacka2D.cpp`) i fajla gde se u okviru istog projekta nalazi main. U fajlu zaglavlja, nakon komentara o nazivu fajla, data je direktiva prevodiocu `#pragma once` koja govori da će fajl zaglavlja biti uključen samo jednom u celom projektu. U narednoj liniji koda nalazi se ključna reč `class` iza koje sledi identifikator klase `Tacka2D`. Početak deklaracije klase `Tacka2D` dat je velikom levom zagradom {, a kraj velikom desnom zagradom iza koje je znak tačka-zarez (;). Ako se ne navede pravo pristupa članovima klase, onda je ono privatno, tako da su atributi `x` i `y` privatni atributi klase `Tacka2D` i tipa su `double`. Ovi deklarisani atributi pripadaju objektu klase `Tacka2D` i za njih se u ovom trenutku ne odvaja memorijski prostor već je samo naglašeno da će svaki objekat klase `Tacka2D` imati svoja dva atributa x i y. Iza njih naveden je modifikator pristupa `public:` što znači da sledi blok javnih članova klase:

- konstruktor bez parametara (konstruktor ne vraća tip) `Tacka2D();`

- metoda `PostaviKoordinate` tipa `void` (`double`, double);
- metoda `UdaljenostDoIshodista` tipa `double(void);`
- metoda `UdaljenostDoTacke` tipa `double(Tacka2D);`
- metoda IspisiPodatke tipa void(`void`).

U fajlu implementacije nakon komentara sledi linija #include `"Tacka2D.h"` kojom se uključuje zaglavlje klase `Tacka2D` da bi se znalo za imena koja se pominju u `Tacka2D.h`. Nakon ove linije je linija koda `#include <cmath>` koja uključuje standardnu matematičku biblioteku iz koje ćemo koristiti funkciju hypot za računanje hipotenuze. Liniju `Tacka2D::Tacka2D()` trebalo bi čitati kao metoda `Tacka2D` bez argumenata iz klase `Tacka2D.` Pošto metoda ima ime klase i ne vraća vrednost (ni void), onda se radi o konstruktoru koji kreira objekat (instancu) klase `Tacka2D.` Ideja je da konstruktor postavi početno stanje objekta inicijalizacijom svih njegovih atributa. Konstruktor `Tacka2D` u svom telu kreiranom objektu tipa `Tacka2D` izvrši inicijalizaciju atributa `x i y` (inicijalizacija atributa objekta) kojom je tačka postavljena u ishodište `(0,0).` Linija `void Tacka2D::PostaviKoordinate(double,double)` se čita kao metoda `PostaviKoordinate` iz klase `Tacka2D` je tipa `void (double, double).` Metode opisuju ponašanja klase i vrše akcije nad objektima klase. U telu ove metode atributima x i y objekta klase Tacka2D se dodeljuju nove vrednosti koje odgovaraju vrednostima fomalnih argumenata, a koji su opet pri pozivu ove metode inicijalizovani stvarnim argumentima za x i y atribute. Linija double Tacka2D::UdaljenostDoIshodista() se čita kao metoda UdaljenostDoIshodista iz klase Tacka2D je tipa double(void). U telu ove metode vraćen je rezultat funkcije hypot koja računa hipotenuzu za prosleđene katete (`return hypot(x,y);`). Slična ovoj metodi je metoda UdaljenostDoTacke koja računa udaljenost između tačke čija je metoda pozvana i druge tačke koja je prosleđena kao formalni argument ove metode. Metoda IspisiPodatke ispisuje na standardnom izlazu koordinate tačke kao uređen par.

U fajlu gde je `main` uključena su zaglavlja za ispis na konzolu i za 2d tačku (`#include <iostream>` i `#include "Tacka2D.h"`). Definisan je makro _ISPISI_PODATKE koji će služiti za uslovno prevođenje. Linijom `Tacka2D A;` kreira se jedna instanca klase `Tacka2D` tako što je pozvan konstruktor (bez argumenata), tako da su sada koordinate tačke A(0,0). Kreiranjem objekta A odvojen je memorijski prostor za smeštanje atributa x i y objekta A i inicijalno su postavljene vrednosti pripadnih atributa.

Napomena: Da je sintaksa bila `Tacka2D A();` onda bi se radilo o deklaraciji funkcije tipa `Tacka2D(void).`

Linija `A.PostaviKoordinate(3.0,4.0);` znači da je objekat A pozvao svoju metodu koja postavlja koordinate tačke A (atribute x i y) na vrednosti (3.,4.). Metoda cout iz standardnog prostora imena (std::cout) omogućuje ispis na

standardni izlaz operatorom <<. Ispisuje se prvo string literal ("Udaljenost tacke A do koordinatnog pocetka je "), zatim rezultat poziva A.UdaljenostDoIshodista(), a onda znak za novi red (<<std::endl;). Sada se kreira tačka B sa koordinatama (9.0,12.0), ispisuje udaljenost od tačke A do tačke B pozivom A.UdaljenostDoTacke(B). Nakon ovoga sledi blok za uslovno prevođenje koji će u slučaju da postoji definisan makro _ISPISI_PODATKE (#if defined _ISPISI_PODATKE) prevesti kod koji ispisuje podatke o tački B koristeći poziv metode IspisiPodatke i broj linije koda u programu. Ako pomenuti makro nije definisan, onda neće biti preveden kod koji ispisuje podatke o tački B, kao i ispisivanje broja linije koda.

Na kraju, poziva se naredba return 0; kojom program završava rad.

Na slikama 2.2.8, 2.2.9, 2.2.10 i 2.2.11 dat je primer korišćenja makroa i klasa. Na slici 2.2.8. dat je sadržaj fajla MojeKonstante.h koji će biti jednom uključen u program i u kome su definisane tri konstante. Prve dve konstante su definisane kao makrozamena (PRVA_RECENICA i DRUGA_RECENICA), dok je treća definisana specifikatorom const i celobrojnog je tipa (DESETKA).

```
//fajl MojeKonstante.h
  #pragma once
  #define PRVA_RECENICA 1
  #define DRUGA_RECENICA 2
  const int DESETKA=10;
```

Slika 2.2.8. Fajl simboličkih konstanti

U fajlu Vic.h smešteno je zaglavlje klase Vic (slika 2.2.9). Privatni blok članova sadrži dva pokazivača na tip char (prvarecenica i drugarecenica). Javni blok čine: konstruktor bez argumenata, metoda PostaviRecenicu tipa void (int, char*) koja postavlja vrednosti na koje ukazuje dati privatni član klase i na kraju, metoda IspisiRecenice koja ispisuje recenice (i ocenu vica).

```
//fajl Vic.h
#pragma once
class Vic{
    private:
      char* prvarecenica;
      char* drugarecenica;
    public:
      Vic();
      void PostaviRecenicu(int rednibrojrecenice,
                           char* recenica);
      void IspisiRecenice();
};
```

Slika 2.2.9. Zaglavlje klase Vic

U fajlu Vic.cpp smeštena je implementacija klase Vic (slika 2.2.10.). Uključena su zaglavlja iostream, MojeKonstante.h i Vic.h, a navedeno je i korišćenje prostora imena std. Konstruktor klase Vic u svom telu postavlja dve rečenice vica na string literal "prazno". Metoda PostaviRecenicu prihvata int argument koji se odnosi na rečenicu i drugi argument koji predstavlja samu rečenicu. U switch naredbi se na osnovu prosleđene informacije o kojoj se rečenici radi vrši dodela teksta datoj rečenici prema konstantama PRVA_RECENICA i DRUGA_RECENICA, dok se kao podrazumevano daje informacija da nema navedenog rednog broja rečenice. Metoda IspisiRecenice ispisuje tekst prve i druge rečenice, a nakon toga i ocenu (koja je uvek int konstanta DESETKA).

```
//fajl Vic.cpp
#include <iostream>
using namespace std;
#include "MojeKonstante.h"
#include "Vic.h"
Vic::Vic(){ prvarecenica = drugarecenica = "prazno"; }
void Vic::PostaviRecenicu(int rednibrojrecenice,char* recenica){
  switch(rednibrojrecenice)
  {
    case PRVA_RECENICA : prvarecenica  = recenica; break;
    case DRUGA_RECENICA: drugarecenica = recenica; break;
    default: cout<<"nema "<<rednibrojrecenice<<" recenice!!!";
  }
}
void Vic::IspisiRecenice(){
  cout<<prvarecenica<<endl;
  cout<<drugarecenica<<endl;
  cout<<"ocena je "<<DESETKA<<endl;
}
```

Slika 2.2.10. Implementacija klase Vic

Na slici 2.2.11. dat je sadržaj fajla u kome se nalazi funkcija main. Uključena su zaglavlja iostream, MojeKonstante.h (dva puta da se pokaže da radi #pragma once), Vic.h i navedeno je korišćenje standardnog prostora imena std. Metoda main je tipa int (int, char*[]), tako da prihvata argumente koji su prosleđeni iz komandne linije. U for petlji ispisuju se parametri komandne linije, pri čemu je nulti parametar naziv programa koji se pokreće. Nakon ovoga, ispisuje se konstanta DESETKA koja je definisana u fajlu MojeKonstante.h. Kreira se objekat v1 tipa Vic pozivom konstruktora bez argumenata. Pozivom metode IspisiRecenice objekta v1 ispisuju se recenice "prazno", "prazno". Sledi postavljanje rečenica pozivom metode PostaviRecenicu koja prihvata: konstantu o kojoj se rečenici radi i string literal koji predstavlja tekst rečenice. Isto kreiranje i postavljanje rečenice se uradi i za kreirani objekat v2 klase Vic.

```
//fajl gde je main
#include <iostream>
using namespace std;
#include "MojeKonstante.h"
#include "MojeKonstante.h"          //ok, moze opet
#include "Vic.h"
int main(int argc, char* argv[])
{
  for(int i=0; i<argc; ++i)
  {
    cout<<"argv["<<i<<"]="<<argv[i]<<endl;
  }
  cout<<"Konstanta DESETKA="<<DESETKA<<endl;
  Vic v1;
  v1.IspisiRecenice();cout<<endl;
  v1.PostaviRecenicu(PRVA_RECENICA,"Govori ko ti je saucesnik!");
  v1.PostaviRecenicu(DRUGA_RECENICA,"Nisam blesav da izdam
rodjenog brata!");

  Vic v2;
  v2.PostaviRecenicu(PRVA_RECENICA,"Chuck Norris ne jede med.");
  v2.PostaviRecenicu(DRUGA_RECENICA,"On zvace pcele.");

  Vic *pvic;
  pvic = &v1;
  pvic->IspisiRecenice();cout<<endl;
  pvic = &v2;
  pvic->IspisiRecenice();cout<<endl;
  pvic->PostaviRecenicu(PRVA_RECENICA,"OOP");
  pvic->PostaviRecenicu(DRUGA_RECENICA,"Lako je.");
  pvic->IspisiRecenice();
  return 0;
}
```

Slika 2.2.11. Primena klase Vic i simboličkih konstanti

Kreira se pokazivač pvic koji ukazuje na objekat klase Vic. Ovom pokazivaču se dodeljuje da ukazuje na objekat v1. Posredno, preko pokazivača pvic poziva se metoda IspisiRecenice objekta v1. Nakon ovoga se pokazivač pvic preusmerava da ukazuje na objekat v2. Posredno se pozivaju metode objekta v2: za ispis rečenica, za postavljanje rečenica i ponovo na ispis rečenica.

```
argv[0]=C:\__Projects\CPP\Proba\Debug\Proba.exe
Konstanta DESETKA=10
prazno
prazno
ocena je 10
Govori ko ti je saucesnik!
```

```
Nisam blesav da izdam rodjenog brata!
ocena je 10
Chuck Norris ne jede med.
On zvace pcele.
ocena je 10
OOP
Lako je.
ocena je 10
```

Slika 2.2.12. Izlaz programa datog na slici 2.2.11.

Na slici 2.2.12. dat je pripadni izlaz za rad programa datog na slici 2.2.11. koji koristi klasu Vic i simboličke konstante. Razmislite da li bi se nešto promenilo da se u fajlu MojeKonstante.h promeni da npr. bude #define PRVA_RECENICA 2 i #define DRUGA_RECENICA 1.

2.3. Oblast važenja

Svako ime u programu ima oblast važenja (*scope*) u kojoj se isto može koristiti. C++ ima oblasti važenja kao što sledi:

- globalna oblast;
- lokalna oblast;
- oblast funkcije;
- oblast klase;
- oblast prostora imena;
- oblast naredbe.

Ako se globalni objekat označi kao static onda je on vidi samo unutar tog fajla (interno vezivanje). Konstante imaju interno vezivanje.

2.3.1. Globalna oblast

Globalnu oblast važenja imaju globalna imena definisana van funkcija i klasa i važe globalno u programu. Napomena: Ako ispred stoji static onda se to ime vidi samo u fajlu definicije. Globalna imena mogu biti: imena klasa, imena funkcija i imena promenljivih proizvoljnog tipa.

2.3.2. Lokalna oblast

Lokalna imena su deklarisana unutar: složene naredbe, lambde, funkcije, metode ili kao formalni argumenti funkcije ili metode i važe od mesta deklaracije do kraja složene naredbe, funkcije, metode, respektivno. Ako se u unutrašnjem bloku deklariše isto ime kao u spoljašnjem bloku, onda ovo ime skriva ime iz spoljašnjeg bloka. Inače, lokalno ime se vidi i u ugnežđenim blokovima. Ako se želi pristup globalnom imenu koje je skriveno lokalnim imenom, onda se koristi operator razrešavanja :: koji se postavlja ispred globalnog imena.

Primer:

```
int i = 10;
void f (int i)          // lokalno ime
{
  ::i = 20;             // pristup globalnom i koje skriva
                        // lokalni argument i
  int j=10;             // lokalno ime j
  {                     // ugnežđeni blok
    char k=20;          // lokalno ime k
    // vide se i,j,k, globalni i se vidi kao ::i
  }
  // vide se i, j, globalni i se vidi kao ::i

}
void g () {
  int i=10;    // loklani i, globalni i se vidi kao ::i
  {
    int i=5;   // novi i u unutrasnjem bloku skriva i
               // iz spoljasnjeg bloka, globalni i se
               // vidi kao ::i
  }
  i=20;        // i iz spoljašnjeg bloka funkcije,
               // globalni i se vidi kao ::i
}
```

2.3.3. Oblast funkcije

Oblast važenja funkcije se odnosi na labele koje se postavljaju na bilo kom mestu u funkciji. Za prelazak izvršavanja koda na mesto labele koristi se naredba goto imelabele; dok se samo ime labele na proizvoljnom mestu u funkciji navodi kao imelabele:;

```
#include <iostream>
using namespace std;
int main()
{
  cout<<"Hello World";
  {
      goto labela;
      cout<<"hm";
  }
  labela:;
  return 0;
} // ispisace se Hello World
```

2.3.4. Oblast klase

Oblast važenja klase odnosi se na članove te klase. Trebalo bi razlikovati da li se neki član klase "vidi" od toga da li postoji pravo da mu se pristupi.

Unutar klase ili prijatelja klase pristupa se svim članovima te klase, bilo preko objekta te klase, ili preko pokazivača na objekat te klase.

U klasama naslednicama može se pritpupiti javnim i zaštićenim članovima osnovne klase. Van klase može se pristupiti samo javnim članovima klase.

2.3.5. Oblast prostora imena

Oblast važenja prostora imena (namespace) odnosi se na deklarisane članove u tom prostoru imena. Do njih se dolazi navođenjem imena prostora imena, operatora :: i samog imena kome se želi pristupiti.

2.3.6. Oblast naredbe

Oblast važenja naredbe odnosi se na imena deklarisana u naredbama for, if, while ili switch. Takva imena su vidljiva od mesta deklarisanja do kraja bloka te naredbe.

2.4. Životni vek objekata

Kreiranjem objekta za njega se zauzima prostor u memoriji i počinje njegov životni vek koji traje sve dok se taj objekat ne ukine, dakle, oslobodi prostor u memoriji koji je zauzimao.

C++ objekti mogu da imaju sledeći životni vek:
- automatski (automatic),
- statički (static),
- dinamički (dynamic),
- privremeni (temporary).

2.4.1. Automatski objekti

Automatski objekat živi od trenutka nailaska toka programa na njegovu deklaraciju do napuštanja oblasti njegovog važenja.

Reč je o lokalnim objektima koji se pri svakom pozivu (nailasku programskog toka na njihovu deklaraciju) ponovo kreiraju. To su argumenti funkcija i metoda i lokalni automatski objekti.

2.4.2. Statički objekti

Postoje globalni i lokalni statički objekti.

Globalni statički objekat živi od početka programa do kraja izvršavanja programa. Ovakvi objekti se kreiraju i inicijalizuju jednom pre poziva main funkcije, a nalaze se u statičkoj oblasti memorije. Reč je globalnim objektima, koji žive koliko i program. Ako se ispred globalnog imena navede reč static ona ima drugo značenje i ne odnosi se na statički životni vek već na interno povezivanje.

Lokalni statički objekat se kreira i inicijalizuje jednom kada program prvi put naiđe na mesto deklaracije u nekom lokalnom bloku. Ispred lokalnog imena se mora navesti reč static da bi ono imalo statički životni vek.

```
int iglob=10;  // globalni statički objekat,
               // inicijalizuje se jednom pre početka programa
               // životni vek je životni vek programa
void f()
{
   static int istat=3;
      // lokalni statički objekat, oblast važenja je funkcija
      // inicijalizuje se jednom, pri prvom pozivu funkcije
      // životni vek je životni vek programa
}
```

String literali su statički po životnom veku.

2.4.3. Dinamički objekti

Dinamički objekat živi od trenutka kada ga eksplicitno kreira programer (operatorom new) do trenutka kada ga programer eksplicitno ukine (operatorom delete).

```
int *pi = new int(5);  // programer kreira objekat tipa int i
                       // inicijalne vrednosti 5 na koji ukazuje
                       // pokazivač pi tipa int*
...
delete pi;             // ukidanje dinamičkog objekta na koga
                       // ukazuje pokazivač pi
```

Reč je o bezimenim objektima kojima se pristupa indirektno preko pokazivača ili mogu biti posredno "imenovani" ako im se dodeli referenca.

2.4.4. Privremeni objekti

Privremeni objekti žive samo dok su potrebni npr. za izračunavanje izraza, za smeštanje povratne vrednosti funkcije. Životni vek privremenog objekta se završava onda kada se upotrebi njegova vrednost. Reč je o bezimenim objektima.

Primer koji obuhvata globalni objekat, lokalni objekat i lokalni statički objekat:

```
#include <iostream>
```

```
int iglob=1;
void f() {
  int iloka=1;        // inicijalizuje se pri svakom pozivu
  static int istalok; // inicijalizuje se samo pri prvom pozivu
                      // biće postavljen na 0 vrednost
  iglob++;            // inkrement globalnog objekta
  iloka++;            // inkrement lokalnog objekta
  istalok++;          // inkrement statičkog lokalnog objekta
  std::cout<<iglob<<" "<<iloka<<" "<<istalok<<" "<<std::endl;
}
int main ()
{
  for(; iglob<5 ; f() ); //četiri poziva funkcije f
  return 0;
}
// IZLAZ:
// 2 2 1
// 3 2 2
// 4 2 3
// 5 2 4
```

2.5. Memorijske oblasti

Memorijska oblast objekta (*storage class*) predstavlja strukturu podataka u memoriji u koju će objekat biti smešten.
Razlikuju se memorijske oblasti za sledeće objekte:

- automatske;
- statičke;
- dinamičke;
- eksterne.

2.5.1. Stek

Na stek (*stack*) se smeštaju automatski objekti. Stek je organizovan kao LIFO (*last-in-first-out*) struktura podataka, tako da se poslednji objekat stavljen na stek prvi skida sa steka.

Pri pozivu datog bloka na stek će se postaviti svi automatski objekti koji pripadaju tom bloku. Pošto se radi o LIFO strukturi, onda se u bloku koriste objekti sa vrha steka tako da ne postoji uticaj na objekte prethodnog poziva bilo kog bloka. Kada program dođe do kraja bloka, onda će se sa steka ukloniti svi automatski objekti koji pripadaju tom bloku.

Ako se stek prepuni dolazi do greške prekoračenja steka (*stack owerflow*). Takođe, moguće je da se u programu pokuša uzeti nešto sa steka koji je prazan tako da dolazi do greške potkoračenja steka (*stack-underflow*).

2.5.2. Statička oblast

U statičku memorijsku oblast se smeštaju po životnom veku statički objekti: globalni objekti na početku izvršavanja programa i lokalni statički objekti pri prvom nailasku programa na deklaraciju tih lokalnih statičkih objekata.

2.5.3. Dinamička oblast

U dinamičku memoriju (*heap, freestore*) se smeštaju dinamički objekti pri njihovom kreiranju (operatorom new), kada se odvoji potreban memorijski prostor za njihovo smeštanje, odnosno, kada se dinamički objekti ukidaju, onda se oslobađa memorijski prostor koji su zauzimali.

Može se desiti da se prekorači veličina hipa kada dolazi do greške: prekoračenje hipa (*heap overflow*). Ako se npr. oslobođena memorija pokuša koristiti preko pokazivača doći će do potkoračenja hipa (*heap-underflow*).

Napomena: Objekat deklarisan sa extern je definisan u nekom drugom fajlu, tako da extern double dparam; ne odvaja memorijski prostor za taj objekat.

2.6. Konverzije tipova

Konverzija tipa nekog objekta u drugi tip može biti implicitna ili eksplicitna. Implicitnu konverziju tipa će obaviti prevodilac, ako je takva konverzija definisana, dok kod eksplicitne konverzije tipa programer navodi o kojoj konverziji tipa je reč.

Implicitne konverzije su ugrađene u jezik i radi se o standardnim konverzijama (*standard conversions*) koje se odnose na ugrađene tipove podataka, osnovne i izvedene klase.

Programer može definisati korisničke konverzije (*user-defined conversions*) iz jednog korisničkog tipa u drugi koristnički tip, ili iz ugrađenog tipa u korisnički tip, kao i iz korisničkog tipa u ugrađeni tip (npr. kompleksan broj u double) .

2.6.1. C stil konverzije tipova

C stil konverzije se navodi tako da se u malim zagradama stavi odredišni tip ispred objekta drugog tipa.

Neki primeri gde je potrebna konverzija, implicitna ili eksplicitna, su kao što sledi:

- switch((int)(double1/double2)) – switch ne prihvata tip double, tako da se koristi eksplicitna konverzija;
- double1 += int1 - operator sabiranja mora da svede tipove na "bogatiji" tip, tako da se korsiti implicitna konverzija;

- f((int)double1) – funkcija prihvata tip int, a poziva se sa argumentom tipa double, tako da je potrebna eksplicitna konverzija;
- return (int)double1; - funkcija vraća int, a navodi se da se vraća tip double , tako da je potrebna eksplicitna konverzija;
- int int1 = (int) double1; - int objekat se inicijalizuje objektom tipa double, tako da je potrebna eksplicitna konverzija.

2.6.2. C++ operatori promene tipa

U C++u se koriste sledeći operatori promene tipa (casting operators):

- `static_cast <novi_tip> (izraz)`
  koristi se za sve dobro definisane konverzije (nema provere validnosti konverzije u vreme izvršavanja):

```
double result = static_cast<double>(4)/5;
```

- `dynamic_cast <novi_tip> (izraz)`
  konvertuje podatak iz jednog tipa u drugi tip pri čemu se vrši provera validnosti konverzije u vreme izvršavanja (ako su navedeni tipovi nekompatibilni vraća 0),

```
class Base
{
  virtual void dummy(){}
};
class Derived: public Base{int i;};
Base * pb = new Derived;
Derived * pd= dynamic_cast<Derived*>(pb);
```

- `const_cast <novi_tip> (izraz)`
  koristi se za uklanjanje const ili volatile osobine promenljive (odredišni tip mora biti isti kao i izvorni tip):

```
const double PI = 3.14;
double* pdouble = const_cast<double*>(&PI);
```

- `reinterpret_cast <novi_tip> (izraz)`
  menja jedan tip u drugi (koristi se za promenu tipa između pokazivača koji ukazuju na nekompatibilne tipove) i koristite ga kada pouzdano znate da na primer `void*` zaista ukazuje na `novi_tip`

```
Zaba zaba;
Baba* pBaba = reinterpret_cast<Baba*>(&zaba);
```

Na slici 2.6.1. dato je zaglavlje klase IspisBroja. Na početku je navedena direktiva za uključivanje fajla jednom u program. Sledi komentar koji opisuje IEEE754 format float broja. Float broj zauzima 4 bajta pri čemu je niži bajt na nižoj memorijskoj lokaciji. Bitovi su redom: 1 bit za znak broja (s), 8 bita za

karakteristiku (karakteristika K je jednaka zbiru broja 127 i eksponenta E) i 23 bita za mantisu (m). Mantisa je oblika 1+m gde vodeća jedinica (skriveni bit) ne ulazi u prikaz. Kada bi vodeća znamenka normalizovanog broja uvek bila 1, ne bi bilo moguće prikazati broj 0. Koristi se sledeći dogovor: kada je K=0 i svi bitovi mantise postavljeni na 0, onda se radi o prikazu realnog broja +0, -0 . Ako je K=255 i svi bitovi mantise su postavljeni na 0, radi se o prikazu +∞ ili -∞. Za prikaz NaN (not a number) koristi se sledeće pravilo: K=255 i postoje bitovi mantise koji nisu 0. Kada je K=0 i postoje bitovi mantise koji nisu 0, radi se o denormalizovanom broju u kome se ne podrazumeva skriveni bit te se smatra da je vodeći bit mantise 0. Kod denormalizovanog broja vrednost eksponenta je fiksirana na -126 (ne koristi se izraz K=binarni eksponent+127). Očito da je 0 poseban oblik denormalizovanog broja. Klasa Ispis Broja ima privatni član fbroj tipa float, a u javnom bloku ima: konstruktor bez parametara, metodu PostaviFloat tipa void (float) te metodu IspisiBinarnoFloatClan tipa void (void).

```
//_________________fajl IspisBroja.h _____________________________
#pragma once
//IEEE 754: float zauzima 4 bajta (na nizoj lokaciji je nizi bajt)
//bitovi(od najznacajnijeg)  znacenje
//1                          (s) znak broja
//8                          (K) karakteristika K=E+127, E=K-127
//23                         (m) mantisa
//broj u zadatku se racuna kao: ((-1) na s)*(1.m)*(2 na E)
class IspisBroja
{
    float fbroj;
  public:
        IspisBroja();
    void PostaviFloat(float broj);
    void IspisiBinarnoFloatClan();
};
//Primer: -0.75
// s = 1 jer je broj negativan
// 0.75 binarno je 1/2 +1/4 odnosno 0.11
// kako je broj predstavljen kao 1.m, onda 1.m = 0.11*(2 na 1),
// te je korekcija za E suprotna tj. E=-1
// kako je E=K-127, K=126 tj. 01111110
// konacno 1 01111110 10000000000000000000000
```

Slika 2.6.1. Zaglavlje klase IspisBroja

Na slici 2.6.2. data je implementacija klase IspisBroja. Na početku je uključeno zaglavlje IspisBroja.h te zaglavlja iostream i iomanip, za ispis i formatiranje ispisa. Uključen je prostor imena std. Konstruktor IspisBroja postavlja član fbroj na 0. Metoda PostaviFloat postavlja član fbroj na vrednost argumenta. Metoda IspisiBinarnoFloatClan prvo uradi konverziju tipa reinterpret_cast kojom tip

adresa člana fbroj konvertuje u pokazivač na unsigned char, a onda tom vrednošću inicijalizuje pokazivač p koji sada ukazuje na niz bajtova koji počinju od adrese na kojoj je smešten član fbroj. Sledi ispis formatiranog teksta da bi se videlo šta će koji bit da predstavlja. Funkcija setw(10) postavlja tekstualni ispis na širinu od 10 karaktera. U dvostrukoj petlji dohvataju se bajtovi float broja počevši od bajta najveće težine (p[3]), a onda se testiraju biti dohvaćenog bajta bitwise operatorom &, počevši od bita najveće težine (1<<7) i ispisuje njihova tekstualna predstava (0 ili 1). Ako se dođe do mesta gde se u ispisu razdvajaju s, K i m onda se pravi razmak (" ").

```
//____________________fajl IspisBroja.cpp _____________________________
#include "IspisBroja.h"
#include <iostream>
#include <iomanip>
using namespace std;
IspisBroja::IspisBroja()
{
  fbroj = 0;
}
void IspisBroja::PostaviFloat(float broj)
{
  fbroj = broj;
}
void IspisBroja::IspisiBinarnoFloatClan()
{
   unsigned char *p = reinterpret_cast<unsigned char*>(&fbroj);
   cout<<"FLOAT BROJ -------- BINARNO PRIKAZAN --------"<<endl;
   cout<<"           s kkkkkkkk mmmmmmmmmmmmmmmmmmmmmmm"<<endl;
   cout<<"-----------------------------------------------"<<endl;
   cout<<setw(10)<<fbroj<<" ";
   for(int i=3; i>=0; i--)
      for(int j=7; j>=0; j--)
      {
         (p[i]&(1<<j))?cout<<"1":cout<<"0";
         if((i==3 && j==7)||(i==2 && j==7)) cout<<" ";
      }
   cout<<endl;
}
```

Slika 2.6.2. Implementacija klase IspisBroja

```
//_____________________fajl gde je main ______________________________
#include <iostream>
using namespace std;
#include "IspisBroja.h"
```

```
int main()
{
   IspisBroja ib;
   ib.PostaviFloat(-0.75f);
   ib.IspisiBinarnoFloatClan();
   return 0;
}

//_____________________________IZLAZ _____________________________
FLOAT BROJ -------- BINARNO PRIKAZAN --------
           s kkkkkkkk mmmmmmmmmmmmmmmmmmmmmmm
-----------------------------------------------
     -0.75 1 01111110 10000000000000000000000
```

Slika 2.6.3. Primena klase IspisBroja

Na slici 2.6.3. data je primena klase IspisBroja. Na početku fajla gde je funkcija main uključeno je zaglavlje iostream, navedeno je uključenje prostora imena std i uključeno je zaglavlje IspisBroja.h. U funkciji main kreiran je objekta ib tipa IspisBroja gde se u konstruktoru postavilo inicijalno stanje člana fbroj na 0. Pozivom metode PostaviFloat za objekat ib postavljeno je stanje člana fbroj na vrednost -0.75. Nakon ovoga pozvana je metoda objekta ib IspisiBinarnoFloatClan koja će ispisati sve bite svih bajtova float broja fbroj.

Na slici 2.6.4. dato je zaglavlje klase PovecajCharIliInt. Klasa ima trivijalno jednu javnu metodu Povecaj koja je tipa void (void*, int).

```
//_____________________fajl PovecajCharIliInt.h _________________
#pragma once
class PovecajCharIliInt{
  public:
    void Povecaj (void* pvoid, int velicina);
};
```

Slika 2.6.4. Zaglavlje klase PovecajCharIliInt

Na slici 2.6.5. data je implementacija klase PovecajCharIliInt. Uključena su zaglavlja PovecajCharIliInt.h, iostream i navedeno je korišćenje prostora imena std. Metoda Povecaj proverava veličinu objekta koji joj je prosleđen preko pokazivača na tip void, dakle, može da bude bilo šta. Pošto je ova metoda zadužena za povećanje char ili int objekta u smislu da broj poveća za 1 , a da karakter poveća na sledeći karakter u alfabetu, onda se proverava da li je prema veličini objekta u pitanju char ili int.

Ako je u pitanju objekat tipa char, onda se vrši promena tipa *void u *char, a onda se preko ovakvog pokazivača pristupa objektu tipa char i isti se inkrementira.

Ako je u pitanju objekat tipa int, onda se vrši promena tipa *void u *int, a onda se preko ovakvog pokazivača pristupa objektu tipa int i isti inkrementira. Data su dva načina promene tipa, C stil i C++ stil korišćenjem operatora static_cast.

```
//_____________________fajl PovecajCharIliInt.cpp _______________
#include "PovecajCharIliInt.h"
#include <iostream>
using namespace std;
void PovecajCharIliInt::Povecaj (void* pvoid, int velicina)
{
   if ( velicina == sizeof(char) )
   {
      char* pchar = (char*)pvoid;  //cast na C nacin
      ++(*pchar);                  //pristup i povecanje
      // cast na C++ nacin, pristup i povecanje
      // (*(static_cast<char*>(pvoid)))++;
   }
   else if (velicina == sizeof(int) )
   {
     int* pint = (int*)pvoid;      //cast na C nacin
     ++(*pint);                    //pristup i povecanje
    // cast na C++ nacin, pristup i povecanje
    // (*(static_cast<int*>(pvoid)))++;
   }
}
```

Slika 2.6.5. Implementacija klase PovecajCharIliInt

```
//_____________________fajl gde je main ____________________________
#include <iostream>
using namespace std;
#include "PovecajCharIliInt.h"
int main ()
{
   PovecajCharIliInt pov;
   char aa('C');
   int bb(1999);

   cout<<"Pre  : aa="<< aa <<"\t"<<"bb="<< bb <<endl;
   pov.Povecaj (&aa,sizeof(aa));
   pov.Povecaj (&bb,sizeof(bb));
   cout <<"Posle: aa="<< aa <<"\t"<<"bb="<< bb << '\n';
   return 0;
}
```

Slika 2.6.6. Primena klase PovecajCharIliInt

U fajlu gde je funkcija main uključena su zaglavlja iostream i PovecajCharIliInt.h te je navedeno korišćenje prostora imena std. Kreirana je jedna instanca pov klase PovecajCharIliInt, inicijalizovan je char aa vrednošću 'C' i inicijalizovan je int bb vrednošću 1999. Ispisane su vrednosti aa i bb pre poziva metoda Povecaj objekta pov. Izvršen je poziv metode Povecaj objekta pov gde su parametri ove metode: adresa objekta aa i njegova veličina. Isto je urađeno i za objekat bb. Nakon ovih poziva ispisane su vrednosti objekata aa i bb (biće D i 2000).

Rezime poglavlja programski jezik C++

U poglavlju programski jezik C++ obrađeni su: elementi koji nisu objektno orijentisani, tipovi podataka, oblasti važenja, životni vek objekata, memorijske oblasti i konverzija tipova.

Zadaci za proveru znanja

1. Napisati program koji koristi klasu Tacka3D koja ima metode za postavljanje i dohvatanje koordinata date tačke, merenja udaljenosti do druge tačke te ispis podataka o tački.
2. Napisati program koji dobija iz konzole podatke o 2 matrice veličine 3x3, a onda koristi klasu Matrica3x3 da pomnoži te dve matrice i da ispiše rezultat tog množenja.
3. Napisati program koji koristi klasu za postavljanje i prikaz double elementa. Prikazuje se binarni sadržaj memorijskih lokacija na kojima je zapisan double element.
4. Dopišite potreban kod da progam radi korektno.

```
#include <iostream>
#include "Complex.h"
using namespace std; const int HM = 3;
int main(){
  Complex c1(10.f,-2.f);  cout<<"c1 = "; c1.Ispisi();
  Complex c2(-2.f, 2.f);  cout<<"c2 = "; c2.Ispisi();
  Complex c3(0,0);
  c3=c1.Saberi(c2); c3.Ispisi(); cout<<endl;
  Complex vekt[HM];
  for(int i=0; i<HM; ++i) vekt[i].Ispisi();
  Complex *pcom = vekt;
  cout<<(*pcom).UdaljenostOdIshodista()<<endl;
  return 0;
}
// IZLAZ
// c1 = 10-2i
// c2 = -2+2i
// 8
// 10+10i
// 20+20i
// 30+30i
// 14.1421
```

3. Objektno orijentisani pristup

Cilj poglavlja je da student ovlada: klasama, objektima, pokazivačem this, konstruktorima (standardnim, konstruktorom kopije i move konstruktorom), destruktorom, inspektorima i mutatorima, statičkim članovima klase, pravima pristupa, prijateljima klase, preklapanjem operatora, nasleđivanjem (javnim, zaštićenim i privatnim), kreiranjem i ukidanjem objekata izvedene klase, višestrukim izvođenjem, virtuelnim klasama, polimorfizmom, apstraktnim klasama i virtuelnim destruktorom.

U tradicionalnom programiranju paradigma je da se razmišlja o koracima koji rešavaju problem. Objektno orijentisani pristup se odnosi na način razmišljanja koji zahteva da se pri projektovanju programa sagledaju i opišu svi problemi na način da se prikažu stanja problema i ponašanja problema, kao i njihova međusobna interakcija. Na ovakav, OOP način, više vremena se zahteva za razmišljanje (projektovanje), a manje za programiranje (kodovanje). Razmišljanje ide ka tome šta delovi sistema rade, a ne kako to rade. Smisao ovoga je da se kao prvo projektuje klasa koja će raditi ono što je potrebno, a da se onda u implementaciji korišćenjem algoritama implementira i kako će to da se izvede. Veoma je bitno da se komunikacija između klasa svede na razumnu (redukovanu) meru te da ta međusobna komunikacija bude strogo kontrolisana.

Ovakav pristup je rešio veliku softversku krizu koja se pojavila krajem sedamdesetih godina 20. veka kada su progami postajali sve veći i kada se postavio problem efikasnog održavanja ogromnog softvera.

Ključni koncepti OOP-a su: klase, objekti, apstrakcija tipova podataka, enkapsulacija, preklapanje operatora, nasleđivanje i polimorfizam.

Apstrakcija tipova podataka (*abstract data types*) podrazumeva da programer proizvoljno definiše svoje apstraktne tipove podataka (klase) koje opisuje njihovim stanjem i ponašanjem te da kreira proizvoljan broj instanci klase (*multiple instances*) i vrši pozive metoda te klase kojima je definisao ponašanje i operacije sa tim primercima klase. Primeri apstraktnih tipova podataka: klase Automobil, BankovniRacun, Student, ClanBiblioteke, MreznaKomunikacija i sl.

Enkapsulacija (*encapsulation*) odnosi se na sakrivanje detalja realizacije klase. Ono što korisnici klase koriste je deklaracioni fajl (ekstenzija .h) u kome se navodi samo šta klasa radi, ali ne i kako to radi. Kada je reč o interfejsu klase ideja je da se korišćenjem modifikatora pristupa odredi koji će članovi klase biti pristupačni samo unutar klase, ili unutar klase naslednice ili će biti nepristupačni ili pristupačni svima. Enkapsulacijom se štiti od grešaka, tako da će se greška tražiti u klasi objekta koji se ne ponaša kako je projektovano.

Preklapanje operatora (*operator overloading*) omogućuje da se za klasu definišu značenja operatora koji već postoje u jeziku. Na ovaj način moguće je npr. kreirati klasu KompleksanBroj i njene 3 instance C1, C2, C3, a onda definisati sabiranje (operator +) te pisati C3 = C1 + C2; Napomena: šta će se dešavati određuje programer definisanjem šta operator radi (npr. neka sabiranje dva objekta tipa Automobil menja cenu zbirnom objektu koja odgovara zbiru cena automobila sabiraka).

Nasleđivanje (*inheritance*) omogućuje da klasa naslednica preuzme osobine roditeljske klase, čime je podržana hijerarhijska klasifikacija. Npr. Zivotinja, Sisar, SlepiMis, drugim rečima slepi miš je sisar, a sisari su životinje. Pri nasleđivanju pravi se pravi nadskup svojstava i ponašanja npr. Avion i BorbeniAvion koji ima svojstva opšteg aviona, ali je i posebno to što je naoružan i što može da puca.

Polimorfizam (*polymorphism*) predstavlja pojavljivanje u više oblika u smislu da se kreira zajednički interfejs koji će omogućiti poziv više metoda u zavisnosti od toga o kom se objektu pozivaocu radi. Ako se metodi ili funkciji čiji je formalni argument referenca ili pokazivač na objekat osnovne klase prosledi referenca ili pokazivač na objekat izvedene klase onda se u telu te metode poziv metode osnovne klase može prevesti u poziv iste metode u izvedenoj klasi ako takva postoji i ako je uključen virtualni mehanizam koji omogućuje polimorfizam. Ovim je postignuto da se novi (izvedeni) tip odaziva na novi način iako je pozvan kao da je stari (osnovni) tip.

3.1. Klase i objekti

Klasa je osnovna organizaciona jedinica programa koja opisuje apstraktni (korisnički) tip. Pojmom klase se opisuje ponašanje objekta i njegovih stanja. Konkretan primerak date klase (instanca) zove se objekat te klase i za svaki kreirani objekat konstruiše se njegovo početno stanje. Objekat neke klase predstavlja promenljivu tog tipa. Stanja i ponašanja predstavljaju članove date klase (*class members*). Stanja objekta su opisana atributima (podacima članovima, *data members*), a ponašanja objekta su opisana metodama (funkcijama članicama, *member functions*). Atributi date klase predstavljaju svojstva, dok metode te klase predstavljaju akcije koje se mogu obavljati nad objektima te klase. Svaki objekat ima svoje atribute koji opisuju njegovo stanje. Ako su atributi objekta objekti drugih klasa onda je reč o ugrađivanju. Metodama se realizuje interfejs klase tako da se odredi šta se može raditi sa objektima te klase gledano iz same klase, iz klasa naslednica ili van klase i klasa naslednica. Objekti klasa međusobno komuniciraju čime mogu da se menjaju njihova stanja.

Posmatra se deklaracija klase Person koja se odnosi na osobu (vidi sliku 3.1). Kod je smešten u fajl Person.h. Prva linija koda #pragma once je direktiva kompajleru koja govori da će ovo zaglavlje samo jednom biti uključeno u program. Neka trivijalno svaka osoba ima atribute: name, surname i age (ime, prezime i starost osobe, respektivno). Neka ova klasa ima metode: SetName, SetSurname, SetAge i Info koje postavljaju vrednosti za ime, prezime, starost te ispisuju infomacije o osobi, respektivno. Zaglavlje metode opisuje kakav je poziv funkcije (lista formalnih argumenata) i kakvog je tipa rezultat metode. Ako metoda služi kao procedura, onda ona vraća tip void.

```
#pragma once

class Person {
  char* name;
  char* surname;
  int   age;
public:

  void SetName(char* formalniargumentime);
  void SetSurname(char* prezime);
  void SetAge(int); // godine
  void Info();
};
```

Slika 3.1. Deklaracija klase Person

Ključna reč class ispred naziva Person je oznaka za početak deklaracije klase Person. Deklaracija je iskaz za uvođenje novog imena u program. Deklaracija prevodiocu (kompajleru) daje informacije o tome šta predstavlja uvedeno ime u smislu šta se sa njim može raditi, čime je omogućena kontrola upotrebe uvedenog imena u samom programu.

Nakon naziva klase Person navedena je otvorena velika zagrada, atributi name tipa pokazivač na tip char, surname tipa pokazivač na tip char i age tipa int (signed int). Ovaj blok članova je podrazumevano privatni jer nije naveden modifikator pristupa članovima klase. Drugim rečima, van klase navedenim atributima ne može se pristupiti direktno već preko datih javnih metoda.

Sledi blok javnih članova (modifikator pristupa public:) koji čine metode SetName, SetSurname, SetAge i Info. Ovde su pomenute metode samo deklarisane u smislu da se zna za nazive i tipove metoda koje sadrži klasa Person, drugim rečima, šta to može da radi klasa Person. U metodi SetName naveden je naziv za formalni argument formalni argument ime tipa char*, kao "detaljniji" opis argumenta. U metodi SetSurname naveden je identifikator formalnog argumenta prezime tipa char*, dok u metodi SetAge nije naveden identifikator već samo tip argumenta (int), što je dozvoljeno jer deklaracija metode govori samo o nazivu metode i listi tipova njenih formalnih argumenata. Na kraju deklaracije klase Person je velika zatvorena zagrada i znak tačka-zarez.

Potrebno je definisati kako navedene metode rade, što se radi van deklaracije u fajlu `Person.cpp` (slika 3.2):

```
#include "Person.h"
#include <iostream>
void Person::SetName(char* ime)
{
  name = ime;
}
void Person::SetSurname(char* prezime)
{
  surname = prezime;
}
void Person::SetAge(int godine)
{
  age = godine;
}
void Person::Info()
{
  std::cout<<name<<" "<<surname<<","<<age<<" godina"<<std::endl;
}
```

Slika 3.2. Definicija klase Person

U prethodnim definicijama prefiks `Person::` ispred naziva metode znači da metoda pripada klasi `Person`. Pri definisanju metode, u implementacionom fajlu, na kraju tela metode nema dodatnog znaka tačka-zarez (`;`) kao što je u deklaraciji klase. Metode `SetName`, `SetSurname` i `SetAge` vrše postavljanje stanja objekta koji ih je pozvao u smislu da modifikuju ime, prezime i godine osobe, respektivno. Metoda `Info` ispisuje vrednosti atributa (stanje objekta): `name`, `surname` i `age`. Sve nabrojane metode su procedure jer vraćaju void. Pre kraja tela procedure ne mora se navesti naredba `return;` (može i da se navede).

U fajlu gde je funkcija main (slika 3.3), linijom koda `#include "Person.h"` uključeno je zaglavlje klase `Person` da bi kompajler imao podatke o imenima koja će biti korišćena u ovom fajlu. Linijom koda `Person osoba;` kreiran je objekat tipa `Person` pozivom odgovarajućeg konstrukora klase. U deklaraciji klase (slika 3.1) nije naveden niti jedan konstruktor te će u tom slučaju kompajer kreirati podrazumevani konstruktor bez parametara.

Napomena:
Ako se kreira samo jedan konstruktor, onda kompajler neće kreirati podrazumevani konstruktor.

```
#include "Person.h"
int main()
```

```
{
  Person osoba;
  osoba.SetName("Nikola");
  osoba.SetSurname("Tesla");
  osoba.SetAge(87);
  osoba.Info();
  return 0;
}
```

Slika 3.3. Korišćenje klase Person

Nakon kreiranja objekta `osoba` slede pozivi metoda `SetName`, `SetSurname` i `SetAge` kojima se postavlja stanje objekta (`Nikola`, `Tesla`, `87`). Sintaksa poziva metode je da se nakon navođenja identifikatora objekta stavi znak tačka (`.`) iza koga sledi naziv metode i lista stvarnih argumenata. Na kraju se poziva metoda `Info` objekta `osoba` koja ispisuje vrednosti atributa objekta `osoba`, a onda sledi naredba `return 0;` za izlazak iz programa.

Potrebno je razlikovati definiciju i deklaraciju klase. Klasa je definisana ako su deklarisani svi članovi klase. Deklaracijom klase u program se uvodi novi naziv korisničkog tipa (class identifikator_klase{ //deklaracija clanova klase};).

Sintaksa čiste deklaracije klase je kao što sledi:

```
class identifikator_klase;
```

Čista deklaracija klase se koristi kada se dve ili više klasa uzajamno referišu. Primer:

```
// fajl Grupa.h
class Clan;
class Grupa
{
  Clan *pc;    // OK
  //...
};

// fajl Clan.h
class Grupa;
class Clan
{
  Grupa *pg;
  //...
};
```

U klasi se može deklarisati pokazivač na objekat iste klase, ali nije dozvoljeno da se deklariše objekat iste klase, jer je za deklaraciju objekta potrebna cela definicija klase.
Primer:

```
class Automobil
{
  Automobil *pa; // OK
  Automobil a;   // NOK
  //...
};
```

Deklaracija klase može biti istovremeno i deklaracija objekata tipa te klase, ili tipova izvedenih iz klase.
Primer:

```
class Automobil
{
  Automobil *pa; // OK
  //...
}
a,   // objekat tipa Automobil
*pa; // pokazivac na objekat tipa Automobil
```

Obratiti pažnju da je naziv klase naziv korisničkog tipa te da se dva korisnička tipa potpuno iste reprezentacije razlikuju pošto se radi o "različitim tipovima", odnosno, neekvivalenciji naziva.
Primer:

```
class A
{
  public:
    int a;
};
class B
{
  public:
    int a;
};
A a;
B b;
int z;     //objekti a, b i z pripadaju različitim tipovima
int f(A);  // funkcija tipa int(A)
int f(B);  // funkcija tipa int(B)
```

Napomena: Obratiti pažnju da uvođenje drugog naziva za neki naziv zadržava ekvivalenciju naziva.

Primer:

```
class A
{
  public:
    int a;
};
typedef A B;  // B je drugo ime za A
```

```
A a;
B b;
a = b;  // OK
```

Napomena: Uvođenjem drugog naziva može se elegantno iskoristiti eksplicitna konverzija.
Primer:

```
typedef int* PI;
PI p4(PI(1));
```

Na slici 3.4. dato je zaglavlje klase KompleksanBroj. Nakon direktive da se fajl jednom uključi u program sledi deklaracija klase KompleksanBroj gde su u privatnom bloku navedeni double članovi za realni i imaginarni deo kompleksnog broja. U javnom bloku konstruktor prihvata dva argumenta tipa double za realni i imaginarni deo kompleksnog broja. Slede metode tipa KompleksanBroj (KompleksanBroj) za: sabiranje (SaberiSa), oduzimanje (OduzmiSa), množenje (PomnoziSa) i deljenje (PodeliSa) dva kompleksna broja, respektivno. Na kraju je data metoda IspisiKompleksanBroj tipa void (void).

```
//_____________________fajl KompleksanBroj.h _________________
//klasa KompleksanBroj bi trebalo da omoguci sabiranje, oduzimanje
//mnozenje i deljenje kompleksnih brojeva kao i ispis kompleksnog
//broja u formi npr. 0, 10, 10+2i, 10-2i, +2i, -2i
#pragma once
class KompleksanBroj
{
   private:
      double realni_deo;
      double imaginarni_deo;
   public:
      KompleksanBroj(double formalni_argument_za_realni_deo,
                     double formalni_argument_za_imaginarni_deo);
      KompleksanBroj SaberiSa (KompleksanBroj drugi_operand);
      KompleksanBroj OduzmiSa (KompleksanBroj drugi_operand);
      KompleksanBroj PomnoziSa(KompleksanBroj drugi_operand);
      KompleksanBroj PodeliSa (KompleksanBroj drugi_operand);
      void IspisiKompleksanBroj();
};
```

Slika 3.4. Zaglavlje klase KompleksanBroj

Na slici 3.5. data je implementacija klase KompleksanBroj. Uključena su zaglavlja: iostream i iomanip za ispis i format ispisa te zaglavlje KompleksanBroj.h. Koristi se std prostor imena i definisan je makro USLOVNO_PREVEDI_EXIT. Konstruktor KompleksanBroj koji prihvata dva argumenta tipa double kreira instancu kompleksnog broja i postavlja početno

stanje atributa realni_deo i imaginarni_deo. Metode SaberiSa i OduzmiSa vrše sabiranje, odnosno, oduzimanje, dva kompleksan broja. Povratna vrednost je privremeni kompleksan broj koji se kreira konstruktorom kome se prosleđuju zbirovi, odnosno, razlike realnih i imaginarnih delova dva kompleksna broja. Ako su dva kompleksna broja A+Bi te C+Di, onda metoda za množenje vraća kompleksan broj čiji je realni deo jednak AC-BD, a imaginarni deo jednak AD+BC. Kod deljenja dva kompleksna broja (A+Bi , C+Di) izvrši se "realisanje" tako da se (A+Bi)/(C+Di) pomnoži sa jedan (C-Di)/(C-Di). Za delioca se dobija C^2+D^2 i mora se ispitati da li je on jednak nuli.

Ako je delilac jednak nuli i definisan je makro `USLOVNO_PREVEDI_EXIT`, onda će se pozvati funkcija exit sa parametrom EXIT_FAILURE (errorlevel je 1). Funkcija exit će pozvati lanac funkcija tipa void (void) koji se postavlja funkcijom atexit. Pošto je u main funkciji funkcijom atexit(PreKraja) postavljena funckija PreKraja, ista će biti pozvana, nakon čega se izlazi iz programa.

Ako je delilac jednak nuli i nije definisan makro `USLOVNO_PREVEDI_EXIT`, onda će se ispisati "Pritisnite taster i sledi poziv funkcije abort()" te će nakon pritiska tastera biti pozvana funkcija abort koja nasilno prekida program i program završava rad (errorlevel je 3).

Ako je delilac različit od nule, onda je količnik dva kompleksna broja A+Bi , C+Di jednak (AC+BD)/(C^2+D^2) za realni deo i (BC-AD) /(C^2+D^2) za imaginarni deo.

Metoda IspisiKompleksanBroj ispisuje prvo relani deo (ako je pozitivan ne ispisuje znak +, noshowpos), a onda, ako je imaginarni deo različit od nule ispisuje tu vrednost sa znakom te ako je vrednost pozitivna, onda ispisuje znak + (showpos).

```
//_____________________fajl KompleksanBroj.cpp _____________________
#include <iostream>
#include <iomanip>
using namespace std;
#include "KompleksanBroj.h"
#define USLOVNO_PREVEDI_EXIT
KompleksanBroj::KompleksanBroj(
                     double formalni_argument_za_realni_deo,
                     double formalni_argument_za_imaginarni_deo)
{
    realni_deo     = formalni_argument_za_realni_deo;
    imaginarni_deo = formalni_argument_za_imaginarni_deo;
}
KompleksanBroj KompleksanBroj::SaberiSa(
                                        KompleksanBroj drugi_operand)
```

```
{
 return KompleksanBroj(realni_deo    + drugi_operand.realni_deo,
                  imaginarni_deo + drugi_operand.imaginarni_deo) ;
}
KompleksanBroj KompleksanBroj::OduzmiSa(
                                     KompleksanBroj drugi_operand)
{
  return KompleksanBroj(realni_deo     - drugi_operand.realni_deo,
                  imaginarni_deo - drugi_operand.imaginarni_deo) ;
}
KompleksanBroj KompleksanBroj::PomnoziSa(
                                     KompleksanBroj drugi_operand)
{
   return KompleksanBroj(
              (realni_deo     * drugi_operand.realni_deo) -
              (imaginarni_deo * drugi_operand.imaginarni_deo) ,
              (realni_deo     * drugi_operand.imaginarni_deo) +
              (imaginarni_deo * drugi_operand.realni_deo) ) ;
}
KompleksanBroj KompleksanBroj::PodeliSa(
                                     KompleksanBroj drugi_operand)
{
   double delilac =
    (drugi_operand.realni_deo     * drugi_operand.realni_deo) +
    (drugi_operand.imaginarni_deo * drugi_operand.imaginarni_deo);
   if(0==delilac) {
#ifdef USLOVNO_PREVEDI_EXIT
      exit(EXIT_FAILURE);//pozvala bi se funkcija PreKraja()
                         //errorlevel = EXIT_FAILURE = 1
#else
   cout<<"Pritisnite taster i sledi poziv funkcije abort()"<<endl;
   system("pause");
   abort();  //nasilni zavrsetak programa, errorlevel=3
#endif
   }
   return KompleksanBroj(
    ( (realni_deo     * drugi_operand.realni_deo) +
      (imaginarni_deo * drugi_operand.imaginarni_deo))/ delilac,
    ( (imaginarni_deo * drugi_operand.realni_deo) -
      (realni_deo     * drugi_operand.imaginarni_deo))/ delilac );
}
void KompleksanBroj::IspisiKompleksanBroj(){
   cout<<noshowpos<<realni_deo;
   if(imaginarni_deo != 0)
   {
     cout<<showpos<<imaginarni_deo<<"i";
   }
```

```
    cout<<endl;
}
```

Slika 3.5. Implementacija klase KompleksanBroj

```
//_____________________fajl gde Vam je main _____________________
#include <iostream>
using namespace std;
#include "KompleksanBroj.h"
void PreKraja()
{
    cout<<"Pozvana je funkcija PreKraja()"<<endl;
    system("pause");
}
int main(){
    atexit(PreKraja);
    KompleksanBroj z1(10.0,-2.);
    cout<<"z1=";
    z1.IspisiKompleksanBroj();
    KompleksanBroj z2(0.0,-2.0);
    cout<<"z2=";
    z2.IspisiKompleksanBroj();
    KompleksanBroj z3(0,0);
    cout<<"z3=";
    z3.IspisiKompleksanBroj();
    z3 = z1.SaberiSa (z2);
    cout<<"z3=z1+z2= ";     z3.IspisiKompleksanBroj();
    z3 = z1.OduzmiSa (z2);
    cout<<"z3=z1-z2= ";     z3.IspisiKompleksanBroj();
    z3 = z1.PomnoziSa(z2);
    cout<<"z3=z1*z2= ";     z3.IspisiKompleksanBroj();
    z3 = z1.PodeliSa (z2);
    cout<<"z3=z1/z2= ";     z3.IspisiKompleksanBroj();
    return 0;
}
//_____________________________IZLAZ:
z1=10-2i
z2=-2i
z3=0
z3=z1+z2= 10-4i
z3=z1-z2= 10
z3=z1*z2= -4-20i
z3=z1/z2= 1+5i
Pozvana je funkcija PreKraja()
Press any key to continue . . .
```

Slika 3.6. Korišćenje klase KompleksanBroj

Na slici 3.6. dat je primer korišćenja klase KompleksanBroj. U fajlu gde je funkcija main, uključena su zaglavlja iostream i KompleksanBroj.h te je navedeno korišćenje prostora imena std. Definisana je funkcija PreKraja tipa void (void) koja ispisuje da je pozvana i pravi pauzu.

U funkciji main naveden je poziv funkcija atexit(PreKraja) koja postavlja u lanac poziva funkciju PreKraja u slučaju da bude pozvana funkcija exit. Kreirani su i ispisani kompleksni brojevi z1, z2 i z3. Nakon toga pozvane su metode objekta z1 za sabiranje, oduzimanje,množenje i deljenje kompleksnog broja gde je kao drugi operand prosleđen kompleksan broj z2. Respektivno se rezultati tih operacija dodeljuju kompleksnom broju z3, a onda ispisuje njegova vrednost. Pošto je kod operacije deljenja za date vrednosti kompleksnog broja z2 delilac jednak nuli i definisan je makro za uslovno prevođenje `USLOVNO_PREVEDI_EXIT`, onda se poziva funkcija exit, a ona funkciju PreKraja koja pravi pauzu.

3.1.1. Unutrašnje klase

Ugnežđena klasa je klasa koja je deklarisana unutar deklaracije druge klase. Ovakva klasa ne predstavlja ugrađivanje i lokalna je za okružujuću klasu. Ako se želi pristup, onda se navodi operator za razrešavanje pristupa (::). Ugnežđena klasa može iz okružujuće klase da koristi imena tipova, statičkih članova i nabrajanja.
Primer:

```
int i;
class Okruzujuca {
  public:
    int i;
    class Ugnezdjena {
      public:
        void fu()
        {
          i++;      // NOK, atribut okružujuće klase
          ::i++;    // pristup globalnom objektu
        }
    };
};
int main()
{
  Okruzujuca::Ugnezdjena u;
  return 0;
}
```

Klasa može biti definisana u definiciji funkcije i kao takva je lokalna u bloku u kome je definisana i može koristiti iz okružujućeg opsega imena tipova, statičke promenljive, extern promenljive, funkcije i nabrajanja. Ovakva klasa ne može imati statičke članove.

```
void func()
{
  class LK    // definicija lokalne klase
  {
    ...
  };
  ...
}
```

3.1.2. Pokazivač this

Pokazivač `this` ukazuje na objekat čija je metoda pozvana. Sada je u metodi objekta moguće, pored direktnog navođenja člana pripadne klase, pristupiti članovima pripadne klase i korišćenjem pokazivača `this` i operatora `->` iza koga sledi identifikator člana pripadne klase. Suštinski, navođenjem člana klase implicitno se pristupa preko `this->`. `This` je konstantni pokazivač na objekat čija je metoda pozvana i predstavlja ugrađeni lokalni objekat metode. Ovaj pokazivač ukazuje na strukturu podataka koja se odnosi na članove klase objekta čija je metoda pozvana. Svaki objekat ima svoju strukturu podataka, pri čemu je metoda realizovana kao jedna kopija koda metode. `This` predstavlja implicitni argument metode objekta.
Primer:

```
KompleksanBroj KompleksanBroj::Saberi(KompleksanBroj drugioperand)
{
  return KompleksanBroj(this->real + drugioperand.real,
                        this->imag + drugioperand.imag);
}
```

U primeru je izvršeno sabiranje kompleksnog broja (čiji su atributi `real` za realni deo i `imag` za imaginarni deo) čija metoda `Saberi` je pozvana i kompleksnog broja `drugioperand` koji predstavlja formalni argument metode `Saberi`. Pozivom konstruktora `KompleksanBroj` kome su prosleđeni argumenti za realni i imaginarni deo kompleksnog broja koji odgovaraju zbiru realnih, odnosno, imaginarnih komponenti kompleksnog broja čija je metoda `Saberi` pozvana i kompleksnog broja `drugioperand`. Elementi `real` i `imag` kompleksnog broja čija je metoda `Saberi` pozvana su pozvani korišćenjem `this->`.

Ako metoda objekta vraća pripadni objekat korišćenjem pokazivača this onda je sintaksa kao što sledi:

```
Automobil Automobil::VratiAutomobil()
{
  return *this;
}
```

Ako metoda objekta vraća referencu na pripadni objekat korišćenjem pokazivača this onda je sintaksa kao što sledi:

```
Automobil& Automobil::VratiAutomobil()
{
  return *this;
}
```

Ako metoda objekta vraća pokazivač na pripadni objekat korišćenjem pokazivača this onda je sintaksa kao što sledi:

```
Automobil* Automobil::VratiAutomobil()
{
  return this;
}
```

3.2. Konstruktor

Konstruktor je metoda koja kreira objekat u "sirovoj" memoriji rezervisanoj za objekat, vrši inicijalizaciju atributa objekta postavljajući tako početno stanje objekta i postavlja informacije o nasleđivanju, polimorfizmu i o primanju poruka. Konstruktor klase ima isti naziv kao i naziv te klase i ne vraća tip (ni void). Poziva se pri kreiranju objekta i može biti preklopljen (*overloaded*). Pri razrešavanju poziva pozvaće se onaj konstruktor koji najbolje odgovara pozivu prema tipovima svojih formalnih argumenata. Kao i metoda, konstruktor može biti javni (dostupan svima), zaštićen (dostupan izvedenim klasama) i privatni (dostupan npr. preko javne statičke metode).

Konstruktor se poziva kao što sledi:

- pri definiciji globalnog objekta;
- pri definiciji lokalnog objekta;
- pri kreiranju objekta koji ima ugrađivanje (pozivaju se konstruktori atributa) pri čemu se u inicijalizatoru konstruktora objekta ništa ne navodi ako atribut nema deklarisan konstruktor, ili atribut ima podrazumevani konstruktor, odnosno, ako atribut ima konstruktor sa argumentima, onda se u inicijalizatoru konstruktora objekta navodi eksplicitna inicijalizacija tog atributa;
- pri kreiranju dinamičkog objekta;
- pri kreiranju privremenog objekta (argument metode (ili funkcije) ili povratne vrednosti metode (ili funkcije));
- pri kreiranju objekta izvedene klase gde se poziva konstruktor osnovne klase;
- pri eksplicitnom pozivu konstruktora (ne prethodi mu identifikator objekta jer će njega tek stvoriti konstruktor).

Podrazumevani konstruktor (*default constructor*) je onaj koji se može pozvati bez argumenata (ili nema argumente ili su svi podrazumevani). Ako nijedan konstruktor nije deklarisan za datu klasu, onda prevodilac implicitno generiše javni podrazumevani konstruktor, pri čemu će pozvati podrazumevane

konstruktore članova objekta. Konstruktor koji se može pozvati bez argumenata potreban je za implicitno kreiranje objekata niza objekata, što se odvija po rastućem indeksu elemenata niza. Pošto konstruktor kreira objekat on se može pozvati i za const i za volatile objekte. Konstruktor ne može biti static niti const niti volatile niti virtual. Konstruktor se **NE** nasleđuje niti je moguće uzeti adresu konstruktora. Konstruktor poseduje pokazivač this.

Privatni konstruktor koristi se za kreiranje singleton klase, koju karakteriše da ima samo jednu instancu. Na slici 3.2.1. dat je jednostavan kod koji opisuje uzorak singleton. Ono što korisnik klase vidi je javna statička metoda GetInstance koja je na nivou klase i pripada klasi kao i svim objektima klase. Pozivom Singleton::GetInstance() dobija se statički pokazivač (m_instance) na objekat klase Singleton ako je isti već kreiran, odnosno, ako nije kreiran, onda će se kreirati nov dinamički objekat klase Singleton te vratiti pozivaocu statički pokazivač na takav objekat (m_instance). Konstruktor klase Singleton samo ispisuje da je isti pozvan. Inicijalna vrednost statičkog pokazivača m_instance je nullptr (ne ukazije ni na kakav objekat). U funkciji main se kreiraju dva pokazivača na objekta klase Singleton (psingleton i psingleton2) pozivom statičke metode Singleton::GetInstance(), a onda se ispisuju vrednosti ova dva pokazivača. Ako su ove vrednosti jednake onda se radi o singletonu.

```
#include <iostream>
using namespace std;
class Singleton {
  public:
    static Singleton* GetInstance() {
     return (m_instance)? m_instance : m_instance=new Singleton;
    }
  private:
    // privatni konstruktor
    Singleton(){ cout << "Kreirana instanca Singleton klase!\n"; }
    ~Singleton() {} // privatni destruktor
    static Singleton *m_instance;           // instanca Singleton
};
Singleton* Singleton::m_instance = nullptr; // inicijalizacija
                                            // statičnog člana

int main(int argc, const char * argv[]) {
    Singleton *psingleton = Singleton::GetInstance();
    cout << psingleton << endl;
    Singleton *psingleton2 = Singleton::GetInstance();
    cout << psingleton2 << endl;
    return 0;
}
// ako je singleton, oba puta će se ispisati ista adresa
```

Slika 3.2.1. Primer singletona

Konstruktor može biti inline, ili definisanjem u fajlu zaglavlja ili eksplicitnim navođenjem inline pri definiciji konstruktora u implementacionom fajlu. Na slici 3.2.2. dat je primer klase KompleksanBroj koja ima konstruktor sa podrazumevanim argumentima 0 za realni i za imaginarni deo kompleksnog broja i koji se realizuje kao inline. Ovakav konstruktor se može pozvati i bez argumenata i sa dva argumenta za realni i imaginarni deo kompleksnog broja. Poziv konstruktora bez argumenata će postaviti realni i imaginarni deo kompleksnog broja na 0. Ako poziv konstruktora ima dva argumenta onda se realni i imaginarni deo kompleksnog broja inicijalizuju redom sa te dve navedene vrednosti. U fajlu zaglavlja navedena je direktiva da se samo jednom može procesuirati klasa KompleksanBroj.h (#pragma once), sledi opis klase KompleksanBroj sa dva privatna atributa za smeštanje imaginarne i realne vrednosti kompleksnog broja, a onda ide javni blok članova: konstruktor, metoda Oduzmi tipa KompleksanBroj (KompleksanBroj) i metoda Ispisi tipa void (). U fajlu implemenatacije pre definicije konstruktora navedena je reč inline. Metoda Oduzmi prihvata objekat tipa KompleksanBroj i vraća objekat tipa KompleksanBroj koji predstavlja razliku objekta čija je metoda pozvana i objekta kb koji je prosleđen ovoj metodi. Oduzimanjem dva kompleksna broja dobija se kompleksan broj kod koga je vrednost realnog dela jednaka razlici realnih vrednosti oba sabirka i gde je vrednost imaginarnog dela jednaka razlici imaginarnih vrednosti oba sabirka. Metoda Ispisi ispisuje kompleksan broj kao uređen par realnog i imaginarnog dela tog kompleksnog broja.
Primer:

```
// fajl KompleksanBroj.h ------------------------------
#pragma once
class KompleksanBroj {
    double realni,imaginarni;
  public:
                KompleksanBroj(double=0, double=0);
  KompleksanBroj Oduzmi(KompleksanBroj);
           void Ispisi();
};

// fajl KompleksanBroj.cpp ----------------------------
#include<iostream>
using namespace std;
#include "KompleksanBroj.h"
inline KompleksanBroj::KompleksanBroj (double r, double i)
{
  realni=r;
  imaginarni=i;
}
KompleksanBroj KompleksanBroj::Oduzmi(KompleksanBroj kb)
{
```

```
  return KompleksanBroj(this->realni     - kb.realni,
                        this->imaginarni - kb.imaginarni);
}
void KompleksanBroj::Ispisi()
{
  cout<<"("<<realni<<","<<imaginarni<<")"<<endl;
}
// fajl gde je main ------------------------------------
#include <KompleksanBroj.h>
int main()
{
  KompleksanBroj kb1;
  KompleksanBroj kb2(10,20);
  KompleksanBroj razlika = kb1.Oduzmi(kb2);
  razlika.Ispisi();
  return 0;
}
```

Slika 3.2.2. Primer inline konstruktora sa podrazumevanim argumentima

U fajlu gde je funkcija `main`, učitano je zaglavlje `KompleksanBroj.h`, a onda u funkciji `main` su kreirani kompleksan broj `kb1` pozivom konstruktora bez argumenata `(0,0)` te je kreiran kompleksan broj `kb2` pozivom konstruktora sa dva argumenta `(10,20)`. Kompleksan broj `razlika` odgovara razlici kompleksnih brojeva `kb1` i `kb2`. Pozvana je metoda `Oduzmi` objekta `kb1` kojoj je prosleđen argument `kb2`. Metoda `Oduzmi` vraća kompleksan broj koji je kreiran konstruktorom `KompleksanBroj` čija su dva argumenta dve razlike: razlika realnih i razlika imaginarnih delova objekta čija je metoda pozvana i objekta koji je prosleđen kao parametar metode `Oduzmi`. Nadalje, u `main` funkciji se za objekat `razlika` poziva metoda `Ispisi` koja ispisuje uređeni par koji predstavlja razliku kompleksnih brojeva `kb1` i `kb2`.

Konstruktor dozvoljava implicitnu konverziju parametra koji mu je prosleđen u objekat klase koji kreira konstruktor. Ako konstruktor može da se pozove sa jednim argumentom datog tipa, onda kreiranje objekta na osnovu argumenta datog tipa predstavlja korisnički definisanu konverziju (*user-defined conversion*) iz datog tipa u tip klase tog objekta.

Napomena: Dozvoljena je samo neposredna korisnički definisana konverzija.

```
class C { public: C(double);};
class K { public: K(C);};
C cc = 3.14;            // neposredna konverzija
K kk = 3.14;            // NOK, posredna konverzija K(C(3.14))
```

Primer:

```
#include <iostream>
```

```
using namespace std;
class Trivijalna {
  int cc;
public:
  Trivijalna() { cc = 3; }
  Trivijalna(int c) { cc= c; }
  int Getcc() { return cc; }
};
const int iDuzina=3;

int main(){
  Trivijalna triv[iDuzina] = { 1, Trivijalna(2) };
  for(int i=0; i < iDuzina; i++)
    cout<<"triv["<<i<< "].Getcc(): "<<triv[i].Getcc()<<"\n";
  system("pause");
  return 0;
}
```

Slika 3.2.3. Konverzija tipa preko konstruktora

Na slici 3.2.3. dat je kod za klasu Trivijalna koja ima: privatni član `cc` tipa `int`, javni konstruktor bez argumenata koji inicijalizuje član `cc` na `3`, javni konstruktor koji prihvata jedan argument tipa `int` i koji tim argumentom inicijalizuje član `cc` te metodu `Getcc` tipa `int()` koja vraća vrednost člana `cc`. Sve metode su inline (definisane u zaglavlju). Konstanta tipa `int iDuzina` poslužiće za postavljanje veličine niza triv na 3 člana tipa `Trivijalna` i poslužiće za `for` petlju u kojoj se ispisuju vrednosti člana `cc` svakog objekta niza. Objekti niza `triv` se kreiraju sledećim redom:

- `triv[0]` se kreira implicitnim pozivom konstruktora koji prihvata tip int i kome se prosleđuje vrednost 1;
- `triv[1]` se kreira eksplicitnim pozivom konstruktora koji prihvata tip int i kome se prosleđuje vrednost 2;
- `triv[2]` se kreira implicitnom pozivom konstruktora bez argumenata, gde će vrednost člana cc biti 3.

Isti efekat bi se postigao ako bi se umesto navedena dva konstruktora stavio jedan konstruktor kao što sledi:

```
Trivijalna(int c=3) { cc= c; }
```

Nadalje, `for` petlja redom ispisuje član `cc` svakog objekta niza. U `main` bi se mogle dodati i sledeće linije koda koje takođe pozivaju eksplicitno i implicitno konstruktor:

```
Trivijalna t1 = 10; // implicitni poziv
t1 = 100;           // implicitni poziv
Trivijalna t2 = Trivijalna(20); // eksplicitni poziv koji
                                // kreira privremeni,
```

```
                                    // bezimeni objekat
                                    // koji se kopira u t2

  Trivijalna t3(30);  // eksplicitni poziv
  Trivijalna t4;      // kao eksplicitni poziv
```

Ako se navede reč `explicit` ispred konstruktora, onda je dozvoljen samo eksplicitan poziv konstruktora.

```
  class Trivijalna {
      int cc;
    public:
      explicit Trivijalna(int c=3){cc=c;};
     int Getcc() { return cc; }
  };

  int main()  {
    Trivijalna t1 = 10; // NOK, nije eksplicitan poziv
    t1 = 100;           // NOK, nije eksplicitan poziv
    Trivijalna t2 = Trivijalna(20); // eksplicitan poziv
    Trivijalna t3(30);              // eksplicitan poziv
    Trivijalna t4;      // kao eksplicitan poziv
    return 0;
  }
```

Inicijalizator konstruktora omogućuje rešavanje problema kao što sledi:

- inicijalizacija konstantnih članova objekta
- inicijalizacija članova objekta koji su reference
- inicijalizacija ugrađenih članova (članovi druge klase)

Dodelom ne bismo mogli da uradimo prethodne 3 stvari.
Primer :

```
  class A {
      int a;
    public: A(int i) {a=i;}
  };
  class B {
      const int consti; //konstanta
      int &refi;        //referenca
      A aa;             //ugrađivanje
    public:
      B(int&)
      {
        consti=i;  // dodela nije dozvoljena
        refi=i;    // mora na mestu deklaracije
        //aa nije ni kreiran
      }
  };
```

Inicijalizacija članova objekta se izvodi u zaglavlju konstruktora navođenjem znaka dvotačka (:) i navođenjem identifikatora člana a onda u zagradi inicijalizacione vrednosti za taj član koja je najčešće izraz u kome učestvuju formalni argumenti konstruktora, pri čemu je delimiter za inicijalizaciju članova zarez. Bez obzira na redosled navođenja inicijalizatora inicijalizacija će ići redom deklaracije članova klase. Nakon inicijalizacije izvršava se telo konstruktora. Napomena: Inicijalizaciju razlikovati od pojma dodele.
Primer za prethodni slučaj:

```
B(int &i)
:
consti(i),  // inicijalizacija konstante
refi(i),    // inicijalizacija reference
aa(i)       // kreiranje ugrađenog objekta
{}
```

Napomena: Inicijalizacijom se ne mogu eksplicitno inicijalizovati članovi nizovi. Ono što nije dozvoljeno je da se unutar tela jednog konstruktora date klase zove drugi konstruktor te iste klase u smislu da "unutrašnji" konstruktor obavi neki posao, a da se onda u telu "glavnog" konstruktora doda još potrebnih dodatnih linija koda. Problem je u tome što bi poziv konstruktora unutar konstruktora napravio poseban objekat koji nema veze sa objektom koji hoćemo da kreiramo. Ako je potrebno da se u konstruktorima obavi neki zajednički posao, onda se kreira zaštićena ili privatna metoda čiji je naziv često Init ili Initialize. Sada se ova metoda zove u telima konstruktora tamo gde je potrebno da se obavi dati posao.
Primer:

```
class XWing {
  //...
  void Init();   // metoda koja sastavi letelicu
public:
  XWing();       // konstruktor koji kreira letelicu
  XWing(Wheapon);// konstruktor koji kreira letelicu i
                 // dodaje naoružanje
  void Put(Wheapon);// postavlja naoružanje
  //...
};
XWing::XWing() {
  Init();        // sastavi letelicu
}
XWing::XWing(Wheapon w) {
  Init();        // sastavi letelicu
  Put(w);        // postavi naoružanje
}
```

Dakle, nepravilno bi bilo kao što sledi:

```
XWing::XWing(Wheapon w) {
  XWing();       // potpuno drugi objekat koji bi za sebe
                 // zvao svoj Init
  Put(w);        // postavi naoružanje
}
```

Primer poziva konstruktora kod ugrađivanja gde klasa ugrađenog objekta nema deklarisan konstruktor ili ima podrazumevani konstruktor:

```
class A
{
  public:
    int ai;
    A():ai(0){}  // može i bez ovoga
};
class B
{
    A aa;
  public:
    int bi;
    B(int i)    // pozvace se implicitno i A() za aa
    {
      bi=i;
    }
};
```

Primer poziva konstruktora kod ugrađivanja gde klasa ugrađenog objekta ima konstruktor sa argumentom:

```
class A
{
  public:
    int ai;
    A(int i):ai(0){}  // konstruktor sa argumentom
};
class B
{
    A aa;
  public:
    int bi;
    B(int i):aa(i)    // poziv A(i)
    {
      bi=i;
    }
};
```

3.2.1. Konstruktor kopije

Inicijalizacijom objekta date klase drugim objektom ili referencom na objekat iste klase vrši se kopiranjem članova tog drugog objekta u pripadne članove prvog objekta. Problem je kada objekat koji želimo da inicijalizujemo ima pokazivač ili referencu na dinamički objekat. Ono što bi prevodilac uradio je da bi kopirao npr. pokazivač tako da bi i taj pokazivač i njegova kopija ukazivali na isti objekat. Ono što se želi je da pri kopiranju oba objekta imaju vlastiti memorijski prostor i svako svoj primerak dinamičkog objekta, gde bi se u novi dinamički objekta samo kopirala vrednost originalnog dinamičkog objekta. Ovo se postiže konstruktorom kopije (*copy constructor*) date klase koji se može pozvati sa jednim argumentom koji je referenca na konstantan objekat te klase i koji se poziva pri konstrukciji objekta drugim objektom iste klase. Pozivi konstruktora kopije se realizuju kao što sledi:

- pri inicijalizaciji objekta operatorom jednako ili operatorom poziva funkcije gde se navodi kopiranje drugog objekta iste klase,

```
C cc1;       // konstruktor
C cc2=cc1;   // konstruktor kopije
C cc3(cc1); // konstruktor kopije
```

- pri inicijalizaciji formalnog argumenta funkcije stvarnim argumentom

```
void f(C);
...
f(cc1);        // zove se konstruktor kopije
```

- pri vraćanju povratne vrednosti funkcije.

```
C f(C cc)
{
  return cc;  // zove se konstruktor kopije
}
```

Primer:

```
class C {
    int *pi;
  public:
    C(int i) : pi(new int(i)) {}  // inline konstruktor
};
int main ()
{
  C cc1(0);     // poziv konstruktora
  C cc2=cc1;    // cc1.pi i cc2.pi ukazuju na isti objekat
  return 0;
}
```

Izmena sa konstruktorom kopije:

```
class C {
    int *pi;
  public:
    C(int i) : pi(new int(i)) {}  // inline konstruktor
```

```
    C(const C& cc)
    : pi(new int(*cc.pi)) {}   // inline konstruktor kopije
                               // pi ukazuje na vlastiti
                               // objekat čija je vrednost
                               // jednaka vrednosti originala
};
int main ()
{
  C cc1(0);     // poziv konstruktora
  C cc2=cc1;    // poziv konstruktora kopije
  return 0;
}
```

Ako klasa nema deklarisan konstruktor kopije, onda prevodilac implicitno generiše javni konstruktor kopije koji vrši redom kopiranje članova originalnog objekta u novokreirani objekat. Za ugrađeni objekat poziva se njegov konstruktor kopije.

Deklaracija: `C(const C&) = default;` znači da će prevodilac generisati konstruktor kopije, dok deklaracija `C(const C&) = delete ;` znači da se ne želi da prevodilac generiše konstruktor kopije.

3.2.2. Move konstruktor

Move konstruktor radi tako da novi objekat preuzme dinamički objekat originalnog objekta ("preusmeravanjem") pri čemu originalni objekat gubi svoj dinamički objekat.
Primer:

```
class ArrayWrapper{
  private:
    int *pvals;
    int size;
  public:
    ArrayWrapper () // podrazum. konstruktor npr. za 32 elementa
      : pvals( new int[ 32 ] ), size( 32 )
    {
    }
    ArrayWrapper (int n) // konstruktor za n elemenata
      : pvals( new int[ n ] ), size( n )
    {
    }
    ArrayWrapper (ArrayWrapper&& other) // move konstruktor
      : pvals( other.pvals ), size( other.size )
    {
      other.pvals = nullptr;  // levi
      other.size = 0;
    }
```

```
    ArrayWrapper (const ArrayWrapper& other) // konstr. kopije
      : pvals( new int[ other.size  ] ), size( other.size )
    {
      for( int i=0; i<size; ++i )
        pvals[ i ] = other.pvals[ i ];
    }
    ~ArrayWrapper ()
    {
      delete [] pvals;
    }
};
```

U prethodnom primeru klasa ArrayWrapper ima dva privatna atributa: pvals (pokazivač na int) i size tipa int.

Javni blok sadrži:

- javni podrazumevani konstruktor koji kreira dinamički niz od 32 elementa tipa `int`;
- konstruktor koji prihvata argument `n` tipa `int` i kreira dinamički niz od `n` elemenata tipa `int`;
- move konstruktor koji :
  - u inicijalizatoru inicijalizuje atribute kreiranog objekta vrednostima atributa prihvaćenog rvalue argumenta `other`, čime kreirani objekat svojim pokazivačem pvals ukazuje na dinamički niz int objekta `other` i gde `size` dobija vrednost broja elemenata tog dinamičkog niza
  - u svom telu postavlja atribute `pvals` i `size` argumenta `other` na `nullptr` i 0, čime `other` više nema dnamički niz objekata;
- konstruktor kopije koji kreira novi dinamički niz u koji se kopiraju elementi dinamičkog niza argumenta `other`;
- destruktor koji briše dinamički niz na koji ukazuje `pvals`.

3.3. Destruktor

Destruktor (*destructor*) je metoda bez argumenata i bez povratnog tipa koja ukida objekat klase. Postoji samo jedan destruktor za klasu. Destruktor ima isti naziv kao i naziv klase sa prefiksom tilda (~). Pri ugrađivanju, prvo se izvrši destruktor objekta, a onda destruktori ugrađenih objekata. Kao i konstruktor i destruktor može biti privatni, zaštićeni ili javni. Ne može se uzimati adresa destruktora. Destruktor se može pozvati i za const i za volatile objekat, pri čemu destruktor ne može biti niti static niti const niti volatile. Destruktor se ne nasleđuje, ali može biti virtuelni (biće pojašnjeno kasnije). Ako nema deklarisanog destruktora za datu klasu, onda će prevodilac kreirati javni destruktor za tu klasu. Kod ukidanja objekta izvedene klase njegov destruktor će pozvati destruktore njegovih atributa i destruktor osnovne klase.

Destruktor se poziva kao što sledi:

- pri eksplicitnom pozivu destruktora (npr. cc.~C(););
- pri napuštanju oblasti važenja automatskog objekta;
- pri ukidanju privremenog objekta koje je pod kontrolom prevodioca (ako se izađe iz opsega važenja ili ako je uzeta vrednost privremenog objekta ili ako se ukida referenca koja upućuje na privremeni objekat);
- pri ukidanju objekta koji ima ugrađivanje kada se pozivaju destruktori njegovih atributa;
- pri ukidanju kreiranog statičkog objekta na završetku izvršavanja programa;
- pri ukidanju dinamičkog objekta;
- pri ukidanju objekta izvedene klase kada se poziva destruktor osnovne klase.

Primer:
Atribut pi klase C tipa int* pri inicijalizaciji će ukazivati na dinamički objekat tipa int, koji će biti ukinut pozivom delete pi koji je naveden u telu destruktora.

```
class C {
    int *pi;
  public:
    C(int i) : pi(new int(i)) {}  // konstruktor
    C(const C& cc)
    : pi(new int(*cc.pi)) {}      // konstruktor kopije
    ~C()                          // destruktor
    {
      delete pi;                  // ukidanje dinamičkog objekta
    }
};
int main ()
{
  C cc1(0);      // poziv konstruktora
  C cc2=cc1;     // poziv konstruktora kopije
  return 0;      // kada objekti cc1 i cc2 prestaju da žive
                 // implicitno se poziva pripadni destruktor
}
```

Napomena: Ako u programu kreirate dinamički objekat, a ukidate samo pokazivač koji ukazuje na taj objekat, onda će on ostati izgubljen i zauzimaće memorijski prostor čime dolazi do curenja memorije (*memory leaking*).

3.4. Inspektori i mutatori

Inspektori (*inspectors*) predstavljaju metode koje ne menjaju atribute (stanje) objekta. Inspektori mogu menjati *mutable* atribute. Metode inspektori se nazivaju konstantim metodama (*constant member functions*).

Inspektori se označavaju ključnom reči `const` koja sledi iza zaglavlja inspektorske metode. U inspektorskoj metodi pokazivač `this` je tipa konstantan pokazivač na konstantan objekat, čime će prevodilac sprečiti izmenu atributa objekta.

Mutatori (*mutators*) su metode koje menjaju atribute objekta.

Primer:

```
class A {
    int i;
    mutable int mi;
  public:
    A (int b = 0)          // konstruktor
    {
      i = mi = b;
    }
    int Geti () const   // inspektor
    {
      mi++;
      return i;
    }
    void Puti (int b=0) // mutator
    {
      i = b;
    }
};
```

Inspektora mogu zvati i konstantni i nekonstantni objekti. Konstantni objekti ne mogu pozivati mutatore.

Primer:

```
void f() {
  const A consta;      //konstantan objekat klase A
  A a     ;            //nekonstantan objekat klase A
  consta.Puti(1);      //NOK
  int i=consta.Geti(); //konstantan objekat zove inspektora
  a.Puti(10);          //nekonstantan objekat zove mutatora
  int j=a.Geti();      //nekonstantan objekat zove inspektora
  //...
}
```

Korišćenjem const_cast može se izbeći korektnost konstantnosti:

```
(const_cast<A*>(&consta))->Puti(10);
```

Volatile metode (navodi se volatile iza zaglavlja metode) imaju pokazivač this tipa konstantan pokazivač na volatile objekat. Volatile objekti mogu pozivati samo volatile metode ili const volatile metode.

Ako je metoda i const i volatile (navodi se const volatile iza zaglavlja metode) onda je pokazivač this tipa konstantan pokazivač na const volatile objekat. Const volatile objekti mogu pozivati samo const volatile metode.

Const objekti mogu pozivati samo const metode ili const volatile metode.

Napomena: Specifikacija metode kao const ili volatile je deo njenog tipa.

U primeru na slici 3.4.1, dat je kod za klasu C koja ima tri metode Const, Volatile i ConstVolatile tipa int (void). Za dati primer gde je u funkciji main kreiran objekat cc koji je const volatile i kao takav može pozvati const volatile metodu. Ako se stavi da je objekat cc const, onda on može pozvati const i const volatile metodu. Ako se stavi da je objekat cc volatile, onda on može pozvati volatile i const volatile metodu.

```
#include <iostream>
using namespace std;
class C
{
public:
	int Const() const;
	int Volatile() volatile;
	int ConstVolatile() const volatile;
};
int C::Const() const { return 5; }
int C::Volatile() volatile { return 50; }
int C::ConstVolatile() const volatile { return 55; }
int main()
{
	const volatile C cc;
	cout << cc.Const() << endl;		// NOK
	cout << cc.Volatile() << endl;		// NOK
	cout << cc.ConstVolatile() << endl;
	return 0;
}
```

Slika 3.4.1. Primer za const, volatile i const volatile metode i objekte

Na slici 3.4.2. dato je zaglavlje klase Automobil. Klasa Automobil ima dva privatna atributa za naziv modela automobila i za njegovu cenu. Javni blok čine: podrazumevani konstruktor, mutatorska metoda za postavljanje stanja objekta klase Automobil (postavi podatke), inspektorska metoda za ispis podataka i inspektorska metoda za uzimanje cene automobila.

```
// fajl Automobil.h
#pragma once
#include <iostream>
```

```
#include <string>
using namespace std;
class Automobil{
  private:
    string model;
    double cena;
  public:
    Automobil();
    void PostaviPodatke(string,double);
    void IspisiPodatke() const;
    double UzmiCenu() const;
};
```

Slika 3.4.2. Zaglavlje klase Automobil

Na slici 3.4.3. data je implementacija klase Automobil. Podrazumevani konstruktor postavlja stanje objekta tipa Automobil na "nema naziv" i cena je 0. Metoda PostaviPodatke postavlja stanje objekta tipa automobil prema prosleđenim parametrima za naziv automobila i cenu automobila. Metoda IspisiPodatke ispisuje vrednosti atributa za naziv i cenu automobila. Metoda UzmiCenu vraća vrednost cene datog automobila.

```
// fajl Automobil.cpp
#include "Automobil.h"
Automobil::Automobil(){
  model = "nema naziv";
  cena = 0;
}
void Automobil::PostaviPodatke(
                     string nazivmodela ,
                     double cenaautomobila)
{
  model = nazivmodela;   cena = cenaautomobila;
}
void Automobil::IspisiPodatke() const
{
  cout<<"Model:"<<model<<endl;   cout<<"Cena :"<<cena <<endl;
}
double Automobil::UzmiCenu() const
{
  return cena;
}
```

Slika 3.4.3. Imlememtacija klase Automobil

Na slici 3.4.4. dato je zaglavlje klase KKopije. Klasa KKopije ima dva privatna atributa: pokazivač na int (pi) i pokazivač na Automobil (pa). Javni blok čine: konstruktor koji prihvata dva argumenta za postavljanje stanja objekta KKopije, konstruktor kopije koji prihvata referencu na konstantan objekat tipa KKopije, inspektorske metode getINT tipa int() i getAUTOMOBIL tipa Automobil*(), mutatorsku metodu set tipa void(int) i na kraju destruktor klase KKopije.

```
// fajl KKopije.h
#pragma once
class Automobil;
class KKopije {
    int *pi;
    Automobil *pa;
  public:
    KKopije(int i, Automobil a);    // konstruktor;
    KKopije(const KKopije&); // konstruktor kopije;
    int        getINT() const;
    Automobil* getAUTOMOBIL() const;
    void set(int i);
    ~KKopije();
};
```

Slika 3.4.4. Zaglavlje klase KKopije

Na slici 3.4.5. data je implementacija klase KKopije. Konstruktor sa dva argumenta kreira dva dinamička objekta za tip int i Automobil na koje ukazuju atributi pokazivači. Konstruktor kopije prihvata referencu na konstantan objekat klase KKopije, a onda kreira novi objekat tipa KKopije koji dobija vlastite dinamičke objekte za int i Automobil (na koje gledaju pripadni atributi pokazivači) postavljajući inicijalne vrednosti navedenih dinamičkih objekata da budu jednake vrednostima na koje ukazuju atributi objekta KKopije na koji upućuje prihvaćena referenca. Metoda getINT vraća vrednost tipa int na koju ukazuje pokazivač pi, dok metoda getAUTOMOBIL vraća objekat tipa Automobil na koji ukazuje pokazivač pa. Metoda set postavlja vrednost tipa int na koji ukazuje pokazivač pi. Destruktor ~KKopije ukida objekte na koje ukazuju pokazivači pi i pa.

```
// fajl KKopije.cpp
#include "KKopije.h"
#include "Automobil.h"
KKopije::KKopije(int i, Automobil a)
:
pi(new int(i)),
pa(new Automobil(a))
```

```
{
}
KKopije::KKopije(const KKopije& kk)
{
  pi=new int(*kk.pi);
  pa=new Automobil(*kk.pa);
}
int KKopije::getINT() const
{
  return *pi;
}
Automobil* KKopije::getAUTOMOBIL() const
{
  return pa;
}
void KKopije::set(int i)
{
  *pi=i;
}
KKopije::~KKopije()
{
  if(pi!=0) delete pi;
  if(pa!=0) delete pa;
}
```

Slika 3.4.5. Imlememementacija klase KKopije

```
// fajl gde je main
#include <iostream>
using namespace std;
#include "KKopije.h"
#include "Automobil.h"
void Ispisi(KKopije kk_a)
{ //napisite metodu u klasi KKopije koja radi isto
  cout<<endl<<"Ispis:"<<endl;
  cout<<" *pi         = "<<kk_a.getINT()<<endl;
  cout<<" *Automobil = "<<endl;
  kk_a.getAUTOMOBIL()->IspisiPodatke();
  cout<<endl;
}
void main(){
  Automobil alfa;
  alfa.PostaviPodatke("Alfa Romeo GT",30000);//int u double
  KKopije kk_a(10,alfa);
```

```
  alfa.PostaviPodatke("Alfa 166",28000);
  KKopije kk_b(kk_a);
  Ispisi(kk_a);
  Ispisi(kk_b);
  kk_a.set(5);
  kk_a.getAUTOMOBIL()->PostaviPodatke("Ford Mustang", 60000);
  Ispisi(kk_a);
  Ispisi(kk_b);
  system("pause");
}
// IZLAZ
Ispis:
*pi        = 10
 *Automobil =
Model:Alfa Romeo GT
Cena :30000

Ispis:
*pi        = 10
 *Automobil =
Model:Alfa Romeo GT
Cena :30000

Ispis:
*pi        = 5
 *Automobil =
Model:Ford Mustang
Cena :60000

Ispis:
*pi        = 10
 *Automobil =
Model:Alfa Romeo GT
Cena :30000
Press any key to continue . . .
```

Slika 3.4.5. Primena klase Automobil i KKopije

Na slici 3.4.5. dat je prikaz korišćenja klasa KKopije i Automobil. Funkcija Ispisi ispisuje vrednosti atributa objekta klase KKopije. U funkciji main kreiran je objekat alfa tipa Automobil sa atributima "Alfa Romeo GT" i 30000. Nakon toga kreiran je objekat kk_a tipa KKopije sa atributima 10, alfa. Nakon ovoga, objektu alfa je promenjeno stanje tako da su novi atributi "Alfa 166", 28000. Sada je pozivom konstruktora kopije klase KKopije, kome je prosleđen objekat kk_a,

kreiran objekat kk_b tipa KKopije. Korišćenjem funkcije Ispisi ispisuju se podaci o objektima kk_a i kk_b (isti ispisi). Na ispis kk_a nije uticala promena stanja objekta alfa jer je konstrukoru koji je kreirao kk_a bio argument alfa prosleđen po vrednosti. Sada sledi promena stanja atributa objekta kk_a tako da sada pripadni atribut pi ukazuje na 5, a pripadni atribut pa ukazuje na objekat tipa Automobil čije je novo stanje "Ford Mustang", 60000. Nakon ove promene slede ispisi objekata kk_a i kk_b koji pokazuju da promena stanja objekta kk_a nije uticala na stanje objekta kk_b (imaju vlastite dinamičke objekte).

3.5. Statički članovi klase

Statički članovi klase (*static members*) su zajednički za sve objekte te klase. Ovi članovi se smeštaju u statičku memorijsku oblast. Statički atribut (*static data member*) deklariše se rečju static koja prethodi tipu atributa. Statički atribut je jedan za celu klasu i sve objekte, dok je nestatički atribut jedan za svaki objekat klase.
Primer:

```
class A {
  static int cl;   // samo jedan cl za celu klasu;
  int ob;         // svaki objekat ima svoj ob
  //...
};
```

Pristup statičkom atributu predstavlja pristup istoj oblasti u memoriji. Statički atributi se nalaze u statičkoj memorijskoj oblasti i kreiraju se na početku programa i traju sve do kraja programa (statički su po životnom veku).

Statiči atribut definiše se u implementacionom fajlu navođenjem tipa atributa, naziva pripadne klase, operatora razrešavanja naziva (::), identifikatora statičkog atributa i dodele, bez navođenja reči static, kao što sledi:

```
int Student::ukupanbrojstudenata = 0;
```

Statički atribut pripada klasi i nije potreban niti jedan kreirani objekat da bi statički atribut postojao, drugim rečima, može mu se pristupiti preko naziva klase i operatora razrešavanja poziva (::), kao što sledi:

```
cout<<Student::ukupanbrojstudenata<<endl;
```

Preko objekta i pokazivača na objekat klase koja ima statički atribut može se standarno pristupiti statičkom atributu, pošto je taj atribut zajednički za sve objekte:

```
cout<<student1.ukupanbrojstudenata<<endl;
cout<<pokazivacnastudent->ukupanbrojstudenata<<endl;
```

Kontrola pristupa statičkom atributu je kao i za nestatičke atribute, osim u slučaju inicijalizacije.

Statički atributi su kao globalni objekti u oblasti važenja pripadne klase.

Statičke metode (*static member functions*) predstavljaju metode koje pripadaju nivou klase i ne pripadaju niti jednom objektu, a samim tim nemaju pokazivač `this`. Statičke metode imaju eksterno povezivanje. Deklaracija statičke metode se izvodi dodavanjem reči `static` na početak deklaracije:

```
static int UzmiUkupanBrojStudenata();
```

Pri definiciji statičke metode ne navodi se reč static:

```
int Student::UzmiUkupanBrojStudenata()
{
  return ukupanbrojstudenata;
}
```

Statičke metode su kao globalne funkcije, ali na nivou klase koja je i oblast važenja. Ovakve metode se koriste kada je potrebna metoda koja nije vezana za objekat već za nivo klase (sve objekte).

Za pristup statičkoj metodi ne mora postojati niti jedan objekat, već se poziv statičke metode može obaviti navođenjem naziva klase, operatora za razrešavanje poziva (::) iza koga sledi identifikator statičke metode sa listom argumenata:

```
cout<<Student::UzmiUkupanBrojStudenata()<<endl;
```

U telu statičke metode može se pristupiti statičkim atributima. Nestatičkim atributima u telu statičke metode može se pristupiti preko objekta ili pokazivača na objekat pripadne klase korišćenjem odgovarajućeg operatora pristupa (`.` ili `->`).

Preko objekta i pokazivača na objekat klase koja ima statičku metodu može se standarno pristupiti toj statičkoj metodi, pošto je ista zajednička za sve objekte:

```
cout<<student1.UzmiUkupanBrojStudenata()<<endl;
cout<<pokazivacnastudent->UzmiUkupanBrojStudenata()<<endl;
```

Statička metoda ne može biti virtuelna. Statičnost ne određuje tip metode.

Primer:

```
class A {                  // privatni blok članova
         int inest;            // nestatiči atribut
  static int jstat;            // statički atribut
public:                    // javni blok članova
              A();               // konstruktor
         void fnest();           // nestatička metoda
  static void gstat(A);          // statička metoda
};
-----------------------------------------------------------------
```

```
int A::jstat = 10;          // definicija statičkog atributa
A::A()                      // definicija konstruktora
{
  inest = 0;
}
void A::fnest()             // definicija nestatičke metode
{
  int i=inest;              // pristup nestatičkom atributu
  int j=jstat;              // pristup   statičkom atributu
}
void A::gstat(A a)          // definicija statička metode
{
  int i=inest;              // NOK, pristup nestatičkom atributu
  int k=a.inest;            // posredno do nestatickog atributa
  int j=jstat;              // pristup statičkom atributu
}
-------------------------------------------------------------------
void main ()
{
  A a;
  X::gstat(a);    // A::gstat, metoda klase
  X::fnest();     // NOK, metoda objekta
  a.fnest();      // metoda objekta
  a.gstat(a);     // metoda klase može i poziv preko objekta
}
```

Na slici 3.5.1. dato je zaglavlje klase `Student`. Dozvoljeno je jednom da se uključi zaglavlje klase `Student` (`#pragma once`), uključena su zaglavlja redom: za ulaz i izlaz podataka, stringove i podešavanja svojstava ispisa koja će trebati u programu. U klasi `Student` naveden je privatni blok članova koji se odnose na: ime, indeks, bodove sa kolokvijuma 1, kolokvijuma 2, testa te bodove sa aktivnosti na nastavi. Na kraju bloka je statički atribut koji se odnosi na broj kreiranih studenata. U javnom bloku su: konstruktor bez formalnih argumenata, metoda za postavljanje stanja objekta tipa `Student`, metoda za ispis podataka o studentu i na kraju je data statička metoda koja vraća broj kreiranih studenata.

```
#pragma once
#include <iostream>
#include <string>
#include <iomanip>
using namespace std;
class Student {
  private:
    string ime;
    string indeks;
```

```
    int kol_1;
    int kol_2;
    int test;
    int aktivnost;
    static int brojKreiranihStudenata;
  public:
         Student();
    void PostaviPodatke(string ime, string indeks,
                        int prvikolokvijum, int drugikolokvijum,
                        int test, int aktivnoststudenta);
    void IspisiPodatke();
    static int VratiBrojStudenata();
};
```

Slika 3.5.1. Fajl zaglavlja klase Student

Na slici 3.5.2. dat je implementacioni fajl klase `Student`. Uključenjem zaglavlja `Student.h` kompajler će znati za potrebne nazive koji se koriste u imeplementacionom fajlu. Sledi definicija statičkog atributa `brojKreiranih Studenata`. Definisan je konstruktor bez argumenata koji postavlja početno stanje novokreiranog objekta tipa `Student`, iza inicijalizatora dvotačka (`:`), a onda se u telu konstruktora inkrementira broj kreiranih studenata i ispisuje broj poziva konstruktora. Metoda `PostaviPodatke` trivijalno vrši kopiranje formalnih argumenata u pripadne nestatičke atribute klase korišćenjem pokazivača `this`. Metoda `IspisiPodatke` ispisuje podatke o studentu navodeći koliko je bilo bodova na kolokvijumima, testu, na aktivnostima na nastavi, ukupan broj bodova i pripadnu ocenu. Za formatiranje ispisa korišćen je poziv `setw` za postavljanje širine ispisa. Na kraju je data definicija statičke metode `VratiBrojStudenata` koja vraća broj kreiranih objekata tipa `Student`.

```
#include "Student.h"
int Student::brojKreiranihStudenata = 0; //ne navodi se static
Student::Student()
:
ime("nema ime"),
indeks("nema indeks"),
kol_1(0),
kol_2(0),
test(0),
aktivnost(0)
{
      brojKreiranihStudenata++;
      cout <<"Broj poziva konstruktora: "
       <<brojKreiranihStudenata << endl;
}//moze se koristiti i metoda VratiBrojStudenata
void Student::PostaviPodatke(string ime, string indeks,
                 int kol_1, int kol_2, int test, int aktivnost){
```

```
      this->ime = ime;
      this->indeks = indeks;
      this->kol_1 = kol_1;
      this->kol_2 = kol_2;
      this->test = test;
      this->aktivnost = aktivnost;}

void Student::IspisiPodatke()
{
  double ukupno = static_cast<double>(kol_1+kol_2+test+aktivnost);
  cout<<endl;
  cout<<"::"<<ime<<", "<<indeks<<endl;
  cout<<"  K1:  K2:  TEST  AKTIVNOST  UKUPNO  OCENA"<<endl;
  cout<<setw(5)<< kol_1;
  cout<<setw(5)<< kol_2;
  cout<<setw(6)<< test;
  cout<<setw(9)<< aktivnost;
  cout<<setw(8)<< (static_cast<int>(ukupno));
  if(ukupno>=51.0) cout<<setw(7)
                       << (6 + static_cast<int>((ukupno-
51.0)/10.0));
  else             cout<<setw(7)<< 5;
}
int Student::VratiBrojStudenata() //ne navodi se static
{
   return brojKreiranihStudenata;
}
```

Slika 3.5.2. Implementacioni fajl klase Student

Na slici 3.5.3. dat je primer upotrebe klase `Student`, sa pripadnim izlazima programa. U fajlu gde je funkcija main uključena su zaglavlja: `iostream`, `string`, `Student.h`, navedeno je da se koristi standardni prostor imena (`using namespace std;`), navedena je konstanta `BROJ` tipa `int` čija je vrednost 2, a onda navedena funkcija `main`.

U funkciji `main` se kreira niz od 2 studenta (što je moguće jer postoji konstruktor koji se može pozvati bez argumenata), zatim referenca na dinamički kreirani objekat tipa `Student`, a onda se ispisuje koliko je studenata kreirano pozivom statičke metode `VratiBrojStudenata`.

U `for` petlji se unose ulazne vrednosti za sva tri kreirana studenta korišćenjem funkcije `getline` i metode `cin`, pri čemu se i poziva metoda `cin.ignore()` da bi se potrošio zaostali enter od prethodnog unosa. Na kraju se u drugoj `for` petlji ispisuju podaci za sva tri studenta potivom metode `IspisiPodatke`, briše dinamički element preko reference koja na njega upućuje, pravi pauza pozivom

sistemske komande "pause" te izlazi iz programa naredbom return. Na kraju je naveden primer ulaza i pripadnog izlaza iz programa.

```
#include <iostream>
#include <string>
#include "Student.h"
using namespace std;
const int BROJ = 2;
int main(void)
{
  Student studenti[BROJ];     //nedinamicki
  Student &rstudent = *new Student;
  cout<<"Broj studenata za koji cemo uneti podatke je "
      <<Student::VratiBrojStudenata()<<"."<<endl;
  for(int i=0; i<=BROJ; ++i){
    string ime, indeks;
    int kol1, kol2, test, aktivnost;
    cout<<"Student "<<i+1<<": ime      : ";getline(cin,ime);
    cout<<"Student "<<i+1<<": indeks   : ";getline(cin,indeks);
    cout<<"Student "<<i+1<<": kol. 1   : ";cin>>kol1;
    cout<<"Student "<<i+1<<": kol. 2   : ";cin>>kol2;
    cout<<"Student "<<i+1<<": test     : ";cin>>test;
    cout<<"Student "<<i+1<<": aktivnost: ";cin>>aktivnost;
cin.ignore();

    if(i!=BROJ)

studenti[i].PostaviPodatke(ime,indeks,kol1,kol2,test,aktivnost);
    else

rstudent.PostaviPodatke(ime,indeks,kol1,kol2,test,aktivnost);
  }
  for(int i=0; i<=BROJ; ++i){
    if(i!=BROJ) studenti[i].IspisiPodatke();
    else           rstudent.IspisiPodatke();
  }
  delete &rstudent;
  cout<<endl;system("pause"); return(0);
}
-------------- IZLAZ IZ PROGRAMA ---------------
Broj poziva konstruktora: 1
Broj poziva konstruktora: 2
Broj poziva konstruktora: 3
Broj studenata za koji cemo uneti podatke je 3.
Student 1: ime      : Pera Peric
Student 1: indeks   : 2014/001
Student 1: kol. 1   : 30
```

```
Student 1: kol. 2   : 30
Student 1: test     : 30
Student 1: aktivnost: 10
Student 2: ime      : Sima Simic
Student 2: indeks   : 2014/002
Student 2: kol. 1   : 20
Student 2: kol. 2   : 20
Student 2: test     : 20
Student 2: aktivnost: 1
Student 3: ime      : Ana Anic
Student 3: indeks   : 2014/003
Student 3: kol. 1   : 25
Student 3: kol. 2   : 25
Student 3: test     : 25
Student 3: aktivnost: 10
::Pera Peric, 2014/001
  K1:  K2:  TEST  AKTIVNOST  UKUPNO  OCENA
   30   30    30        10     100     10
::Sima Simic, 2014/002
  K1:  K2:  TEST  AKTIVNOST  UKUPNO  OCENA
   20   20    20         1      61      7
::Ana Anic, 2014/003
  K1:  K2:  TEST  AKTIVNOST  UKUPNO  OCENA
   25   25    25        10      85      9
Press any key to continue . . .
```

Slika 3.5.3. Primer upotrebe klase Student

3.6. Prava pristupa i prijatelji klase

Specifikatorima pristupa članovima omogućena je dostupnost članova kao što sledi:

- private, privatni pristup gde je član dostupan samo u okviru pripadne klase;
- protected, zaštićeni pristup gde je član dostupan unutar klase i u izvedenim klasama;
- public, javni pristup gde je član dostupan svima.

Dozvoljeno je da se specifikatori navode u obliku `specifikator:` ispred svakog člana klase, ili za blok članova klase, pri čemu pravo pristupa vredi od navođenja specifikatora pristupa do navođenja sledećeg specifikatora pristupa ili kraja klase.

Napomena: Prvo se gleda vidljivost člana, a onda i da li mu se može pristupiti prema datom specifikatoru pristupa.
Primer:

```
class Osnovna
{
  private:
    int priv;
    void ff(int);
  protected:
    int prot;
    void ff(double);
  public:
    int publ;
};
...
class Izvedena : public Osnovna
{
  void ppp() {
    priv=10;          // NOK, vidi se, ali je nedostupan
    prot=10;          // vidi se i dostupan je
    publ=10;          // vidi se i dostupan je
    ff(3);            // void ff(int) je nedostupna
    ff(3.14);         // vidi se i dostupna je
  }
};
```

Prijatelji klase (*friend*) su funkcije, metode ili klase za koje ne vrede specifikatori pristupa koji su navedeni u klasi u kojoj je navedeno prijateljstvo. Prijateljskoj funkciji se može dostaviti kao argument i privremeni objekat nastao implicitnom konverzijom tipa. Prijateljstvo nije nasledno niti je tranzitivno. Prijateljstvo se navodi korišćenjem reči *friend* ispred naziva prijatelja. Vrede pravila razrešavanja poziva kod poziva *friend* funkcije ili metode.

Primer:

```
class A {
  private:
    friend class B;          // prijateljska klasa
    friend int C::met(A);    // prijateljska metoda klase C
    friend int glob(A);      // prijateljska funkcija;
    int priv;
};
...
class B
{
   // svi članovi klase A su dostupni
};
...
class C
{
  ...
  int met(A aa)
  {
    return aa.priv*10;
  }
  ...
};
```

```
int glob(A aa)
{
  return aa.priv*aa.priv;
};
...
```

Napomena: Prijateljska funkcija se obično koristi za pristup članovima više klasa (svima je jednak prijatelj) gde se ne vrši promena vrednosti člana (stanja objekta).

3.7. Preklapanje operatora

Preklapanje operatora podrazumeva da se postojeći operatori mogu definisati i za apstraktne tipove, pri čemu efekat preklopljenog operatora određuje programer. Pošto se radi o preklapanju postojećih operatora, onda nije moguće promeniti prioritet, grupisanje ili broj operanada koje preklopljeni operatori prihvataju u odnosu na originalne operatore niti je moguće uvesti nove operatore.

Operatori koji se ne mogu preklopiti su: sizeof, tačka (.), tačka-zvezdica (.*), dve dvotačke (::), operator uslovne dodele (?:).

Sintaksa deklaracije operatorske metode za preklapanja operatora je da se navede povratni tip, zatim reč operator koju sledi sam operator i lista formalnih argumenata koji predstavljaju operande tog operatora. Jedan operand operatorske metode predstavlja objekat koji je tu metodu pozvao tako da je lista argumenta za 1 manja od potrebnog broja operanada. Kod operatorske funkcije u listi formalnih argumenata se navodi onoliko operanada koliko zahteva dati operator. Operatorske metode i funkcije ne mogu imati podrazumevane argumente.

U izrazima se preklopljeni operatori koriste u "prirodnoj" notaciji nad operandima ili eksplicitno navođenjem poziva operatorske funkcije ili operatorske metode.

Primer operatorske metode za operator +

```
Argument operator+ (Argument); // jedan operand je objekat
                               // čiji je + pozvan,
                               // a drugi operand je Argument
Argument a1, a2;
Argument a3 = a1 + a2 ;  // isto što i a3=a1.operator+(a2);
```

Primer operatorske funkcije za operator +

```
Argument operator+ (Argument, Argument);
// oba operanda su argumenti operatorske funkcije
```

```
Argument a1, a2;
Argument a3 = a1 + a2 ;  // isto što i a3=operator+(a1,a2);
```

Zbog istog poziva može postojati odgovarajuća operatorska funkcija ili operatorska metoda. Operatorska funkcija može uraditi implicitnu konverziju stvarnih argumenata u tipove formalnih argumenata (operatorska metoda ne). Primer:

```
class KompleksanBroj {
    double realni;
    double imaginarni;
  public:
    KompleksanBroj (double re = 0, double im = 0)
    :
    realni(re),
    imaginarni(im)
    {}
    friend KompleksanBroj operator+
                              (KompleksanBroj, KompleksanBroj);
};
KompleksanBroj operator+(KompleksanBroj c1, KompleksanBroj c2)
{
  return KompleksanBroj(c1.realni     + c2.realni,
                        c1.imaginarni + c2.imaginarni);
}
int main () {
  KompleksanBroj c = 1.2345 + KompleksanBroj(5.0,5.0);
  // pozvace se
  // operator+(KompleksanBroj(1.2345),KompleksanBroj(5.0,5.0));
  return 0;
}
```

Ako bi se u prethodnom primeru umesto operatorske funkcije sabiranja koristila operatorska metoda sabiranja:

```
KompleksanBroj KompleksanBroj::operator+(KompleksanBroj c2)
const
{
  return KompleksanBroj(realni     + c2.realni,
                        imaginarni + c2.imaginarni);
}
```

ne bi bio dozvoljen poziv:

```
KompleksanBroj c = 1.2345 + KompleksanBroj(5.0,5.0);
// KompleksanBroj(1.2345).operator+(KompleksanBroj(5.0,-1.0))
```

jer nije dozvoljena implicitna konverzija 1.2345 u KompleksanBroj.

Ako operatorska funkcija menja vrednost nekog od operanada, onda taj operand mora biti *lvalue*. Formalni argument za operand koji se menja mora biti referenca na nekonstantan objekat, a pripadni stvarni argument objekat (bez konverzije). Formalni argument za operand koji se ne menja može se pozvati stvarnim argumentom gde se može uraditi i konverzija (operator+= je friend).

```
KompleksanBroj& operator+=
                    (KompleksanBroj &levi, KompleksanBroj desni)
{
  levi.realni     += desni.realni;
  levi.imaginarni += desni.imaginarni;
  return cl;
}
```

Prema prethodnom je kao što sledi:

```
KompleksanBroj c1;
double d=1.2345;
c += d;
d += c;    // NOK
```

Operatorska metoda za prethodnu operaciju += koja menja vrednost operanda (objekta) koji je ovu metodu pozvao ima sintaksu kao što sledi:

```
KompleksanBroj& KompleksanBroj::operator+=
                                          (KompleksanBroj desni)
{
  this->realni     += desni.realni;
  this->imaginarni += desni.imaginarni;
  return *this;
}
```

Operatorska funkcija za unarni operator ima jedan argument (operand), dok operatorska metoda za unarni operator nema argument jer je operand upravo objekat koji je pozvao ovu metodu.
Primer:

```
// funkcija
KompleksanBroj operator!(KompleksanBroj operand)
{
  return KompleksanBroj(operand.realni,-operand.imaginarni);
}

// metoda
KompleksanBroj KompleksanBroj::operator!()
{
  return KompleksanBroj(this->realni,-this->imaginarni);
}
```

Sintaksa za operatore inkrementiranja (dekrementiranja) je kao što sledi:

- `operator++()` za prefiksno inkrementiranje, lvalue
- `operator--()` za prefiksno dekrementiranje, lvalue
- `operator++(int)` za postfiksno inkrementiranje, int je bilo koje vrednosti
- `operator--(int)` za postfiksno dekrementiranje, int je bilo koje vrednosti

Primer:

```
// deklaracija
C& operator++();    // ++ prefiksni, lvalue te vraća referencu
C  operator++(int); // postfiksni ++

// definicija
C& C::operator++()
{
  // predinkrement koji definiše programer za dati tip
  return *this;
}
C  operator++(int)
{
  // postinkrement koji definiše programer za dati tip
  return *this;
}

// pozivi
C cc;     // objekat klase C
++cc;     // isto kao i poziv cc.operator++();
cc++;     // isto kao i poziv cc.operator++(0);
```

Operator poziva funkcije () je binarna operacija gde je prvi operand objekat koji ide pre (), a drugi operand je lista argumenata koja se smešta u (). Operatorska metoda poziva funkcije je nestatička metoda. Sintaksa operatora poziva funkcije je kao što sledi:

```
C  cc;
cc();                 // poziv cc.operator();
cc(arg1, arg2);       // poziv cc.operator()(arg1, arg2);
cc(arg1, arg2, arg3); // poziv cc.operator()(arg1, arg2, arg3);
```

Primer:

```
class C {
    //...
  public:
    //...
    int operator()(int arg1, int arg2)
    {
      return arg1*arg2;
    }
    //...
};
```

```
int main ()
{
  C cc;
  int mnozenje = cc(5,5);  // kao i poziv cc.operator()(5,5);
  return 0;
}
```

U primeru datom na slici 3.7.1. data je klasa Matrica koja omogućuje kreiranje dvodimenzionalne matrice elemenata tipa double realizovane dinamički korišćenjem pokazivača na pokazivač na tip double.

Zbog jednostavnosti, implementacija klase Matrica je inline. Uključena su potrebna zaglavlja za ispis i formatiranje ispisa (iostream, iomanip) te je uključen prostor imena std. Klasa Matrica ima privatni blok koga čine atributi red (za redove) i kol (za kolone) tipa int i ppd tipa double** (za smeštanje elemenata matrice).

U javnom bloku su:

- Konstruktor koji prihvata dva argumenta tipa int i koji inicijalizuje članove red i kol vrednostima pripadnih argumenata. U telu konstruktora se kreira dinamički niz u koji može da se smesti "red" pokazivača na double, a onda se za svaki takav pokazivač kreira dinamički prostor u koji se može smestiti "kol" elemenata tipa double.
- Operator poziva funkcije operator() prihvata dva argumenta koji određuju poziciju elementa u matrici. Elementu se pristupa sa ppd[r][k], što znači da je element u redu r i u koloni k. Pošto operator vraća referencu na tip double moguće je i uzeti vrednost elemenata sa date pozicije u matrici, ali i postaviti novu vrednost tog elemenata.
- Metoda UzmiVrednost samo vraća vrednost elementa u matrici na poziciji (r,k). Korišćenjem pointerske aritmetike bira se pokazivač za red u kome se element nalazi *(ppd+r), a onda element koji se nalazi u koloni k u navedenom redu *(*(ppd+r)+k).
- Operatorska metoda operator<< omogućuje ispis elemenata matrice po redovima i kolonama. Za dohvatanje elemenata ova metoda koristi operatorsku metodu operator(). Ispis se obavlja u dvostrukoj petlji. Ova metoda vraća referencu na izlazni tok koja je njen parametar, tako da je omogućeno dalje korišćenje izlaznog toka za ispis. Kombinacija fixed i setprecision(2) omogućije ispis double broja sa dve decimalne tačke, dok setw(10) omogućuje ispis broja double na 10 mesta, računajući i decimalnu tačku.
- Destruktor oslobađa dinamički prostor koji matrica zauzima, tako da prvo oslobodi dinamički prostor za kolone svakog reda, a onda i dinamički prostor redova (pokazivača na double).

```
#include <iostream>
```

```
#include <iomanip>
using namespace std;
class Matrica
{
	int red;
	int kol;
	double **ppd;
public:
	Matrica(int r, int k)
	:
	red(r),
	kol(k)
	{
		ppd = new double*[red];
		for (int i = 0; i < red; i++) ppd[i] = new
double[kol];
	}
	double& operator()(int r, int k)  //lvalue
	{
		return ppd[r][k];
	}
	double UzmiVrednost(int r, int k)
	{
		return *(*(ppd + r) + k);
	}
	ostream& operator<<(ostream &os)
	{
		for (int r = 0; r < red; r++)
		{
			for (int k = 0; k < kol; k++)
			{
			   cout << fixed << setprecision(2) <<setw(10)
				<< this->operator()(r, k);
			}
			cout << endl;
		}
		return os;
	}
	~Matrica()
	{
		for (int i = 0; i < red; i++) delete[] ppd[i];
		delete[] ppd;
	}
};
```

*Slika 3.7.1. Klasa Matrica realizovana pomoću double***

U primeru datom na slici 3.7.2. data je klasa Matrica2 koja omogućuje kreiranje dvodimenzionalne matrice realizovane dinamički korišćenjem pokazivača na tip double.

Zbog jednostavnosti, implementacija klase Matrica2 je inline. Uključena su potrebna zaglavlja za ispis i formatiranje ispisa (iostream, iomanip) te je uključen prostor imena std. Klasa Matrica2 ima privatni blok koga čine atributi red (za redove) i kol (za kolone) tipa int i pd tipa double* (za smeštanje elemenata matrice).

U javnom bloku su:

- Konstruktor koji prihvata dva argumenta tipa int i koji inicijalizuje članove red i kol vrednostima pripadnih argumenata. U telu konstruktora se kreira dinamički niz elemenata tipa double u koji može da se smesti red*kol elemenata.
- Konstruktor kopije koji: kopira vrednosti red i kol od originala, kreira novi dinamički prostor na koji ukazuje pokazivač pd i kopira vrednosti elemenata originalne matrice u novi dinamički prostor.
- Operator poziva funkcije operator() prihvata dva argumenta koji određuju poziciju elementa u matrici. Elementu se pristupa sa pd[r*kol+k], što znači da je element u jednodimenzionalnom nizu smešten na poziciji r*kol+k, jer je manji indeks kolona. Pošto operator vraća referencu na tip double moguće je i uzeti vrednost elemenata sa date pozicije u matrici, ali i postaviti novu vrednost tog elemenata.
- Metoda UzmiVrednost samo vraća vrednost elementa u matrici na poziciji (r,k). Korišćenjem pointerske aritmetike bira se pozicija na kojoj se nalazi element *(pd+r*kol+k), tako da napisano znači da je od početka niza na koji ukazuje pokazivač pd element udaljen za r*kol+k elemenata.
- Operatorska metoda operator<< omogućuje ispis elemenata matrice po redovima i kolonama. Implementirana je isto kao i za klasu Matrica.
- Destruktor oslobađa dinamički prostor koji matrica zauzima, tako da oslobađa dinamički prostor niza double.

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

using namespace std;
class Matrica2
{
    int red;
    int kol;
    double *pd;
  public:
     Matrica2(int r, int k): red(r), kol(k)
     {
```

```
            pd = new double[red*kol];
        }
        Matrica2(const Matrica2 &mm2)
        :
        red(mm2.red),
        kol(mm2.kol),
        pd(new double[mm2.red*mm2.kol])
        {
            for(int i=0; i<red*kol; i++) pd[i]=mm2.pd[i];
        }
        double& operator()(int r, int k)
        {
              return pd[r*kol+k];
        }
        double  UzmiVrednost( int r, int k )
        {
              return *(pd+r*kol+k);
        }
        ostream& operator<<(ostream &os)
        {
              for(int r=0; r<red; r++)
              {
                    for(int k=0; k<kol; k++)
                    {
                          cout<<fixed<<setprecision(2)<<setw(10)
                              <<this->operator()(r,k);
                    }
                    cout<<endl;
              }
              return os;
        }
        ~Matrica2()
        {
          delete [] pd;
        }
};
```

*Slika 3.7.2. Klasa Matrica realizovana pomoću double**

Na slici 3.7.3. dat je primer upotrebe klasa Matrica i Matrica2. U fajlu gde je funkcija main uključena su zaglavlja iostream i iomanip, Matrica.h i Matrica2.h.

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "Matrica.h"
#include "Matrica2.h"
int main()
```

```
{
	int r=3;
	int k=2;
	Matrica mm(r,k);
	for (int red = 0; red < r; red++)
		for (int kol = 0; kol < k; kol++)
		  mm(red, kol) = static_cast<double>(red * 10 + kol);
	mm(0, 0) = 100.0;
	cout << "mm" << endl;
	mm << cout;
	Matrica2 m1(r,k);
	for(int red=0; red<r; red++)
		for(int kol=0; kol<k; kol++)
		  m1(red, kol) = static_cast<double>(red * 10 + kol);
	Matrica2 m2 = m1;
	m1(0, 0) = 555.0;
	cout << "m1" << endl;
	m1 << cout;
	cout << "m2" << endl;
	m2 << cout;
	return 0;
}
IZLAZ
mm
    100.00      1.00
     10.00     11.00
     20.00     21.00
m1
    555.00      1.00
     10.00     11.00
     20.00     21.00
m2
      0.00      1.00
     10.00     11.00
     20.00     21.00
```

Slika 3.7.3. Primena klase Matrica i Matrica2

U funkciji main kreiran je objekat mm tipa Matrica koja ima r redova (3) i k kolona(2). Matrica mm je popunjena elementima double (primenjen je operator promene tipa) po redovima 0.0 , 1.0, 10.0, 11.0, 20.0 i 21.0. Pozvana je operatorska metoda poziva funkcije, tako da je postavljeno da element matrice mm koji je na poziciji (0,0) ima vrednost 100.0. Nakon ovoga je pozvana operatorska metoda za ispis matrice.

Kreiran je objekat m1 tipa Matrica2 koji sadrži iste elementie kao i matrica mm. Pozvan je konstruktor kopije koji kreira objekat m2 tipa Matrica2 gde je original objekat m1. Sada se za objekat m1 postavlja da je vrednost elementa koji se

nalazi na poziciji (0,0) jednaka 555. Sledi ispis matrice m1 i m2. Izlaz pokazuje da matrica m2 ima vlastiti dinamički prostor koji je dodeljen pri pozivu konstruktora kopije jer promena vrednosti elemenata matrice m1 nije uticala na isti element matrice m2.

Operatorska metoda indeksiranja operator[] je nestatička binarna operacija. Sintaksa je kao što sledi:

```
TIP& C::operator[](int i);    // deklaracija
C cc;                         // objekat za poziv
lvrednost = cc[indeks];      // poziv, lvrednost je tipa TIP
```

Operatorska metoda posrednog pristupa članu preko pokazivača operator-> je nestatička unarna operacija. Ako povratna vrednost ove metode nije pokazivač, onda se nastavlja dalji poziv operatora -> (*drill-down behavior*) dok se ne dođe do pokazivača, a onda se uradi poziv ->.

Primer 1:

```
#include <iostream>
struct A
{
  void fa()
  {
    std::cout<<"fa";
  };
};
struct B {
  A* pA;
  A* operator->()
  {
    return pA;
  };
};
int main()
{
  B bb;
  bb->fa();    // poziv je (bb.operator->())->f().
  return 0;
}
```

Poziv bb-> je poziv bb.operator->() koji vraća pokazivač pA, a onda se uradi pA->fa().

Primer 2:

```
#include <iostream>
struct A
{
  int ia;
};
```

```
struct B
{
  A* pA;
  A* operator->()
  {
    return pA;
  }
};
struct C
{
  B* pB;
  B& operator->()
  {
    return *pB;
  }
};
int main()
{
    A aa = { 10 };
    B bb = { &aa };
    C cc = { &bb };
    std::cout << aa.ia << bb->ia << cc->ia; // ispisuje "101010"
    return 0;
}
```

Izraz aa.ia predstavlja vrednost javnog atributa ia objekta aa. Poziv bb->ia je poziv bb.operator->() koji vraća pA, a onda se uradi (kao u prethodnom primeru) pA->ia. Poziv cc->ia je poziv cc.operator->() koji vraća referencu na objekat tipa B na koji ukazuje pokazivač pB, a onda se za taj objekat poziva operatorska funkcija operator->() koji vraća pokazivač pA, a onda se uradi pA->ia.

Operatorska metoda dereferenciranja operator* je nestatička unarna operacija. Primer:

```
  #include <iostream>
  class A {
        int ia;
      public:
        A(int i){ia=i;}
        int& operator*() { return ia; }
        int Get(){return ia;}
  };
  int main()
  {
    A* pa = new A(10);
    *pa = 20;             // poziv pa.operator*()
    std::cout<<pa->Get(); // ispis 20
```

```
    delete pa;
    return 0;
}
```

U primeru je kreiran pokazivač pa na dinamički objekat tipa A čiji je atribut ia=10. Dereferenciranje *pa je poziv pa.operator*() koji vraća referencu na atribut ia objekta na koji ukazuje pa. Referenca je lvalue tako da je moguće izvršiti dodelu vrednosti (20). Na kraju se ispisuje vrednost atributa ia objekta na koji ukazuje pokazivač pa pozivom metode Get, nakon čega se ukida dinamički objekat na koga ukazuje pa.

Operatorska funkcija new, koja se koristi za kreiranje dinamičkog objekta datog tipa, poziva se pre konstruktora date klase. Ova funkcija prvo pokuša da alocira memorijski prostor za dinamički objekat datog tipa i ako uspe, onda vraća pokazivač na taj dinamički objekat, u suprotnom vraća nullptr. Sintaksa poziva globalnog operatora new je kao što sledi:

```
::operator new(velicina_potrebnog_memorijskog_prostora);
```

Omogućeno je da programer postavi novu funkciju new-handler koja je zadužena ako nema memorijskog prostora pri pokušaju kreiranja dinamičkog objekta. To se izvodi postavljanjem nove, korisničke, funkcije tipa void () korišćenjem funkcije *set_new_handler* (zaglavlje <cnew>). Funkcija *set_new_handler* prihvata kao parametar pokazivač na novu funkciju tipa void() (koja je novi *handler*), a vraća stari *handler*. Sintaksa je kao što sledi:

```
void (*set_new_handler( void(*)() ))();
```

Operatorska metoda operator new je statička metoda (ne mora ni da se navede static) koja se poziva pre konstruktora date klase i koja obezbeđuje potreban memeorijski prostor za dinamički objekat koji će se kreirati. Ova metoda ima jedan obavezan argument tipa size_t (zaglavlje <cstddef>) koji se odnosi na veličinu objekta u bajtovima, a ostale proizvoljne argumente može da dodaje programer. Povratna vrednost metode operator new je tipa void*.

Operatorska funkcija operator delete poziva se pri oslobađanju memorijskog prostora kao što sledi:

```
::operator delete(pokazivac_na_prostor_koji_se_oslobadja);
```

Operatorska metoda operator delete je statička metoda (ne mora ni da se navede static) koja će biti pozvana posle destruktora date klase i koja oslobađa memorijski prostor koji je zauzimao dinamički objekat. Obavezni argument ove metode je tipa void*, a neobavezni je tipa size_t. Povratna vrednost metode operator delete je tipa void.

```
#include <cstddef>
class C {
    //...
  public:
```

```
    void* operator new(size_t, double hm1, int hm2);
    void* operator new(size_t vel);
    void operator delete (void *pv);
    //...
};
```

U datom primeru prva deklaracija operatora new ima 3 parametra: prvi je obavezan i to je veličina memorijskog prostora koja je potrebna za dinamički objekat, dok su druga dva parametra one koje kreira programer za svoje potrebe. Poziv bi bio (parametri hm1, hm2 i C za size_t):

```
C* pc = new (hm1, hm2) C;
```

Druga deklaracija operatora new sa 1 parametrom je uobičajena i poziv je:

```
C* pc = new C;
```

Treća deklaracija se odnosi na operator delete i tu se kao parametar prosleđuje pokazivač (adresa) na memorijski prostor koji je potrebno osloboditi.

Navedene metode operator new i operator delete se nasleđuju i ne mogu biti virtuelne.

Operatorska metoda dodele operator= je nestatička metoda koja se ne nasleđuje i koja obavlja dodelu atributa jednog objekta drugom. Ako nema preklopljenog operatora dodele, onda će prevodilac generisati operator dodele.

```
class C {
    int *pi;
  public:
    C(int i) : pi(new int(i)) {}  // konstruktor
    C(const C& cc) : pi(new int(*cc.pi)){} // konstruktor kopije
    C& operator=(const C& cc)  // operator dodele
    {
      if (this==&cc){}         // isti objekat - ne treba nista
      else                     // različit objekat - ide dodela
      {
        delete pi;
        pi=new int(*cc.pi);
      }
      return *this;            // vrati referencu, lvalue
    }
    ~C() { delete pi; }        // destruktor

};
int main ()
{
  C cc1(10), cc2(20);   // poziv konstruktora
  C cc3 = cc2;          // poziv konstruktora kopije
  cc1 = cc2;            // poziv operatora dodele
```

```
    return 0;
}
```

U datom primeru kod operatorske metode dodele povratna vrednost je referenca na objekat date klase jer je poziv oblika aa=bb, dakle, lvalue. U kodu je trivijalno dato da ako je reč o refleksivnoj dodeli (aa=aa) onda se samo prosledi referenca na objekat. Ako je u pitanju dodela gde su dva različita objekta, onda se oslobodi prostor koji je zauzimao prvi objekat, dodeli mu se novi memorijski prostor, prvi objekat se inicijalizuje vrednošću drugog objekta i na kraju se vrati referenca na prvi objekat.

Operatorska metoda konverzije operator TIP() je unarna operacija koja nema povratni tip i omogućuje konverziju (implicitno i eksplicitno) objekta date klase u objekat datog tipa (ugrađeni ili korisnički). Ova operatorska metoda se nasleđuje i može biti virtuelna.

```
//...
class Automobil{
    string naziv;
    double cena;
  public:
    //...
    Automobil(string naziv,double cena)
                                        :naziv(naziv),cena(cena){}
    operator double() { return cena; }
    //...
};
int main() {
  Automobil audi("Audi A3",25000.0);
  double d1=audi;
  double d2=static_cast<double>(audi);
  return 0;
}
```

U datom primeru u funkciji main kreiran je objekat audi tipa Automobil sa atributima "Audi A3" i 25000.0. Nakon ovoga, izvršena je implicitna konverzija tipa Automobil u tip double, a potom i eksplicitna konverzija iz tipa Automobil u tip double.

Data je tabela 3.7.1. u kojoj su navedeni elementi koji se ne nasleđuji, koji nemaju povratni tip, koji ne nogu biti virtuelni.

Tabela 3.7.1. Elementi koji se ne nasleđuju, koji nemaju povratni tip, koji nisu virtuelni

Ne nasleđuje se	konstruktor destruktor operator = prijatelji

Nema povratni tip	konstruktor destruktor konverzija
Ne može biti virtuelan	konstruktor operator new operator delete prijatelji

Operatorska metoda operator>> klase istream (zaglavlje <iostream>) je namenjena (preklopljena) za unos vrednosti za sve ugrađene tipove. Za korisničke tipove potrebno je preklopiti takav operator ili kao prijateljsku operatorsku funkciju ili kao operatorsku metodu date klase. Objektu klase istream pridružuje se jedna datoteka za ulaz. Deklaracija operatorske metode operator>> klase istream je kao što sledi:

```
istream& operator>>(ugradjenitip &referenca_na_ugradjeni_tip);
```

Biblioteka iostream ima globalni statički objekat cin klase istream kome je pridružen standardni ulazni uređaj. Objekat cin je u prostoru imena std. Sintaksa korišćenja je kao što sledi:

```
int ii;
std::cin>>ii;  // unos vrednosti ii sa tastature
```

Operatorska metoda operator<< klase ostream (zaglavlje <iostream>) je namenjena (preklopljena) za ispis vrednosti za sve ugrađene tipove. Za korisničke tipove potrebno je preklopiti takav operator ili kao prijateljsku operatorsku funkciju ili kao operatorsku metodu date klase. Objektu klase ostream pridružuje se jedna datoteka za izlaz. Deklaracija operatorske metode operator<< klase ostream je kao što sledi:

```
ostream& operator<<(ugradjenitip ugrtip);
```

Biblioteka iostream ima globalni statički objekat cout klase ostream kome je pridružen standardni izlazni uređaj. Objekat cout je u prostoru imena std. Sintaksa korišćenja je kao što sledi:

```
int ii;
std::cout<<ii;  // slanje vrednosti ii na ekran
```

Ono što je zajedničko za operatorske metode operator>> i operator<< je da su binarne operacije gde je prvi operand referenca na pripadni tok, a drugi operand je referenca na tip za operator>>, odnosno, objekat tipa koji se ispisuje za operator<<. Pošto ove metode vraćaju referencu na pripadni tok onda je moguće nataviti sa sledećom operacijom učitavanja ili ispisa (grupišu sa leva na desno):

```
  int i1, i2, i3;
  std::cin >>i1>>i2>>i3;
  std::cout<<i1<<i2<<i3;
```

U primeru na slici 3.7.4. dato je zaglavlje klase KompleksanBroj. Data je direktiva da se zaglavlje jednom uključi u program, a onda su uključena zaglavlja: cmath (matematička biblioteka), iostream (za učitavanje i ispis) i fstream (za rad sa datotekama). Klasa KompleksanBroj ima dva privatna atributa tipa double za smeštanje realne i imaginarne vrednosti kompleksnog broja. Javni konstruktor koji se može pozvati bez argumenata ima podrazumevane argumente za atribute klase KompleksanBroj postavljene na 0. Sledi deklaracija operatorskih metoda: za binarne operacije sabiranja (+), oduzimanja (-), množenja (*), deljenja (/) kompleksnih brojeva, za unarnu operaciju vraćanje konjugovano-kompleksnog broja (!) te operatorske prijateljske funkcije operator<< za ispis na ekran i operator<< za ispis u fajl.

```
//_____________________KompleksanBroj.h
#pragma once
#include <cmath>
#include <iostream>
#include <fstream>
using namespace std;
class KompleksanBroj
{
  private:
    double real;
    double imag;
  public:
    KompleksanBroj(double=0,double=0);
    KompleksanBroj operator +(KompleksanBroj);
    KompleksanBroj operator -(KompleksanBroj);
    KompleksanBroj operator *(KompleksanBroj);
    KompleksanBroj operator /(KompleksanBroj);
    KompleksanBroj operator !();
                //operator ! koristice se za dobijanje
                //konjugovano kompleksnog broja, za a+bi to je a-bi
    friend ostream&  operator <<(ostream &s,KompleksanBroj &c);
                //za ispis na ekran
    friend ofstream& operator <<(ofstream &s,KompleksanBroj &c);
                //za ispis u fajl
};
```

Slika 3.7.4. Zaglavlje klase KompleksanBroj sa preklapnjem operatora

Na slici 3.7.5. dat je sadržaj implementacionog fajla KompleksanBroj.cpp. Uključeno je zaglavlje KompleksanBroj.h da bi se znalo koja imena u klasi KompleksanBroj postoje i šta znače. Dat je konstruktor koji inicijalizuje realni i

imaginarni deo kompleksnog broja. Slede definicije operatorskih metoda za sabiranje (a+bi + c+di = a+c +i(b+d)), oduzimanje (a+bi – (c+di) = a-c +i(b-d)), množenje ((a+bi)*(c+di) = ac-bd +i(ad+bc)) deljenje ((a+bi)/(c+di) = ((ac+bd)+i(bc-ad))/(c^2+d^2)) i vraćanje konjugovano kompleksne vrednosti (a+bi => a-bi). Jedan operand u navedenim operacijama je objekat čija je metoda pozvana. Kod operacije deljenja se ispituje da li je delilac jednak nuli te ako jeste nasilno se izlazi iz programa pozivom funkcije exit(EXIT_FAILURE).

```
//_____________________KompleksanBroj.cpp
#include "KompleksanBroj.h"
KompleksanBroj::KompleksanBroj(double real, double imag)
:
real(real), imag(imag)
{}
KompleksanBroj KompleksanBroj::operator+
                                    (KompleksanBroj drugi_operand)
{
  return KompleksanBroj(this->real + drugi_operand.real,
                this->imag + drugi_operand.imag) ;
}
KompleksanBroj KompleksanBroj::operator-
                                    (KompleksanBroj drugi_operand)
{
  return KompleksanBroj(this->real - drugi_operand.real,
                this->imag - drugi_operand.imag) ;
}
KompleksanBroj KompleksanBroj::operator*
                                    (KompleksanBroj drugi_operand)
{
  return KompleksanBroj((this->real * drugi_operand.real) -
                (this->imag * drugi_operand.imag) ,
                (this->real * drugi_operand.imag) +
                (this->imag * drugi_operand.real) );
}
KompleksanBroj KompleksanBroj::operator/
                                    (KompleksanBroj drugi_operand)
{
  double delilac = pow(drugi_operand.real,2) +
                   pow(drugi_operand.imag,2);
  if(0==delilac) {
    cout<<"Delilac Vam je 0 !!!"<<endl;
    cout<<"Pritisnite taster za izlazak iz programa..."<<endl;
    system("pause");
    exit(EXIT_FAILURE);
  }
  return KompleksanBroj(
```

```
    ( (this->real * drugi_operand.real) +
      (this->imag * drugi_operand.imag) ) / delilac,
    ( (this->imag * drugi_operand.real) -
      (this->real * drugi_operand.imag) ) / delilac );
}
KompleksanBroj KompleksanBroj::operator!()
{
  return KompleksanBroj(this->real, -this->imag);
}
```

Slika 3.7.5. Implementacija klase KompleksanBroj sa preklapanjem operatora

Na slici 3.7.6. dat je sadržaj fajla gde je funkcija main. Definisane su operatorske funkcije za ispis kompleksnog broja na ekran i u fajl. Ove operatorske funkcije imaju dva argumenta: prvi je referenca na tok (*stream*), a drugi je referenca na obejkat klase KompleksanBroj. Ove operatorske funkcije vraćaju referencu na prihvaćeni tok. Ispis atributa kompleksnog broja ima oblik na dve decimale (setprecision(2) i fixed). Ispisaće se realni deo kompleksnog broja (bez znaka + ako je pozitivan, noshowpos) te imaginarni deo kompleksnog broja sa znakom (showpos za ispis znaka +) i na kraju imaginarna jedinica uz uslov da je imaginarni deo različit od nule. U komentaru je dat kod za ispis atributa kompleksnog broja u obliku uređenog para (real, imag).

```
//_____________________fajl gde je main
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
using namespace std;
#include "KompleksanBroj.h"
// ispis na ekran
ostream& operator<<(ostream& os, KompleksanBroj& c)
{
  //format ispisa:  Re + Im * i
  os<<setprecision(2)<<fixed;
  os<<noshowpos<<c.real;
  if(c.imag != 0) os<<showpos  <<c.imag<<"i";
  //os<<"("<<c.real<<","<<c.imag<<")"<<endl; // ispis (Re,Im)
  return os;
}

// ispis u fajl
ofstream& operator<<(ofstream& os, KompleksanBroj& c)
{
  //OBLIK:  Re + Im*i
  os<<setprecision(2)<<fixed;
  os<<noshowpos<<c.real;
```

```
  if(c.imag != 0) os<<showpos  <<c.imag<<"i";
  return os;
}
int main()
{
  KompleksanBroj c1(10.0,-10.0),c2(5.0,6.0),c3;
  //KompleksanBroj c1(10.0,-10.0),c2,c3; // da bi delilac=0
  c3 = c1+c2; cout<<"("<<c1<<")+("<<c2<<")="<<c3<<endl;
  c3 = c1-c2;  cout<<"("<<c1<<")-("<<c2<<")="<<c3<<endl;
  c3 = c1*c2; cout<<"("<<c1<<")*("<<c2<<")="<<c3<<endl;
  c3 = c1/c2; cout<<"("<<c1<<")/("<<c2<<")="<<c3<<endl;
  cout<<"Konjugovana vrednost "<<c1<<" je "<<(!c1)<<endl;

  string strulaz="ulaz.txt";//ili std::string
  ifstream infile (strulaz,ifstream::binary); //ili std::ifstream
  if (infile.is_open())
  {
    cout<<"sadrzaj fajla "<<strulaz<<" je :"<<endl;
    char c = infile.get();
    while (infile.good()) {
      cout << c;   c = infile.get();
    }
    cout<<endl;
  }
  else
  {
    cout<<"Ne moze se otvoriti "<<strulaz<<endl;
    system("pause"); return 1;
  }
  string strizlaz="izlaz.txt";
  ofstream outfile(strizlaz,ofstream::binary);
  if(outfile.is_open()==0){
    cout<<"Ne moze se otvoriti "<<strizlaz<<endl;
    system("pause"); return 1; }
  infile.clear();
  infile.seekg (0,ios::end);  //ili infile.end
  long size = infile.tellg();//velicina fajla ulaz.txt
  infile.seekg (0,infile.beg);
  char* buffer = new char[size];
    //dinamicka alokac. za sadrzaj ulaz.txt
  infile.read (buffer,size);
    //ucitavanje sadrzaja ulaz.txt u buffer
  outfile.write (buffer,size);
    //snimanje ucitanog sadrzaja u izlaz.txt
  delete[] buffer; //da oslobodite dinamicki blok
  outfile<<c1<<endl;
  outfile.write ("Ovo je kraj    OOPa.",20);
```

```
  long pos = outfile.tellp();
  outfile.seekp (pos-13);
  outfile.write ("pocetak",7);
  outfile.close(); infile.close(); system("pause"); return 0;
}

_________________________________ FAJL ulaz.txt ____________________
Objektno orijentisano programiranje.

_________________________________I Z L A Z_____________________________
(10.00-10.00i)+(5.00+6.00i)=15.00-4.00i
(10.00-10.00i)-(5.00+6.00i)=5.00-16.00i
(10.00-10.00i)*(5.00+6.00i)=110.00+10.00i
(10.00-10.00i)/(5.00+6.00i)=-0.16-1.80i
Konjugovana vrednost 10.00-10.00i je 10.00+10.00i
sadrzaj fajla ulaz.txt je :
Objektno orijentisano programiranje.

_________________________________ FAJL izlaz.txt ____________________
Objektno orijentisano programiranje.
10.00-10.00iOvo je pocetak OOPa.
Press any key to continue . . .
```

Slika 3.7.6. Korišćenje izlaznog toka na primeru klase KompleksanBroj

U funkciji main kreirana su tri kompleksna broja c1, c2 i c3, a onda je kompleksnom broju c3 dodeljen kompleksan broj dobijen operacijama sabiranja, oduzimanja, množenja i deljenja kompleksnih brojeva c1 i c2 redom. Ispisuje se konjugovana vrednost kompleksnog broja c1.

Kreira se objekat infile klase ifstream koji se odnosi na fajl "ulaz.txt" koji dati fajl posmatra binarno. Proverava se da li je fajl otvoren i ako jeste, onda se uzima znak (bajt) iz fajla i proverava da li je došlo do kraja fajla (eofbit) ili se desila logička greška (failbit) ili se desila greška pri čitanju (badbit). Ako je sve dobro prošlo (metoda good()), onda se u petlji ispisuje znak, ponovo učitava znak i proverava da li je čitanje bilo ispravno i tako do kraja fajla. Ako fajl nije otvoren ispisuje se poruka i nakon pauze program završava rad.

Kreira se objekat outfile tipa ofstream koji se odnosi na fajl "izlaz.txt" koji se posmatra binarno. Ako fajl nije otvoren, onda se ispisuje poruka i nakon pauze program završava rad. Ako je fajl uspešno otvoren poziva se metoda clear() za ulazni fajl "ulaz.txt" koja briše bilo koji bit greške (preporučujem), postavlja se get pointer fajla na kraj ulaznog fajla (`infile.seekg (0,ios::end);`), a onda uzima vrednost get pointera ulaznog fajla (`infile.tellg()`) i dodeljuje promenljivoj size koja odgovara veličini fajla. Sada se kreira dinamički niz (bafer) veličine size. Učitava se ulazni fajl u bafer (`infile.read (buffer,size);`), a onda se sadržaj tog bafera smešta u izlazni fajl

(`outfile.write (buffer,size);`). Nakon ovoga oslobađa se dinamički blok (`delete[] buffer;`) te upisuje u izlazni fajl kompleksan broj c1, a onda i `20` karaktera stringa `"Ovo je kraj OOPa."`. Uzima se put pointer fajla (`outfile.tellp();`) i dodeljuje promenljivoj `pos`. Put pointer fajla pomera se za 13 mesta manje od trenutne pozicije (`outfile.seekp (pos-13);`) i od te pozicije upisuje 7 karaktera stringa `"pocetak"`. Na samom kraju se zatvaraju tokovi na ulazni i na izlazni fajl, pravi pauza i izlazi iz programa.

3.8. Nasleđivanje

Nasleđivanje se izvodi tako da se u zaglavlju izvedene klase nakon naziva te klase stavi znak dvotačka (:), a onda se napiše lista klasa iz kojih se izvodi izvedena klasa. Pored ovoga potrebno je pre izvedene klase navesti potpunu deklaraciju datih osnovnih klasa. Ako iza znaka dvotačka nije naveden način izvođenja onda se smatra se da se radi o privatnom izvođenju.
Ne nasleđuju se: konstruktor, destruktor, operatorska metoda operator = i prijatelji.

Postoje tri načina izvođenja klasa iz osnovnih klasa: javni (public), zaštićeni (protected) i privatni (private).

3.8.1. Javno izvođenje

Javnim izvođenjem:

- javni članovi osnovne klase postaju javni članovi izvedene klase;
- zaštićeni članovi osnovne klase postaju zaštićeni članovi izvedene klase;
- privatni članovi osnovne klase su direktno nedostupni iz izvedene klase.

Javnim izvođenjem izvedena klasa je vrsta osnovne klase (*a kind of*).
Primer:

```
class Osnovna {
  private:
    int iosnpri;
    int fosnpri();
  protected:
    int iosnzas;
    int fosnzas();
  public:
    int iosnjav;
    int fosnjav();
};
class Izvedena : public Osnovna
{
  public:
    void f()
```

```
    {
      iosnjav++;        // je javni član i u izvedenoj klasi
      int j = fosnjav();// je javni član i u izvedenoj klasi
      iosnzas++;        // je zaštićeni član i u izvedenoj klasi
      int j = fosnzas();// je zaštićeni član i u izvedenoj klasi
      // nema direktnog pristupa privatnom članu iosnpri
      // nema direktnog pristupa privatnom članu fosnpri
    }
};
```

3.8.2. Zaštićeno izvođenje

Zaštićenim izvođenjem:

- javni članovi osnovne klase postaju zaštićeni članovi izvedene klase;
- zaštićeni članovi osnovne klase postaju zaštićeni članovi izvedene klase;
- privatni članovi osnovne klase su direktno nedostupni iz izvedene klase.

```
class Osnovna {
  private:
    int iosnpri;
    int fosnpri();
  protected:
    int iosnzas;
    int fosnzas();
  public:
    int iosnjav;
    int fosnjav();
};
class Izvedena : protected Osnovna
{
  public:
    void f()
    {
      iosnjav++;        // je zaštićeni član u izvedenoj klasi
      int j = fosnjav();// je zaštićeni član u izvedenoj klasi
      iosnzas++;        // je zaštićeni član i u izvedenoj klasi
      int j = fosnzas();// je zaštićeni član i u izvedenoj klasi
      // nema direktnog pristupa privatnom članu iosnpri
      // nema direktnog pristupa privatnom članu fosnpri
    }
};
```

3.8.3. Privatno izvođenje

Privatnim izvođenjem:

- javni članovi osnovne klase postaju privatni članovi izvedene klase;
- zaštićeni članovi osnovne klase postaju privatni članovi izvedene klase;

- privatni članovi osnovne klase su direktno nedostupni iz izvedene klase.

Privatnim izvođenjem osnovna klasa je deo izvedene klase (*a part of*).

```
class Osnovna {
  private:
    int iosnpri;
    int fosnpri();
  protected:
    int iosnzas;
    int fosnzas();
  public:
    int iosnjav;
    int fosnjav();
};
class Izvedena : protected Osnovna
{
  public:
    void f()
    {
      iosnjav++;         // je privatni član u izvedenoj klasi
      int j = fosnjav();// je privatni član u izvedenoj klasi
      iosnzas++;         // je privatni član u izvedenoj klasi
      int j = fosnzas();// je privatni član u izvedenoj klasi
      // nema direktnog pristupa privatnom članu iosnpri
      // nema direktnog pristupa privatnom članu fosnpri
    }
};
```

Razlikuju se:
- direktna osnovna klasa (navedena u listi deklaracija izvedene klase)
- indirektna osnovna klasa (je osnovna klasa neke od direktnih osnovnih klasa)

Pristupanje članu osnovne klase može se napisati i eksplicitno kao što sledi:

```
OsnovnaKlasa::nazivclanaosnovneklase
```

što je podesno ako je naziv člana izvedene klase isti kao i naziv člana osnovne klase (skrivanje naziva).
Primer:

```
class A
{
  public:
    int f(int);
};
class B : public A
{
  public:
```

```
    void f(int ii)
    {
      if(ii==1) return;
      int i = A::f();
    }
};
```

Dozvoljeno je samo da se vrati originalno pravo pristupa članu osnovne klase koje je datim izvođenjem promenjeno u izvedenoj klasi kao što sledi:

```
class Osnovna {
  private:
    int iosnpri;
    int fosnpri();
  protected:
    int iosnzas;
    int fosnzas();
  public:
    int iosnjav;
    int fosnjav();
};
class Izvedena : private Osnovna
{
  public:
    Osnovna::iosnjav;
    Osnovna::fosnjav;
  protected:
    Osnovna::iosnzas;
    Osnovna::fosnzas;
};
```

Dostupnost osnovne klase podrazumeva da su u tom mestu koda dostupni njeni javni članovi. Kod privatnog izvođenja osnovna klasa je dostupna samo u metodama izvedene klase (i prijateljima izvedene klase). Kod privatnog izvođenja ako se "gleda" van izvedene klase ne "vidi" se da je data osnovna klasa deo izvedene klase.

Pravilo je da se pokazivač koji ukazuje na objekat izvedene klase može konvertovati u pokazivač koji ukazuje na objekat osnovne klase ako je osnovna klasa na tom mestu koda dostupna.

Primer:

```
class Osnovna {...};
class JavnoIzvedena  : public  Osnovna {...};
class PrivatnoIzvedena : private Osnovna {...};
int main () {
  JavnoIzvedena javizv;
  Osnovna *posn1 = &javizv;  // OK, dostupnost osnovne klase
```

```
    PrivatnoIzvedena priizv;
    // ne bi moglo Osnovna *posn2 = &priizv; jer je osnovna klasa
    // nedostupna iz ovog dela koda
    return 0;
  }
```

Prema dostupnosti, ako je izvođenje javno, onda je moguće konvertovati pokazivač koji ukazuje na objekat izvedene klase u pokazivač koji ukazuje na objekat osnovne klase. Obratna konvezija nije dozvoljena.

Primer:

```
class Osnovna {...};
class JavnoIzvedena : public Osnovna {...};
int main()
{
  JavnoIzvedena javizv;
  JavnoIzvedena *pjavizv = &javizv;
  Osnovna *posn = pjavizv;          // OK, dostupnost
  Osnovna osn;
  Osnovna *posn2 = &osn;
  // nije dozvoljeno JavnoIzvedena *pjavizv2 = posn2;
  return 0;
}
```

Prema dostupnosti, ako je izvođenje privatno, onda je konverzija pokazivača koji ukazuje na objekat izvedene klase u pokazivač koji ukazuje na objekat osnovne klase moguća samo unutar metoda izvedene klase ili prijatelja izvedene klase.

Primer:

```
class Osnovna {...};
class PrivatnoIzvedena : private Osnovna
{
  ...
  void f()
  {
    PrivatnoIzvedena *ppriizv = this;
    Osnovna *posn = pprivizv;            // OK, dostupnost
    ...
  }
};
int main()
{
```

```
  PrivatnoIzvedena priizv;
  PrivatnoIzvedena *ppriizv = &priizv;
  // nije dozvoljeno Osnovna *posn = pprivizv; //NOK, nedostupnost
  return 0;
}
```

Prema dostupnosti, ako je osnovna klasa dostupna, onda se može objekat osnovne klase inicijalizovati objektom pripadne izvedene klase.
Primer:

```
class Osnovna {...};
class JavnoIzvedena : public Osnovna {...};
int main()
{
  JavnoIzvedena javizv;
  Osnovna osn = javizv;  // izvedena je tipa osnovna
  return 0;
}
```

3.8.4. Kreiranje i ukidanje objekta izvedene klase

Pozivanjem adekvatnog konstruktora (razrešavanje poziva) izvedene klase, u inicijalizatoru se poziva i odgovarajući konstruktor osnovne klase prema razrešavanju poziva kojim se inicijalizuje podobjekat osnovne klase. Redosled inicijalizacije članova izvedene klase odgovara redosledu njihove deklaracije. Nakon navedene inicijalizacije sledi izvršavanje tela konstruktora izvedene klase.
Primer 1:

```
class Tacka2D {
  double x;
  double y;
public:
  Tacka2D(double, double);
};
Tacka2D::Tacka2D(double x,double y)
:
x(x),
y(y)
{}
class Tacka3D : public Tacka2D
{
    double z;
  public:
    Tacka3D(double, double, double);
};
```

```
Tacka3D::Tacka3D (double x, double y, double z)
:
Tacka2D(x,y),
z(z)
{}
```

U datom primeru klasa Tacka2D je osnovna klasa klase Tacka3D. Izvođenje je javno. Redosled poziva pri kreiranju objekta klase Tacka3D je da se prvo zove konstruktor podobjekta osnovne klase Tacka2D gde se inicijalizuju članovi x i y, zatim sledi inicijalizacija člana z u izvedenoj klasi i na kraju se izvršava telo konstruktora izvedene klase Tacka3D.

Redosled poziva pri ukidanju objekta izvedene klase je :

- destruktor izvedene klase;
- destruktori članova redosledom suprotnim od redosleda kojim su inicijalizovani;
- destruktor osnovne klase.

Primer:

```
#include <iostream>
using namespace std;
class Ugradjeni_1 {
  public:
    Ugradjeni_1()  {cout<<"Konstruktor: Ugradjeni_1"<<endl;}
    ~Ugradjeni_1() {cout<<"Destruktor : Ugradjeni_1"<<endl;}
};
class Ugradjeni_2 {
  public:
    Ugradjeni_2()  {cout<<"Konstruktor: Ugradjeni_2"<<endl;}
    ~Ugradjeni_2() {cout<<"Destruktor : Ugradjeni_2"<<endl;}
};
class Osnovna {
  public:
    Osnovna ()   {cout<<"Konstruktor: Osnovna"<<endl;}
    ~ Osnovna () {cout<<"Destruktor : Osnovna"<<endl;}
};
class Izvedena : public Osnovna {
    Ugradjeni_2 ugr2;
    Ugradjeni_1 ugr1;
  public:
    Izvedena ()   {cout<<"Konstruktor: Izvedena"<<endl;}
    ~ Izvedena () {cout<<"Destruktor : Izvedena"<<endl;}
};
int main ()
{
  cout<<"----- Konstrukcija -----"<<endl;
  Izvedena izv;
```

```
  cout<<"----- Destrukcija  -----"<<endl;
  izv.~Izvedena();
  system("pause");
  return 0;
}
```

```
Izlaz:
----- Konstrukcija -----
Konstruktor: Osnovna
Konstruktor: Ugradjeni_2
Konstruktor: Ugradjeni_1
Konstruktor: Izvedena
----- Destrukcija ------
Destruktor : Izvedena
Destruktor : Ugradjeni_1
Destruktor : Ugradjeni_2
Destruktor : Osnovna
```

U datom primeru se u funkciji main kreiranjem objekata klase Izvedena pozivaju se redom kao što sledi:

- konstruktor osnovne klase Osnovna, čime se kreira podobjekat izvedene klase Izvedena;
- konstruktor za ugrađeni objekat klase Ugradjeni_2 za inicijalizaciju objekta ugr2, jer je on na redu prema deklaraciji u zaglavlju klase Izvedena;
- konstruktor za ugrađeni objekat klase Ugradjeni_1 za inicijalizaciju objekta ugr1, jer je on na redu prema deklaraciji u zaglavlju klase Izvedena;
- telo konstruktora klase Izvedena.

Nakon ovoga u kodu sledi eksplicitni poziv destruktora za kreirani objekat klase Izvedena (dat je primer sintakse takvog poziva) koji poziva redom kao što sledi:

- destruktor izvedene klase Izvedena;
- zatim se poziva destruktor klase
- destruktor za ugrađeni objekat klase Ugradjeni_1 za inicijalizaciju objekta ugr1 jer se ide unazad u odnosu na deklaraciju u zaglavlju klase Izvedena;
- konstruktor za ugrađeni objekat klase Ugradjeni_2 za inicijalizaciju objekta ugr2, jer se ide unazad u odnosu na deklaraciju u zaglavlju klase Izvedena;
- destruktor osnovne klase Osnovna.

Strukture podataka osnovne klase, njene izvedene klase i redom dalje izvedene klase počinju na istoj memorijskoj lokaciji jer se dodatni atributi izvedene klase dodaju iza postojećih atributa osnovne klase i tako redom do poslednje izvedene klase u tom nizu. Ovim je jasno zašto se konverzija pokazivača koji ukazuje na objekat izvedene klase može konvertovati u pokazivač koji ukazuje na objekat osnovne klase i tako redom do "prve" osnovne klase.

3.8.5. Višetruko izvođenje

Ako je neka klasa izvedena iz dve klase koje imaju istu osnovnu klasu, onda postoje dva podobjekta te osnovne klase u izvedenoj klasi. Navođenje samo podobjekta osnovne klase je dvosmisleno jer se ne zna o kom podobjektu od dva je reč. Da bi se ovo izbeglo mora se eksplicitno navesti o kom podobjektu se radi (kojom granom izvođenja se do njega dolazi).

Primer:

```
class Osnovna
{
  public:
    int iosn;
};
class Izvedena_1 : public Osnovna {};
class Izvedena_2 : public Osnovna {};

class Izvedena_1_2 : public Izvedena_1, public Izvedena_2
{
  public:
    void povecaj_iosn();
};
void Izvedena_1_2::povecaj_iosn()
{
   Izvedena_1::iosn++;//1. podobjekat preko grane Izvedena_1
   Izvedena_2::iosn++;//2. podobjekat preko grane Izvedena_2
   //ne može iosn++; jer se ne zna koji je od dva podbojekta
}
```

Napomena: U deklaraciji izvođenja jedna direktna osnovna klasa se može navesti samo jednom.

Sada je potrebno za prethodni slučaj sa zajedničkom indirektnom osnovnom klasom napisati pravilnu konverziju pokazivača koji ukazuje na objekat izvedene klase Izvedena_1_2 u pokazivač koji ukazuje na objekat indirektne osnovne klase Osnovna:

```
Izvedena_1_2 *pizvedena_1_2;

Osnovna *posnovna =
  dynamic_cast<Osnovna*>
    (dynamic_cast<Izvedena_1*>(pizvedena_1_2));
    //za podobjekat preko Izvedena_1

Osnovna *posnovna =
  dynamic_cast<Osnovna*>
    (dynamic_cast<Izvedena_2*>(pizvedena_1_2));
    //za podobjekat preko Izvedena_2
```

3.8.6. Virtuelne klase

Ako se želi jedinstvenost podobjekta indirektne osnovne klase, onda se mora u deklaraciji navesti da je ta osnovna klasa virtuelna.
Primer:

```
class Osnovna
{
  public:
    int iosn;
};
class Izvedena_1 : virtual public Osnovna {};
class Izvedena_2 : virtual public Osnovna {};

class Izvedena_1_2 : public Izvedena_1, public Izvedena_2
{
  public:
    void povecaj_iosn();
};
void Izvedena_1_2::povecaj_iosn()
{
  Izvedena_1::iosn++;  // podobjekat preko grane Izvedena_1
  Izvedena_2::iosn++;  // podobjekat preko grane Izvedena_2
  iosn++; // može jer je jedan podobjekat
          // klase Osnovna za klase Izvedena_1 i Izvedena_2
}
```

Pravilo domincije se odnosi na pristupanje članovima koji imaju isti naziv, koji su dostupni, ali se nalaze u različitom nivou hijerarhije izvođenja. U tom slučaju se ne radi o dvosmislenosti poziva, već se koristi pravilo dominacije po kome će se pozvati onaj član koji je u hijerarhiji na višoj poziciji.
Primer:

```
#include <iostream>
using namespace std;
class Osnovna
{
  public:
    void f(){cout<<"Osnovna, f"<<endl;}
};
class Izvedena_1 : public virtual Osnovna
{
  public:
      void f(){cout<<"Izvedena_1, f"<<endl;}
};
class Izvedena_2 : public Izvedena_1, public virtual Osnovna
{
  public:
```

```
    void Pozovi_f() {
      f();                          // Izvedena_1, f
      Izvedena_1::Osnovna::f();     // Osnovna, f
      Osnovna::f();                 // Osnovna, f
    }
};
int main()
{
    Izvedena_2 iz2;
    iz2.Pozovi_f();
    return 0;
}
```

U datom primeru je hijerarhija takva da metoda f klase Izvedena_1 dominira metodom f klase Osnovna. Ovim će se poziv f() u metodi Pozovi_f klase Izvedena_2 odnositi na poziv metode f u klasi Izvedena_1. Iza toga navedeni su eksplicitni pozivi metode f u klasama Izvedena_1 i Osnovna.

Prvo se inicijalizuju virtuelne osnovne klase, a onda nevirtuelne osnovne klase. Ovim je rešeno kako inicijalizovati jedan zajednički podobjekat zbog virtuelnih klasa.

Na slici 3.8.1 dat je slučaj nasleđivanja koji ima strukturu dijamanta (diamond). Klasa JMBG (jedinstveni matični broj) ima konstruktor koji postavlja jmbg javni član. Klasa Otac i klasa Majka javno nasleđuju virtuelnu klasu JMBG i u pripadnom konstruktoru postavljaju ime oca, odnosno, ime majke te pripadni JMBG podobjekat. Klasa Dete je javno izvedena iz klasa Otac i Majka i u svom konstruktoru kreira podobjekte Otac, Majka, ali i virtuelni podobjekat JMBG. Klasa Dete poseduje metodu UzmiJMBG tipa char* (void) koja vraća jmbg član podobjekta tipa JMBG.

```
//JMBG-OTAC_MAJKA_DETE
#include <iostream>
using namespace std;
class JMBG
{
  public:
      JMBG(char* jmbg): jmbg(jmbg){}
      char* jmbg;
};
class Otac: public virtual JMBG
{
  public:
      Otac(char *ime, char* jmbg): JMBG(jmbg), ime(ime){}
      char *ime;
};
class Majka: public virtual JMBG
```

```
{
  public:
      Majka(char *ime, char* jmbg): JMBG(jmbg), ime(ime){}
      char *ime;
};
class Dete: public Otac, public Majka
{
  public:
      Dete(char* ime, char* imeoca, char* imemajke, char* jmbg)
      :
      Otac(imeoca, jmbg),
      Majka(imemajke,jmbg),
      JMBG("007")
      {}
      char* UzmiJMBG(){return jmbg;}
};
int main ()
{
  Dete ja("Perica","Svetozar","Mara","0101968");
  cout<<ja.UzmiJMBG()<<endl;
  return 0;
}
```

Slika 3.8.1. Nasleđivanje koje čini strukturu dijamanta

U funkciji main se kreira objekat tipa Dete čijem konstruktoru se prosleđuju argumenti za ime deteta, ime oca, ime majke i jmbg deteta. U inicijalizatoru se inicijalizuje jedinstveni virtuelni podobjekat JMBG sa parametrom "007", tako da ni podobjekat tipa Otac niti podobjekat tipa Majka neće inicijalizovati zajednički podobjekat tipa JMBG. Izlaz će biti "007".

Inicijalizacija više direktnih osnovnih klasa je navedena u inicijalizatoru u zaglavlju konstruktora izvedene klase, pri čemu redosled poziva konstruktora direktnih osnovnih klasa odgovara redosledu njihovih deklaracija u zaglavlju izvedene klase. Nakon poziva konstruktora direktnih osnovnih klasa, sledi inicijalizacija članova izvedene klase (poziv odgovarajućih konstruktora ako je u pitanju ugrađivanje) i na kraju izvršavanje tela konstruktora izvedene klase. Primer:

```
class Osnovna_1
{
    int iosn_1;
  public:
    Osnovna_1(int i): ison_1(i){}
};
class Osnovna_2
{
```

```
    int iosn_2;
  public:
    Osnovna_2(int i): ison_2(i){}
};
class Izvedena : public Osnovna_1, public Osnovna_2
{
    int iizv;
  public:
    Izvedena(int i)
    :
    Osnovna_2(i),
    Osnovna_1(i),   // ili Osnovna_1(i),Osnovna_2(i)
    iizv(i)
    {}
};
// redosled bi bio
// podobjekat Osnovna_1, podobjekat Osnovna_2, iizv, Izvedena
```

3.8.7. Polimorfizam

Polimorfizam u smislu paradigme "jedan interfejs, a više metoda" može se posmatrati u vreme prevođenja i u vreme izvršavanja programa. U vreme prevođenja ova paradigma je podržana preklapanjem metoda i preklapanjem operatora. U vreme izvršavanja ova paradigma je podržana vurtualnim metodama. Polimorfizam je mehanizam koji omogućuje da se pri pozivu virtuelne metode osnovne klase kojoj se pristupa preko pokazivača ili reference na objekat izvedene klase poziva odgovarajuća redefinisana virtuelna metoda izvedene klase. Ova metoda izvedene klase je takođe virtuelna i nadjačava (*overrides*) virtuelnu metodu osnovne klase, pri čemu mora biti istog naziva i istog tipa kao nadjačana metoda. Ne postoji virtuelna funkcija, već samo metoda. Virtuelna metoda pripada objektu i može biti prijateljica drugoj klasi.
Primer:

```
#include <iostream>
using namespace std;
class Poligon {
  protected:
    double sirina;
    double visina;
  public:
    void Postavi (double s, double v)
    {
      sirina = s;
      visina = v;
    }
```

```
    virtual double Povrsina ()
    {
      return 0;
    }
    virtual void StaJe ()
    {
      cout<<"Poligon"<<endl;
    }
};
class Trougao: public Poligon {
  public:
    double Povrsina ()
    {
      return (sirina * visina / 2.0);
    }
    void StaJe ()
    {
      cout<<"Trougao"<<endl;
    }
};
class Pravougaonik: public Poligon {
  public:
    double Povrsina ()
    {
      return sirina * visina;
    }
    void StaJe ()
    {
      cout<<"Pravougaonik"<<endl;
    }
};
void PrikaziStaJe(Poligon* ppoligon)
{
  ppoligon->StaJe();
}
int main () {
  Poligon *ppoligon0 = new Poligon;
  ppoligon0->Postavi(5.0,6.0);
  cout<<ppoligon0->Povrsina()<<endl; // 0
  PrikaziStaJe(ppoligon0);           // Poligon
  delete ppoligon0;
  Poligon *ppoligon3 = new Trougao;
  ppoligon3-> Postavi(5.0,10.0);
  cout<<ppoligon3->Povrsina()<<endl; // 25
  PrikaziStaJe(ppoligon3);           // Trougao
  Poligon *ppoligon4 = new Pravougaonik;
  ppoligon4-> Postavi(10.0,20.0);
```

```
  cout<<ppoligon4->Povrsina()<<endl; // 200
  PrikaziStaJe(ppoligon4);           // Pravougaonik
  return 0;
}
```

U datom primeru navedene su deklaracije i definicije klasa: Poligon, Trougao i Pravougaonik. Osnovna klasa Poligon ima dva privatna atributa širinu i visinu tipa double, javnu metodu Postavi tipa void(double,double), virtuelnu metodu Povrsina tipa double() i virtuelnnu metodu StaJe tipa void(). Metoda Postavi setuje atribute klase. Virtuelna metoda Površina vraća 0, dok virtuelna metoda StaJe ispisuje reč "Poligon". Izvedena klasa Trougao javno nasleđuje klasu Poligon i koristi metodu Postavi iz osnovne klase za postavljanje vrednosti atributa sirina i visina. U klasi Trougao virtuelna metoda Povrsina nadjačava istoimenu i istotipsku virtuelnu metodu osnovne klase tako da računa i vraća vrednost površine trougla za postavljene vrednosti atributa sirina i visina. Nadjačana virtuelna metoda StaJe klase Trougao ispisuje reč "Trougao". Izvedena klasa Pravougaonik javno nasleđuje klasu Poligon i koristi metodu Postavi iz osnovne klase za postavljanje vrednosti atributa sirina i visina. U klasi Pravougaonik virtuelna metoda Povrsina nadjačava istoimenu i istotipsku virtuelnu metodu osnovne klase tako da računa i vraća vrednost površine pravougaonika za postavljene vrednosti atributa sirina i visina. Nadjačana virtuelna metoda StaJe klase Pravougaonik ispisuje reč "Pravougaonik". Funkcija PrikaziStaJe prihvata pokazivač na objekat osnovne klase Poligon i poziva preko prosleđenog pokazivača metodu StaJe.

U funkciji main kreiran je pokazivač ppoligon0 na objekat tipa Poligon koji je inicijalizovan pokazivačem na dinamički kreirani objekat klase Poligon. Preko ovog pokazivača pozvana je metoda Postavi koja postavlja vrednosti atributa klase Poligon na 5.0 i 6.0, a onda na isti način virtuelna metoda Povrsina klase Poligon koja ispisuje 0 (tako je stavljeno u kodu). Pozivom funkcije PrikaziStaJe kojoj je prosleđen pokazivač na objekat klase Poligon poziva se virtuelna metoda StaJe klase Poligon koja ispisuje reč "Poligon". Ovde se ne aktivira virtuelni mehanizam jer pokazivač "zaista" ukazuje na objekat klase Poligon. Nakon ovoga ukida se dinamički objekat na koga ukazuje pokazivač ppoligon0.

Sledeće je da je u funkciji main je kreiran pokazivač ppoligon3 na objekat tipa Poligon koji je inicijalizovan pokazivačem na dinamički kreiran objekat klase Trougao. Preko ovog pokazivača pozvana je nevirtuelna metoda Postavi koja postavlja vrednosti atributa klase Poligon na 5.0 i 10.0, a onda je na isti način pozvana i vituelna metoda Povrsina klase Poligon. Sada se aktivira virtuelni mehanizam jer ovaj pokazivač "zaista" ukazuje na objekat klase Trougao, tako da se proverava da li u izvedenoj klasi postoji nadjačana virtuelna metoda Povrsina i ako postoji ona se poziva čime će se specijalno za trougao izračunati njegova površina prema postavljenim podacima za visinu i širinu (vrednost 25). Nakon ovoga, pozvana je funkcija PrikaziStaJe kojoj je prosleđen pomenuti

pokazivač i gde se preko njega poziva virtuelna metoda StaJe klase Poligon. Sada se aktivira virtuelni mehanizam jer ovaj pokazivač "zaista" ukazuje na objekat klase Trougao, tako da se proverava da li u izvedenoj klasi postoji nadjačana virtuelna metoda StaJe i ako postoji ona se poziva čime će se specijalno za trougao ispisati reč "Trougao". Nakon ovoga ukida se dinamički objekat na koga ukazuje pokazivač ppoligon3.

Nakon ovoga, u funkciji main je kreiran pokazivač ppoligon4 na objekat tipa Poligon koji je inicijalizovan pokazivačem na dinamički kreiran objekat klase Pravougaonik. Preko ovog pokazivača pozvana je nevirtuelna metoda Postavi koja postavlja vrednosti atributa klase Poligon na 10.0 i 20.0, a onda je na isti način pozvana i vituelna metoda Povrsina klase Poligon. Sada se aktivira virtuelni mehanizam jer ovaj pokazivač "zaista" ukazuje na objekat klase Pravougaonik, tako da se proverava da li u izvedenoj klasi postoji nadjačana virtuelna metoda Povrsina i ako postoji ona se poziva čime će se specijalno za pravougaonik izračunati njegova površina prema postavljenim podacima za visinu i širinu (vrednost 200). Nakon ovoga, pozvana je funkcija PrikaziStaJe kojoj je prosleđen pomenuti pokazivač i gde se preko njega poziva virtuelna metoda StaJe klase Poligon. Sada se aktivira virtuelni mehanizam jer ovaj pokazivač "zaista" ukazuje na objekat klase Pravougaonik, tako da se proverava da li u izvedenoj klasi postoji nadjačana virtuelna metoda StaJe i ako postoji ona se poziva čime će se specijalno za pravougaonik ispisati reč "Pravougaonik". Nakon ovoga ukida se dinamički objekat na koga ukazuje pokazivač ppoligon4 i program završava naredbom return 0.

3.8.8. Apstraktna klasa

Klasa koja ima bar jednu apstraktnu metodu je apstraktna klasa (*abstract class*). Apstraktna metoda je metoda koja je samo deklarisana, a nije definisana i sintaksa je da se nakon deklaracije ovakve metode stavi = 0 (*pure virtual function*). Apstraktna klasa ne može imati svoje instance. Ideja apstrakcije je da se postavi osnovna klasa koja predstavlja generalizaciju problema, dok će klase izvedene iz apstraktne klase predstavljati specijalizaciju problema.
Primer:

```
#include <iostream>
using namespace std;
class Poligon {
  protected:
    double sirina;
    double visina;
  public:
    void Postavi (double s, double v) { sirina = s; visina = v; }
    virtual double Povrsina () = 0;
    virtual void StaJe () { cout<<"Poligon"<<endl; }
```

```
};
class Trougao: public Poligon {
  public:
    double Povrsina () { return (sirina * visina / 2.0);    }
    void StaJe ()      { cout<<"Trougao"<<endl;              }
};
class Pravougaonik: public Poligon {
  public:
    double Povrsina ()    { return sirina * visina;         }
    void StaJe ()         { cout<<"Pravougaonik"<<endl;     }
};
void PrikaziStaJe(Poligon* ppoligon) {  ppoligon->StaJe();   }
int main () {
  //Poligon *ppoligon0 = new Poligon; // NOK, apstraktna klasa
  Poligon *ppoligon3 = new Trougao;
  ppoligon3-> Postavi(5.0,10.0);
  cout<<ppoligon3->Povrsina()<<endl; // 25
  PrikaziStaJe(ppoligon3);           // Trougao
  Poligon *ppoligon4 = new Pravougaonik;
  ppoligon4-> Postavi(10.0,20.0);
  cout<<ppoligon4->Povrsina()<<endl; // 200
  PrikaziStaJe(ppoligon4);           // Pravougaonik
  return 0;
}
```

U datom primeru je virtuelna metoda Povrsina u klasi Poligon postavljena kao apstraktna metoda, čime je i klasa Poligon postala apstraktna. Ovim je onemogućeno kreiranje instanci klase Poligon, jer nije definisano ponašanje objekta klase Poligon za metodu Povrsina.

Napomena: Ako navedena metoda u izvedenoj klasi ima isti naziv kao i virtuelna metoda osnovne klase, ali nije istog tipa, onda se radi o skrivanju naziva (rečeno ranije).
Primer:

```
#include <iostream>
using namespace std;
class Osnovna
{
  public:
    virtual void f()
    {
      cout<<"Osnovna, void f()"<<endl;
    }
};
class Izvedena : public Osnovna
{
```

```
  public:
    void f(double d) // nevirtuelna, skriva naziv f iz osnovne
    {
      cout<<"Izvedena,void f(double):"<<d<<endl;
    }
    //double f() { return 3.14;}// NOK, ova f nije tipa void()
};
int main()
{
  Osnovna *posn = new Izvedena;
  posn->f();   // f iz osnovne nije nadjacana u izvedenoj klasi
  delete posn;
  Izvedena *pizv = new Izvedena;
  // pizv->f();       // NOK, f iz izvedene skriva f iz osnovne
  pizv->f(3.14);             // f iz Izvedene, tipa void(double)
  delete pizv;
  return 0;
}
```

U datom primeru, izvedena klasa ne nadjačava metodu f iz osnovne klase. Ovo znači da će poziv metode f preko pokazivača posn (koji je inicijalizovan pokazivačem koji ukazuje na dinamički objekat klase Izvedena) pozvati baš metodu f iz osnovne klase. Ako se kreira pokazivač na izvedenu klasu pizv koji je inicijalizovan pokazivačem koju ukazuje na dinamički objekat klase Izvedena, onda poziv metode f() je pogrešan jer takva se ne vidi iz izvedene klase. Ona koja se vidi je f tipa void(double) tako da je poziv f(3.14) korektan. Pogrešno bi bilo da se u izvedenoj klasi deklariše metoda (double f();) koja se od virtuelne metode iz osnovne klase (void f();) razlikuje samo po tipu koji vraća.

Eksplicitni poziv virtuelne metode neće aktivirati virtuelni mehanizam. Primer:

```
class Osnovna
{
  public:
    virtual void f(){ cout<<"Osnovna, void f()"<<endl;    }
};
class Izvedena : public Osnovna
{
  public:
    void f()
    {
      Osnovna::f();
      cout<<"I jos nesto da se uradi"<<endl;
    }
};
```

U virtuelnoj metodi u izvedenoj klasi dat je eksplicitni poziv virtuelne metode u osnovnoj klasi koji će izvršiti virtuelnu metodu osnovne klase, a onda sledi dalji kod u smislu da se još nešto dodatno uradi. Ovo je primer ponovnog korišćenja koda.

Kao i sa pokazivačima, takvo je stanje i sa referencama kada je u pitanju virtuelni mehanizam.

Primer:

```
#include <iostream>
using namespace std;
class Osnovna
{
  public:
    virtual void f(){ cout<<"Osnovna, void f()"<<endl;      }
};
class Izvedena : public Osnovna
{
  public:
    void f() { cout<<"Izvedena,void f()"<<endl;      }
};
void DinamickoVezivanje(Osnovna &rosn)
{
  rosn.f(); // za dati main primer poziva se f u izvedenoj klasi
}
void StatickoVezivanje(Osnovna osn)
{
  osn.f(); // za dati main primer poziva se f u osnovnoj klasi
}
int main()
{
  Izvedena izv;
  DinamickoVezivanje(izv); // dynamic binding
  StatickoVezivanje(izv);  // static binding
  return 0;
}
```

Ako funkcija ili metoda kao formalne argumente ima objekte, onda se u fazi prevođenja koda zna tip objekta i koristiće se statičko povezivanje (*static binding*), što znači da se neće aktivirati virtuelni mehanizam. Ako funkcija ili metoda kao formalne argumente ima pokazivače na objekte ili reference na objekte, onda se u fazi prevođenja koda ne zna stvarni tip objekta i koristiće se dinamičko povezivanje (*dynamic binding*) čime će se aktivirati virtuelni mehanizam.

Pošto se pravo pristupa virtuelnim metodama u hijerarhiji može razlikovati onda je pravo pristupa virtuelnoj metodi određen prvom deklaracijom. Na

primer, u slučaju da je virtuelna metoda u osnovnoj klasi po pravu pristupa javna, a da je njena nadjačana metoda u izvedenoj klasi privatna, aktiviranjem virtuelnog mehanizma neće se sprečiti pristup nadjačanoj metodi.

Primer:

```
#include <iostream>
using namespace std;
class Osnovna {
  public:
    virtual void f(){cout<<"Javna f u klasi Osnovna"<<endl;}
};
class Izvedena : public Osnovna
{
  private:
    void f(){cout<<"Privatna f u klasi Izvedena"<<endl;}
};
int main ()
{
  Osnovna *posn= new Izvedena; // ili new Izvedena()
  posn->f();   // "Privatna f u klasi Izvedena"
  return 0;
}
```

U datom primeru u funkciji main pozivom metode f preko pokazivača na objekat osnovne klase koji je inicijalizovan pokazivačem na dinamički objekat izvedene klase "pristupa" se javnoj metodi f osnovne klase. Pošto je poziv bio preko pokazivača ili reference na objekat izvedene klase, onda se proverava da li je metoda f virtuelna i da li je ona redefinisana u izvedenoj klasi te pošto jeste, poziva se metoda f u izvedenoj klasi bez obzira na to što je privatna po pravu pristupa.

Ako se u telu konstruktora ili destruktora pozove virtuelna metoda ne aktivaira se virtuelni mehanizam.

Primer:

```
#include <iostream>
using namespace std;
class Osnovna {
  public:
    Osnovna()
    {
      f();
    }
    virtual void f() { cout<<"Javna f u klasi Osnovna"<<endl; }
    ~Osnovna()
    {
      f();
    }
};
class Izvedena : public Osnovna
{
  private:
    void f(){cout<<"Privatna f u klasi Izvedena"<<endl;}
```

```
};
int main (){
  Izvedena izv; // konstruktor osnovne "Javna f u klasi Osnovna"
  return 0;
} // na kraju programa biće pozvan destruktor izvedene klase
  // "Javna f u klasi Osnovna"
```

U main funkciji kreiran je objekat izv klase Izvedena. Videli smo ranije da kreiranje objekta izvedene klase ide tako da se prvo kreira podobjekat osnovne klase pozivom odgovarajućeg konstruktora osnovne klase. U konstruktoru osnovne klase pozvana je virtuelna metoda f i ako bi se sada aktivirao virtuelni mehanizam onda bi se pozvala metoda f u izvedenoj klasi, pri čemu objekat izvedene klase još nije do kraja kreiran. Prema tome, virtuelna metoda f će u konstruktoru biti pozvana kao obična metoda. Na kraju programa, pozivom destruktora izvedene klase, pozvaće se destruktor izvedene (podrazumevani) te osnovne klase koji ovde poziva metodu f kao običnu metodu.

3.8.9. Virtuelni destruktor

Neka je u funkciji main kreiran pokazivač posn na objekat osnovne klase koji je inicijalizovan pokazivačem na dinamički objekat izvedene klase. Nakon toga ukida se objekat na koji ukazuje pokazivač posn.

Ako je destruktor osnovne klase nevirtuelan:

- poziva se destruktor osnovne klase kojim se ukida podobjekat izvedene klase;
- ne poziva se konstruktor izvedene klase tako da ostaje deo izvedene klase neoslobođen.

Ako je destruktor osnovne klase virtuelan:

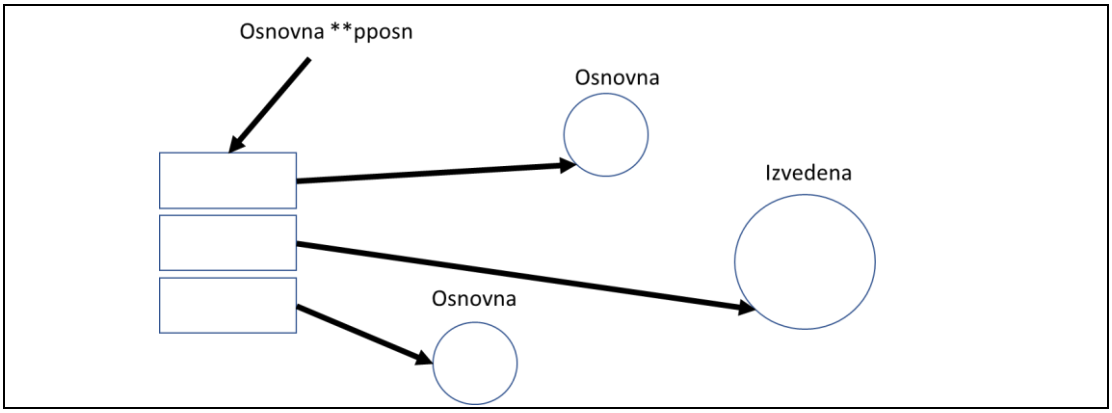
- poziva se virtuelni destruktor osnovne klase i pošto pokazivač posn ukazuje na objekat izvedene klase aktiviraće se virtuelni mehanizam te će biti pozvan destruktor izvedene klase;
- unutar destruktora izvedene klase implicitno će se pozvati destruktor osnovne klase;
- ovim će biti oslobođen ceo memorisjki prostor koji je zauzimao objekat izvedene klase.

```
#include <iostream>
using namespace std;
class Osnovna
{
  public:
    Osnovna ()
    {
       cout<<"Osnovna, konstruktor \n";
    }
```

```
    virtual ~Osnovna ()
    {
      cout<<"Osnovna, destruktor  \n";
    }
};
class Izvedena : public Osnovna
{
  public:
    Izvedena ()
    {
      cout<<"Izvedena, konstruktor \n";
    }
    ~Izvedena()
    {
      cout<<"Izvedena, destruktor  \n";
    }
};
int main()
{
  Osnovna * posn = new Izvedena;
  delete posn;
  return 0;
}
```

Slika 3.8.2. Primer virtuelnog destruktora

Pošto je izvedena klasa pravi nadskup osnovne klase, struktura podataka izvedene klase je veća od strukture podataka osnovne klase. Zbog ovoga je pravilno da se za nizove koristi pokazivač na pokazivač na objekat osnovne klase (Osnovna **pposn;). Ovim se postiže organizacija podataka u memoriji kao na slici 3.8.3. Dereferencijom posn[indeks] dolazi se do niza pokazivača (na slici su to 3 pravougaonika) koji ukazuju na objekte osnovne (dva manja kruga) ili izvedene klase (veći krug). Daljim dereferenciranjem *(posn[indeks]) dobija se konkretan objekat. Ako bi se koristio samo pokazivač na osnovnu klasu (Osnovna *posn), onda bi se postavljanjem objekata i osnovne i izvedene klase pri korišćenju pointerske aritmetike za posn (računa veličinu objekata osnovne klase) u opštem slučaju došlo po pogrešne pozicije za slučaj da se traži element niza koji nije nulti.



Slika 3.8.3. Organizacija niza koji sadrži pokazivače na objekte osnovnih i izvedenih klasa

Npr. neka je veličina objekta osnovne klase 10 bajta, veličina objekta izvedene klase 15 bajta i neka je u niz postavljen prvo objekat izvedene, pa objekat osnovne klase. Ovo bi značilo da od pozicije na koju ukazuje posn ide prvo 15 bajta za objekat izvedene klase, pa 10 bajta za objekat osnovne klase. Za posn[0] sve je korektno, ali za posn[1] nije jer je pomeraj 10 bajta umesto 15.

```
#pragma once
#include <iostream>
#include <string>
using namespace std;
class VozacNaAutoputu {
protected:
	static double elektronskaNaplata;
	string imeVozaca;
	double vrednostNaKartici;
public:
	VozacNaAutoputu(string imeVozaca, double vrednostNaKartici);
	virtual void PlatiPutarinu();
	string GetImeVozaca() const;
	double GetStanjeNaKartici() const;
	virtual ~VozacNaAutoputu();
};
```

Slika 3.8.4. Zaglavlje klase VozacNaAutoputu

Na slici 3.8.4. dato je zaglavlje klase VozacNaAutoputu. Na početku je data direktiva za jedno prevođenje, uključena su zaglavlja iostream i string te navedeno korišćenje prostora imena std. Klasi VozacNaAutoputu pripada privatni član elektronskaNaplata i odnosi se na cenu putarine. Članovi objekta klase VozacNaAutoputu su imeVozaca i vrednostNaKartici (platnoj kartici). Javni članovi su: konstruktor koji prihvata dva argumenta koji se odnose na članove objekta, virtuelna metoda PlatiPutarinu te inspektorske metode GetImeVozaca i GetStanjeNaKartici i na kraju virtuelni destruktor.

```
#include "VozacNaAutoputu.h"
using namespace std;
VozacNaAutoputu::VozacNaAutoputu(string imeVozaca, double
vrednostNaKartici)
:
imeVozaca(imeVozaca),
vrednostNaKartici(vrednostNaKartici)
{
	cout << "VozacNaAutoputu: konstruktor" << endl;
}
double VozacNaAutoputu::elektronskaNaplata = 500.00;
void VozacNaAutoputu::PlatiPutarinu()
{
  this->vrednostNaKartici -= VozacNaAutoputu::elektronskaNaplata;
}
string VozacNaAutoputu::GetImeVozaca() const
{
	return this->imeVozaca;
}

double VozacNaAutoputu::GetStanjeNaKartici() const
{
	return this->vrednostNaKartici;
}
VozacNaAutoputu::~VozacNaAutoputu()
{
	cout << "VozacNaAutoputu : destruktor" << endl;
}
```

Slika 3.8.5. Implementacija klase VozacNaAutoputu

Na slici 3.8.5. data je implementacija klase VozacNaAutoputu. Uključeno je zaglavlje VozacNaAutoputu.h. Konstruktor klase VozacNaAutoputu u inicijalizatoru postavlja početno stanje objekta (imeVozaca, vrednostNaKartici), a u telu konstruktora daje ispis da je reč o konstruktoru klase VozacNaAutoputu. Definisana je vrednost člana elektronskaNaplata koji pripada klasi VozacNaAutoputu na iznos od 500.00. Metoda PlatiPutarinu umanjuje vrednost stanja na elektronskoj kartici za vrednost putarine (elektronske naplate). Ispektorske metode GetImeVozaca i GetStanjeNaKartici vraćaju stanje članova klase: imeVozaca i vrednostNaKartici, respektivno. Na kraju, virtuelni destruktor ispisuje da je reč o destruktoru klase VozacNaAutoputu.

```
#pragma once
#include "VozacNaAutoputu.h"
class PolicijacNaAutoputu : public VozacNaAutoputu
{
```

```
public:
    PolicijacNaAutoputu(string imevozaca, double stanjenakartici);
    void PlatiPutarinu();
    ~PolicijacNaAutoputu();
};
```

Slika 3.8.6. Zaglavlje klase PolicajacNaAutoputu

Na slici 3.8.6. dato je zaglavlje klase PolicajacNaAutoputu. Na početku je data direktiva za jedno prevođenje i uključeno je zaglavlje VozacNaAutoputu.h. Klasa PolicajacNaAutoputu javno je izvedena iz klase VozacNaAutoputu. Klasa VozacNaAutoputu ima javni blok članova: konstruktor koji prihvata argumente imevozaca i stanjenakartici, virtuelnu metodu PlatiPutarinu i virtuelni destruktor.

```
#include "PolicajacNaAutoputu.h"
using namespace std;

PolicijacNaAutoputu::PolicijacNaAutoputu(string imevozaca, double
stanjenakartici)
: VozacNaAutoputu(imevozaca, stanjenakartici)
{
	cout << "PolicajacNaAutoputu : konstruktor" << endl;
}

void PolicijacNaAutoputu::PlatiPutarinu()
{
	cout << "Za policajca je besplatna putarina" << endl;
}

PolicijacNaAutoputu::~PolicijacNaAutoputu()
{
	cout << "PolicajacNaAutoputu : destruktor" << endl;
}
```

Slika 3.8.7. Implementacija klase PolicajacNaAutoputu

Na slici 3.8.7. data je implementacija klase PolicajacNaAutoputu. Uključeno je zaglavlje VozacNaAutoputu.h i navedeno korišćenje prostora imena std. Konstruktor klase PolicajacNaAutoputu u inicijalizatoru poziva kreiranje podobjekta klase PolicajacNaAutoputu prosleđujući mu svoje parametre imevozaca i stanjenakartici, a u telu konstruktora daje ispis da je reč o konstruktoru klase PolicajacNaAutoputu. Virtuelna metoda PlatiPutarinu ne menja stanje na kartici "policajca" već samo ispisuje o kojoj metodi je reč. Na kraju, virtuelni destruktor ispisuje da je reč o destruktoru klase PolicajacNaAutoputu.

```
#include <iostream>
```

```
#include "VozacNaAutoputu.h"
#include "PolicajacNaAutoputu.h"
using namespace std;
const int BROJ_CLANOVA = 2;
int main(void)
{
  VozacNaAutoputu* clanovi[BROJ_CLANOVA];
  cout << endl; clanovi[0] = new VozacNaAutoputu("Mirko",
                                                 1000.00);
  cout << endl; clanovi[1] = new PolicijacNaAutoputu("Slavko",
                                                 1000.00);
  cout << endl;
  for (int i = 0; i < BROJ_CLANOVA; i++)
  {
      clanovi[i]->PlatiPutarinu();
      cout << clanovi[i]->GetImeVozaca() <<
              ", stanje nakon placanja putarine je "
           << clanovi[i]->GetStanjeNaKartici()<<endl;
  }
  for (int i = 0; i < BROJ_CLANOVA; i++)
  {
      delete clanovi[i];
  }
  system("pause");
  return 0;
}

//IZLAZ
VozacNaAutoputu: konstruktor

VozacNaAutoputu: konstruktor
PolicajacNaAutoputu : konstruktor

Mirko, stanje nakon placanja putarine je 500
Za policajca je besplatna putarina
Slavko, stanje nakon placanja putarine je 1000
VozacNaAutoputu : destruktor
PolicajacNaAutoputu : destruktor
VozacNaAutoputu : destruktor
Press any key to continue . . .
```

Slika 3.8.8. Primer niza pokazivača na članove osnovne klase

Na slici 3.8.8. dat je primer niza pokazivača na članove osnovne klase koji će da ukazuju i na članove osnovne klase i na članove pripadne izvedene klase. Uključena su zaglavlja iostream, VozacNaAutoputu i PolicajacNaAutoputu, navedeno je korišćenje imena prostora std te konstanta `BROJ_CLANOVA` koja ima vrednost 2.

U funkciji main kreira se statički niz koji sadrži navedni broj pokazivača na objekte osnovne klase VozacNaAutoputu. Kreiraju se dva člana: pokazivač na objekta klase VozacNaAutoputu (parametri: Mirko, 1000.00) i pokazivač na objekta klase PolicajacNaAutoputu (parametri: Slavko, 1000.00). U prvom slučaju pozvan je konstruktor osnovne klase (ispis VozacNaAutoputu: konstruktor), a u drugom prvo konstruktor podobjekta osnovne klase, a onda konstruktor objekta izvedene klase (ispis VozacNaAutoputu: konstruktor, PolicajacNaAutoputu : konstruktor).

U petlji se za svakog člana poziva da plati putarinu i ispisuje se stanje na kartici nakon plaćanja putarine. Preko pokazivača na objekat osnovne klase VozacNaAutoputu poziva se metoda PlatiPutarinu klase VozacNaAutoputu i stanje na kartici će biti umanjeno za isnos putarine (ispis da Mirko ima stanje na kartici 500). Preko pokazivača na objekat izvedene klase pozivanjem metode PlatiPutarinu aktiviraće se virtuelni mehanizam i biće aktivirana metoda PlatiPutarinu klase PolicajacNaAutoputu, tako da policajcu neće biti naplaćena putarina (ispis da Slavko ima nepromenjeno stanje na kartici: 1000).

Na kraju se u petlji ukidaju objekti na koje ukazuju navedeni pokazivači. Preko pokazivača koji ukazuje na objekat osnovne klase VozacNaAutoputu pozvaće se samo njegov destruktor (ispis VozacNaAutoputu : destruktor), dok će poziv preko pokazivača koji "zaista" ukazuje na objekat izvedene klase biti aktiviran virtuelni mehanizam tako da će biti pozvan konstruktor izvedene klase koji implicitno na kraju zove destruktor podobjekta klase VozacNaAutoputu, tako da je ispis: PolicajacNaAutoputu : destruktor, VozacNaAutoputu : destruktor.

Rezime poglavlja objektno orijentisani pristup

U poglavlju objektno orijentisani pristup obrađene su: klase, objekti, pokazivač this, konstruktori (standardni, konstruktor kopije i move konstruktor), destruktor, inspektori i mutatori, statički članovi klase, prava pristupa, prijatelji klase, preklapanje operatora, nasleđivanje (javno, zaštićeno i privatno), kreiranje i ukidanje objekata izvedene klase, višestruko izvođenje, virtuelne klase, polimorfizam, apstraktne klase i virtuelni destruktor.

Zadaci za vežbanje

1. Napisati program koji koristi klasu MojStek koja omogućuje kreiranje strukture koja radi kao LIFO (Last In First Out) i koja u dinamičkom nizu čuva elemente tipa Student (atributi: ime, prezime, broj indeksa). MojStek realizovati tako da ima konstruktor kopije i move konstruktor. Pokazati da su konstruktor kopije i move konstruktor korektni.
2. Napisati program koji u statičkom nizu sadrži elemente tipa Brod. Veličina niza određena je celobrojnom konstantom. Atributi su: naziv modela broda, cena broda, id broda te statički atribut broj kreiranih brodova. Atribut id broda je jedinstven za konkretan brod. Metode su: IspisiPodatke, PostaviNaziv, PostaviCenu, UzmiNaziv, UzmiCenu, UzmiID te statičke metode: UzmiBrojKreiranihBrodova. U programu se iz konzole unose podaci o brodovima. Kreirati konstantan objekat tipa Brod, popuniti ga podacima i ispisati podatke o njemu.
3. Napisati klasu Vektor i klasu Matrica koji su realizovani dinamički i čuvaju podatke tipa double. I vektor i matrica mogu biti proizvoljnih dimenzija. Napisati prijateljsku funkciju koja množi dati vektor sa odgovarajućom matricom. Ako prosleđeni parametri (vektor i matrica) nisu matematički korektni ispisati takvu informaciju. Unos parametara vektora i matrice je iz konzole.
4. Napisati klasu KompleksanBroj koja ima implementirane operatore: +=, ++, --, <<, >>. Korišćenjem navednih operatora: uneti podatke o kreiranom kompleksnom broju (>>), primeniti jedan po jedan operatore: +=, -- i ++, nakon čega se u fajl Rezultati.hm upisuje stanje kompleksnog broja (<<).
5. Napisati program koji modeluje paradigmu dijamanta kod nasleđivanja klasa: Zivotinja, Sisar, Leteci, SlepiMis. Životinja slepi miš je leteći sisar.
6. Napisati program koji modeluje paradigmu dijamanta kod nasleđivanja klasa: Vozilo, Automobil, Brod, Amfibija. Vozilo amfibija je automobil koji može da plovi.

7. Napisati klasu ClanBibioteke i klasu PocasniClanBiblioteke koja je izvedena iz klase ClanBiblioteke. Član biblioteke plaća članarinu, dok počasni član plaća 50% članarine. Napisati dinamički niz koji sadrži pokazivače na članove biblioteke. Implementirati metodu PlatiClanarinu koja umanjuje vrednost na računu datog člana u zavisnosti od toga da li se radi o članu ili počasnom članu biblioteke. Pokazati da će se odazvati korektna metoda PlatiClanarinu u slučaju naplate članarine za svakog člana na koga ukazuje element niza.

8. Dopisati šta je potrebno da bi program bio korektan.

```
#include "Complex.h"
#include "MyMatrix.h"
#include "Dete.h"
#include "ArrayWrapper.h"
//DODATI FUNKCIJU f
int main(){
  Complex Z1(-5.0,5.0), Z2(10.0,-10.0), Z3;
  Z3=Z1+Z2; Z3<<cout<<endl;            //(5, -5)
  !Z3;      Z3<<cout<<endl;            //(5, 5)
  // MyMatrix ima podatak clan int *m koji ukazuje na elemente
matrice
  int r(3),c(3);
  MyMatrix mym(r,c);
  for(int red=0; red<r; red++) for(int kol=0; kol<r; kol++)
    mym(red,kol)=red*10+kol;
  MyMatrix mym2 = mym;
  mym2(0,0)=555;
  mym<<cout<<endl;
      //0    1    2
      //10   11   12
      //20   21   22
  // Klasa Dete je javno izvedena iz klasa Otac i Majka,
  // klase i Otac i Majka izvedene su javno i virtuelno iz klase
  // JMBG
  // konstruktor klase Otac i Majka prijhvataju 2 argumenta: ime i
  // jmbg
  // konstruktor klase JMBG prihvata 1 argument jmbg
  Dete ja("ImeDeteta", "ImeOca", "ImeMajke", "jmbgbroj");
  // pokazite da se prvo inicijalizuje podobjekat klase JMBG
  cout << ja.uzmijmbg() << endl;
  //pokazite odakle se poziva konstruktor JMBG
  //ArrayWrapper ima atribute: int* pint;  int size;
  //move konstruktor prihvata int koji predstavlja velicinu niza
  ArrayWrapper av = f(16);
  av.Ispisi(); //0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  return                                                    0;
}
```

4. Šabloni

Cilj poglavlja je da student ovlada: šablonima klasa, specijalizacijom šablona i šablonima metoda.

Šabloni predstavljaju mogućnost da se napiše opšti kod iz koga će kompjler da generiše konkretan kod u zavisnosti od poziva za kreiranje datog koda. Ovo vredi i za funkcije i za klase.

4.1. Šablon klase

Šablon klase ili generička klasa (*generic class*) predstavlja paradigmu da se napiše "parametarski" kod koji se odnosi na istu realizaciju stanja i ponašanja objekta, ali za različite entitete.

Šablon pri deklaraciji ima navedenu reč template znak < listu parametara i znak >, nakon čega sledi naziv šablona i dalje sve kao kod deklaracije klase, pri čemu se koristi naziv identifikatora koji je naveden u bloku <> template. Deklaracija šablona je globalna. Može biti više deklaracija jednog šablona, pri čemu je definicija za dati šablon jedinstvena. Za tip se koristi reč class ili typename.

Šablon za trivijalnu klasu koja ima jedan element datog tipa koji se pripadnim Get i Set metodama postavlja i uzima je kao što sledi:

```
template <class T>
class Simple {
    T tip;
  public:
    Simple (T tip);
    void Set(T tip);
    T Get();
};
```

Klasa koje je generisana iz šablona naziva se šablonska klasa (*template class*) i predstavlja konkretizaciju koda za datu klasu (tip). Konkretna šablonska klasa se dobija (generiše) npr. pozivom kao što sledi:

```
Simple<int> celi(50);

Automobil bmw("X1",50000.0); // videti ranije datu klasu Automobil
Simple<Automobil> kola(bmw);

Simple<double> realni(3.14);
```

Navedeni su nazivi šablona i stvarnih argumenata šablona u bloku <>. U datim primerima kreiraju se šablonske klase kod kojih umesto tipa T stoji konkretno int, Automobil i double, respektivno, odgovarajućim pozivima Simple<TIP>.

Neka je sada modifikacija takva da je novi šablon SimpleMAX ima formalne argumente šablona class i int:

```
template <class T, int MAX>
class SimpleMAX {
    T tip[MAX];
  public:
    Simple ();
    void Put(T tip, int index);
    T Get(int index);
};
```

Sada se umesto čuvanja jednog elementa tipa T, čuva MAX elemenata tipa T u obliku statičkog niza. Konkretna šablonska klasa se dobija (generiše) npr. pozivom kao što sledi:

```
SimpleMAX<int,10> celih10();

Simple<Automobil,20> parkingza20();

Simple<double,30> realnih30();
```

Ovim pozivima će kompajler generisati konkretne klase `SimpleMAX<int,10>`, `Simple<Automobil,20>` i `Simple<double,30>` koje predstavljaju nezavisne korisničke tipove, bez obzira što im je "ista" realizacija (izgled). Ovde je za formalni parametar šablona int postavljen stvarni argument 10, 20, 30, respektivno (nema class ispred int). Stvarni argumenti šablona čiji formalni argumenti šablona nisu class (u primeru je bio int MAX) moraju biti konstantni izrazi, adrese statičkih članova klasa, adrese objekata ili funkcija sa eksternim povezivanjem.

Stvarni argumenti šablonske klase moraju potpuno odgovarati formalnim argumentima šablona.

Metoda šablona klase koja ima argumente koji su i argumenti šablona klase je šablonska metoda.

Definisanje metoda šablona klase se radi u fajlu zaglavlja.

Primer definicije metoda za napred navedeni šablon klase Simple je kao što sledi:

```
template<class T>
Simple<T>::Simple(T tip)
:
tip(tip)
{
}
template<class T>
void Simple<T>::Set(T tip)
```

```
{
  this->tip = tip;
}
template<class T>
T Simple<T>::Get()
{
  return tip;
}
```

Pozivi metoda bi bili:

```
Simple<double> sd(3.14);
sd.Set(1.41);
std::cout<<sd.Get()<<std::endl;
```

Prijateljska funkcija generičke klase ne mora biti generička funkcija. U sledećem primeru prijateljska funkcija VelicinaUBajtovima nije generička, tako da je ona ista za sve šablonske klase generisane iz šablona klase Simple. Za razliku od nje, funkcija SrednjaVrednost je generička, tako da će svaka generička klasa koja je generisana iz šablona klase Simple imati svoj primerak prijateljske funkcije SrednjaVrednost gledano prema datom tipu T. Prijateljska funkcija Ispis će biti generisana kao jedna za sve generičke klase iz šablona klase Simple, ali će zbog njenog argumenta biti generisana generička klasa Simple<double>.

```
template <class T>
class Simple {
    friend void VelicinaUBajtovima();
    friend T SrednjaVrednost(Simple<T>);
    friend void Ispis(Simple<double>);
     T tip;
  public:
    Simple (T tip);
    void Set(T tip);
    T Get();
};
```

Na početku je navedeno da je svaka generička klasa generisana iz istog šablona klase odvojen entitet, tako da će u tom smislu svaka takva klasa da ima vlastite statičke članove i vlastite statičke lokalne objekte.

Bilo kakvo pominjanje naziva šablona dovodi do generisanja šablonske klase ili šablonske funkcije.
Napomena: Nešablonska funkcija koja je preklopljena sa odgovarajućom šablonskom verzijom predstavlja definiciju te funkcije za konkretne tipove.
Primer:

```
template<class T>
T Pola(Simple<T> smax)
```

```
{
  // za sve slucajeve osim za slucaj Pola<int>
}

int Pola(Simple<int> smax)
{
 // za slucaj Pola<int>
}
```

Ova definicija funkcije int Pola(Simple<int>) će se koristiti za pozive gde je argument Simple<int>, a za ostale slučajeve će se generisati iz šablona.

Na slici 4.1. dat je primer šablona klase i pripadne specijalizacije šablona klase. Šablon mycontainer sadrži element tipa T, konstruktorom postavlja početnu vrednost tog argumenta i metodom increase inkrementira vrednost člana element i vraća tako inkrementiranu vrednost. Sintaksa specijalizacije šablona klase mycontainer je takva da se navede template<>, a onda se dalje deklariše specijalni slučaj šablona klase mycontainer, npr. za tip char. Iza sledi definicija specijalizacije šablona mycontainer za tip char, gde se sve vreme navodi tip char kao argument za tip šablona. U datom konstruktoru se postavlja char element član, a u metodi increase se definiše "inkrement" tako da se malo slovo postavi da je veliko, a ako je veliko da takvo i ostane, pri čemu se nova vrednost slova vrati naredbom return.

```
#include <iostream>
using namespace std;
// sablon klase
template <class T>
class mycontainer
{
    T element;
  public:
    mycontainer (T arg) {element=arg;}
    T increase () {return ++element;}
};
// specijalizacija sablona klase
template <>
class mycontainer <char>
{
    char element;
  public:
    mycontainer (char arg) ;
    char increase ();
};

mycontainer<char>::mycontainer(char arg)
```

```
{
  element=arg;
}
char mycontainer<char>::increase()
{
    if ((element>='a')&&(element<='z'))
      element+='A'-'a';
    return element;
}
int main ()
{
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.increase() << endl;
  system("pause");
  return 0;
}
```

Slika 4.1. Primer specijalizacije šablona klase

U funkciji main se kreiraju dve šablonske klase mycontainer<int> i mycontainer<char> odgovarajućim pozivima gde se zahteva kreiranje objekata myint(7) i mychar('j'). Klasa mycontainer<int> će odgovarati definiciji opšteg šablona klase mycontainer, dok će klasa mycontainer<char> odgovarati definiciji specijalizacije šablona klase mycontainer za tip char.

```
#pragma once
template <class T>
class Stek
{
  private:
    T* pstek;
    int velicina ;
    int indeks ;
    static const int MAX = 1000;
  public:
    Stek (int = 16) ;
    ~ Stek () { delete [] pstek ; }
    bool Stavi(const T&);
    bool Uzmi(T&) ;
    bool Prazan() const { return indeks == -1 ; }
    bool Pun() const  { return indeks == velicina - 1 ; }
} ;
template <class T>
Stek<T>::Stek(int s)
{
    velicina = (s > 0 && s < MAX) ? s : 16 ;
```

```
    indeks = -1 ;
    pstek = new T[velicina] ;
}
template <class T>
bool Stek<T>::Stavi(const T& element)
{
  if (false==Pun())
  {
    pstek[++indeks] = element ;
    return true ;
  }
  return false ;
}
template <class T>
bool Stek<T>::Uzmi(T& element)
{
  if (false==Prazan())
  {
    element = pstek[indeks--] ;
    return true ;
  }
  return false ;
}
```

Slika 4.2. Primer šablona za stack

Na slici 4.2. dato je zaglavlje i definicija šablona Stek (fajl Stek.h). Stek (stack) je LIFO (last in first out) struktura podataka u koju se stavlja i skida podatak gledano na vrh steka. Poslednji podatak koji se stavi na vrh steka biće prvi skinut sa vrha steka. Šablon klase Stek ima privatni blok članova: pokazivač na stek (pstek), veličina steka (velicina), indeks elementa na steku (indeks) i celobrojnu konstantu MAX koja se odnosi na maksimalnu veličinu steka. Javni blok šablona klase Stek čine: konstruktor sa podrazumevanim int elementom jednakim 16 (odnosi se na podrazumevanu veličinu steka), destruktor koji briše dinamički kreirani niz na koji ukazuje član pstek, metoda Stavi koja povećava stek datog tipa stavljajući na vrh steka element tog tipa, metoda Uzmi koja smanjuje stek datog tipa uzimajući element tog tipa sa vrha steka, inspektorska metoda Prazan koja vraća logički tip kao odgovor na pitanje da li je stek prazan (ako je indeks -1) i na kraju inspektorska metoda Pun koja vraća logički tip kao odgovor na pitanje da li je stek pun (ako indeks odgovara vrednosti velicina-1). Sledi definicija konstruktora Stek<T> koji prihvata jedan argument (podrazumevano je 16) i na osnovu koga se postavlja vrednost člana velicina (ako je van dozvoljenih granica 0...MAX biće 16, u suprotnom će biti koliko je navedeno pri pozivu), indeks ima početnu vrednost -1 (stek je prazan), kreira se dinamički niz veličine velicina koji sadrži elemente tipa T i na koji ukazuje član pstek. Definiše se metoda Stavi koja prihvata referencu na konstantan

objekat tipa T i koja proverava da li je stek pun te ako jeste vraća false, inače ažurira indeks, stavlja element na vrh steka i vraća true. Definiše se metoda Uzmi koja prihvata referencu na objekat tipa T i koja proverava da li je stek prazan te ako jeste vraća false, inače uzima element sa vrha steka, ažurira indeks i vraća true.

```
#include <iostream>
#include "Stek.h"
using namespace std;
int main()
{
  Stek<double> stekdouble(8);
  double doubleelement = 3.14; // mora biti double
  cout << "Stavljam double elemente na stek stekdouble." << endl;
  while (stekdouble.Stavi(doubleelement))
  {
      cout << doubleelement << ' ';
      doubleelement += 1.0;
  }
  cout << endl << "Stek stekdouble je pun !" << endl
       << endl << "Uzimanje elemenata sa steka stekdouble."
<<endl;
  while (stekdouble.Uzmi(doubleelement))
  {
    cout << doubleelement << ' ';
  }
  cout << endl<< "Stek stekdouble je prazan !" << endl;cout<<endl;
  Stek<int> stekint;
  int intelement = 1;
  cout << "Stavljam int elemente na stek stekint." << endl;
  while (stekint.Stavi(intelement))
  {
      cout << intelement << ' '; intelement += 1;
  }
  cout << endl << "Stek stekint je pun !" << endl
       << endl << "Uzimanje elemenata sa steka stekint." << endl;
  while (stekint.Uzmi(intelement))
  {
    cout << intelement << ' ';
  }
  cout << endl << "Stek stekint je prazan !" << endl;
  return 0;
}

IZLAZ

Stavljam double elemente na stek stekdouble.
```

```
3.14 4.14 5.14 6.14 7.14 8.14 9.14 10.14
Stek stekdouble je pun !

Uzimanje elemenata sa steka stekdouble.
10.14 9.14 8.14 7.14 6.14 5.14 4.14 3.14
Stek stekdouble je prazan !

Stavljam int elemente na stek stekint.
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Stek stekint je pun !

Uzimanje elemenata sa steka stekint.
16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
Stek stekint je prazan !
```

Slika 4.3. Primer upotrebe šablona za stek

Na slici 4.3. data je funkcija main u kojoj se kreira objekat stekdouble tipa Stek<double> čijem konstruktoru se prosleđuje vrednost 8 (veličina steka). Kreiran je doubleelement tipa double čija je početna vrednost 3.14. U while uslovu se se poziva metoda Stavi klase Stek<double> kojoj se kao argument prosleđuje doubleelement. Ako je povratna vrednost ove metode jednaka false onda je stek stekdouble pun te nije moguće staviti novi element na stek stekdouble i while petlja završava rad. Ako je povratna vrednost metode Stavi jednaka true onda je uspešno stavljen novi element na stek stekdouble, izvršava se telo petlje gde se ispisuje vrednost elementa doubleelement, a zatim se elementu doubleelement dodaje vrednost 1.0 i ide u novi ciklus while petlje. Ako metoda Stavi vrati false onda to znači da je stek stekdouble pun i while petlja završava rad nakon čega se ispisuje da je stek stekdouble pun. Nakon ovoga sledi while petlja koja za računanje svog uslova koristi povratnu vrednost poziva metode Uzmi. Metoda Uzmi vraća da li je uspela da skine element sa vrha steka stekdouble te ako jeste vraća true tako da se izvršava telo while petlje u kojoj se ispisuje vrednost uzetog elementa i ponovo ide u sledeći ciklus while petlje. Ako metoda Uzmi vrati false, znači da je stek prazan, while petlja završava i nakon toga se ispisuje da je stek stekdouble prazan.

Sada se kreira objekat stekint tipa Stek<int> koji je kreiran pozivom konstruktora bez argumenata, čime je postavljena vrednost veličine steka na 16 elemenata tipa int. Kreiran je intelement tipa int čija je početna vrednost 1. U while uslovu se poziva metoda Stavi klase Stek<int> kojoj se kao argument prosleđuje intelement. Ako je povratna vrednost ove metode jednaka false onda je stek stekint pun te nije moguće staviti novi element na stek stekint i while petlja završava rad. Ako je povratna vrednost metode Stavi jednaka true onda je uspešno stavljen novi element na stek stekint, izvršava se telo petlje gde se ispisuje vrednost elementa intelement, a zatim se elementu intelement dodaje vrednost 1 i ide u novi ciklus while petlje. Ako metoda Stavi vrati false onda to

znači da je stek stekint pun i while petlja završava rad nakon čega se ispisuje da je stek stekint pun. Nakon ovoga sledi while petlja koja za računanje svog uslova koristi povratnu vrednost poziva metode Uzmi. Metoda Uzmi vraća da li je uspela da skine element sa vrha steka stekint te ako jeste vraća true tako da se izvršava telo while petlje u kojoj se ispisuje vrednost uzetog elementa i ponovo ide u sledeći ciklus while petlje. Ako metoda Uzmi vrati false, znači da je stek stekint prazan, while petlja završava i nakon toga se ispisuje da je stek stekint prazan.

4.2. Šablon funkcije

Šablon funkcije ili generička funkcija (*generic function*), kao i šablon klase, obezbeđuje za "opšti" tip isti izgled funkcije (isto ponašanje) na osnovu koga će pri odgovarajućem pozivu kompajler generisati funkciju koja koristi konkretan tip.

Šablonska funkcija (*template function*) je konkretna funkcija koju kompajler generiše iz šablona funkcije.

U deklaraciji šablona funkcije moraju se svi formalni argumenati šablona pojaviti kao formalni argumenti funkcije. Formalni argument šablona funkcije može se pojaviti kao formalni argument nekog šablona klase. Primer koji koristi šablon vektor iz std prostora imena:

```
template <class T> void Fsortiranje (std::vector<T>&v);
```

Neka se sada radi o šablonu funkcije koja sortira niz od n elemenata "opšteg" tipa, kao što sledi:

```
template <class T>
void Fsortiranje (T *niz, int n) {
  for (int i=0; i<n-1; i++)
    for (int j=i+1; j<n; j++)
      if (niz[i]>niz[j])
      {
        T temp = niz[i];
        niz[i] = niz[j];
        niz[j] = temp;
}
```

Sada bi za poziv:

```
int arr[20];

...
Fsortiranje(arr, sizeof(arr)/sizeof(int));
```

kompajler generisao šablonsku funkciju tipa void Fsortiranje(int*, int) koja može da sortira int niz.

Ako bi se koristilo poziv:

```
Automobil parking[20];

...
Fsortiranje(parking, sizeof(parking)/sizeof(Automobil));
```

kompajler bi generisao šablonsku funkciju tipa void Fsortiranje(Automobil*, int) koja može da sortira niz objekata klase Automobil. Da bi ovakva šablonska funkcija bila korektna, moraju se za klasu Automobil implementirati operatori > i =, čime bi zamena elemenata bila uspešna.

Generisanje šablonske funkcije zahteva definiciju pripadnog šablona funkcije, dok generisanje poziva funkcije zahteva samo deklaraciju šablona funkcije.

Pošto šablon funkcije može da generiše preklopljene šablonske funkcije, onda se primenjuju pravila za razrešavanje poziva. Šablonska funkcija može biti preklopljena i sa nešablonskim funkcijama istog naziva. Prvo se traži nešablonska funkcija koja dogovara pozivu, a onda šablon funkcije iz koje se može generisati šablonska funkcija koja odgovara pozivu pri čemu nema trivijalnih konverzija.

```
void f(std::vector<complex>& complexv,std::vector<int>& intv)
{
  sort(complexv); // poziv sort(std::vector<complex>);
  sort(intv);     // poziv sort(std::vector<int>);
}
```

Moguće je generisati funkciju iz šablona funkcije navođenjem deklaracije konkretne funkcije. Primer:

```
template<class T>
T Saberi(T a,T b)
{
  return a+b;
}
int Saberi (int,int); // deklaracija
void f(int i1, int i2, char c1, char c2) {
  int  m1= Saberi(i1,i2);//generiše se  int Saberi(int,int);
  char m2= Saberi(c1,c2);//generiše se char Saberi(char,char);
  int  m3= Saberi(i1,c1);// OK, zbog deklaracije
}
```

Navođenjem deklaracije int Saberi(int,int); generisaće se funkcija koja će pri pozivu Saberi(i1,c1) moći konverzijom da razreši ovaj poziv. Kada ne bi bila navedena navedena deklaracija program bi dojavio grešku.

Rezime poglavlja šabloni

U poglavlju šabloni obrađeni su: šabloni klasa, specijalizacija šablona i šabloni metoda.

Zadaci za proveru znanja

1. Napisati program koji koristi šablonsku funkciju BubbleSort. Sortirati niz od M elemenata koji sadrži tip Student prema broju indeksa i niz od N elemenata koji sadrži slučajno generisane proste brojeve.
2. Napisati program koji koristi šablonsku klasu Red (FIFO). U redu za šalter može biti maksimalno N Osoba. Napraviti simulator koji puni red, prazni red i ispisuje promenu stanja reda.
3. Napisati program koji koristi šablonsku klasu Max. Klasa Max sadrži niz od 10 elemenata proizvoljnog tipa i ima metodu VratiSrednji koja vraća element niza koji je najbliži srednjoj vrednosti svih članova niza. Pokazati ispravnost koda na primeru tipova double i max. Napisati specijalizaciju klase Max za tip Automobil gde metoda VratiSrednji vraća automobil koji ima vrednost cena_automobila/broj_rata najbližu srednjoj vrednosti cena_automobila/broj_rata za sve automobile smeštene u niz.

5. Izuzeci

Cilj poglavlja je da student ovlada korišćenjem mehanizma postavljanja i obrade izuzetaka. Obrađuju se try-catch blok, korisnički definisani izuzeci, propagacija izuzetka, postavljanje ograničenja funkcije u smislu postavljanja izuzetka, funkcija terminate, funkcija unexpected, funkcija abort, makro assert te korišćenje klase std::exception .

Mehanizam obrade izuzetaka radi odvojeno od osnovnog toka programa. Informacija o izuzetku može biti bilo kog tipa (ugrađenog, korisničkog). Ideja je da se u osnovni kod doda kod za obradu izuzetka koji može nastati u tom delu koda. Informacija o tome koji je izuzetak nastao prenosi se nezavisno od standardnog prenosa argumenata i vrednosti koju vraća funkcija ili metoda. Ako se izuzetak pojavi u delu koda koji može da ga obradi, onda će program odmah preći na obradu datog izuzetka.

Ne treba ni preterivati u smislu da se svuda postavlja kod za obradu izuzetaka, već samo u onim delovima koda gde izuzetak može da se pojavi.

U slučaju nastanka izuzetka u delu koda koji obrađuje izuzetak omogućeno je da se kontrola toka programa i informacije o izuzetku prenese kroz hijerarhiju poziva svo do mesta gde će se dati izuzetak obraditi.

Izuzetak se može postaviti korišćenjem reči throw (kao bacanje lasa) iza koje sledi objekat nekog tipa T. Sada će se aktivirati mehanizam koji prenosi informaciju da se desio izuzetak tipa T na mesto na kome se izuzetak tipa T obrađuje.

U sledećem primeru metoda Alarmi prihvata argument tipa int čiji bitovi predstavljaju zastavice o upadu na dati ulaz. U telu ove metode postavljan je try-catch blok koji obrađuje izuzetak tipa int. U bloku try se postavlja osnovni tok programa koji može da postavi izuzetak.

Ako je bilo koja zastavica podignuta u argumentu svialarmi, onda se postavlja izuzetak tipa int (simbolička int konstanta UPAD) sintaksom throw UPAD. Izvršavanjem naredbe throw iza koga je navedena vrednost tipa int kontrola toka programa se prebacuje na najbliže mesto u hijerarhiji poziva obrade izuzetka koje ima catch iskaz sa argumentom tipa int kome se dodeljuje vrednost UPAD. Ovim je informacija o izuzetku prenesena na mesto obrade.

Iz liste catch pripadnog try bloka bira se onaj catch čiji tip argumenta odgovara tipu izuzetka. Ako je više takvih slučajeva bira se prvi na koji se naiđe.

Dakle, nakon postavljanja izuzetka prelazi se na catch (*exception handler*) blok u kome postoji catch koji obrađuje int tip izuzetka. Ovaj catch prihvata tip int gde je kao formalni argument bloka za obradu izuzetka naveden identifikator izuz u koga će se kopirati vrednost UPAD. U telu catch izraza navodi se kod za

obradu izuzetka tako da se proverava koje su zastavice podignute i na osnovu njih se preduzimaju odgovarajuće akcije.

```
// negde u kodu se definisane int konstante UPAD, LEVI_ULAZ,
// DESNI_ULAZ,...

void Sigurnost::Alarmi(int svialarmi)
{
  try
  {
    if (svialarmi)
      throw UPAD; //postavljanje izuzetka da se desio upad u posed
    //... ostali kod
  }
  catch (int izuz)
  {
     // ovde ide obrada izuzetka
     if(LEVI_ULAZ & izuz)
     {
       // obrada slucaja da je upad na npr. levi ulaz
     }
     if(DESNI_ULAZ & izuz)
     {
       // obrada slucaja da je upad na npr. levi ulaz
     }
     ...
  }
}
```

Obrada izuzetka ne mora biti smeštena u bloku u kome je throw, već može biti smeštena u višem nivou hijerarhijskih poziva. Uvek se ide do najbližeg mesta u hijerarhiji poziva koji može da obradi izuzetak datog tipa. Da bi se omogućilo ovakvo kretanje po hijerarhiji poziva izvedeno je da se izrazom iza throw inicijalizuje privremeni objekat koji se stavlja u statičku memoriju, tako da je dostupan na svim nivoima poziva, a time i na pripadnom catch ulazu u kome će inicijalizovati njegov formalni argument.

```
class MojIzuzetak
{
  // neka je uradjena deklaracija i definicija klase
};

void Sigurnost::Pozvana(int uslov)
{
  if (uslov)
    throw MojIzuzetak();
  //...
}
```

```
void Sigurnost::Pozivajuca()
{
  int uslov;
  // racunanje uslova
  try {
      Pozvana(uslov);
      throw "Izuzetak tipa char*";
  }
  catch (char *p) {
    cout<<p;
  }
  catch (MojIzuzetak& rMI) {
    cout<<"Izuzetak tipa MojIzuzetak";
  }
  cout<<"Kod iza catch bloka"<<endl;
}
```

U pseudo primeru iznad, u metodi Pozivajuca klase Sigurnost deklarisan je lokalni int uslov, a nakon toga neka se izračuna njegova vrednost. U try bloku poziva se metoda Pozvana klase Sigurnost i prosleđuje joj se kao argument vrednost promenljive uslov. U metodi Pozvana se testira uslov i ako je isti ispunjen, onda se postavlja izuzetak tipa MojIzuzetak(). Pošto u bloku metode Pozvana ne postoji obrada izuzetka, onda se obrada pomera na viši nivo u hijerarhiji poziva tako da se obrada izuzetka premešta na try-catch blok u metodi Pozivajuca. U catch bloku prvo je ispitivanje da li je tip koji se obrađuje char*, pošto nije ide se dalje po catch bloku. Sledeći catch obrađuje izuzetke tipa MojIzuzetak gde je formalni argument tipa referenca na objekat klase MojIzuzetak i u složenom bloku se ispisuje "Izuzetak tipa MojIzuzetak".

Ako je uslov koji se prosleđuje metodi Pozvana jednak 0, onda se u metodi Pozvana neće baciti izuzetak i nakon povratka iz metode Pozvana sledeća linija koda u metodi Pozivajuca je throw "Izuzetak tipa char*"; čime se postavlja izuzetak tipa char*, odlazi se u pripadni catch blok i testira da li je formalni argument prvog catch izraza char*. Pošto prvi catch obrađuje izuzetke takvog tipa onda se ispisuje tekst "Izuzetak tipa char*".

Nakon obrađenog izuzetka programski tok nastavlja iza catch bloka tako da se izvršava kod koji ispisuje "Kod iza catch bloka".

Na slici 5.1. dat je primer obrade izuzetka.

```
#include <iostream>
using namespace std;
class DeljenjeNulom
{
      double deljenik;
public:
```

```
      DeljenjeNulom(double deljenik)
      :
      deljenik(deljenik)
      {}
      void Opis()
      {
        cout << "Delili ste " << deljenik
             << " sa nulom !!!" << endl;
      }
};
int main()
{
      double brojnik;
      double nazivnik;
      try
      {
            cout << "unesite brojnik  = "; cin >> brojnik;
            cout << "unesite nazivnik = "; cin >> nazivnik;
            if (0 == nazivnik) throw DeljenjeNulom(brojnik);
            cout << brojnik / nazivnik << endl;
      }
      catch (DeljenjeNulom &dn)
      {
            dn.Opis();
      }
      catch (...)
      {
        cout <<
             "NIje bilo deljenje nulom, nego neki drugi izuzetak"
             << endl;
      }
      return 0;
}
```

Slika 5.1. Primer obrade izuzetka

U prethodnom primeru, ako se za nazivnik unese 0 izvršiće se throw iza koga je kreiran objekat pozivom konstruktora klase DeljenjeNulom. Ovim objektom će, u pripadnom catch bloku koji obrađuje izuzetak tipa DeljenjeNulom, biti inicijalizovana referenca koja upućuje na taj objekat.

Sintaksa catch(...) omogućuje da se prihvati izuzetak bilo kog tipa. Ova sintaksa se može koristiti i kao što sledi:

```
try
{
  throw "Salji dalje";
}
catch (...)
```

```
  {
    cout<<"Delimicna obrada, a onda saljem dalje"<<endl;
    throw;
  }
```

Ovde se radi o tome da se postavljanjem izuzetka u telu catch bloka on delimično obrađuje, a onda se postavljanjem throw bez argumenta (iza sledi samo tačka-zarez) taj isti izuzetak prosleđuje na viši hijerarhijski nivo poziva na dalju obradu.

Za tip izuzetka TI koji se postavlja i tip formalnog argumenta za catch TC vrede pravila slaganja u smislu da su oba istog tipa ili je TC osnovna klasa klase TI koja je dostupna na mestu postavljanja izuzetka ili se radi o pokazivačima gde se TI na mestu postavljanja izuzetka može konvertovati u TC.

```
class KomunikacioniKanalIzuzetak
{
   //...
   public:
      virtual void Obradi(){//obrada izuzetka kanala}
};
class GPRSIzuzetak  : public KomunikacioniKanalIzuzetak
{
  //...
      void Obradi(){//obrada izuzetka GPRS kanala}
};
class TCPIPOMIzuzetak  : public KomunikacioniKanalIzuzetak
{
  //...
      void Obradi(){//obrada izuzetka TCPIPOM kanala}
};
void PosaljiPoruku(char* poruka)
{
  try
  {
    // pokusaj slanja poruke raznim kanalima
  }
  catch (GPRSIzuzetak &gprsiz)
  {
    gprsiz.Obradi(); // obrada greske pri slanju GPRS kanalom
  }
  catch (KomunikacioniKanalIzuzetak &kkiz)
  {
    gprsiz.Obradi(); // obrada greske pri slanju za sve
                     // ostale kanale
  }
```

```
  catch(...)
  {
    // obrada svih preostalih izuzetaka
  }
}
```

U primeru iznad dato je korišćenje polimorfizma u catch bloku. Ono na šta treba obratiti pažnju je redosled navođenja tipova izuzetaka koji se obrađuju. Redom se ide od sitnije granulacije ka krupnijoj, dakle, prvo je navedena obrada za tip izvedene klase GPRSIzuzetak, zatim obrada za sve izuzetke tipa osnovne klase KomunikacioniKanalIzuzetak ili drugih klasa koje su iz nje izvedene (zbog reference formalnog catch argumenta i polimorfizma) i na kraju je navedena obrada svih ostalih tipova izuzetaka.

Napomena: Kada je reč o nasleđivanju i polimorfizmu uvek se postavlja catch redosled u kome ide prvo obrada specifičnih tipova (izvedena klasa), a na kraju obrada svih ostalih tipova (osnovna klasa).

Ako za dati izuzetak nema nijednog odgovarajućeg catch iskaza, onda se nastavlja traženje odgovarajućeg catch iskaza u okružujućem try-catch bloku. Ako se ne nađe odgovarajući catch iskaz, poziva se funkcija `terminate`.

Funkcija `void terminate()` se poziva:

- kada mehanizam za obradu izuzetaka ne nalazi mesto za obradu postavljenog izuzetka;
- kada je stek poziva poremećen;
- kada se u destruktoru pozvanom pri odmotavanju steka postavi izuzetak.

Moguće je postaviti svoju funkciju koju će pozvati funkcija `terminate` korišćenjem funkcije `set_terminate`:

```
typedef void (*PFVV)();
PFVV set_terminate(PFVV);
```

Funkcija `set_terminate` prihvata kao argument funkciju tipa `void void` i nju će zvati funkcija `terminate`, a vraća prethodno postavljenu funkciju (tipa `void void`) koju bi zvala funkcija terminate.

Napomena: Iz funkcije terminate se podrazumevano zove funkcija `abort()`.

Pri postavljanju izuzetka i odlaska do mesta obrade tog izuzetka dešava se kao što sledi:

- stek poziva se vraća u stanje u koje je bio na mestu obrade izuzetka (odmotavanje steka - *stack unwinding*);
- za sve automatske objekte koji su konstruisani od trenutka ulaska u try blok u kome se izuzetak obrađuje biće pozvani destruktori;

- ako konstruktor nekog elementa niza postavi izuzetak, onda se zovu destruktori samo konstruisanih elemenata niza;
- ako je objekat delimično konstruisan, a desi se izuzetak, onda će se zvati samo destruktori konstruisanih članova i podobjekata.

Ako se želi ograničenje da funkcija ili metoda može postaviti samo određene tipove izuzetaka onda se u deklaraciji funkcije ili metode navodi lista tih dozvoljenih izuzetaka koje neka metoda ili funkcija može postaviti (direktno ili indirektno):

```
void PosaljiPoruku(char* poruka) throw (Klasa1, Klasa2)
{
  //...
}
```

ovo je isto kao da je napisano sledeće:

```
void PosaljiPoruku(char* poruka)
{
  try
  {
    //...
  }
  catch (Klasa1 &k1) {throw;}
  catch (Klasa2 &k2) {throw;}
  catch (...) {unexpected();}
}
```

U prethodnom je navedeno da metoda PosaljiPoruku može postaviti izuzetak tipa Klasa1 ili Klasa2 Ako se postavi izuzetak tipa koji nije naveden u throw listi deklaracije metode ili funkcije, onda će se pozvati funkcija `unexpected`.

Napomene: Ako funkcija ili metoda nema throw specifikaciju u deklaraciji onda može da postavi izuzetak bilo kog tipa, a ako je lista throw prazna (throw()) onda ta funkcija ili metoda ne može postaviti izuzetak. Specifikacija izuzetaka nije deo tipa funkcije ili metode.

Ako funkcija postavi izuzetak tipa koji nije u throw listi, onda se poziva funkcija `unexpected tipa void void`.

Moguće je postaviti svoju funkciju koju će pozvati funkcija `unexpected` korišćenjem funkcije `set_unexpected`:

```
typedef void (*PFVV)();
PFVV set_unexpected(PFVV);
```

Funkcija `set_unexpected` prihvata kao argument funkciju tipa `void void` i nju će zvati funkcija `unexpected`, a vraća prethodno postavljenu funkciju (tipa `void void`) koju bi zvala funkcija `unexpected`.

Napomena: Iz funkcije `unexpected` se podrazumevano zove funkcija `terminate`.

Funkciju abort poziva makro assert koji ispituje tvrdnju koja je njegov argument i ako je tvrdnja netačna biće pozvana funkcija abort. U primeru na slici 5.2. u funkciji main kreiran je int objekat aa inicijalizovan vrednošću 10, dva pokazivača na tip int: bb i cc koji su inicijalizovani na NULL. Nakon ovoga postavljeno je da pokazivač bb ukazuje na objekat aa. Pozivom ispisa broja (print_number) za parametar bb biće ispisano 10, dok će za parametar cc tvrdnja u assert makrou biti netačna i biće ispisan razlog prekida rada programa.

```
#include <iostream>
#include <cassert>
using namespace std;
void print_number(int* myInt) {
      assert(myInt != NULL);
      cout << (*myInt) << endl;
}
int main()
{
      int aa = 10;
      int * bb = NULL;
      int * cc = NULL;
      bb = &aa;
      print_number(bb);
      print_number(cc);
      return 0;
}
IZLAZ (na Linuxu)
10
a.out: main.cpp:6: void print_number(int*): Assertion `myInt!=__nu
ll' failed.
Aborted (core dumped)
```

Slika 5.2. Primer assert

Makro assert se može koristiti na sledeće načine:

```
int sum = (assert(a > 0), a) + (assert(b < 0), b);
```

ili

```
assert(a > 0 && b < 0)
int sum = a + b;
```

što predstavlja isti kod.

Na slici 5.3. dat je primer gde se koristi obrada (poznatih) izuzetka iz standardne biblioteke.

```
#include <iostream>
#include <exception>  // za izuzetak bad_alloc
using namespace std;
```

```
int main()
{
   cout << "Unesite koliki niz int elemenata zelite: ";
   try
   {
      int input = 0;
      cin >> input;
      int* numArray = new int [input];
      delete[] numArray;
   }
   catch (std::bad_alloc& exp)
   {
      cout << "Objasnjenje: " << exp.what() << endl;
   }
   catch(...)
   {
      cout << "Svi ostali izuzeci" << endl;
   }
   return 0;
}
```

Slika 5.3. Obrada izuzetka tipa std::bad_alloc

Uključena su zaglavlja iostream i exception, za unos, ispis i za izuzetak std::bad_alloc te je navedeno korišćenje prostora imena std. Ako biste u funkciji main uneli vrednost -1 došlo bi postavljanja izuzetka std::bad_alloc i bilo bi ispisano metodom what da je u pitanju "bad array new length". Obrađen je objekat exp klase std::bad_alloc. Ova klasa je naslednica klase std::exception (zaglavlje exception).

Neke nasledice klase std::exception su klase:

bad_alloc – neuspeh operatora new;

bad_cast – eksplicitna konverzija pogrešnog tipa;

ios_base::failure – postavljaju ga funkcije i metode biblioteke iostream.

Rezime poglavlja izuzeci

U poglavlju izuzeci obrađeni su: try-catch blok, korisnički definisani izuzeci, propagacija izuzetka, postavljanje ograničenja funkcije u smislu postavljanja izuzetka, funkcija terminate, funkcija unexpected, funkcija abort, makro assert te korišćenje klase std::exception.

Zadaci za proveru znanja

1. Napisati program koji obrađuje izuzetak kod traženja inverzne matrice 3x3 za slučaj da je determinanta 0.
2. Napisati program koji obrađuje izuzetak ako je ulazni parametar metode prost broj čija je vrednost veća od 500, a manja od 1000.
3. Napisati program koji obrađuje izuzetak ako je broj poziva rekurzivne metode veći od 20.
4. Napisati program koji koristi korisničke funkcije za terminate i unexpected.
5. Napisati program koji koristi klasu MojIzuzetak koja nasleđuje klasu std::exception i koja ima nadjačanu metodu what tipa const char* () const throw() te koja ispisuje tekst koji opisuje izuzetak. Za postavljanje izuzetka uzeti proizvoljan slučaj.

6. Uvod u standardnu biblioteku (C++ i STL)

Cilj poglavlja je da student ovlada korišćenjem nekih često korišćenih klasa i funkcija iz standardne biblioteke (misli se i na std prostor imena i na Standard Template Library – standardnu biblioteku šablona): string, array, list, vector, stack, pair, queue, deque, map i sort.

Kontejner je struktura podataka koja sadrži elemente istog tipa (stek, red, liste, vektori, red sa dva kraja). Za pristup elementima kontejnera koriste se iteratori. Deklaracija iteratora za npr. listu celih brojeva je: std::list<int>::iterator iter, tako da se onda koriste metode liste: begin() i end() za obilazak liste.

Primer za ispis elemenata liste koja sadrži cele brojeve:

```
std::list<int> intlista;
std::list<int>::iterator it;
for(it=intlista.begin(); it!=intlista.end();it++)
  std::cout<<*it<<std::endl;
```

6.1. Klasa string

Standardna C++ biblioteka omogućuje rad sa stringovima (nizovima karaktera) korišćenjem klase string.

Formalno je klasa string kao što sledi:

```
typedef basic_string<char> string;
```

Na slici 6.1.1. dat je jednostavan program koji kreira dva stringa str1 i str2, a onda vrši dodelu kojom str3 ima sadržaj kao i str1 (sledi ispis sadržaja str3) te nakon toga vrši konkatenaciju str1 i str2 koja se dodeljuje stringu str3 (gde opet sledi ispis sadržaja str3) i na kraju se ispisuje koliko karaktera ima str3.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
      string str1 = "C++";
      string str2 = " je OK";
      string str3 = str1;
      cout << "str3 : " << str3 << endl;
      str3 = str1 + str2;
      cout << "str1 + str2 : " << str3 << endl;
      int len = str3.size();
      cout << "str3.size() :  " << len << endl;
      return 0;
}
IZLAZ
```

```
str3 : C++
str1 + str2 : C++ je OK
str3.size() : 9
```

Sika 6.1.1. Konkatenacija stringova

Na slici 6.1.2. dat je primer nekih metoda klase string. Pripadni izlazi programa dati su na slici 6.1.3. Kreira se string str1 kome se prosleđuje string literal "Popokatepetl", a nakon toga se ispisuje taj string. Metoda stringa length (sinonim je size) ispisuje dužinu stringa str1 (vrednost 12).

Metoda stringa at vraća referencu na karakter na indeksu koji se prosleđuje kao parametar, tako da je moguće uzeti ili promeniti karakter na datoj poziciji u stringu. U primeru je dato da se ispiše uzeti karakter na indeksu 1 (karakter o). Sledeće dve metode koje su date su front i back koje daju referencu na prvi i poslednji karakter datog stringa (karakteri P i l).

String str2 inicijalizovan je stringom str1. Radi se o nezavisnoj kopiji tako da promena stringa str2 (str[2]='P') ne utiče na originalni string str1 (Nezavisna kopija PoPokatepetl originala Popokatepetl).

Data je inicijalizacija stringa str3 sa dva parametra, pri čemu prvi prarametar govori koliko je ponavljanje drugog parametra (karakter '*').

String str4 inicijalizovan je podstringom stringa str1 koji počinje na 8. indeksu i dužine je 3 karaktera ("pet").

String str5 je inicijalizovan korišćenjem iteratora stringa str1, tako da je korišćena metoda begin i pripadni ofset 8 i 11 čime je isto dobijen prethodni podstring ("pet").

String str6 je inicijalizovan dodelom stringa str5, a onda je obrisan niz u stringu str metodom clear, koja nije uticala na str6.

Inicijalizacija niza karaktera pomoću stringa se rešava metodom c_str.

Dodavanje na kraj stringa obavlja metoda append. Metoda append sa jednim parametrom ima niz koji se dodaje na kraj datog stringa (ili konkatenacija +=), dok verzija sa 3 parametra prihvata string, početni indeks podniza tog stringa i dužinu tog podniza.

Traženje početnog indeksa uzorka u datom stringu obavlja metoda find kojoj se prosleđuje uzorak koji se traži. Konstanta strng::npos predstavlja najveću vrednost indeksa (2^{64}-1). Ako se uzorak ne nađe u stringu, onda metoda find vraća vrednost -1.

Metoda substring može da prihvati jedan ili dva parametra. Ako ova metoda ima jedan parametar, onda se on odnosi na početni indeks podstringa datog stringa i vraća podstring koji ide od datog indeksa sve do kraja stringa. Ako metoda

substring ima dva argumenta, onda se radi o početnom indeksu podstringa datog stringa i dužini tog podstringa u karakterima.

Metoda erase briše podstring datog stringa od datog indeksa za datu dužinu podstringa u karakterima.

Metoda replace ima tri argumenta: prvi je početni indeks podstringa datog stringa, drugi je dužina tog podstringa u karakterima i treći je niz koji će zameniti taj podstring.

Prebacivanje iz malog u veliko slovo obavlja funkcija toupper. U "for-each" petlji se uzima redom referenca na slovo datog stringa, a onda se na njega deluje funkcijom toupper.

Metoda stringa compare poredi stringove po abecednoj vrednosti sa drugim stringom koji je njen argument. Ako su stringovi isti metoda compare vraća 0. Ako je prvi string "veći" od drugog stringa, onda ova metoda vraća pozitivan celi broj, a ako je prvi string manji od drugog onda ova metoda vraća negativan celi broj.

```
#include <string>
#include <iostream>
using namespace std;
int main()
{
  string str1("Popokatepetl");
  cout << str1 << endl;
  int len = str1.length(); // lenghth() ili size()
  cout << "Duzina stringa " << str1 << " je " << len << endl;
  char ch = str1.at(1);    // ili str1[1]
  cout << "Na indeksu 1 stringa "<< str1 << " je karakter "
       << ch << endl;
  char prvi = str1.front();  // ili str1[0]
  cout << "Prvi karakter je " << prvi << endl;
  char poslednji = str1.back();   // ili str1[str1.length()-1]
  cout << "Poslednji karakter je " << poslednji << endl;

  string str2(str1);
  str2[2] = 'P';
  cout << "Nezavisna kopija "<< str2 << " originala "
       << str1<<endl;

  string str3(30, '*');  // 30 zvezdica
  cout << "30 zvezdica "<< str3 << endl;

  string str4(str1, 8, 3); // string inicijalizovan podstringom
                           // string str1 i to od indeksa 8
                           // naredna 3 karaktera
```

```
cout << "Podstring (indeksi 8,9. i 10. karakter) stringa "
     << str1 << " je "<< str4 << endl;

string str5(str2.begin() + 8, str2.begin() + 8 + 3); // iterator
cout << "Isti podstring preko iteratora  "<<str5 << endl;

string str6 = str5;
str5.clear();
cout <<"Nezavisna kopija "<< str6 << endl; // nezavisna kopija

const char* charstr = str1.c_str(); // stringa u niz karaktera
cout << "Iz stringa u niz karaktera "<<charstr << endl;

cout << "Dodavanje !!! na kraj stringa " << str1;
str1.append("!!!"); // dodavanje !!! na kraj stringa
                    // moze i str1 += "!"
cout << " je " << str1 << endl;

// dodavanje podstringa na kraj stringa
string strSlova("ABCDEFGH");
cout << "Dodavanje podstringa " << strSlova.substr(2, 3)
     << " stringa " << strSlova << " na kraj stringa "
     << str1 << " je ";
    str1.append(strSlova, 2, 3);  // dodavanje na kraj
                                  // podstringa od 2. karaktera
                                  //sledeca 3 karaktera
    cout << str1 << endl;

// trazenje indeksa uzorka, ako se ne nadje uzorak vraca se -1
string strTrazi("pet");
if (str1.find(strTrazi) != string::npos)
    cout << "Podstring "<< strTrazi << " je nadjen na indeksu "
         << str1.find(strTrazi) << " u stringu "<< str1 << endl;
    else
          cout << "Podstring " << strTrazi
               << " nije nadjen u stringu "<< str1 << endl;

//  podstring stringa od indeksa koliko karaktere
cout << "Podstring stringa " << str1
     << " od indeksa 3 duzine 4 karaktra je "
     <<str1.substr(3, 4) << endl;

// podstring od indeksa 3 do kraja stringa
cout << "Podstring stringa "<< str1
     << " od indeksa 3 do kraja stringa je "
     << str1.substr(3) << endl;
```

```
  // brisanje od indeksa 3 sledecih 4 karaktera
  string brojanje("0123456789");
  cout << "........................................."
       << brojanje << endl;
  brojanje.erase(3, 4);
  cout << "Brisanje od indeksa 3 sledeca 4 karaktera "
       <<brojanje << endl;

  // iteratorsko brisanje, od indeksa do iskljucno indeksa
  brojanje = "0123456789";
  brojanje.erase(brojanje.begin() + 5, brojanje.end() - 2);
  cout << brojanje << endl;

  // menjanje 4 karaktera od indeksa 2 sa tackicama
  brojanje = "01234567";
  cout << "........................................."
       << brojanje << endl;
  brojanje.replace(2, 4, "....");
  cout << "Menjanje 4 karaktera od indeksa 2 sa tackicama "
       << brojanje << endl;

  // mala u velika slova
  string strSlucaj("AAbbCCdd");
  cout << "String " << strSlucaj;
  for (auto &ch : strSlucaj) ch = toupper(ch);
  cout << " prebacen u velika slova je " << strSlucaj << endl;

  // komparacija stringova
  string strABC("ABC");
  string strDEF("DEF");
  string strGHI("GHI");
  cout << "Komparacija stringova " << strDEF << " i " << strABC
       << " vraca " << strDEF.compare(strABC) << endl;
  cout << "Komparacija stringova " << strDEF << " i " << strDEF
       << " vraca " << strDEF.compare(strDEF) << endl;
  cout << "Komparacija stringova " << strDEF << " i " << strGHI
       << " vraca " << strDEF.compare(strGHI) << endl;
  strABC.at(0) = 'W';
  cout << strABC;
  return 0;
}
```

Sika 6.1.2. Neke metode klase string

```
Duzina stringa Popokatepetl je 12
Na indeksu 1 stringa Popokatepetl je karakter o
Prvi karakter je P
Poslednji karakter je l
```

```
Nezavisna kopija PoPokatepetl originala Popokatepetl
30 zvezdica ******************************
Podstring (indeksi 8,9. i 10. karakter) stringa Popokatepetl je pet
Isti podstring preko iteratora  pet
Nezavisna kopija pet
Iz stringa u niz karaktera Popokatepetl
Dodavanje !!! na kraj stringa Popokatepetl je Popokatepetl!!!
Dodavanje podstringa CDE stringa ABCDEFGH na kraj stringa Popokatepetl!!!
je Popokatepetl!!!CDE
Podstring pet je nadjen na indeksu 8 u stringu Popokatepetl!!!CDE
Podstring stringa Popokatepetl!!!CDE od indeksa 3 duzine 4 karaktra je
okat
Podstring stringa Popokatepetl!!!CDE od indeksa 3 do kraja stringa je
okatepetl!!!CDE
.....................................0123456789
Brisanje od indeksa 3 sledeca 4 karaktera 012789
0123489
.............................................01234567
Menjanje 4 karaktera od indeksa 2 sa tackicama 01....67
String AAbbCCdd prebacen u velika slova je AABBCCDD
Komparacija stringova DEF i ABC vraca 1
Komparacija stringova DEF i DEF vraca 0
Komparacija stringova DEF i GHI vraca -1
WBC
```

Sika 6.1.3. Izlaz koda datog na slici 6.1.2.

Na slici 6.1.3. dat je izlaz koda koji je nastao izvršavanjem koda sa slike 6.1.2. Probajte da menjate parametre datih metoda i pratite pripadne izlaze programa.

6.2. Klasa array

Klasa array se koristi za smeštanje fiksnog broja objekata datog tipa. Formalno je klasa array kao što sledi:

```
template < class T, size_t N > class array;
```

```
#include <iostream>
#include <array>
#include <random>
class Hm{
    static int brojInstanci;
    mutable int brojpozivametodaobjekta;//izmenjiv iz const f-je
    int id;
  public:
    Hm()
    :
    brojpozivametodaobjekta(1)
    {
```

```
      id = ++brojInstanci;
    }
    int BrojPozivaMetoda() const
    {
      return ++brojpozivametodaobjekta;
    }
    static int BrojInstanci()
    {
      return brojInstanci;
    }
    int UzmiID()
    {
      brojpozivametodaobjekta++;
      return id;
    }
};
int Hm::brojInstanci=0;
int main ()
{
  std::default_random_engine generator;
  std::uniform_int_distribution<int> distribution(1,39);//1..39
  Hm niz[10];
  std::cout<<"broj instanci Hm je "
           <<Hm::BrojInstanci()<<std::endl;

  for(auto i=0; i<10; i++)
  {
    int slucajni = distribution(generator);
    for(auto j=0; j<slucajni; j++)    {
       niz[i].BrojPozivaMetoda();
    }
    std::cout<<"broj pristupa metodama objekta niz["
             <<i<<"]="<<niz[i].BrojPozivaMetoda()<<std::endl;
  }
  std::array<Hm,5> myarray = { };
  for ( auto it = myarray.begin(); it != myarray.end(); ++it )
    std::cout << ' ' << (it)->UzmiID();
  for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
    std::cout << ' ' << (it)->BrojPozivaMetoda();
  std::cout << "\nprvi     :" << myarray.front().UzmiID()
           << std::endl;
  std::cout << "poslednji:" << myarray.back().UzmiID()
           << std::endl;
  myarray.front() = Hm();
  std::cout<<myarray.at(0).UzmiID()<<std::endl;
  std::cout<<myarray[0].UzmiID()<<std::endl;
  for(Hm& x : myarray )
```

```
    std::cout<<x.UzmiID()<<",";
  std::cout<<std::endl;
  for ( auto rit=myarray.rbegin() ; rit < myarray.rend(); ++rit )
     std::cout << rit->UzmiID() << ",";
  return 0;
}
```

Sika 6.2.1. Primer primene klase array

Na slici 6.2.1. dat je primer korišćenja klase array. Opisana je klasa Hm koja ima blok privatnih članova: statički celobrojni član brojInstanci, celobrojni mutable član brojpozivametodaobjekta i celobrojni id.

Javni blok klase Hm čine:

- konstruktor koji inicijalizuje broj poziva metoda objekta na 1, povećava statički član broj instanci i takvu vrednost dodeljuje id članu;
- metoda BrojPozivaMetoda koja inkrementira broj poziva metoda objekta;
- statička metoda BrojInstanci koja vraća koliko je bilo kreiranih instanci od startovanja programa;
- metoda UzmiID ažurira broj poziva metoda objekta i vraća id objekta.

Van zaglavlja klase definisan je statički član brojInstanci (vrednost 0). U funkciji main kreiran je generator slučajnih brojeva i uniformna distribucija celih brojeva tako da će biti generisani slučajni brojevi u rasponu od 1 do 39.

Kreiran je niz od 10 elemenata tipa Hm, a onda se ispisuje broj kreiranih instanci klase Hm.

U for petlji, koja ima 10 iteracija, generiše se slučajnu broj od 1 do 39 koristeći uniformnu distribuciju i generator. Tako generisani broj predstavlja broj iteracija u unutrašnjoj for petlji. U unutrašnjoj petlji poziva se za tekući objekat niza `niz` određen vanjskom petljom metoda BrojPozivaMetoda. Nakon završene unutrašnje petlje ispisuje se za tekući element niza `niz` BrojPozivaMetoda.

Kreira se array od 5 elemenata tipa Hm, čiji je identifikator myarray. U for petlji korišćenjem iteratora objekta myarray (metode begin() i end()) ispisuju se redom id članovi elemenata koje čuva objekat myarray.

Sledeća iteratorska petlja koristi metode cbegin() i cend() da bi pozvala inspektorsku metodu BrojPozivaMetode svakog objekta koga čuva objekat myarray, uz ispisavanje povratne vrednosti navedene metode.

Daje se primer korišćenja metoda front i back za dohvatanje elemenata na početku i kraju array objekta. Ove metode vraćaju referencu tako da su lvalue. Metodom at ili indeksiranjem dohvata se element koji je smešten u array čiji je indeks naveden kao parametar.

Navedeno je koriščenje for petlje u smislu for-each, gde je u zagradama navedeno kako će biti uzet element koji je u array polju (u primeru preko

reference), znak dvotačka i onda sledi array identifikator (myarray). Ova petlja ispisuje povratnu vrednost metode UzmiID za sve elemente koji su smešteni u myarray.

Na kraju, dato je ispisivanje id elementa pozivom metode UzmiID, ali iteratorima koji se kreću od kraja ka početku (metode: rbegin(), rend()).

6.3. Struktura pair

Struktura pair predstavlja rešenje za par objekata pri čemu svaki od elemenata može biti proizvoljnog tipa.

Formalno je struktura pair kao što sledi:

```
template <class T1, class T2> struct pair;
```

```
#include <utility>      // std::pair
#include <iostream>     // std::cout
int main() {
      std::pair<int, char> foo(10, 'a');
      std::pair<int, char> bar(90, 'z');

      foo.swap(bar);

      std::cout << "foo contains: " << foo.first;
      std::cout << " and " << foo.second << '\n';

      return 0;
}
```

Sika 6.3.1. Primer primene strukture pair

Na slici 6.3.1. dat je mali primer strukture pair. Parovi foo i bar sadrže po par objekta tipa int, char. Pozvana je metoda swap objekta foo sa paramterom bar da bi parovi zamenili međusobno vrednosti. Sledi ispis prvog i drugog elementa para objekta foo.

6.4. Klasa list

Klasa list pripada STL biblioteci i koristi se za kreiranje liste objekata datog tipa koji se navodi kao parametar šablona klase. Objekti liste nisu u kontinualnom memorijskom prostoru, već su kao dvostruko spregnuti "razbacani" po memoriji. Na ovaj način je obezbeđeno dobro insertovanje i brisanje elemenata liste. Mana je sekvencionalan pristup elementu liste. Insertovanje ili brisanje elementa liste ne utiče na validnost iteratora (elementi liste ostaju na svom mestu u memoriji).

Na slici 6.4.1. dat je program koji koristi klasu list. Uključena su zaglavlja: algorithm, iostream i list. U funkciji main kreiran je objekat lista1 tipa std::list<int> koji je inicijalizovan listom celobrojnih vrednosti (vrednosti 1, 2, 3, 4). Metoda push_front(0) objekta lista1 dodaje element 0 na početak liste. Metoda push_back(6) objekta lista1 dodaje element 6 na kraj liste. Funkcija find traži i vraća vrednost std::list<int> iteratora koji se odnosi na element čija je vrdnost 6 od početka liste lista1 (lista1.begin()) do kraja liste lista1 (lista1.end()). Ako se iteratorom ne dođe do samog kraja liste, onda je nađen element liste lista1 koji ima vrednost 6 te se na njegovo mesto insertuje ceo broj čija je vrednost 5. Na kraju se ispisuju svi elementi liste lista1.

```
#include <algorithm>
#include <iostream>
#include <list>
int main()
{
	// kreiranje liste celobrojnih vrednosti
	std::list<int> lista1 = { 1, 2, 3, 4 };

	// dodavanje elementa na početak liste
	lista1.push_front(0);
	// dodavanje elemenata na kraj liste
	lista1.push_back(6);

	// nadji vrednost iteratora koji se odnosi na element liste
	// cija je vrednost 6
	std::list<int>::iterator it =
			std::find(lista1.begin(), lista1.end(), 6);
	// ako je nadjen element čija je vrednost 6,
	// onda insertuj ceo broj 5 na mesto na koje se odnosi
	// iterator
	if (it != lista1.end())
	{
		lista1.insert(it, 5);
	}

	// ispis svih elemenata liste lista1
	for (int n : lista1)
	{
		std::cout << n << '\n';
	}
	return 0;
}
```

Slika 6.4.1. Primer klase list

Na slici 6.4.2. dat je primer korišćenja metode sort klase list. Uključena su zaglavlja: iostream, list i string. Funkcija compare_nocase prihvata dve reference na konstantan string. Koristi se neoznačeni ceo broj ii kao indeks da bi se proveravalo da se nije stiglo do kraja kraćeg stringa. U telu while naredbe proverava se odnos dva, prethodno pretvorena (tolower), mala slova koji se nalaze na istom indeksu u oba stringa te se u slučaju da su različiti vraća rezultat provere. Ako su isti, onda se ide na sledeći par slova koji su na istom indeksu u oba stringa. Ako se dođe do kraja jednog stringa, onda se proveri da li je drugi string duži i vrati se takva provera. Ovaj primer poredi stringove bez osetljivosti na velika i mala slova.

```
#include <iostream>
#include <list>
#include <string>
bool compare_nocase(
               const std::string& first, const std::string& second)
{
  unsigned int ii = 0;
  while ((ii < first.length()) && (ii < second.length()))
  {
    if (tolower(first[ii]) < tolower(second[ii])) return true;
    else
      if (tolower(first[ii]) > tolower(second[ii])) return false;
    ++ii;
  }
  return (first.length() < second.length());
}
int main()
{
      std::list<std::string> lista2;
      std::list<std::string>::iterator it;
      lista2.push_back("Mercedes");
      lista2.push_back("Audi");
      lista2.push_back("BMW");
      lista2.push_back("bicikl");

      lista2.sort();
      std::cout << "lista2:";
      for (it = lista2.begin(); it != lista2.end(); ++it)
            std::cout << ' ' << *it;
      std::cout << '\n';

      lista2.sort(compare_nocase);
      std::cout << "lista2:";
      for (it = lista2.begin(); it != lista2.end(); ++it)
            std::cout << ' ' << *it;
```

```
	std::cout << '\n';
	return 0;
}
IZLAZ
lista2: Audi BMW Mercedes bicikl
lista2: Audi bicikl BMW Mercedes
```

Slika 6.4.2. Sortiranje liste

Na slici 6.4.2. u funkciji main kreirana je lista stringova lista2, a kreiran je i iterator za listu stringova it. U listu su stavljena 4 stringa ("Mercedes", "Audi", "BMW" i "bicikl"). Pozvana je metoda sort objekta lista2 bez parametara, a onda je u iteratorskoj petlji koristeći metode begin i end objekta lista2 prođeno kroz sve elemente sortirane liste lista2 (*it) te je dat njihov ispis. Stringovi su poređani prema osetljivosti na velika i mala slova.

Nakon ovoga urađena je poziv metode sort objekta lista2, ali joj je prosleđen parametar koji ukazuje na funkciju compare_nocase. Sada se opet ide kroz iteratorsku petlju i ispisuju elementi liste lista2. Stringovi su poređani prema neosetljivosti na velika i mala slova.

6.5. Klasa vector

Klasa vector pripada STL biblioteci i koristi se za kreiranje vektora objekata datog tipa koji se navodi kao parametar šablona klase. Objekti vektora su raspoređeni u kontinualnom memorijskom prostoru. Ovo nije baš podesno za ubacivanje i brisanje elementa vektora, a može se javiti i problem ako pri insertovanju elementa nema dovoljno kontinualnog memorijskog prostora gledano sa kraja vektora, tako da je potrebno da se izvrši dealokacija i premeštanje na novi kontinualni memorisjki prostor koji je dovoljno velik za novi broj elemenata. Prednost je direktan pristup elementu vektora. Insertovanje ili brisanje elementa vektora može uticati na validnost iteratora.

```
#include <iostream>
#include <vector>
int main()
{
	// kreiranje vektora celobrojnih vrednosti
	std::vector<int> vektor1 = { 1, 2, 3, 4 };

	// dodavanje elemenata na kraj vektora
	vektor1.push_back(5);
	vektor1.push_back(6);

	// ispis svih elemenata
	for (int n : vektor1)
	{
```

```
            std::cout << n << '\n';
        }
        return 0;
}
```

Slika 6.5.1. Primer klase vector

U primeru datom na slici 6.5.1. uključena su zaglavlja iostream i vector. kreiran je vektor koji sadrži cele brojeve 1,2,3 i 4. Pozvane su metode za dodavanje elemenata na kraj vektora (push_back), tako da su dodati elementi 5 i 6. Na kraju se u petlji ispisuje sadržaj elemenata vektora.

6.6. Klasa queue

Klasa queue predstavlja red koji predstavlja FIFO (First In First Out) strukturu podataka. Primer je red ispred šaltera (prvi koji dođe biće uslužen).

```
#include <iostream>
#include <queue>
int main ()
{
  std::queue<int> myqueue;
  int myint;

  std::cout << "Unesite cele brojeve (0 je za kraj):"<<std::endl;

  do {
    std::cin >> myint;
    if(myint) myqueue.push (myint);
  } while (myint);

  std::cout << "myqueue obrada: ";
  while (!myqueue.empty())
  {
    std::cout << ' ' << myqueue.front();
    myqueue.pop();
  }
  std::cout << '\n';

  return 0;
}
IZLAZ
Unesite cele brojeve (0 je za kraj):
3 8 5 9 2 4 0
myqueue obrada: 3 8 5 9 2 4
```

Slika 6.6.1. Primer klase queue

Na slici 6.6.1. dat je primer korišćenja klase queue. Uključena su zaglavlja iostream i queue. U funkciji main kreiran je objekat tipa reda celih brojeva (myqueue). U do petlji unose se celi brojevi dok se ne unese 0. Svaki uneseni celi broj se stavlja u red myqueue korišćenjem metode push. U while petlji se proverava da li red nije prazan, tako da sve dok postoji bar jedan element u redu, biće obrađen najstariji element reda prema vremenu dospeća u red. Elementi se uzimaju korišćenjem metode front pri čemu se vrši ispis tog elementa.

6.7. Klasa deque

Klasa deque (double ended queue) predstavja red kome se mogu dodavati elementi na oba kraja.

```
#include <iostream>
#include <deque>
int main()
{
	std::deque<int> mydeque = { 10,20,30 };

	mydeque.emplace_front(111);
	mydeque.emplace_front(222);
	mydeque.emplace_back(888);
	mydeque.emplace_back(999);

	std::cout << "mydeque contains:";
	for (auto& x : mydeque)
		std::cout << ' ' << x;
	std::cout << '\n';
	return 0;
}
```

Slika 6.7.1. Primer klase deque

U primeru na slici 6.7.1. uključena su zaglavlje iostream i deque. U objekat mydeque tipa deque koji sadrži cele brojeve smešteni su brojevi 10, 20 i 30. Pozivanjem metode emplace_front dodati su elementi na početak reda (brojevi 111 pa 222), dok su pozivanjem metode emlace_back dodati brojevi 888 pa 999 na kraj reda. U pelji se ispisuju elementi koje sadrži objekat mydeque (222 111 10 20 30 888 999).

6.8. Klasa stack

Klasa stack se koristi za relizaciju steka kao LIFO (Last In First Out) strukture podataka. LIFO struktura je kao šaržer u kome će poslednji stavljeni metak biti prvi ispaljen.

```
#include <iostream>
#include <stack>
using namespace std;
int main()
{
    stack<int> mystack;
    mystack.push(1);
    mystack.push(2);
    mystack.push(3);
    mystack.push(4);
    // u steku su gledano od vrha: 4, 3, 2, 1

    mystack.pop();
    // ostaje 3, 2, 1
    mystack.pop();
    // ostaje 2, 1

    while (!mystack.empty()) {
        cout << ' ' << mystack.top();
        mystack.pop();
    } // ispisuje 2 1
}
```

Slika 6.8.1. Primer klase stack

U primeru na slici 6.8.1. uključena su zaglavlja: iostream i stack te je navedeno korišćenje prostora imena std. U funkciji main kreiran je objekat mystack tipa stack koji sadrži cele brojeve. Korišćenjem metode push stavljeni su redom celi brojevi redom: 1, 2, 3 i 4. Pozvana je dva puta metoda pop koja uzima i izbacuje najmlađi element steka gledano prema vremenu stavljanja na stek. Uzeti su elementi 4 pa 3 tako da je ostalo 2 i 1 gledano od vrha steka. U while petlji proverava se nije li stek prazan te ako ima elemenata metodom pop se uzima i izbacuje iz steka najmlađi element (ispisivaće se 2 1).

6.9. Klasa map, mulitmap

Klasa map omogućuje asocijativno mapiranje, parovima ključ vrednost, pri čemu su ključevi jedinstveni. Klasa multimap omogućuje asocijativno mapiranje, parovima ključ vrednost, pri čemu ključevi nisu jedinstveni.

```
#include <iostream>
#include <map>
using namespace std;
int main () {
    multimap<char,int> niz;
    niz.insert(make_pair('x',50));
```

```
    niz.insert(make_pair('y',100));
    niz.insert(make_pair('y',150));
    niz.insert(make_pair('y',200));
    niz.insert(make_pair('z',250));
    niz.insert(make_pair('z',300));
    for (char c='x'; c<='z'; c++)
    {
        cout << "Postoji " << niz.count(c)
             << " elemen. ciji kljuc je " << c << ":";
        multimap<char,int>::iterator it;
        for (
              it=niz.equal_range(c).first;
              it!=niz.equal_range(c).second;
              ++it
        )
        std::cout << ' ' << (*it).second;
        std::cout << '\n';
    }
    return 0;
}
IZLAZ
Postoji 1 elemen. ciji kljuc je x: 50
Postoji 3 elemen. ciji kljuc je y: 100 150 200
Postoji 2 elemen. ciji kljuc je z: 250 300
```

Slika 6.9.1. Primer klase multimap

Na slici 6.9.1. dat je primer korišćenja multimap klase. Uključena su zaglavlja iostream i map. U funkciji main kreiran je objekat niz tipa multimap koji ima asocijaciju char "u" int. U objekat niz insertovani su parovi pozivom metode insert koja kao parametar prima par kreiran pozivom funkcije make_pair. Dodati su parovi: x 50, y 100, y 150, y 200, z 250 i z 300. U for petlji koja prolazi karakterima malih slova od c do z ispisuje se broj istih ključeva za pripadno slovo kao ključ. Kreira se pripadni iterator tipa multimap<char,int>::iterator, a onda se u iteratorskoj petlji niza niz koristi metoda equal_range čiji parametar predstavlja ključ na osnovu koga ova metoda vraća par koji određuje opseg u kome se nalazi dati ključ. Od tekućeg iteratora uzima se vrednost koja predstavlja par i ispisuje se vrednost za dati ključ ((*it).second). Na kraju je dati izlaz pri izvršavanju programa.

6.10. Sortiranje: qsort, sort

Na slici 6.6.1 dat je primer korišćenja funkcije qsort.

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

```
class Automobil{
   string naziv;
   double cena;
public:
   Automobil(string naziv, double cena)
   :
   naziv(naziv),
   cena(cena)
   {}
   void Ispisi()
   {
     cout << naziv.data() << " " << cena << std::endl;
   }
   double GetCena()
   {
     return cena;
   }
};
int poredi(const void *a, const void *b)
{
   return static_cast<int>(  (*(Automobil*)a).GetCena()
                           - (*(Automobil*)b).GetCena()  );
}
int main(void)
{
  Automobil salon[] = {
    { "Audi", 80000 },
    { "BMW", 30000 },
    { "Mercedes", 100000 }
  };
  qsort(salon,
        sizeof(salon)/sizeof(Automobil),
        sizeof(Automobil),
        poredi);
  for (int i = 0; i < sizeof(salon) / sizeof(Automobil); i++)
    (salon + i)->Ispisi();
  return 0;
}
IZLAZ
BMW 30000
Audi 80000
Mercedes 100000
```

Slika 6.10.1. Primer primene funkcije qsort

Na slici 6.10.1. dat je primer korišćenja funkcije qsort. Već je ranije bilo reči o klasi Automobil. U ovom primeru ova klasa ima privatne članove naziv i cena (tipovi string i double). U javnom bloku su: konstruktor koji ima dva argumenta

i koji postavlja stanje objekta, metoda Ispisi koja ispisuje stanje objekta i metoda GetCena koja vraća cenu automobila datog objekta.

Funkcija poredi vraća int vrednost koja može biti pozitivna, nula ili negativna. Dva argumenta navedene funkcije su tipa const void *, tako da funkcija može da prihvati pokazivače na bilo objekte bilo kog tipa. Zbog ovoga je potrebno kastovanje (mix c i c++) kojim se pokazivač na void kastuje u pokazivač na potrebni tip objekta, a onda se dereferenciranjem uzima taj objekat (ovde tipa Automobil) i poziva se njegova metoda GetCena. Vrši se oduzimanje cena dva automobila, a onda promena tipa u int jer je cena tipa double.

U funkciji main kreiran je niz salon koji ima tri objekta tipa Automobil (redom: Audi 80000, BMW 30000, Mercedes 100000). Pozvana je metoda qsort čiji su parametri: početak niza, broj objekata u nizu, veličina objekta i pokazivač na funkciju tipa int (const void *, const void *) koja vrši poređenje i koja vraća int vrednost (negativnu, 0 ili pozitivnu što zavisi od poređenja). Na osnovu te vrednosti se vrši ili ne vrši zamena mesta poređenih elemenata. Dat je izlaz sortiranih automobila prema ceni automobila.

```
#include <iostream>
#include <algorithm>    // std::sort
#include <vector>

bool myfunction (int i,int j) { return (i<j); }

struct myclass {
  bool operator() (int i,int j) { return (i<j);}
} myobject;

int main () {
  int myints[] = {6,2,1,5,4,3,7,8};
  std::vector<int> myvector (myints, myints+8);
               // 6 2 1 5 4 3 7 8

  // podrazumevani operator <
  std::sort (myvector.begin(), myvector.begin()+4);
  //(1 2 5 6) 4 3 7 8

  // koriscenje funckije myfunction kao komparatora
  std::sort (myvector.begin()+4, myvector.end(), myfunction);
  // 1 2 5 6 (3 4 7 8)

  // koriscenje objekta kao komparatora
  std::sort (myvector.begin(), myvector.end(), myobject);
  // 1 2 3 4 5 6 7 8
```

```
  // print out content:
  std::cout << "myvector contains:";
  for (
        std::vector<int>::iterator it=myvector.begin();
        it!=myvector.end();
        ++it
  )
    std::cout << ' ' << *it;
  std::cout << '\n';
  return 0;
}
IZLAZ
1 2 3 4 5 6 7 8
```

Slika 6.10.2. Primer primene funkcije sort

Na slici 6.10.2. dat je primer funckije sort. Uključena su zaglavlja: iostream, algorithm i vector. Definisana je funkcija myfunction tipa bool (int, int) koja vraća da li je prvi argument manji od drugog.

Definisana je struktura myclass koja ima definisan operator poziva funkcije koja prihvata dva argumenta i vraća da li je prvi argument manji od drugog. Kreiran je i objekat myobject navedenog tipa.

U funkciji main kreiran je niz myints koji sadrži cele brojeve 6,2,1,5,4,3,7 i 8. Kreiran je vektor myvector koji sadrži cele brojeve i inicijalizovan je nizom myints elemenata koji su redom na indeksima 0 do isključno 8. Stanje vektora myvector je 6 2 1 5 4 3 7 8.

Pozvana je funkcija sort koja sortira prva 4 elementa vektora myvector (myvector.begin(), myvector.begin()+4) korišćenjem podrazumevanog operatora poređenja <. Stanje vektora myvector je 1 2 5 6 4 3 7 8.

Pozvana je funkcija sort koja sortira od 4. elementa vektora myvector pa nadalje (myvector.begin()+4, myvector.end()) korišćenjem komparatorske funkcije myfunction. Stanje vektora myvector je 1 2 5 6 3 4 7 8.

Pozvana je funkcija sort koja sortira sve elemente vektora myvector (myvector.begin(), myvector.end()) korišćenjem objekta myobject čija će operatorska metoda poziva funkcije (koja prihvata 2 argumenta) biti korišćena za komparaciju ovih elemenata. Stanje vektora myvector je 1 2 3 4 5 6 7 8.

Rezime poglavlja uvod u standardnu biblioteku (C++ i STL)

U poglavlju uvod u standardnu biblioteku (C++ i STL) obrađeni su često korišćeni objekti i klase iz standardne biblioteke: string, array, list, vector, stack, pair, queue, deque, map i sort.

Zadaci za vežbanje

1. Napisati program koji koristi array za smeštanje 30 elemenata Fibonačijevog niza (1, 1, 2, 3, 5, 8, 13, ...). Ispisati rezultat deljenja datog elementa i njegovog prethodnika, počevši od drugog elementa niza.
2. Napisati program koji sortira listu koja sadrži tip Student. Sortiranje se vrši prema jednom od kriterijuma: indeks, ime ili prezime.
3. Napisati program koji simulira punjenje i pražnjenje steka koristeći stek koji sadrži elemente klase Student.
4. Napisati program koji simulira red čekanja u pošti koristići queue koji sadrži elemente tipa Penzioner.
5. Napisati program koji koristi mapiranje ključ-vrednost kao Automobil-ocena na NCAP testu (testovi sigurnosti automobila).
6. Napisati program koji sortira vektor koji sadrži elemente tipa Student po dva kriterijuma: broj indeksa i ime studenta.

7. Pametni pokazivači

Cilj poglavlja pametni pokazivači je da student ovlada primenom pametnih pokazivača: unique_ptr, shared_ptr i weak_ptr te da shvati problem i rešenje ciklične zavisnosti.

Ranije je rečeno da korišćenjem standardnih pokazivača programer određuje životni vek objekta koristeći operatore new i delete. Ideja sa pametnim pokazivačima je da programer kreira pametni pokazivač na objekat, a da ukidanje objekta vrši pametni pokazivač.

Za rad sa pametnim pokazivačima potrebno je uključenje zaglavlja memory. Pametni pokazivači pripadaju standardnom prostoru imena std.

7.1.1. Pokazivač unique_ptr

Šablon pametnog pokazivača unique_ptr je kao što sledi:

```
template <typename D, typename Deleter> class unique_ptr;
```

Pametni pokazivač unique_ptr ukazuje samo na jedan objekat (ili jedan niz objekata), tako da prevodilac neće dozvoliti sledeće:

```
std::unique_ptr<int> upi1(new int);
std::unique_ptr<int> upi2 = upi1;   // NOK
```

Pametni pokazivač unique_ptr je "ekskluzivni" vlasnik objekta ili niza na koji ukazuje.

Ono što bi moglo da se uradi u ovom smislu je da se koristi funkcija std::move kao što sledi:

```
std::unique_ptr<int> upi1(new int);
std::unique_ptr<int> upi2 = std::move(upi1);
```

pri čemu će se pri dealokaciji objekta na koji ukazuje unique_ptr pozvati operator delete.

Sintaksa za kreiranje unique_ptr pokazivača koji ukazuje na niz objekata je kao što sledi:

```
std::unique_ptr<Automobil[]> upi1(new Automobil[20]);
std::unique_ptr<Automobil> upi2(new Automobil[20]);
```

ili korišćenjem make_unique:

```
std::unique_ptr<int[]> upiarr = std::make_unique<int[]>(20);
```

pri čemu će se pri dealokaciji niza objekta na koji ukazuje unique_ptr pozvati operator delete[].

Deleter izraz se koristi da se specificira programerov način dealokacije onoga na šta ukazuje unique_ptr. U primeru datom na slici 7.1.1. klasa ObrisiObjekat ima šablonsku operatorsku metodu poziva funkcije (prihvata 1 argument, T*) koja će pri dealokaciji objekta obrisati objekat na koji ukazuje dati unuque_ptr i ispisaće se da je objekat dealociran. Program ispisuje celobrojnu vrednost 10 koju dobija pozivom dereferenciranja *upi1, te dva puta "Obrisano!".

```
#include <memory>
#include <iostream>
class ObrisiObjekat
{
public:
    template <typename T> void operator()(T* up)
    {
          delete up;
          std::cout << "Obrisano!" << std::endl;
    }
};
int main()
{
  std::unique_ptr<int, ObrisiObjekat> upi1(new int(10));
  std::cout << (*upi1) << std::endl;
  ObrisiObjekat oo;
  std::unique_ptr<double, ObrisiObjekat&> upi2(new double, oo);
  return 0;
}
```

Slika 7.1.1. Primer deleter izraza za unuque_ptr

Ako je ideja da se dohvati obični pokazivač sadržan u pametnom pokazivaču, onda se koristi metoda get:

```
std::unique_ptr<double> upi1(new double(1.41));
std::cout << (*(upi1.get()));
```

tako da bi se u navedenom kodu ispisalo 1.41.

7.1.2. Pokazivač shared_ptr

Šablon pametnog pokazivača shared_ptr je kao što sledi:

```
template <typename T> class shared_ptr;
```

Pametni pokazivač shared_ptr ukazuje na deljene objekte, tako da je dozvoljeno sledeće:

```
std::shared_ptr<double> spi1(new double (2.73));
```

```
std::shared_ptr<double> spi2 = spi1;
```

Interno, shared_ptr koristi brojač referenci na deljeni objekat (preko kontrolnog objekta koga stvara konstruktor shared_ptr) i ako broj referenci shared_ptr pokazivača (*shared counter*) na taj deljeni objekat padne na nulu, onda se vrši dealokacija objekta.

Kontrolni objekat poseduje:
- pokazivač koji ukazuje na deljeni objekat;
- brojač shared_ptr referenci (*shared counter*);
- brojač weak_ptr referenci (*weak counter*).

Kontrolni objekat zadužen za brojače referenci na deljeni objekat živi dok postoji bar jedan weak_ptr koji "gleda" na deljeni objekat jer mora da im pruži informaciju o alokaciji deljenog objekta.

Kreiranjem shared_ptr pokazivača radi se alokacija za kontrolni objekat i radi se alokacija za deljeni objekat:

```
std::shared_ptr<double> spi1(new double (1.618));
```

da bi se poboljšale performanse programa koristi se funkcija sdt::make_shared koja uradi jednu alokaciju za smeštanje deljenog objekta i kontrolnog objekta:

```
std::shared_ptr<double> spi(make_shared<double>(1.618));
```

Kod ovog tipa pametnog pokazivača može se desiti curenje memorije ako postoje bar dva objekta koji imaju shared_ptr pokazivač kao član koji ukazuje na onaj drugi objekat. Ovim se oni ciklično (međusobno) održavaju. Na slici 7.1.2. dat je program u kome se kreiraju dva objekta Elvis i Tom klase RnR na koje ukazuje po jedan shared_ptr. Oba objekta poseduju član tipa shared_ptr<RnR> koji su postavljeni da ukazuju na onaj drugi objekat.

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;
class RnR
{
public:
	string pevac;
	shared_ptr<RnR> ortak;
	RnR(string pevac)
	:
	pevac(pevac)
	{
		cout << "Konstruktor: " << pevac << endl;
	}
	~RnR()
```

```
	{
		cout << "Destruktor : "<< pevac << endl;
	}
	void PostaviOrtaka(shared_ptr<RnR> ortak)
	{
		this->ortak = ortak;
	}
};
int main()
{
	shared_ptr<RnR> Elvis(make_shared<RnR>("Elvis Presley"));
	shared_ptr<RnR> Tom(make_shared<RnR>("Tom Jones"));
	Elvis->PostaviOrtaka(Tom);
	Tom->PostaviOrtaka(Elvis);
	return 0;
}
IZLAZ
Konstruktor: Elvis Presley
Konstruktor: Tom Jones
```

Slika 7.1.2. Primer ciklične povezanosti i curenja memorije za shared_ptr

Rezultat prethodnog koda će biti da curi memorija jer se ne mogu pozvati destruktori ni za objekat Elvis ni za objekat Tom.

7.1.3. Pokazivač weak_ptr

U cilju izbegavanja curenja memorije ako postoji ciklične povezanost objekata na koje ukazuju pokazivači tipa shared_ptr uveden je weak_ptr pokazivač. Ovaj pokazivač postavlja slabu vezu u cikličnoj povezanosti.

Na slici 7.1.3. dat je modifikovan prethodni primer u kome se koriste i weak_ptr pokazivači kao član klase RnR te kao formalni argument metode PostaviOrtaka. Sada je izlaz programa korektan jer se pozivaju i destruktori objekata Elvis i Tom.

```
#include <iostream>
#include <string>
#include <memory>
using namespace std;
class RnR
{
public:
	string pevac;
	weak_ptr<RnR> ortak;
	RnR(string pevac)
	:
	pevac(pevac)
```

```
	{
		cout << "Konstruktor: " << pevac << endl;
	}
	~RnR()
	{
		cout << "Destruktor : "<< pevac << endl;
	}
	void PostaviOrtaka(weak_ptr<RnR> ortak)
	{
		this->ortak = ortak;
	}
};
int main()
{
	shared_ptr<RnR> Elvis(make_shared<RnR>("Elvis Presley"));
	shared_ptr<RnR> Tom(make_shared<RnR>("Tom Jones"));
	Elvis->PostaviOrtaka(Tom);
	Tom->PostaviOrtaka(Elvis);
	return 0;
}
IZLAZ
Konstruktor: Elvis Presley
Konstruktor: Tom Jones
Destruktor : Tom Jones
Destruktor : Elvis Presley
```

Slika 7.1.3. Primer upotrebe weak_ptr

Za proveru da li je objekat na koji ukazuje weak_ptr nedealociran koristi se metoda lock koja stvara privremeni shared_ptr objekat. Ako je pomenuti objekat dealociran, onda je shared_ptr jednak nuli.

Na slici 7.1.4. dat je kod koji bi trebalo dodati u prethodni primer. Funkcija DaLiJeZiv prihvata parametar tipa weak_ptr<int>, a onda pozivom njegove metode lock() kreira objekat spFROMwp tipa shared_ptr<int>. Sledi provera da li je spFROMwp „prazan“. Ako jeste ispisuje se da je objekat živ, inače se ispisuje da objekat nije živ.

```
// dodati funkciju DaLiJeZiv iznad funkcije main
void DaLiJeZiv(std::weak_ptr<int> weak)
{
	std::shared_ptr<int> spFROMwp(weak.lock());
	if (spFROMwp)
	{
		std::cout << "Ziv je ! " << (*spFROMwp) << endl;
	}
	else
	{
```

```
            std::cout << "Nije ziv. " << endl;
      }
}
//dodati u funkciju main pre naredbe return 0
      shared_ptr<int> spceo(make_shared<int>(1000));
      weak_ptr<int> wp = spceo;
      DaLiJeZiv(wp);
      spceo.reset();
      DaLiJeZiv(wp);
```

Slika 7.1.4. Primer upotrebe metode lock pametnog pokazivača weak_ptr

U funkciji main dodata je linija koda koja kreira objekat spceo tipa shared_ptr<int> koji ukazuje na objekat int koji ima vrednost 1000. Kreira se objekat wp tipa weak_ptr<int> koji ukazuje na objekat na koji ukazuje spceo. Poziva se funkcija DaLiJeZiv(wp) i ispisaće se "Ziv je !", nakon čega se uništava objekat na koji "gleda" spceo pozivom metode reset(). Novi poziv funkcije DaLiJeZiv(wp) ispisaće "Nije ziv.".

Rezime poglavlja pametni pokazivači

U poglavlju pametni pokazivači obrađeni su pokazivači unique_ptr, shared_ptr i weak_ptr. Objašnjena je njihova primena te problem i rešenje cikličnе zavisnosti.

Zadatak za vežbanje

Korišćenjem pametnih pokazivača napisati program koji simulira klub ljudi koji su ljubitelji jedne od tri rase pasa. Međusobno su ortaci u klubu oni koji imaju istu rasu psa. Nasumično, grupa dobija novog člana (vlasnika nasumične rase od dozvoljenih rasa). Nasumično, neki član izlazi iz grupe. Povezati novog člana prema rasi psa. Izbeći cikličnu povezanost.

8. Niti

Cilj poglavlja je da student ovlada nitima kao lakim procesima koji se primenjuju tamo gde je moguće razdvojiti celine koda koje se mogu izvršavati paralelno. Student bi trebalo da ovlada: kreiranjem niti, vezivanjem niti za datu funkciju, postavljanjem argumenta funkcije koju nit izvršava te sinhnronizacijom niti u smislu završetka njihovog rada.

```
#include <iostream>
#include <thread>
#include <chrono>
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
int main()
{
    std::thread t;
    std::cout << "pre startovanja niti, joinable: "
              << std::boolalpha << t.joinable() << std::endl;

    t = std::thread(foo);
    std::cout << "posle startovanja niti, joinable: "
              << t.joinable() << std::endl;

    t.join();
    std::cout << "posle join metode, joinable: " << t.joinable()
              << std::endl;
    return 0;
}
IZLAZ
pre startovanja niti, joinable: false
posle startovanja niti, joinable: true
posle join metode, joinable: false
```

Slika 8.1. Primer upotrebe klase thread

Na slici 8.1. dat je primer upotrebe klase thread. Uključena su zaglavlja iostream, thread i chrono. Funkcija foo tekuću nit koja izvršava ovu funkciju postavlja u stanje spavanja u trajanju od 1 sekunde.

U funkciji main kreiran je objekat t tipa thread. Ispisuje se stanje joinable objekta t koje je false, što znači da se nit ne izvršava. Niti t se dodeljuje da izvršava funkciju foo (`std::thread(foo)`) i nakon toga se "odmah" ispisuje vrednost joinable koja je sada true, jer nit će provesti bar 1 sekundu u funkciji

foo. Nakon ovoga poziva se metoda join niti t koja utiče na to da glavni program čeka da nit obavi svoj posao. Nakon toga se ispisuje stanje joinable koje je sada false.

```
#include <iostream>
#include <thread>
#include <chrono>
#include <string>
void foo(int i, std::string str, int pauza)
{
    for(;i>0;i--)
    {
        std::cout<<str<<std::endl;
        std::this_thread::sleep_for
                            (std::chrono::milliseconds(pauza));
    }
}
int main()
{
    std::thread t1, t2;
    t1 = std::thread(foo, 5, "Rock", 10);
    t2 = std::thread(foo, 5, "Pop",30);
    t1.join();
    t2.join();
    std::cout << "Sve niti su zavrsile posao" << std::endl;
    return 0;
}
IZLAZ
Pop
Rock
Rock
Rock
Pop
Rock
Rock
Pop
Pop
Pop
Sve niti su zavrsile posao
```

Slika 8.2. Primer upotrebe dve niti

Na slici 8.2. dat je primer upotrebe dve niti. Uključena su zaglavlja iostream, thread, chrono i string.

Funkcija foo prihvata tri argumenta: broj iteracija za for petlju, string koji će biti ispisan u svakoj iteraciji i celobrojnu vrednost koja se odnosi na milisekunde

pauze koja će biti postavljena u for petlji. U for petlji koja ima dati broj iteracija se ispisuje dati string i pravi data pauza.

U funkciji main kreirana su dva objekta tipa thread (t1 i t2). Nakon ovoga startovane su niti t1 i t2 koje izvršavaju, svaka svoju, funkciju foo. Nit t1 "svojoj" funkciji foo prosleđuje parametre da se petlja vrti 5 puta, da se ispisuje "Rock" i da se pravi pauza od 10ms. Nit t2 "svojoj" funkciji foo prosleđuje parametre da se petlja vrti 5 puta, da se ispisuje "Pop" i da se pravi pauza od 30ms. Sledeća linija koda je t1.join() koja sinhrono čeka da se nit t1 završi. Kada se završi nit t1 ide se dalje. Sada je poziv t2.join() te ako je u međuvremenu nit t2 već završila svoj posao odmah se ide dalje, inače se čeka da t2 završi svoj posao. Nakon što su obe niti završile svoj posao, ispisuje se "Sve niti su zavrsile posao".

Rezime poglavlja niti

U poglavlju niti obrađeno je kreiranje niti, vezivanje niti za datu funkciju, postavljanje argumenta funkcije koju nit izvršava te kako se sinhnronizuju niti u smislu završetka njihovog rada.

Zadaci za proveru znanja

1. Napisati program koji koristi dve niti pri čemu jedna nit menja slučajno prvu polovinu slova početne reči ("QWERTYUI"), a druga nit menja drugu polovinu reči početne reči. Prva nit pravi pauzu od 20ms, a druga od 30ms. Cilj je dobiti reč "Mercedes". Svaka nit menja jedno slovo ako nije pogođeno i pravi pauzu. Ispisuje se svaka promena početne reči. Svaka nit vrši maksimalno 100 promena slova. Program završava kada se napravi 200 promena slova ili kada se dobije reč "Mercedes".
2. Napisati program za množenje matrica dimenzija 3x3 korišćenjem tri niti, pri čemu je jedna nit zadužena za jednu kolonu odgovora.
3. Napisati dve niti od kojih jedna povećava promenljivu ceobroj (početna vrednost je 10) za 1 i pravi pauzu od 10ms, a druga umanjuje promenjivu ceobroj za 1 i pravi pauzu od 20ms. Kada ceobroj bude jednak 20 program završava rad.

9. Lambda

Cilj poglavlja je da student ovlada lambda funkcijama za pisanje kompaktnijeg i čitljivijeg koda. Opisuje se format lambda izraza, prenos parametara po vrednosti i referenci, korišćenje vanjskih promenljivih, postavljanje tipa povratne vrednosti lambda funkcije, navođenje parametara poziva te primeri tela lambda funkcije.

Lambda izraz predstavlja zapis anonimne funkcije. Format lambda funkcije je redom kao što sledi:

- `[ nacin_korišćenja_vanjskih_promenljivih ]`
  u srednjim zagradama postavlja se način korišćenja vanjskih promenljivih u kojima kao prefiks ispred identinfikatora promenljive može biti:
  - znak `=` za prenos po vrednosti;
  - znak `&` za prenos po referenci;

    Napomena: ako nema identifikatora iza znaka, onda se sve varijable lambde koriste po znaku (= sve po vrednosti, & sve po referenci).
    Napomena: ako je navedeno npr. [&, prom1] onda se samo prom1 koristi po vrednosti, a ostale po referenci.

    Napomena: ako je navedeno npr. [=, &prom2] onda se samo prom2 koristi po referenci, a ostale po vrednosti.
- `( parametri_poziva )`
  parametri poziva lambda funkcije idu kao lista poziva standardne funkcije, sa navođenjem tipa i pripadnog identifikatora parametra;
- `-> povratni_tip`
  iza znaka -> se navodi povratni tip rezultata lambda funkcije;
- `{ telo funkcije }`
  telo lambda funkcije je kao i kod standardne funkcije smešteno u složeni blok.

Na slici 8.1. dat je program koji sortira niz elemenata tipa double. U funkciji main dat je niz x koji ima 3 elementa tipa double. Pozvana je funkcija abssort koja prihvata dva argumenta: pokazivač na tip double i broj elemenata. Stvarni argumenti su adresa niza x i broj elemenata niza x. U funkciji abssort pozvana je standardna metoda std::sort kojoj su sada prosleđena 3 argumenta: adresa od koje počinje niz, adresa na kojoj isključno završava niz i kao treći argument data je lambda funkcija. Prazne srednje zagrade govore da nema vanjskih promenljivih koje će se koristiti u ovoj lambda funkciji. Povratni tip lambda funkcije nije naveden, tako da će se vratiti tip koji odgovara izrazu iza naredbe return u lambda funkciji. U listi argumenata nalaze se dva argumenta tipa double. U telu lambda funkcije porede se po apsolutnoj vrednosti ova dva argumenta i rezultat tog poređenja je povratna vrednost lambda funkcije. Ovim

pozivom će elementi niza biti sortirani u rastućem redosledu prema apsolutnim vrednostima elemenata.

U funkciji main se nakon poziva funkcije abssort ispisuju elementi niza x po redosledu pripadnih rastućih indeksa. Na kraju je dat izlaz izvršavanja programa.

```
#include <algorithm>
#include <cmath>
#include <iostream>
void abssort(double* x, unsigned n)
{
      std::sort
      (
        x,
        x + n,
        []
        (double a, double b)
        {
          return (std::abs(a) < std::abs(b));
        }
      );
}
int main()
{
      double x[] = { 1.1, -3.3, 2.2, };
      abssort(x, sizeof(x) / sizeof(double));
      for (int i = 0; i < sizeof(x) / sizeof(double); i++)
            std::cout << x[i] << std::endl;
      return 0;
}
IZLAZ
1.1
2.2
-3.3
```

Slika 8.1. Lambda izraz

Na slici 8.2. dat je program koji koristi lambda izraz (funkciju) kao treći parametar for_each petlje (zaglavlje algorithm). U funkciji main kreiran je vektor v koji ima 9 elemenata celobrojnog tipa čije su vrednosti od 1 do 9. Ove vrednosti su postavljene u petlji u kojoj se vrednosti redom od 1 do 9 postavljaju na kraj vektora. Kreirana je celobrojna promenljiva evenCounter i inicijalizovana na 0.

U for_each pozivu navedena su tri argumenta:

- prvi argument je iterator koji se odnosi na početni element vektora v (v.begin()) odakle će se uzimati element po element i prosleđivati kao parametar lambda funkciji;
- drugi argument je iterator koji se odnosi na kraj vektora v (v.end()) dokle će ići iteracija uzimanja elemenata vektora i prosleđivanja lambda funkciji;
- treći argument je lambda funkcija.

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
      vector<int> v;
      for (int i = 1; i < 10; ++i) { v.push_back(i); }
      int evenCount = 0;
      for_each
      ( v.begin(),
        v.end(),
        [&evenCount]
        (int n)
        {
            cout << n;
            if (n % 2 == 0)
            {
              cout << " is even " << endl;
              ++evenCount;
            }
            else
            {
              cout << " is odd " << endl;
            }
         }
      );
      cout << "Broj parnih brojeva u vektoru v je "
           << evenCount << endl;
      return 0;
}
IZLAZ
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
```

```
9 is odd
Broj parnih brojeva u vektoru v je 4
```

Slika 8.2. Primer korišćenja lambda izraza sa vanjskom promenljivom

U ovoj lambda funkciji navedeno je da će se vanjski objekat evenCount koristiti u lambda funkciji kao referenca. Ovo znači da će se svaka izmena evenCount elementa u lambda funkciji reflektovati na original. Nije naveden povratni tip lambda funkcije tako da će to biti tip koji vraća izraz iza naredbe return. Naredba return nije navedena, tako da ostaje podrazumevana naredba return; što znači da će povratni tip lambda funkcije biti void. Ova lambda funkcija prihvata jedan argument tipa int i to je element vektora koji će joj biti prosleđen prolaskom kroz for_each petlju. U telu lambda funkcije ispisuje se prihvaćeni int element, onda se proverava da li element ima parnu ili neparnu vrednosti pri čemu se ispisuje ta informacija. Ako je tekući element paran, onda se inkrementira brojač parnih brojeva u vektoru (evenCount). Nakon završetka poziva for_each u funkciji main se ispisuje vrednost promenljive evenCount. Na kraju slike 8.2. dat je pripadni izlaz pri izvršavanju opisanog programa.

```
#include <iostream>
int main()
{
  double prom1(5.1), prom2(8.1);
  auto lam = [=,&prom2] (double a, double b)->int
             { prom2=88.8; return (a*b*prom1); };
 int i = lam(3.14,2.71);
      std::cout << i << ","<<prom1<<","<<prom2<<std::endl;
      return 0;
} //IZLAZ 43,5.1,88.8
```

Slika 8.3. Lambda sa korišćenjem vanjskih promenljivih

Na slici 8.3. dat je primer lambda funkcije lam gde se vanjska promenljiva prom1 koristi po vrednosti, a prom2 po referenci. Prihvataju se dva argumenta tipa double, a vraća tip int koji odgovara proizvodu stvarnih argumenata i promenljive prom1 kastovanu u int. Lambda lam menja stanje prom2 koja se koristi po referenci.

Rezime poglavlja lambda

U poglavlju lambda obrađeno je korišćenje lambda izraza. Pokazani su načini korišćenja vanjskih promenljivih, postavljanje povratnog tipa, lista argumenata i telo lambda funkcije.

Zadaci za proveru znanja iz poglavlja lambda:

1. Napisati lambda funkciju koja sortira niz koji sarži tip Automobil prema ceni automobila.
2. Napisati lambda funkciju koja iz prostih brojeva manjih od 1000 prikazuje one koji se završavaju cifrom 7.
3. Napisati lambda funkciju koja daje rešenja date kvadratne jednačine.

PRILOZI

Prilog 1. Neke C++ biblioteke

Prilog 2. ASCII tabela (osnovna)

Prilog 3. Odabrane metode nekih standardnih klasa koje se koriste u ovom materijalu

Prilog 1. Neke C++ biblioteke

Uslužne biblioteke	
<cstdlib>	alati opšte namene (kontrola programa, dinamička rapodela memorije, nasumični brojevi, sortiranje i pretraživanje)
<csignal>	funkcije i makro konstante za upravljanje signalima
<csetjmp>	makro (i funkcija) koji snima (i skače) na kontekst izvršavanja
<cstdarg>	rukovanje neodređenim brojem argumenata
<typeinfo>	uslužne informacije o tipu za vreme izvršavanja
<typeindex> (od C++11)	std::type_index klasa
<type_traits> (od C++11)	informacije o tipu za vreme izvršavanja
<bitset>	std::bitset šablon
<functional>	objekti funkcija, pozivi funkcija, operacije povezivanja i omotači referenci
<utility>	razne uslužne komponente
<ctime>	C uslužna biblioteka za vreme i datum
<chrono> (od C++11)	C++ uslužna biblioteka za vreme
<cstddef>	standardni makroi i definisani tipovi
<initializer_list> (od C++11)	std::initializer_list šablon
<tuple> (od C++11)	std::tuple klasa
<any> (od C++17)	std::any klasa
<optional> (od C++17)	std::optional šablon
<variant> (od C++17)	std::variant šablon
Biblioteke za dinamičko upravljanje memorijom	
<new>	uslužna biblioteka za niski nivo upravljanja memorijom
<memory>	uslužna biblioteka za visoki nivo upravljanja memorijom
<scoped_allocator> (od C++11)	biblioteka za multinivovske alokatore
<memory_resource> (od C++17)	polimorfni alokatori i memorisjski resursi
Biblioteke za numerička ograničenja	
<climits>	ograničenja za celobrojne tipove
<cfloat>	ograničenja za tip float
<cstdint> (od C++11)	tipovi fiksne veličine i ograničenja za druge tipove

<cinttypes> (od C++11)	formatiranje makroa, intmax_t i uintmax_t matematika i konverzije
<limits>	standardni način ispitivanja svojstava aritmetičkih tipova
Biblioteke upravljanje greškama	
<exception>	biblioteka za upravljanje izuzecima
<stdexcept>	standardni objekti izuzetaka
<cassert>	makro za uslovno prevođenje koji poredi njegove argumente sa nulom
<system_error> (od C++11)	sistemske greške
<cerrno>	makro koji sadrži poslednji broj greške
Biblioteke stringova	
<cctype>	funkcije za određivanje tipa u char podacima
<cwctype>	funkcije za određivanje tipa u wide-char podacima
<cstring>	razne funkcije za rukovanje char stringovima
<cwchar>	razne funkcije za rukovanje wide-char stringovima
<cuchar> (od C++11)	C funkcije za konverziju Unicode karaktera
<string>	std::basic_string šablon
<string_view> (od C++17)	std::basic_string_view šablon
<charconv> (od C++17)	konverzije std::to_chars i std::from_chars
Kontejnerske biblioteke	
<array> (od C++11)	std::array kontejner
<vector>	std::vector kontejner
<deque>	std::deque kontejner
<list>	std::list kontejner
<forward_list> (od C++11)	std::forward_list kontejner
<set>	asocijativni kontejneri std::set i std::multiset
<map>	asocijativni kontejneri std::map i std::multimap
<unordered_set> (od C++11)	neuređeni asocijativni kontejneri std::unordered_set i std::unordered_multiset
<unordered_map> (od C++11)	neuređeni asocijativni kontejneri std::unordered_map i std::unordered_multimap
<stack>	kontejnerski adapter std::stack
<queue>	kontejnerski adapteri std::queue i std::priority_queue
Biblioteka iteratora	
<iterator>	iteratori

Biblioteke algoritama	
<algorithm>	biblioteka raznih algoritama
<execution> (od C++17)	izvršavanje paralelnih algoritama
Numerička biblioteka	
<cmath>	opšte matematičke funkcije
<complex>	baratanje kompleksnim brojevima
<valarray>	manipulacija nizom vrednosti
<random> (od C++11)	generator i distribucija slučajnih brojeva
<numeric>	numeričke operacije nad podacima u kontejnerima
<ratio> (od C++11)	bratanje razlomcima
<cfenv> (od C++11)	funkcije za float (statusi i okruženje)
Biblioteke za ulaz i izlaz	
<iosfwd>	deklaracija klasa za IO
<ios>	std::ios_base class, std::basic_ios šabloni i neki definisani tipovi
<istream>	std::basic_istream šabloni i neki definisani tipovi
<ostream>	std::basic_ostream, std::basic_iostream šabloni i neki definisani tipovi
<iostream>	nekoliko standardnih objekata za IO
<fstream>	std::basic_fstream, std::basic_ifstream, std::basic_ofstream šabloni i neki definisani tipovi
<sstream>	std::basic_stringstream, std::basic_istringstream, std::basic_ostringstream šabloni i neki definisani tipovi
<iomanip>	kontrola formata ulaza i izlaza
<streambuf>	std::basic_streambuf šablon
<cstdio>	C IO funkcije
Biblioteka za lokallizaciju	
<locale>	biblioteka za lokalizaciju
<clocale>	C biblioteka za lokalizaciju
Biblioteka regularnih izraza	
<regex> (od C++11)	podrška za regularne izraze
Biblioteka za atomske operacije	
<atomic> (od C++11)	podrška za atomske operacije
Biblioteka za niti	
<thread> (od C++11)	std::thread klasa
<mutex> (od C++11)	primitive za mutex
<shared_mutex> (od C++14)	primitive za zaštitu deljenih podataka kojima paralelno pristupaju niti

<future> (od C++11)	primitive za asinhrono izračunavanje
<condition_variable> (od C++11)	klasa za blokiranje jedne ili više niti dok druga nit modifikuje deljenu varijablu (uslov)
Biblioteka za fajl-sistem	
<filesystem> (od C++17)	std::path klasa
<cstdlib>	usluge opšte namene:kotrola programa, dinamička alokacija memorije, slučajni brojevi, sortiranje i pretraživanje
<csignal>	funkcije i makroi upravljanja signalima
<csetjmp>	biblioteka za korisničko upravljanje sekvencom poziva
<cstdarg>	upravljanje neodređenim brojem argumenata
<typeinfo>	informacije o tipu za vreme izvršavanja
<typeindex> (od C++11)	std::type_index
<type_traits> (od C++11)	informacije o tipu za vreme prevođenja
<bitset>	std::bitset šablon
<functional>	objekti funkcija, pozivi funkcija, operacije povezivanja i omotači referenci
<utility>	razne uslužne komponente
<ctime>	C uslužna biblioteka za vreme i datum
<chrono> (od C++11)	C++ uslužna biblioteka za vreme
<cstddef>	standardni makroi i definisani tipovi
<initializer_list> (od C++11)	std::initializer_list šablon
<tuple> (od C++11)	std::tuple šablon
<any> (od C++17)	std::any klasa
<optional> (od C++17)	std::optional šablon
<variant> (od C++17)	std::variant šablon

Prilog 2. ASCII tabela (osnovna)

dec	okt	hex	ch	dec	oct	hex	ch	dec	oct	hex	ch	dec	oct	hex	ch
0	0	00	**NUL** (null)	32	40	20	space	64	100	40	**@**	96	140	60	**`**
1	1	01	**SOH** (start of header)	33	41	21	**!**	65	101	41	**A**	97	141	61	**a**
2	2	02	**STX** (start of text)	34	42	22	**"**	66	102	42	**B**	98	142	62	**b**
3	3	03	**ETX** (end of text)	35	43	23	**#**	67	103	43	**C**	99	143	63	**c**
4	4	04	**EOT** (end of transmission)	36	44	24	**$**	68	104	44	**D**	100	144	64	**d**
5	5	05	**ENQ** (enquiry)	37	45	25	**%**	69	105	45	**E**	101	145	65	**e**
6	6	06	**ACK** (acknowledge)	38	46	26	**&**	70	106	46	**F**	102	146	66	**f**
7	7	07	**BEL** (bell)	39	47	27	**'**	71	107	47	**G**	103	147	67	**g**
8	10	08	**BS** (backspace)	40	50	28	**(**	72	110	48	**H**	104	150	68	**h**
9	11	09	**HT** (horizontal tab)	41	51	29	**)**	73	111	49	**I**	105	151	69	**i**
10	12	0a	**LF** (line feed - new line)	42	52	2a	*****	74	112	4a	**J**	106	152	6a	**j**
11	13	0b	**VT** (vertical tab)	43	53	2b	**+**	75	113	4b	**K**	107	153	6b	**k**
12	14	0c	**FF** (form feed - new page)	44	54	2c	**,**	76	114	4c	**L**	108	154	6c	**l**
13	15	0d	**CR** (carriage return)	45	55	2d	**-**	77	115	4d	**M**	109	155	6d	**m**
14	16	0e	**SO** (shift out)	46	56	2e	**.**	78	116	4e	**N**	110	156	6e	**n**
15	17	0f	**SI** (shift in)	47	57	2f	**/**	79	117	4f	**O**	111	157	6f	**o**
16	20	10	**DLE** (data link escape)	48	60	30	**0**	80	120	50	**P**	112	160	70	**p**
17	21	11	**DC1** (device control 1)	49	61	31	**1**	81	121	51	**Q**	113	161	71	**q**
18	22	12	**DC2** (device control 2)	50	62	32	**2**	82	122	52	**R**	114	162	72	**r**
19	23	13	**DC3** (device control 3)	51	63	33	**3**	83	123	53	**S**	115	163	73	**s**
20	24	14	**DC4** (device control 4)	52	64	34	**4**	84	124	54	**T**	116	164	74	**t**
21	25	15	**NAK** (negative acknowledge)	53	65	35	**5**	85	125	55	**U**	117	165	75	**u**
22	26	16	**SYN** (synchronous idle)	54	66	36	**6**	86	126	56	**V**	118	166	76	**v**
23	27	17	**ETB** (end of transmission block)	55	67	37	**7**	87	127	57	**W**	119	167	77	**w**
24	30	18	**CAN** (cancel)	56	70	38	**8**	88	130	58	**X**	120	170	78	**x**
25	31	19	**EM** (end of medium)	57	71	39	**9**	89	131	59	**Y**	121	171	79	**y**
26	32	1a	**SUB** (substitute)	58	72	3a	**:**	90	132	5a	**Z**	122	172	7a	**z**
27	33	1b	**ESC** (escape)	59	73	3b	**;**	91	133	5b	**[**	123	173	7b	**{**
28	34	1c	**FS** (file separator)	60	74	3c	**<**	92	134	5c	****	124	174	7c	**|**

29	35	1d	**GS** (group separator)	61	75	3d	=	93	135	5d	]	125	175	7d	}
30	36	1e	**RS** (record separator)	62	76	3e	>	94	136	5e	^	126	176	7e	~
31	37	1f	**US** (unit separator)	63	77	3f	?	95	137	5f	_	127	177	7f	**DEL**

Prilog 3.

Odabrane metode nekih standardnih klasa koje se koriste u ovom materijalu

Neke metode klase std::stack

empty() tipa bool (void)
metoda proverava da li je stek prazan
ako jeste vraća true
ako nije vraća false

size() tipa int (void)
metoda vraća vrednost koja predstavlja koliko elemenata sadrži stek

push(el) tipa void (T)
metoda dodaje novi element tipa T na vrh steka (najmlađi element)

pop() tipa T (void)
metoda čita i uklanja element tipa T sa vrha steka (najmlađi element)

top() tipa T (void)
metoda samo čita element tipa T sa vrha steka, ali ga ne uklanja sa steka

Neke metode klase std::queue

empty() tipa bool (void)
metoda proverava da li je red prazan
ako jeste vraća true
ako nije vraća false

size() tipa int (void)
metoda vraća vrednost koja predstavlja koliko elemenata sadrži red

push(el) tipa void (T)
metoda dodaje novi element tipa T u red

pop() tipa T (void)
metoda čita i uklanja element tipa T iz reda

back() tipa T (void)
metoda čita, ali ne uklanja element sa kraja reda (najmlađi element)

front() tipa T (void)
metoda čita, ali ne uklanja element sa početka reda (najstariji element)

Neke metode klase std::list

empty() tipa bool (void)
metoda proverava da li je lista prazna
ako jeste vraća true
ako nije vraća false

size() tipa int (void)
metoda vraća vrednost koja predstavlja koliko elemenata sadrži lista

push_back(el) tipa void (T)
metoda dodaje novi element tipa T na kraj liste

push_front(el) tipa void (T)
metoda dodaje novi element tipa T na početak liste

pop_back() tipa T (void)
metoda čita i uklanja element tipa T sa kraja liste

pop_front() tipa T (void)
metoda čita i uklanja element tipa T sa početka liste

sort() tipa void (void)
metoda sortira listu za podrazumevanu komparaciju a<b (rastući redosled)

clear() tipa void (void)
metoda koja briše sve elemente liste

reverse() tipa void(void)
metoda obrće elemente liste tako da je prvi poslednji tako redom

Neke metode klase std::vector

empty() tipa bool (void)
metoda proverava da li je vektor prazan
ako jeste vraća true
ako nije vraća false

size() tipa int (void)
metoda vraća vrednost koja predstavlja koliko elemenata sadrži vektor

push_back(el) tipa void (T)
metoda dodaje novi element tipa T na kraj vektora

pop_back() tipa T (void)
metoda čita i uklanja element tipa T sa kraja vektora

front() tipa T& (void)
metoda pristupa prvom elementu (početak) vektora

back() tipa T& (void)
metoda pristupa poslednjem elementu (kraj) vektora

at(index) tipa T&(int)
metoda pristupa elementu koji se u vektoru nalazi na indeksu index

LITERATURA

[1] Sidhartha Rao: "*Naučite samostalno C++*", 1. izdanje, Kompjuter biblioteka, Begrad, 2017.

[2] B. Stroustrup : "*The C++ Programming Language*", Addison Wesley, 4th edition, 2013.

[3] Laslo Kraus : "*Programski jezik C++ sa rešenim zadacima*", 10. izdanje, Akademska misao, 2016.

[4] Laslo Kraus : "*Rešeni zadaci iz programskog jezika C++*", 5. izdanje, Akademska misao, 2016.

[5] Dragan Milićev : "Objektno orijentisano programiranje na jeziku C++", Mikro knjiga, 2005.

[6] S. Meyers : "*Effective Modern C++: 42 Specific Ways to Improve Your Useof C++11 and C++14.*", O'Reilly Media, Cambridge, MA, USA, 2015.

[7] N. M. Josuttis : "*The C++ Standard Library: A Tutorial and Reference*", Addison Wesley, Upper Saddle River, NJ, USA, 2nd edition, 2012.

[8] A. Williams : "*C++ Concurrency in Action*", Manning Publications, Shelter Island, NY, USA, 2012.

[9] Jacek Galowicz : "*C++17 STL Cookbook*", PACKT PUBLISHING, 2017.

[10] John Horton : "*Beginning C++ Game Programming*", PACKT PUBLISHING, 2016.

[11] Fedor G. Pikus : "*Hands-On Design Patterns with C++*", PACKT PUBLISHING, 2019.

[12] B. Stroustrup : "*Programming: Principles and Practice Using C++*", Addison Wesley, Upper Saddle River, NJ, USA, 2009.

[13] Wisnu Anggoro : "*Learning C++ Functional Programming*", PACKT PUBLISHING, 2017.

[14] Stefan Björnander : "*C++ Windows Programming*", PACKT PUBLISHING, 2016.

[15] Brian Overland : "*C++ opušteno*", 2. izdanje, CET Beograd: 2014.

[16] Bruce Eckel : "*Misliti na jeziku C++*", Mikro knjiga Beograd, 2014.

[17] Brian Overland : "*C++ jasnim jezikom*", Mikro knjiga, 2013.

[18] Paul Deitel, Harvey Deitel : "How to Program", Pearson, 10th. edition, 2016.

[19] http://www.cplusplus.com

[20] https://en.cppreference.com

[21] http://isocpp.org

[22] https://www.geeksforgeeks.org

INDEKS POJMOVA

U

V

W

Z