VELEUČILIŠTE U SPLITU
ODJEL RAČUNARSTVA

Nikica Plazibat

Objektno orijentirano programiranje

BILJEŠKE S PREDAVANJA – 2001/2002

Split, listopad 2002

Priređeno u suradnji s ECSAT d.o.o. Split

Sadržaj

Bilješke s predavanja iz predmeta *"Objektno orijentirano programiranje"* predstavljaju prvu verziju teksta. Predavanja su održana u drugom semestru školske godine 2001/2002.

Bilješke su nastale na osnovu predavanja koja je autor održao na Veleučilištu, tj. dijela "Power Point" prezentacija, te iz nekoliko različitih izvora. Primjeri iz prezentacija nisu isti, ali je zadržana koncepcija s predavanja.

Budući da je ovo prva verzija, logično je očekivati da postoje skrivene greške. Svaki komentar je dobrodošao.

Nikica Plazibat, dipl. ing.

Prije Uvoda

Prvo i drugo poglavlje opisuju povijest programskih jezika, tj. kojim su se putem razvijali načini programiranja, da bi evoluirali do objektno orijentiranih programskih jezika. Drugo poglavlje vodi ukratko kroz nekoliko tehnika te opisuje njihove karakteristike i mane, te na kraju nas navodi na objektno orijentirano programiranje, kao programiranje najbliže čovjekovom načinu razmišljanja.

Treće poglavlje ukratko opisuje karakteristike objektno orijentiranih jezika, opisuje što su klasa, objekt, nasljeđivanje i polimorfizam koji predstavljaju kamen temeljac svakog objektno orijentiranog programskog jezika
Također, u trećem poglavlju slikovito je opisan primjer objektno orijentiranog programa koji simulira sustav kontrole leta, s prikazom podataka o letećim objektima na radarskom ekranu.
Primjer zorno ilustrira prednosti objektno orijentiranog programiranja nad strukturiranim programiranjem. Primjer zračne kontrole leta provlačit će se kroz neka poglavlja preko primjera (nasljeđivanje i polimorfizam).

Četvrto i peto poglavlje počinje opisom C++ programskog jezika. Opisani elementi slični su elementima C programskog jezika.

Šesto poglavlje opisuje pokazivače i reference. Pokazivači su dio standarda u C-u, međutim novost u C++-u su reference, koje imaju sličnosti s pokazivačima, ali u odnosu na pokazivače imaju olakšano korištenje.
Alokacija i dealokacija memorije pomoću *new* i *delete* operatora opisana je u nastavku šestog poglavlja te je dana usporedba tih operatora sa funkcijama *malloc* i *free* iz C programskog jezika.

Poglavlje sedam kreće u opis osnovnih klasa, koje predstavljaju osnove objektnog programiranja. Tu su opisane klase, podatkovni i funkcijski članovi, prava pristupa članovima klase, te kako se iz klase koja predstavlja korisnički definirani tip kreira objekt.

Osmo poglavlje opisuje još jednu prednost C++ jezika, preopterećene operatore. Svim ugrađenim operatorima (zbrajanje, oduzimanje, pridruživanje, ... može se promijeniti funkcija koju vrše. Također za pojedine se klase mogu definirati funkcije koje pojedini operatori vrše, tako da se može definirati da operator zbrajanja vrši točno određene operacije nad objektom, npr. zbrajanje dvaju string objekata vrši konkatenaciju, dodavanje jednog stringa drugom kao krajnji rezultat.

Deveto poglavlje opisuje polja objekata, pokazivače na objekte, te pokazivače na polja objekata, te opisuje i njihove međusobne sličnosti.

Deseto i jedanaesto poglavlje obrađuju jedan od najvažnijih aspekata OOP-a: nasljeđivanje i polimorfizam. Opisuju kako se klase mogu nasljeđivati, hijerarhiju nasljeđivanja, nadjačavanje funkcija osnovne klase. Uvodi se pojam virtualnih funkcija. Također se opisuje virtualno nasljeđivanje, te na kraju čiste virtualne funkcije koje su produkt ADT-a (apstraktnih tipova podataka).

Dvanaesto poglavlje opisuje statičke podatkovne i funkcijske članove klase, te u čemu je njihova razlika u odnosu na obične članove klase, tj. da oni pripadaju području klase a ne području objekta kreiranog iz klase.

Trinaesto poglavlje opisuje privatni i zaštićeni način nasljeđivanja, prijateljske funkcije i klase koje imaju pristup svim podatkovnim i funkcijskim članovima klase bez obzira na pravo pristupa klase čiji su prijatelji.
Također se u ovom poglavlju opisuje kako se mogu preopteretiti redirekcijski globalni operatori, tako da se npr. *cout* objekt može naučiti kako da ispiše neki objekt kada se pridruži preko operatora redirekcije "<<".

Četrnaesto poglavlje bavi se ulazno izlaznim tokovima, šire opisuje objekte za unos podataka sa tipkovnice, te ispis na ekran. kod objekta za unos podataka *cin* opisuje dodatne funkcijske članove za manipulaciju međuspremnikom (bufferom), dok kod objekta za ispis podataka na ekran opisuje funkcijske članove za formatiranje ispisa. Također su opisane klase za rukovanje protokom podataka iz i u datoteke, te primjeri njihovog korištenja.

Petnaesto poglavlje opisuje predloške (engl. "templates") koji omogućuju pisanje općenitog koda, koji se može koristiti za različite tipove podataka. Ovdje su opisane dvije vrste predložaka: predlošci funkcija i predlošci klasa.

Šesnaesto, posljednje poglavlje bavi se iznimkama (engl. "exceptions"), koje služe pri hvatanju grešaka koje se javljaju u toku rada programa, npr. kada program ostane bez memorijskih resursa, ako se pokuša pristupiti nepostojećem članu polja, ili ako korisnik unese krivi tip podatka. Naravno programer treba "naučiti" program kako da reagira na takve predvidljive greške. U ovom poglavlju je opisano kako se "bacaju" iznimke kad program detektira pogrešku, te objašnjava kako se obrađuje takva pogreška u dijelu koda koji "hvata" iznimku.

1. Uvod

Predavanja iz predmeta O*bjektno orijentirano programiranje* vode u svijet objektno orijentiranog programiranja kroz osnove C++ jezika.

1.1. Povijest

Računala se od svojih početaka koriste za rješavanje aplikativnih (realnih) problema. Zadatak software-a je da riješi provaliju između koncepta aplikacije i koncepta rada računala. S jedne strane treba riješiti neki problem (srediti podatke o zaposlenicima neke firme, izračunati njihove plaće svakog mjeseca ili poslati raketu na mjesec), dok se s druge strane nalazi način rada koje računalo razumije (električni impulsi, strojni jezik). Tipičan programski zadatak je prevesti aplikacijski koncept u računalni koncept.

Prvi programi su bili ograničene složenosti. Rastom snage računala postavljali su se zahtjevi za sve kompleksnijim software-skim projektima. Povezivanje provalije između načina čovjekova načina razmišljanja i rada računala postajalo je sve teže.

Napredak software-ske tehnologije vođen je željom da se ova razlika što lakše premosti.

Želja čovjeka da premosti razliku između koncepta ljudskog razmišljanja i razumijevanja i koncepta na kojem je zasnovan rad računala predstavlja pokretačku silu razvoja software-skih alata.

2. Pregled tehnika programiranja

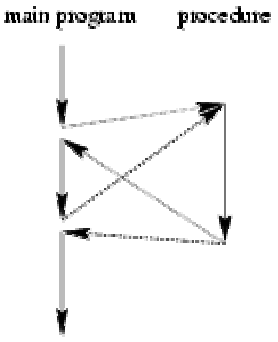
Dat ćemo ukratko pregled nekih od tehnika programiranja koje su se koristile do danas. To su nestrukturirano, proceduralno i modularno programiranje.
Preko tih tehnika programiranja evoluiralo se do objektnog programiranja.

2.1. Nestrukturalno programiranje

U pravilu početnici u programiranju počinju pisati programe koji se sastoje samo od glavne "main" funkcije. Rad sa podacima je u tzv. globalnom području, tj. svi podaci u programu dostupni su svim dijelovima programa. Ova tehnika programiranja postaje kontraproduktivna kada program naraste. Na primjer ako se jedan izraz treba koristiti na više od jednog mjesta u programu, jedini način je da se kopira na više mjesta u samom programu. Osim toga postaje nepregledan. To nas vodi na ideju da dijelove koda koji se ponavljaju izdvojimo u zasebne cjeline, damo im imena i način kako da ih se poziva te uzima rezultat operacija. Tako smo došli do procedura.

2.2. Proceduralno programiranje

S proceduralnim programiranjem može se grupirati ponavljajući slijed naredbi (statements) na jednom mjestu. Poziv procedure skače na mjesto u programu gdje je ona smještena. Nakon što je procedura izvršena, program se nastavlja izvršavati od mjesta odakle je procedura pozvana. Nakon što su uvedeni parametri koji se prosljeđuju proceduri, te mogućnost poziva procedura iz procedure (pod procedure, "sub procedures"), programi su se mogle strukturirano pisati, bili su razumljiviji, i što je veoma važno, "error-free". Na slici 2-1**.** vidimo tok programa sa pozivom procedura.



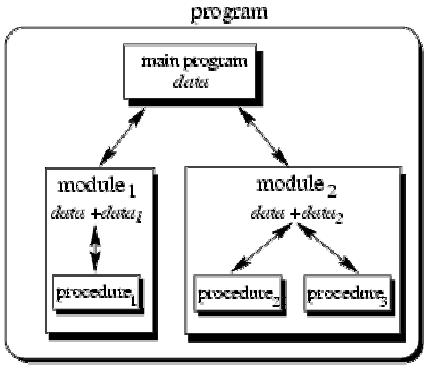
Slika 2-1

Sada se program može promatrati kao niz poziva procedura (funkcija). Glavni program je odgovoran da proslijedi podatke pojedinim pozivima funkcija, podaci se obrađuju unutar funkcije, te kada je program gotov, prikazuje se rezultirajući podatak.

Sada imamo program koji je podijeljen u male dijelove nazvane procedure ili funkcije. To nam omogućava upotrebu općih procedura ili grupa procedura isto tako i u drugim programima. One moraju biti odvojeno dostupne. Kao rezultat toga imamo modularno programiranje koje omogućava grupiranje procedura u module.

2.3. Modularno programiranje

Kod modularnog programiranja procedure zajedničkih *funkcionalnosti* grupirane su zajedno u odvojene module. Stoga se program više ne sastoji od jednog dijela, već je podijeljen u nekoliko manjih dijelova koji su povezani preko poziva procedura. Model modularnog programa prikazan je na slici 2-2.



Slika 2-2

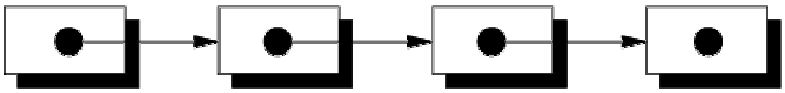
Svaki modul može imati svoje podatke. To omogućava svakom modulu da može rukovati unutrašnjim stanjima, koji su modificirani pozivima procedura modula. Ipak postoji samo jedno stanje po modulu i jedan modul može egzistirati najviše jednom u cijelom programu.

2.4. Primjer sa strukturama podataka

Primjer koristi strukturu za spremanje podataka. Postoji razne strukture podataka, npr. liste, stabla, polja, buffer-i (queues) itd. Svaka od tih struktura podataka može biti karakterizirana po dvije stvari*:* ***strukturi*** i ***pristupom*** *(engl. "access methods").*

2.4.1. Rukovanje običnom (single listom)

Obična lista ima jednostavnu strukturu, sastoji se od elemenata koji su međusobno povezani.



Slika 2-3 Obična lista

Jednostavna linkana lista ima *pristupne funkcije (access methods)* za dodavanje novih elemenata na kraj liste i brisanje elementa sa početka liste. Primjer takve liste je "queue" lista kao FIFO buffer (engl. *"first in first out"*).

Ovaj primjer samo ilustrira koncept i probleme koji postoje, on nije kompletan niti optimiziran.

Pretpostavimo da u jeziku C želimo napraviti program za listu. Ako listu gledamo u okviru modularnog programiranja kao ***opću strukturu (common data structure)***, implementirat ćemo je u posebnom modulu. To zahtijeva dvije datoteke: definiciju interface-a i implementacijsku datoteku. Ovdje ćemo dati samo jednostavni pseudo kod koji bi trebao biti lako razumljiv.

```
/*
 * Interface definition for a module which implements
 * a singly linked list for storing data of any type.
 */

MODULE Singly-Linked-List-1

BOOL list_initialize();
BOOL list_append(ANY data);
BOOL list_delete();
list_end();
ANY list_getFirst();
ANY list_getNext();
BOOL list_isEmpty();

END Singly-Linked-List-1
```

Definicija interface-a opisuje što je dostupno, a ne kako je implementirano. Kako je lista realizirana, skriveno je u implementacijskoj datoteci. Ovo je jedan od osnovnih principa programiranja: skrivanje informacija (information hiding).

Ovo omogućava promjenu implementacije, npr. može se upotrijebiti jači algoritam, ali koji troši više memorijskih resursa za spremanje elemenata, bez potrebe za spremanjem za promjenu ostalih modula unutar programa. Poziv funkcija modula izvan modula ostaje isti.

Ideja interface-a je sljedeća: Prije korištenja liste program mora inicijalizirati listu pozivom funkcije list_initialize(), da bi inicijalizirao lokalne varijable modula.

Dvije funkcije *list_append i list_delete* implementiraju funkcionalnosti dodavanja i brisanja elemenata liste.

Funkcija list_append() uzima jedan argument, podatak određenog tipa. To je bitno jer se lista može koristiti u nekoliko različitih "okolina", stoga tip podatkovnog elementa koji će se spremati u listi nije unaprijed poznat. Kao posljedicu moramo koristiti specijalni tip podatka "ANY", a koji nam omogućava da dodijelimo podatak bilo kojeg tipa.

Procedura *list_end()* treba biti pozvana kada program završi, omogućavajući modulu da počisti interno korištene varijable (dinamički korištenu memoriju).

Dvije sljedeće procedure: list_getFirst() i list_getNext(), nude jednostavan mehanizam za prolazak kroz listu:

```
ANY data;

data <- list_getFirst();

WHILE data IS VALID DO
doSomething(data);
data <- list_getNext();
END
```

Sada imamo list modul koji nam omogućava da koristimo listu sa bilo kojim tipom podatka.

Sada se postavlja pitanje: što ako je potrebno više od jedne liste?

2.4.2. Rukovanje s više lista u programu

Sada možemo redizajnirati naš list modul, da bude u mogućnosti prihvatiti više od jedne liste. Stoga je potrebno definirati novu interface datoteku, koja uključuje definiciju *list_handle* varijable. Ovaj rukovatelj (pokazivač), se koristi u svakoj korištenoj proceduri modula da jedinstveno identificira korištenu listu. Nova interface datoteka sada će izgledati ovako:

```
/*
     * A list module for more than one list.
     */

    MODULE Singly-Linked-List-2

    DECLARE TYPE list_handle_t;

    list_handle_t list_create();
                  list_destroy(list_handle_t this);
    BOOL          list_append(list_handle_t this, ANY data);
    ANY           list_getFirst(list_handle_t this);
    ANY           list_getNext(list_handle_t this);
```

```
    BOOL          list_isEmpty(list_handle_t this);

    END Singly-Linked-List-2;
```

DECLARE_TYPE – tip varijable je uveden za varijablu *list_handle_t*. Ovdje nije specificirano kako je ovaj *handle* stvarno predstavljen ili implementiran. Isto tako skrivamo implementacijske detalje ovog tipa u implementacijskoj datoteci.

Ponovno koristimo liste-create funkciju, ovaj put da bismo dobili *handle* na novu praznu listu. Svaki poziv ostalih procedura sada sadrži specijalni parametar *this* koji identificira koja je lista u pitanju. Sve procedure sada funkcioniraju na osnovu parametra *this*, umjesto jedne globalne liste u modulu.

Sada bismo mogli reći da smo kreirali "list objekt". Svaki ovakav objekt mogli bismo jedinstveno identificirati pomoću njegovog handle-a, te na njega možemo primijeniti samo one funkcije (metode) koje su definirane da koriste taj handle (rukovatelj).

2.4.3. Problemi modularnog programiranja

2.4.3.1. Eksplicitno kreiranje i uništavanje

U gornjem primjeru, svaki put kada se želi koristiti listu, mora se eksplicitno deklarirati *handle* i pozvati *list_create()* funkciju da bi se dobio ispravan rukovatelj. Nakon korištenja liste mora se eksplicitno pozvati *list_destroy() sa handle-om* liste koju želimo uništiti. Ako se lista koristi unutar procedure, to bi izgledalo ovako:

```
PROCEDURE foo() BEGIN
        list_handle_t myList;
        myList <- list_create();

        /* Do something with myList */
        ...

        list_destroy(myList);
    END
```

Uspoređujući listu sa ostalim tipovima podataka, npr. integer-om, on je deklariran unutar određenog područje (scope), npr. unutar procedure. Kad ga definiramo, možemo ga koristiti. Kada se izađe iz područja gdje je on definiran (program izađe iz funkcije), taj određeni integer je izgubljen. On je automatski kreiran i uništen. Neki prevodioci (compileri) čak i inicijaliziraju integer na specifičnu vrijednost npr. nulu.

U čemu je različit list "objekt"? Područje (života) liste je također definirano, stoga ona mora biti kreirana kad se uđe u "životno" područje i uništena kad se iz njega izađe. U vrijeme inicijalizacije lista treba biti inicijalizirana kao prazna. Dobra bi bila mogućnost definiranja liste na sličan način kao i varijable integer tipa. To bi izgledalo kao u primjeru koda:

```
PROCEDURE foo() BEGIN
        list_handle_t myList; /* List is created and initialized */

        /* Do something with the myList */
```

```
   ...
END /* myList is destroyed */
```

Prednost ovakvog načina rada bila bi u tome što bi prevoditelj preuzeo pozivanje inicijalizacijske i terminacijske procedure kada bude potrebno. To bi omogućilo da se lista ispravno izbriše.

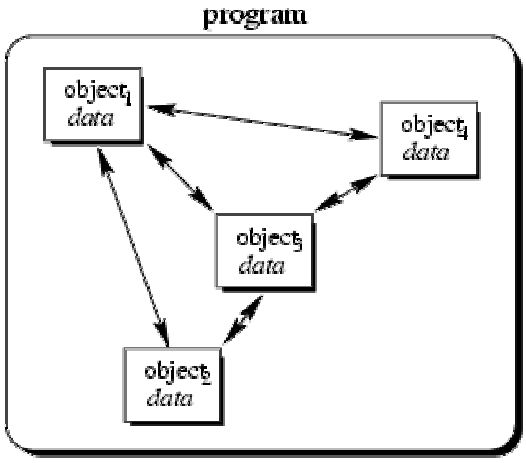
2.4.3.2. Objedinjavanje podaka i operacija

U modularno programiranju podaci su odvojeni *(engl. "decoupled")* od operacija nad njima. Programiranje je bazirano na operacijama umjesto nad podacima. Moduli grupiraju zajedničke operacije (npr. list_...() operacije) u jednu cjelinu. Te se operacije koriste prosljeđujući im eksplicitno podatke nad kojima će funkcije raditi.

U objektnoj orijentaciji, struktura je organizirana oko podataka. Izabiru se podaci koji najbolje zadovoljavaju potrebe. Zbog toga program postaje strukturiran prema podacima umjesto prema funkcijama, dakle obrnuto. Podaci specificiraju ispravne operacije. U objektnom programiranju moduli grupiraju podatke engl. *(engl. "data representation")* zajedno.

2.5. Objektno orijentirano programiranje

U objektno orijentiranom programiranju postoji svijet objekata koji međusobno komuniciraju. Svaki od tih objekata ima svoje stanje.



Slika 2-4

Pretpostavimo da imamo nekoliko lista u našem primjeru. Kod modularnog programiranja potrebno je eksplicitno kreirati (i uništiti rukovatelje (handle)) na listu. Nakon toga se mogu koristiti procedure modula za modificiranje svake liste, preko rukovatelja liste.

Nasuprot tome, u objektno orijentiranom programiranju, možemo imati objekata (lista) koliko želimo. Umjesto pozivanja procedure kojoj se mora proslijediti ispravni rukovatelj, šalje se poruka određenom list-objektu. Ugrubo, svaki objekt implementira svoj modul, te omogućava međudjelovanje više lista istovremeno.

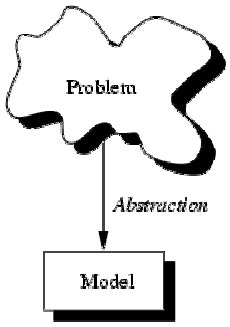
Svaki objekt je odgovoran za svoje ispravno inicijaliziranje i uništavanje. To znači da više nije potrebno eksplicitno pozivanje procedura za kreiranje i uništavanje liste.

Do sada opisano moglo bi se shvatiti kao samo jedna ljepša tehnika pisanja modularnih programa, kad bi to bilo sve što objektno orijentirano programiranje može ponuditi. Ali nije tako.

2.5.1. Apstraktni tipovi podataka

Neki autori opisuju objektno orijentirano programiranje kao programiranje apstraktnih tipova podataka i njihove međusobne veze.

Prva stvar s kojom smo suočeni kada pišemo program je realni problem koji pokušavamo riješiti. “Real-time” problemi su u početku uglavnom zamagljeni i prva stvar koju trebamo napraviti je pokušati razumjeti problem, te odvojiti bitne od nebitnih detalja. Pokušavamo definirati apstraktni model problema. Proces modeliranja apstraktnog modela zove se apstrakcija i prikazan je na slici.



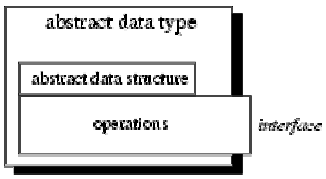
Slika 2-5

Kada se fokusira na problem potrebno je definirati karakteristike problema:

- koji su podaci uključeni
- koje se operacije koriste

Apstrakcija je strukturiranje zamagljenog problema *(engl “nebulous”)* u entitet s određenim podacima i operacijama. Podaci i operacije nad njima ne mogu se odvojiti jedno od drugog.

Strukturi podataka može se pristupiti preko definiranih operacija. Na slici vidimo ADT koji se sastoji od apstraktne strukture podataka i operacija nad njima.



Slika 2-6

Samo operacije su vidljive izvana i one definiraju “interface”.

Kada se na osnovu ADT-a kreira podatak, to se zove instanca abstraktnog tipa podatka.

Apstraktni tip podataka (ADT) ima sljedeće karakteristike

- eksportira tip
- eksportira skup operacija (Ovaj se dio zove interface)
- operacije interface-a su jedini pristupni mehanizam strukturi podataka

Princip skrivanja strukture podataka od vanjskog svijeta, i omogućen pristup istoj preko strogo definiranog interface-a zove se *enkapsulacija*

Abstraktni tipovi podataka (ADT) omogućavaju stvaranje instanci s dobro definiranim osobinama i ponašanjem *(engl. "properties and behaviour")*. U objektnoj orijentaciji iz ADT-a proizlaze *klase*. Stoga klasa definira osobine *(engl. "properties")* objekta, koji su instance u objektno orijentiranoj okolini.
ADT definira funkcionalnosti stavljajući glavni naglasak na podatke, njihovu strukturu i operacije nad njima te početne uvjete *(engl. "preconditions")*.

3. Objektno orijentirani programski jezici

3.1. Uvod u OOP – pojam klase

U kasnim ‘60-im, software-ski inženjeri zaključili su da je moguća kombinacija tehnologija izrade software-a i napretka u programskim jezicima. Rezultirajuća konstrukcija u programskim jezicima je bila klasa (engl. *class*).

Klasa implementira koncepte apstrakcije, modularizacije i skrivanja podataka. To radi grupirajući korisnički definirane tipove podataka, zajedno s procedurama i funkcijama koje se mogu primjeniti na njih.

Klasa omogućava nasljeđivanje (engl. *inheritance)*. Koncept nasljeđivanja povezan je s postupnim *uobličavanjem,* te također omogućava ponovno iskorištavanje već postojećeg koda (eng. *reusement*), klasa i struktura podataka u klasama.

Klasa je u programskim jezicima napravila revoluciju sličnu onoj koju je u elektronici napravio razvoj integriranih krugova. Kao što se elektronički sustavi mogu izrađivati korištenjem već postojećih IC-ova, te po potrebi njihovim modificiranjem, tako se i u izradi software-skih sustava može raditi korištenjem ili modifikacijom klasa.

Objektno orijentirano programiranje obećava da će dosta smanjiti spomenutu provaliju.

Prvi objektno orijentirani jezik bio je SIMULA, razvijen sredinom i kasnih ’60 ih godina u Norveškoj. SMALLTALK je jezik koji je popularizirao koncept objektno orijentiranog programiranja, razvijen početkom ’70-ih godina. Bio je veoma prihvaćen kod istraživanja umjetne inteligencije.

Sedamdesetih godina prošlog stoljeća ovi jezici su bili dostupni samo istraživačkim laboratorijama. Pravi bum objektno orijentiranih programskih jezika počinje početkom ‘80-ih, pojavom komercijalnih programa. SMALLTALK-80 predstavljen je 1983 godine. Ostali objektno orijentirani jezici su također postali komercijalno dostupni: C++, Objective C, Eiffel, Object Pascal i CommonLisp Object system. Ipak C++ proizlazi kao glavni i dominantni objektno orijentirani programski jezik.

Ključni koncepti koje treba podržavati svaki objektno orijentirani programski jezik su da treba:

- biti baziran na objektima i klasama
- podržavati nasljeđivanje
- podržavati polimorfizam

3.1.1. Objekti

Osnova modela objektno orijentiranog jezika je objekt. Objekt je u svakodnevnom svijetu definiran kao "nešto" s granicama. Objekt objedinjava podatke (engl. *properties*) i funkcijske članove ili metode *(engl. "methods")*. Podatkovni članovi opisuju stanje objekta. Metode (funkcijski članovi) omogućavaju objektu da provodi svoje ponašanje. Metode objekta imaju pristup podatkovnim članovima objekta, tj. međusobno dijele podatke objekta. Veoma je važno koje su metode i podatkovni članovi objekta vidljivi izvana. Objekt prima poruke *(engl. "messages")* iz vanjskog svijeta. Objekt izvršava jednu od svojih metoda kao rezultat prijema poruke iz vanjskog svijeta (recimo drugog objekta). Poruka kaže što objekt treba napraviti, dok metoda kaže kako to treba napraviti.

Objektno orijentirana aplikacija je skup objekata koji rade zajedno u smislu postizanja ukupnog cilja.

3.1.2. Klase:

Objektno orijentirani jezici opisuju objekte pomoću klasa. Klasa je opis sličnih objekata. Klasa definira objekte i metode individualnih objekata. Programski jezici trebaju omogućiti kreiranje objekata iz njihovih klasa. Klasa mora omogućiti rukovanje memorijom (engl. "memory-management"), alokaciju i dealokaciju memorijskog prostora objekta, oslobađajući programera tog zadatka.

Važno je da klasa postane strukturni entitet za sve programe. Opis klase djeluje kao modul, koji omogućava podjelu programa u manje dijelove.

3.1.3. Nasljeđivanje:

Nasljeđivanje, vrlo važna karakteristika objektno orijentiranog programiranja, predstavlja mogućnost izvođenja novih klasa iz postojećih, te njihova modificiranja, umjesto izrade iz početka, što omogućava povećanje produktivnosti, korištenjem već postojeće biblioteke programskog koda, (engl. "reusability", vrlo važna karakteristika OOP).
Klase i njihove pod klase (izvedene tj. naslijeđene klase), formiraju hijerarhiju klasa. Instanca izvedene klase je isto tako instance svih njenih super-klasa (klasa viših po hijerarhiji).

3.1.4. Polimorfizam:

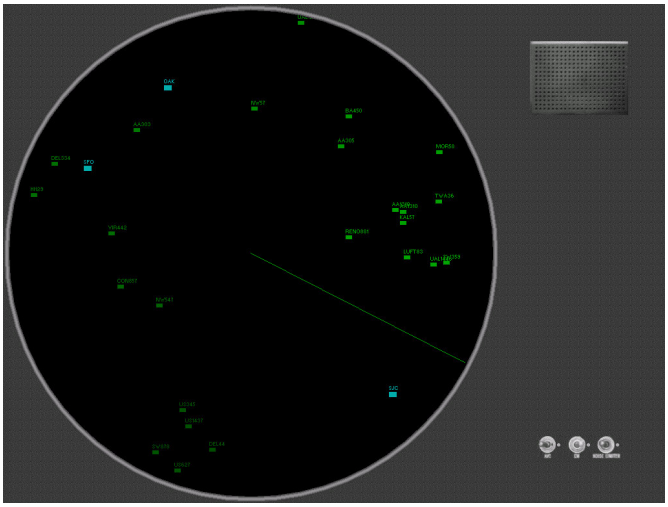
Polimorfizam u prijevodu znači "moći poprimiti više oblika", također je važna karakteristika objektno orijentiranog programiranja. Polimorfizam je baziran na

pretpostavci da tip ili klasa varijable ne mora biti ista tipu ili klasi objekta na koju se varijabla odnosi. Dok sam objekt ne može biti polimorfan, varijabla koja ga predstavlja i koja može pokazivati na sam objekt može imati više tipova. Varijabla deklarirana da bude određene klase može se odnositi na objekt upravo te klase, ili objekt bilo koje od njenih izvedenih podklasa *(engl. "sub classes")*.
Drugim riječima klasa varijable deklarira minimum zahtjeva koje objekt može imati. Polimorfizam omogućava specificiranje algoritama na višem, apstraktnijem nivou. Tipičan primjer je algoritam za printanje elemenata u listi. Kod polimorfizma algoritam može biti izveden na višem, apstraktnom nivou: "pretraži listu i pošalji poruku za printanje svakom elementu". Lista može sadržavati objekte različitih klasa, bitno je samo da svaka od tih klasa ima definiranu *print* metodu.

3.2. Primjer objektno orijentiranog programa:

Objektno orijentirano programiranje predstavlja nov pristup programiranju. U ovom primjeru predstavljen je informativni pogled na osnovni koncept OOP-a: objekti, klase, nasljeđivanje, enkapsulacija i polimorfizam.

Pretpostavimo da trebamo razviti novi software koji prikazuje jednostavnu verziju radarskog ekrana koji se koristi u kontroli zračnog prometa. Na ekranu se treba prikazivati zračni prostor oko i iznad aerodroma, te lokacija i kretanje aviona u tom prostoru. Slika 3-1 prikazuje takav ekran. Aerodrom je prikazan u sredini, te su prikazani svi avioni koji se nalaze u radarskom prostoru.



Slika 3-1 slika radarskog ekrana

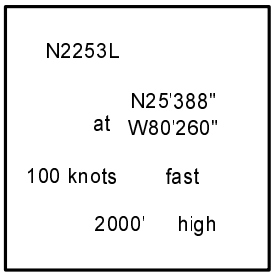
Konvencionalni pristup razvoju takvog software-a može se zasnivati prvo na funkcionalnostima koje pruža sistem: prikupljanje radarskih informacija i obnavljanje podataka na ekranu. Glavne funkcionalnosti (prikupi i obnovi), bile bi korištene da se izvede kompletna dekompozicija sistema. U objektno orijentiranom pristupu ide se na drugi način.

Ako se uzme objektno orijentirani pogled na problem, fokusiramo se na *entitete* koje postoje u aplikaciji. Umjesto da se gledaju globalne funkcionalnosti, može se problem sagledati u mnogo manjim dijelovima, entitetima ili objektima, koji se pojavljuju u aplikaciji. Svaki objekt objedinjuje male funkcionalnosti i malo podataka. Tablica 3-1 prikazuje ciljeve i rezultate dva različita pristupa.

	Pristup projektiranju	
	Konvencionalni	Objektno orijentirani
Cilj	identificirati glavne funkcije	identificirati glavne objekte
rezultati	sakupiti radarske informacije refresh ekrana	zrakoplovi radarski ekran radarski prijemnik

Tablica 3-1

U našem primjeru, avioni postaju objekti, oni sadrže sljedeće informacije: identifikacijski broj, lokaciju, visinu, smjer i brzinu. Slika 3-2 prikazuje objekt "avionXX" sa specifičnim vrijednostima za njihovu identifikaciju, koordinate, brzinu i visinu.

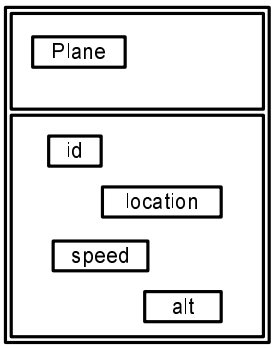


Slika 3-2

Osim toga avion ima funkcionalnosti tj. ponašanje *(engl. "behaviour")*: u stvarnom svijetu avioni lete, u našem primjeru oni mijenjaju koordinate, brzinu, visinu ili smjer.

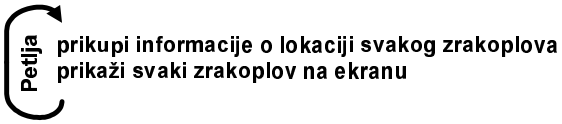
Kretanje aviona je detektirano radarskim sistemom koji generira podatke o svakom pojedinom avionu, te se podaci prikazuju na radarskom ekranu (monitoru).
Sljedeća funkcionalnost aviona je ta da se njegovi podaci prikazuju na radarskom ekranu.
Avion je dobar primjer objekta.

U centru naše aplikacije je objekt avion. Umjesto da opisujemo svaki avion posebno, više smisla ima opisati sve avione kao grupu ili klasu. Klase su najvažnije komponente objektno orijentiranog programa. Velik dio programiranja odlazi na definiranje klasa. Slika 3.2.3 prikazuje karakteristike koje su zajedničke za sve objekte-avione.



Slika 3-3

Dijagram specificira te klase. Klase se koriste za kreiranje objekata. Svi su objekti kreirani iz klasa, te se za objekt kaže da je on instanca *(engl "instance")* određene klase. Ovaj se sistem može sagledati kao skup objekata-aviona. Cjelokupni program funkcionira kao obična petlja, koja uzima podatke o položaju aviona sa radara, te zatim zove objekte-avione da se iscrtaju na ekranu. Slika 3.2.4 prikazuje pseudo-kod upravljačke petlje .



Slika 3-4: pseudo kod upravljačke petlje

U ovom kontekstu, možemo promatrati drugo važno svojstvo OOP-a: enkapsulaciju (engl. *encapsulation", objedinjavanje podataka i funkcija*). O objektu možemo razmišljati na način da ima određene granice, koje štite njegovu unutrašnjost tako da je čine nevidljivom. Klasa objekta definira koji aspekti objekta su vidljivi i dostupni. Npr. klasa *avion* može definirati da su samo dvije operacije dostupne za objekt avion: update informacija o lokaciji aviona i prikazivanje aviona na ekranu. Ova separacija onog što je unutra i nedostupno od onog što je dostupno izvana je jedna od osnovnih prednosti objektno orijentiranog programiranja.

Objektno orijentirano programiranje nudi i više od toga. Ovaj primjer je do sada rukovao samo jednim tipom zrakoplova. Svijet nije tako jednostavan. Postoje različite vrste letjelica: jedrilice, baloni, helikopteri, avioni sa jednim i više motora, mlazni avioni, itd. U našem primjeru, bilo bi važno raspoznati razliku između lakih i teških aviona, komercijalnih i malih privatnih aviona, ili aviona kojim se upravlja vizualno, te aviona koji posjeduju uređaje za navigaciju. Svaka od ovih letjelica spada u svoju kategoriju. Definirajući klasu, objektno orijentirano programiranje omogućava da se ta klasa može naslijediti iz druge klase. Od osnovne se klase nasljeđuju struktura podataka i funkcionalnost. Da bi se specificirao laki putnički zrakoplov, dovoljno je znati kako se on razlikuje od običnih zrakoplova. Npr. laki putnički zrakoplov koristi oznaku na repu kao identifikacijski broj, dok komercijalni zrakoplovi koriste broj leta kao oznaku.

Nova verzija software-a za praćenje zračnog prometa može specificirati tri klase: osnovni (basic) zrakoplov opisan gore i dvije pod klase koje nasljeđuju sve karakteristike klase osnovnog zrakoplova, te dodaju nove karakteristike svakoj, koje specificiraju kako se svaka od njih identificira. Nadalje, dvije nove klase mogu redefinirati ili promijeniti prikazivanje na radarskom ekranu, koje može uključivati prikaz identifikacijske oznake zrakoplova.
Sada se može promatrati iduća važna karakteristika objektno orijentiranih programskih jezika: *polimorfizam*. *Polimorfizam* znači "mnogo oblika". Označava da se iako su svi objekti prikazani na ekranu zrakoplovi, *može prikazati i više*: oni mogu imati više od jednog oblika ili tipa, koje pripadaju više od jedne klase. Neki od njih su mali zrakoplovi, dok su neki komercijalni letovi. Što se naše glavne petlje u programu tiče, to je petlja koja uzima informacije s radarskog uređaja i prikazuje ih na ekranu. Zrakoplovi su prikazani korištenjem njihove "Display()" funkcionalnosti. Što se tiče upravljačke petlje, dovoljno je da osnovna klasa definira funkcionalnosti prikaza (display-a) . To što mali zrakoplov i putnički zrakoplov imaju različite funkcionalnosti za prikaz identifikacije ne mijenja glavnu petlju.

Neke prednosti i fleksibilnost objektno orijentiranog programiranja proizlaze od razdvajanja imena (poziva) funkcije i njene stvarne implementacije (tijela). Poziv funkcionalnosti *(engl. "invocation of a funcionality")* naziva se "traženje servisa od objekta". Izvršavanje njegove implementacije opisuje se kao "primatelj zahtjeva izvršava odgovarajući servis". Odvajanje zahtjeva i servisa veoma povećava fleksibilnost u objektno orijentiranom programiranju. Svaki individualni objekt zrakoplov prima zahtjev i izvršava zahtijevani servis. Kao posljedicu, naš program "proračunava" ***šaljući*** zahtjeve svim objektima koji sudjeluju u obradi.

U globalu, možemo promatrati objektno orijentirani program kao skup objekata koji komuniciraju "***slanjem"i "primanjem"*** zahtjeva za servisima.

4. Elementi C++ programa

4.1. Jednostavni program

```
#include <iostream.h>

int main()
{
        cout << "Hello World!\n";
        return 0;
}
```

Ovo je jednostavni program u C++-u koji na ekranu ispisuje tekst “Hello world”.

Iako jednostavan u njemu može se vidjeti nekoliko zanimljivih detalja:

1. Simbol # predstavlja signal pretprocesoru. Pretprocesor se starta prilikom startanja prevoditelja. On čita izvorni kod tražeći linije koje počinju karakterom ’#’. “*include*” je pretprocesorska instrukcija koja u prijevodu znači da je ono što slijedi iza nje ime datoteke, koja se treba nalaziti u direktoriju u kojem se nalaze .h datoteke prevoditelja. Nakon što se datoteka učita, naredba se zamijeni sadržajem datoteke.
2. Program počinje radom ulaskom u funkciju “main()”. Svaki C++ program sadrži ovu funkciju. Funkcija ‘main()’ se automatski poziva prilikom starta programa. Ona ima svoje ulazne i izlazne parametre. U ovom slučaju nema ulaznih parametara (tipa je void), te vraća cjelobrojnu vrijednost. Povratna vrijednost se vraća onome tko je pokrenuo program, obično operativnom sistemu, signalizirajući kako je program završio s radom, npr. vrijednost 1 signalizira greška, za vrijednost 0 program je normalno završio. Ovaj primjer vraća vrijednost 0. Svi programi koji podržavaju ANSI standard deklariraju main funkciju da povratni parametar bude tipa *int*
   Tijelo funkcije main nalazi se unutar zagrada ‘{‘ i ‘}’.
3. Objekt *cout* se koristi za ispis poruke na ekran. Njemu srodan objekt *cin* ima obrnutu funkciju, čita vrijednosti sa “ekrana” odnosno ulaznog *toka.*
   Operator ‘<<’ zove se operator redirekcije. Sve što dolazi iza njega ispisuje se na ekranu. Ako se ispisuje niz (string) karaktera, moraju se nalaziti unutar znakova dvostrukih navodnika (double quote).
   U jeziku C također se koristi ‘<<’ ali u funkciji operacije pomaka (shift-anja tj. množenja broja s 2^n, gdje je n pomak bitova u lijevo) unutar byte-a u lijevo za određen broj karaktera.
   Posljednja dva karaktera ‘\n’ kazuju *cout* objektu da ubaci novi red prilikom ispisa na ekran. Ovaj “escape” karakter je dio seta kontrolnih karaktera koji su isti kao i u C jeziku.
4. Svaka naredbu unutar funkcije, kao i u C jeziku završava se znakom ‘;’

4.2. Kratki osvrt na cout

U prvom primjeru programa u C++-u prikazana je upotreba *cout* objekta sa operatorom redirekcije '<<' za ispis stringa karaktera. Objekt *cout* osim prikaza stringa može na izlazni tok (ekran) poslati i druge podatke: varijable, rezultate matematičkih operacija itd... Kako to funkcionira, za sada nas ne treba brinuti. U sljedećem primjeru vidi se kakve podatke možemo proslijediti na izlazni tok.

```
1:    // Listing 2.2 using cout
2:
3:       #include <iostream.h>
4:       int main()
5:       {
6:          cout << "Hello there.\n";
7:          cout << "Here is 5: " << 5 << "\n";
8:          cout << "The manipulator endl writes a new line to the screen." <<  endl;
9:          cout << "Here is a very big number:\t" << 70000 << endl;
10:         cout << "Here is the sum of 8 and 5:\t" << 8+5 << endl;
11:         cout << "Here's a fraction:\t\t" << (float) 5/8 << endl;
12:         cout << "And a very very big number:\t" << (double) 7000 * 7000 << endl;
13:         return 0;
14: }
```

```
Hello there.
Here is 5: 5
The manipulator endl writes a new line to the screen.
Here is a very big number:      70000
Here is the sum of 8 and 5:     13
Here's a fraction:              0.625
And a very very big number:     4.9e+07
```

U primjeru vidimo nekoliko detalja:

- Operatori redirekcije se mogu ulančati jedan iza drugog.
- Unutar stringa moguće je korištenje "escape" sekvenci kao i u C jeziku: '\n' predstavlja novi red, te '\t' predstavlja tab karakter, koji služi za poravnavanje ispisa.
- Možemo ispisati brojeve ili rezultate matematičkih operacija (npr. zbrajanje)
- Formatiranje ispisa brojeva na primjer, realnih brojeva, te ispisa brojeva u "scientific" formatu, može se dobiti konvertiranjem *(engl. "cast")* broja u željeni tip.
- Umjesto direktnog upisa broja iza operatora redirekcije, može se koristiti varijabla određenog tipa, kojeg *cout* prepoznaje prema tipu, te vrši ispis.
- *endl* služi za insertiranje oznake za novi red u izlazni tok. Radi isto kao i '\n' karakter. On predstalja tzv. *modifikator.*

Objekt koji uzima podatke sa ulaznog toka je *cin.* Operator redirekcije '>>' sada ima obrnuti smjer, te svaki podatak koji dođe sa ulaznog toka preusmjerava programski određene varijable u memoriji, specificirane iza operatora redirekcije.

```
...

int     iBrojGodina, iTežina;
```

```
cout << "Upisi dva broja:";
cin >> a >> b;
cout << "Zbroj dva broja iznosi" << a+b << endl;

 ...
```

Napomena: Da bismo lakše razlikovali operatore za ispis i učitavanje podataka '<<' i '>>' možemo shvatiti kao da pokazuju smjer prijenosa podataka.

4.3. Komentari

4.3.1. Tipovi komentara

Prilikom pisanja programa očigledno je što koja linija koda radi. Ali nakon nekoliko mjeseci taj isti kod je vrlo teško razumjeti. Osim toga naš kod će možda neko drugi koristiti ili održavati. Zbog toga unutar programa stavljamo komentare. Oni se ignoriraju od strane prevoditelja, a mogu pomoći onome tko kod čita, opisati što određeni dio koda radi.

Postoje dvije vrste komentara u C++-u:

- linijski komentari *(engl. "double slash ")*
- standardni "C-style" komentari *(engl. "star-slash")*

Linijski komentari (engl. "C++ style comment") označavaju se s *"//"* i kazuju prevoditelju da je sve što slijedi do kraja linije komentar te da taj dio ignorira prilikom prevođenja.

"Star-slash" (engl."C style comment") komentar počinje s *"/*"* karakterima i kazuje prevoditelju da ignorira sve karaktere do trenutka dok ne naiđe na *"*/"* niz karaktera.

U principu C++ tip komentara "//" koristi se za pisanje komentara, dok se C tip komentara "/*..*/" koristi za blokiranje određenog dijela programa. C++ tip komentara može se nalaziti unutar C tipa komentara.

4.3.2. Korištenje komentara

Kao opće pravilo, komentari se koriste na početku datoteke izvornog koda i kazuju što program zapravo radi.

```
/************************************************************

Program:      Hello World

File:         Hello.cpp

Function:     Main ()
```

```
Description:  Program ispisuje tekst "Hello world" na konzolu

Author:       Mate Matic (mm)

Environment:   Visual C++ 6.0, konzolna aplikaciju, C jezik, racunalo PII 433
               128 MB RAM

Notes:        Uvodni testni program

Revisions:    1.00  10/1/02 (mm) Prva verzija
              1.01  10/2/02 (tm) Kozmeticke izmjene

************************************************************/
```

Svaka funkcija također treba imati komentare opisujući što radi, koje vrijednosti prima i koje vraća. Svaki izvod koji je imalo nejasan treba biti iskomentiran.

Ipak ne treba pretjerivati sa komentarima. Ono što je samo po sebi jasno ne treba komentirati. Komentar treba opisivati zašto se nešto događa u kodu, a ne kako se događa.

5. Elementi C++-a

5.1. Varijable i konstante

Varijabla je mjesto gdje program sprema podatke. Ime varijable je *labela* stvarne memorijske lokacije.

Prilikom definicije varijable prevoditelju treba reći kolika je veličina varijable: integer, char itd...

5.1.1. Veličina integer-a

Veličina varijable tipa integer može varirati od tipa računala i platforme. Negdje je 2 a negdje 4 byte-a.

Za pregled veličine pojedinih tipova podataka, koristi se ugrađena funkcija u svaki prevoditelj *sizeof ()*, koja daje veličinu objekta koji joj se proslijedi. Primjer:

```
int iSizeOfInt;
...
iSizeOfInt = sizeof( int );
...
```

U tablici 5.1.1-1 dani su tipovi varijabli u C++-u i njihove veličine.

Type	**Size**	**Values**
unsigned short int	2 bytes	0 to 65,535
short int	2 bytes	-32,768 to 32,767
unsigned long int	4 bytes	0 to 4,294,967,295
long int	4 BYTES	-2,147,483,648 to 2,147,483,647
int (16 bit)	2 bytes	-32,768 to 32,767
int (32 bit)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned int (16 bit)	2 bytes	0 to 65,535
unsigned int (32 bit)	2 bytes	0 to 4,294,967,295
char	1 byte	256 character values
float	4 bytes	1.2e-38 to 3.4e38
double	8 bytes	2.2e-308 to 1.8e308

Table 5.1.1-1

SIGNED I UNSIGNED VARIJABLE

Nekad su nam potrebni negativni brojevi, a nekad nisu. Sve cjelobrojne varijable dolaze u dva oblika: *signed i unsigned*. Prilikom deklaracije varijabli ispred tipa se stavlja jedna od ove dvije oznake. *signed* oznaka ispred cjelobrojne varijable znači da ona može poprimiti negativne vrijednosti, dok *unsigned* znači da cjelobrojna varijabla može poprimiti samo pozitivne vrijednosti. Ukoliko se prilikom deklaracije varijable izostavi ovaj prefiks, pretpostavlja se da je postavljeno *signed.*
Primjer raspona vrijednosti koje može poprimiti integer varijabla veličine 2 byte-a.

unsigned int (16 bit)	0-65535
signed int (16 bit) int (16 bit)	od -32768 do +32767

5.1.2. Definiranje varijable

Varijabla se definira navođenjem tipa iza kojeg slijedi razmak od jednog ili više karaktera, te navođenjem imena varijable. Ime varijable može biti bilo koja kombinacija slova, brojeva i karaktera '_', s tim da prvi znak mora biti slovo ili znak '_'. Ime varijable ne smije sadržavati razmak (space)

Iako ime varijable može biti bilo koje, s tim da nije jedno od predefiniranih riječi u C++ jeziku, preporučuje se da ima neki smisleni naziv, tj. opisuje što varijabla sadrži. tako je lakše shvatiti funkciju te varijable u programu. Npr: očito je koja je od ove dvije varijable razumljivija:

```
...
int J7845jkd;
int iKutnaBrzina;
...
```

Jezik C++ je "case sensitive", tj. velika i mala slova se smatraju različitim. Varijabla imena brzinaVjetra i BRZINAVJETRA su dvije različite varijable.

Možemo kreirati više varijabli u istoj liniji, odvajajući ih zarezom. Osim toga nekim varijablama prilikom kreiranja možemo dodijeliti vrijednost. Dodjela početne vrijednosti prilikom kreiranja varijable zove se *inicijalizacija.*

```
...
int iBrzinaVjetra,  iDuljinaElise = 15;
...
```

5.1.3. typedef

Umjesto da prilikom kreiranja varijable navodimo ugrađeni tip varijable, možemo definirati zamjensko ime tipa podatka tzv. "alias" pomoću ključne riječi *typedef*. U biti kreiramo sinonim za tip podatka. Primjer:

```
typedef unsigned short int USHORT;
typedef unsigned int SerialPortSpeed;
...
USHORT errorCode = 0;
SerialPortSpeed speed = 19200;
...
```

Program korištenjem typedef postaje pregledniji, programer više ne treba razmišljati o tipu varijable kao *unsigned short int* već kao tipa *SerialPostSpeed*. S ovim program postaje pregledan, a programiranje manje sklono greškama.

5.1.4. "Wrapping arround" integer

Prilikom korištenja varijabli treba paziti na maksimalnu vrijednost koju mogu sadržavati. Ako npr. unsigned integer varijabla (npr. 2byte-a) u programu dosegne svoju maksimalnu vrijednost koja iznosi 65535, povećanje postavlja varijablu na nulu. Isto tako ako se signed int varijabli vrijednosti +32767 poveća vrijednost za jedan, vrijednost koju će ona sadržavati biti će –32768.

5.1.5. Karakteri i brojevi

Karakter varijabla tipa char je veličine 1 byte i može sadržavati 256 kombinacija vrijednosti. ASCII karakter set i njegov ISO ekvivalent se koristi za interpretiranje slova, brojeva i specijalnih znakova.
Na primjer malo slovo 'a' po ASCII standardu ima vrijednost 97. Isto tako sva ostala mala i velika slova i specijalni karakteri imaju vrijednost između 1 i 128.
Treba paziti da vrijednost varijable char 5 ne predstavlja karakter '5'. Njegov kod po ASCII standardu je 53, kao što je slovo 'a' predstavljeno brojem 97.

5.1.6. Specijalni karakteri

C++ prepoznaje neke specijalne karaktere karaktere za formatiranje. Ti specijalni karakteri se u kod ubacuju iza '\' koji se zove "*escape*" ili "*backslash*" karakter. *Escape* mijenja značenje karaktera koji slijedi iza njega.

Primjer karaktera za novi red izgleda:

```
char newLineChar = '\n';
```

karakter	**značenje**
\n	novi red
\t	tab
\b	povrat za jedno mjesto lijevo
\"	znak dvostrukih navodnika
\'	znak jednostrukih navodnika
\?	oznaka upita
\\	"backslash"

5.1.7. Konstante

Konstante također sadrže podatke, ali koji se ne mijenjaju tijekom rada programa. Konstantu se mora inicijalizirati prilikom kreiranja, te se poslije ne može za nju postaviti nova vrijednost.
Postoje dvije vrste konstanti literarne i simboličke:

5.1.7.1. Literarne konstante

Literarne (doslovna) konstanta je vrijednost koja se direktno ukuca u program gdje je potrebna. Na primjer:

```
int iBrojKomada = 135;
```

Broj 135 predstavlja literarnu konstantu. Broju 135 ne možemo dodijeliti vrijednost, te je ne možemo promijeniti.

5.1.7.2. Simboličke konstante

Simboličke konstante su predstavljene imenom kao i varijable, s razlikom što nakon inicijalizacije njena vrijednost ne može biti promijenjena. Primjer:

```
iBrojKomada = iBrojKutija * iBrojKomadaPoKutiji;
```

gdje *iBrojKomadaPoKutiji* predstavlja konstantu.

Postoje dva načina deklaracije simboličkih konstanti: preko *#define* direktive pretprocesoru i preko ključne riječi *const*.

```
#define iBrojKomadaPoKutiji 15
```

ili

```
const int iBrojKomadaPoKutiji = 15;
```

Oba načina pružaju mogućnost postavljanja vrijednosti konstante na jednom mjestu u programu i laku izmjenu i rekompajliranje ako se parametar promijeni, ali ipak danas se preferira drugi način korištenja.
U prvom slučaju pretprocesor prije početka prevođenja programa, na svim mjestima gdje se pojavljuje konstanta umjesto konstante upisuje broj 15, dok se u drugom kreira konstantna varijabla koja se koristi u programu. Prednost drugog načina je što konstantna varijabla ima *definiran tip*.

5.1.8. Enumerirane konstante

Enumerirane konstante omogućuju kreiranje novih tipova podataka i nakon toga definiciju varijabli tih tipova. Varijable enumeriranih tipova su ograničene na skup određenih vrijednosti. Primjer korištenja enumeriranih tipova:

```
enum color { CRVENA, PLAVA, ZELENA, BIJELA, CRNA };
...

color bgColor = BIJELA;
...
bgColor = CRVENA;
...
```

Prema tome *color* je tip podatka točno određenih vrijednosti, dok je *bgColor* ime varijable koja može poprimiti jednu od vrijednosti boja. Ako vrijednosti konstanti unutar znakova navodnika nisu specificirane, vrijednost prve konstante je *nula.* Vrijednosti sljedećih konstanti povećavaju se za jedan.
Ukoliko je naznačena vrijednost konstante u enumeraciji, ona poprima tu vrijednost, a sve koje je slijede, ako im vrijednost nije navedena, poprimaju vrijednost uvećanu za jedan, u ovisnosti o udaljenosti od posljednje definirane vrijednosti. Na primjer:

```
enum color { CRVENA=100, PLAVA, ZELENA=500, BIJELA, CRNA=700};
```

Sada vrijednosti konstanti koje nisu specificirane iznose: PLAVA iznosi 101, BIJELA iznosi 501.

Treba napomenuti da enumerirana varijabla ima tip *unsigned int*, te da je enumerirana konstanta jednaka integer varijabli.

5.2. Izrazi i naredbe (Expressions and statements)

5.2.1. Statements

U C++-u naredbe (*"statements"*) upravljaju izvođenjem programa, izračunavaju izraze ("*expressions*"), pa čak mogu i ne raditi ništa ("null statement"). Naredbe u C++ moraju završiti karakterom ';' .

Jedna od najčešćih naredbi je naredba pridruživanja *(engl. "assignment statement")*. Primjer,

```
x = a + b;
```

kaže: dodijeli varijabli *x* vrijednost zbroja varijabli *a* i *b*. Operator pridruživanja "assignment operator" pridružuje ono što je na desnoj strani (a to je rezultat operacije zbrajanja) onome što se na nalazi na lijevoj strani.

5.2.2. Blokovi i složene naredbe

Kao što možemo koristiti jednostavnu naredbu, sastavljenu od jednog retka, isto tako možemo koristiti složenu naredbu, blok naredbu. Blok počinje s otvorenom zagradom '{' i završava s '}'. Blok ne završava karakterom ';'.

```
{
```

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

5.2.3. Izrazi

Sve iz čega kao rezultat proizlazi vrijednosti zove se izraz *(engl. "expression")*. Npr. 3+2 vraćaju 5 te je to izraz.

Primjeri izraza su:

15.12
PI
iBrojOkretajaUSekundi

Primjer složenog izraza:

```
x = a + b;
```

U ovom slučaju rezultat zbroja *a* i *b se* pridružuje vrijednosti *x*, te izraz također vraća vrijednost varijable *x.* Zbog ovog posljednjeg ispravna je sljedeća naredba:

```
y = x = a + b;
```

Ona pridružuje vrijednost zbroja *a* i *b* varijabli *x* te nakon toga, vrijednost varijable *x* varijabli *y*.

5.2.4. Operatori

Svi *operatori* su simboli koji uzrokuju da prevoditelj izvrši neku akciju. *Operatori* djeluju nad *operandima*, a u C++-u svi operandi su *izrazi.*
Postoje dvije kategorije operatora:

- operatori pridruživanja
- matematički operatori

5.2.4.1. operator pridruživanja ('=')

Operator pridruživanja uzrokuje da operand s lijeve strane pridruživanja promijeni vrijednost u vrijednost koja je s desne strane pridruživanja.

Operandi koji se nalaze s lijeve strane operatora pridruživanja imaju oznaku *lvalue,* dok se oni s desne strane nazivaju *rvalue*. Konstante su tipa rvalue. One ne mogu biti rvalue. Možemo pisati:

```
x = 3456;
```

ali ne možemo pisati

```
3456 = x;
```

Kao što se vidi ne mogu sve *rvalue* biti *lvalue*, dok vrijedi da sve *lvalue* mogu biti *rvalue*.

5.2.4.2. Matematički operatori

Postoji pet matematičkih operatora: zbrajanje (+), oduzimanje (-), množenje (*), dijeljenje (/), te modulo operator (%).

Prva četiri operatora su sama po sebi razumljiva. Modulo operator kao rezultat vraća ostatak dijeljenja nad cijelim brojem. Primjer:

```
int a;
...
a = 10 % 3;
...
```

varijabla *a* ima vrijednost 1.

5.2.4.3. Kombiniranje operatora pridruživanja i matematičkih operatora

Kada je potrebno nekoj varijabli dodati neku vrijednost i rezultat operacije spremiti u tu istu varijablu, prvo što nam pada na pamet je:

```
temp = temp + 2;
```

Jednostavniji način pisanja iste naredbe je:

```
temp += 2;
```

Obje rade isto.

Slično se može koristiti i za druge operacije (-=), (/=), (*=), (%=).

5.2.4.4. inkrement i dekrement

Jedna od najčešće korištenih operacija je dodavanje ili oduzimanje varijable za jedan. Za to se može koristiti inkrement odnosno dekrement operator. Evo primjera:

```
a = a + 1;
a += 1;
a++;
```

korištenje inkrement operatora

Sve tri linije koda rade isto, povećavaju vrijednost varijable za 1. Isto vrijedi i za (--) dekrement operator.

5.2.4.5. prefix i postfix

Operatori za inkrement i dekrement dolaze u dvije mogućnosti: prefix i postfix. Sintaksa je slijedeća:

prefix `iGodineOve++;`
postfix `++iGodineOve;`

U jednostavnim naredbama upotreba jednog od ova dva operatora ne pravi nikakvu razliku, ali u složenim izrazima kada se poveća (smanji) vrijednost varijable t ima utjecaja.

Prefix operator se izvodi (vrijednost varijable se promijeni) prije dodjeljivanja vrijednosti, dok se kod postfix operatora vrijednost varijable promijeni nakon dodjele vrijednosti. Primjer:

```
int a = ++x;
```

povećaj vrijednost *x* i dodjeli je varijabli *a*

```
int b = x++;
```

dodijeli vrijednost *x* varijabli te povećaj x za 1.

5.2.5. Prioritet matematičkih operacija

Kod složenih izraza izvođenje operacija ovisi o prioritetu operatora. Prvo se proračunavaju dijelovi izraza koji imaju veći prioritet. Npr. množenje ima veći prioritet od zbrajanja, dok zagrade imaju veći prioritet od množenja, itd. Na primjer:

```
x = 5 + 3 + 8 * 9 + 6 * 4;
```

će biti 104 dok će

```
x = 5 + (3 + 8) * (9 + 6 * 4);
```

kao rezultat imati 368.

5.2.6. Priroda istine

U C++ jeziku nula se smatra kao *false* ili neistina, dok se sve ostale vrijednosti smatraju kao istina, tj. *true*. Ako je izraz jednak nuli tada je *false*, dok se za sve ostale vrijednosti smatra da je *true.*

5.2.6.1. operatori usporedbe

Operatori usporedbe se koriste za određivanje da li su dva izraza jednaka, ili je jedan od njih veći ili manji od drugog. Svaka operacija usporedbe rezultira kao 1 (*true)* ili 0 (*false*). Primjer je operacija usporedbe jednakosti dvaju integer varijabli:

```
iBrzinaPrenosa == iPostavljenaBrzina;
```

Ako su ove dvije varijable jednake, rezultat operacije je 1, u suprotnom je 0. Isto vrijedi i za ostale operatore usporedbe. U tablici 5-1 navedeni su operatori usporedbe:

Ime	***Operator***	***Primjer***	***Rezultat***
JEDNAK	==	`100 == 50;`	`false`
		`50 == 50;`	`true`
nije jednak	!=	`100 != 50;`	`true`
		`50 != 50;`	`false`
veći od	>	`100 > 50;`	`true`
		`50 > 50;`	`false`
veći ili jednak	>=	`100 >= 50;`	`true`
		`50 >= 50;`	`true`
manji od	<	`100 < 50;`	`false`
		`50 < 50;`	`false`
manji ili jednak	<=	`100 <= 50;`	`false`
		`50 <= 50;`	`true`

Tablica 5-1

5.2.7. if naredba

If naredba omogućava testiranje uvjeta, npr. da li su dvije varijable jednake, te se grana na različite dijelove koda ovisno o rezultatu.

Najjednostavnija *if* naredba je:

```
if (izraz)
        naredba;
```

Izraz u zagradama može biti bilo koji izraz, ali je obično usporedba vrijednosti. Ako izraz kao rezultat ima vrijednost 0, smatra se da je *false* i naredba koja ide iza njega se ne izvršava. Ako je rezultat različit od 0, smatra se da je *true* te se ono što slijedi iza *if* naredbe izvršava. To je prikazano su sljedećem primjeru.

```
if ( nekiBroj > drugiBroj )
        nekiBroj = drugiBroj;
```

Da bismo umjesto naredbe od jedne linije izvršili više njih, ako je uvjet ispravan, iza *if* naredbe stavljamo niz naredbi u blok naredbi unutar '{' i '}' zagrada.

```
if (bigNumber > smallNumber)
{
     bigNumber = smallNumber;
     cout << "bigNumber: " << bigNumber << "\n";
     cout << "smallNumber: " << smallNumber << "\n";
```

```
}
```

5.2.8. else

U slučaju da moramo obaviti neke akcije ako ispitivani uvjet nije točan, koristimo else naredbu, sada naredba ispitivanja izgleda ovako:

```
if (expression)
     statement;
else
     statement;
```

Moguće je korištenje više *if –else* naredbi jedne unutar druge:

```
if (expression1)
{
    if (expression2)
        statement1;
    else
    {
        if (expression3)
            statement2;
        else
            statement3;
    }
}
else
    statement4;
```

5.2.9. Logički operatori

Ponekad je potrebno koristiti više od jednog relacijskog ispitivanja u isto vrijeme, tj. ispitivati više uvjeta. Za takvo ispitivanje koristimo logičke operatore AND, OR i NOT.

Operator	Simbol	Primjer
AND	&&	izraz1 && izraz2
OR	izraz1 | | izraz2	
NOT	!	!izraz

Primjer:

```
if ( (x == 5) && (y == 5) )
```

5.2.10. Kondicionalni (ternalni) operator

Kondicionalni operator je jedini ternarni operator u C++ jeziku, tj. jedini operator koji prima 3 izraza.

On prima tri izraza i vraća vrijednost:

(expression1) ? (expression2) : (expression3)

Gornja linija se čita: Ako izraz1 je *true* vrati vrijednost izraza 2, u protivnom vrati vrijednost izraza 3. Tipično ova povratna vrijednost će biti vezana na varijablu.

Primjer:

```
z = (x > y) ? x : y;
```

5.3. Funkcije

Funkcija je u osnovi pod-program koji djeluje nad podacima i vraća rezultat. Svaki C++ program ima bar jednu funkciju – funkciju *main()*. Kada se starta program, *main* funkcija se poziva automatski i po potrebi poziva druge funkcije.
Svaka funkcija ima ime, a kad program naiđe na poziv funkcije, izvođenje programa se grana na tijelo te funkcije. Kada funkcija završi, izvođenje programa se nastavlja iza mjesta odakle je bio poziv funkcije.

Prilikom pisanja programa koriste se dvije skupine funkcija, ugrađene i korisnički definirane. Ugrađene funkcije dolaze uz prevoditelja (razvojnu okolinu).

5.3.1. Deklaracija i definicija funkcije

Korištenje određene funkcije u programu zahtijeva da je *deklariramo* i definiramo. *Deklaracija* kazuje prevoditelju ime, povratni tip te tipove parametara funkcije. *Definicija* kazuje prevoditelju kako funkcija radi. Funkcija se ne može pozvati ako prethodno nije deklarirana. *Deklaracija* funkcije se često naziva prototip funkcije.

Postoje tri načina deklaracije funkcije:

- napisati prototip u posebnoj datoteci, i nakon toga pomoću #*include* direktive uključiti je u program.
- napisati prototip u datoteci gdje se funkcija koristi
- definirati funkciju na mjestu koje je iznad svih mjesta gdje se nalazi poziv funkcije

Funkcijski prototip je naredba koja završava znakom ';'. Sadrži povratni tip funkcije, ime funkcije i listu parametara. Parametri funkcije sa njihovim tipovima odvojeni su međusobno zarezom. Prototip funkcije i definicija funkcije se mora točno slagati oko povratnog tipa, imena i liste parametara inače prevoditelj javlja grešku prilikom prevođenja. Prototip funkcije ipak ne treba sadržavati imena parametara funkcije, već samo njihov tip. Ipak prilikom navođenja imena funkcija je razumljivija. Stoga oba prototipa funkcije su ispravna.

```
long Area ( int, int );
```

```
long Area ( int length, int width );
```

Sve funkcije imaju povratni tip. Ako povratni tip nije specificiran, podrazumijeva se da je povratni tip *int.* Funkcija može ne vraćati ništa. Povratni tip takve funkcije je tipa *void.*
Funkcije ne moraju imati ulazne parametre.

Definicija funkcije sastoji se od *header-a* (zaglavlja) funkcije i tijela funkcije.
Header funkcije izgleda isto kao i prototip, ali bez ';' karaktera na kraju.
Tijelo funkcije (engl "function body") je skup naredbi unutar vitičastih zagrada '{' i '}'. Sve naredbe unutar funkcije moraju biti terminirane ';' karakterom.
Ako funkcija vraća neku vrijednost, mora završiti naredbom *return*, iako se naredba *return* može nalaziti još bilo gdje unutar tijela funkcije.

```
return_type function_name ( [type parameterName]...)
{
    statements;
}
```

5.3.2. Izvršavanje funkcije

Pozivom funkcije, izvršavanje programa se nastavlja na prvoj naredbi funkcije iza otvorene zagrade '{'. Funkcija može pozivati druge funkcije, te može pozivati samu sebe (rekurzija).

5.3.2.1. lokalne varijable

Osim što se varijable mogu proslijediti funkciji, mogu se i deklarirati unutar tijela funkcije. Te se varijable zovu lokalne varijable, pošto egzistiraju samo unutar tijela funkcije. Kada funkcija završi, te varijable više nisu dostupne.

Parametri proslijeđeni funkciji smatraju se lokalnim varijablama i koriste se kao da su definirani unutar tijela funkcije.

5.3.2.2. globalne varijable

Varijable definirane izvan svih funkcija vide se u okviru cijelog programa, i nazivaju se globalne varijable, uključujući funkciju *main*.

Lokalna varijabla definirana unutar funkcije skriva globalnu varijablu istog imena. To znači: ako postoje dvije varijable istog imena, jedna lokalno definirana i druga globalno, promjena vrijednosti varijable odnosi se na onu lokalnu. Da bi se u ovom slučaju pristupilo globalnoj varijabli, mora se koristiti operator za razlučivanje "::".
Primjer:

```
int i = 3;

int funkcija (void)
{
        int i;
        i = 2;
```

```
        return i;
}
```

Funkcija vraća vrijednost 2, ali vrijednosti globalne varijable i ostaje 3. Da bi se pristupilo i promijenilo vrijednost globalne varijable mora unutar funkcije pisati:

```
...

::i = 2;

...
```

Lokalne varijable vidljive su samo unutar kompletnog bloka funkcije. Lokalna varijabla se može u funkciji deklarirati unutar jednog bloka, te je ona vidljiva samo unutar tog bloka. Na primjer:

```
void myFunc()
{
        int x = 8;

        cout << "\nIn myFunc, local x: " << x << endl;

        {
                cout << "\nIn block in myFunc, x is: " << x;
                int x = 9;
                cout << "\nVery local x: " << x;
        }

        cout << "\nOut of block, in myFunc, x: " << x << endl;
}
```

Varijabla *x* je deklarirana unutar funkcije i to je lokalna varijabla. Ali druga varijabla *x* je deklarirana unutar bloka unutar funkcije i njen životni prostor je samo unutar tog bloka. Dok je funkcija unutar tog bloka ta varijabla *x* skriva lokalnu varijablu funkcije *x.* Nakon što program izađe iz bloka varijabla unutar bloka prestaje postojati, te se promjena vrijednosti varijable *x* odnosi na lokalnu varijablu funkcije.

5.3.3. Korištenje funkcije kao parametra funkcije

Funkcija se može koristiti kao parametar druge funkcije. Pošto funkcija vraća vrijednosti, ta se vrijednosti prosljeđuje drugoj funkciji kao argument. Primjer:

```
Answer = (double(triple(square(cube(myValue)))));
```

5.3.4. Parametri funkcije

Već je rečeno da se parametri funkcije unutar funkcije vide kao lokalne varijable. Prilikom ulaska u funkciju, stvara se kopija parametara proslijeđenih funkciji. To znači da promjenom vrijednosti tih varijabli unutar funkcije, završetkom funkcije vrijednosti tih parametara u dijelu programa koji je pozvao funkciju ostaju nepromijenjene.
Ovaj način prenošenja parametara u funkciju naziva se prenošenje po vrijednosti *(engl. "passing by value")*.

5.3.5. Podrazumijevane vrijednosti funkcija

Za svaki parametar prilikom poziva funkcije mora se proslijediti vrijednost. Jezik C++ uvodi u odnosu na C mogućnost da parametrima funkcije postavimo podrazumijevane vrijednosti prilikom deklaracije funkcije, dok definicija funkcije ostaje ista. Primjer:

```
long funkcija ( int a = 30 );
```

Ime parametra nije neophodno unutar deklaracije pa se može pisati i ovako:

```
long funkcija ( int = 30 );
```

Isto tako ime parametra unutar deklaracije i definicije može biti različito, pošto je podrazumijevana vrijednost vezana s rednim brojem parametra, a ne imenom

Ako se prilikom poziva funkcije navede parametar, njegova se vrijednost prosljeđuje unutar funkcije. Ako se parametar ne navede, uzima se podrazumijevana vrijednost u deklaraciji, kao vrijednost varijable unutar funkcije. Primjer:

```
int a = 30;
...
funkcija ( a );
...
funkcija();
...
```

Ako je podrazumijevana vrijednost 30 ova dva poziva će imati isti rezultat.

Bilo koji parametar funkcije može imati podrazumijevanu vrijednost s tim da parametri funkcije koji slijede iza njega moraju također imati definiranu podrazumijevanu vrijednost.

5.3.6. Preopterećenje funkcija

U C++ jeziku moguće je kreirati više funkcija istog imena. Funkcije se mogu razlikovati po listi parametara, različitim brojem i tipovima parametara. Primjer:

```
int myFunction (int, int);
int myFunction (long, long);
int myFunction (long);
```

Preopterećenje funkcije se naziva i funkcijski *polimorfizam.* U prijevodu znači "poprimiti mnogo oblika".

Napomena: Ne mogu se preopteretiti dvije funkcije istog imena, broja i tipova parametara, a različitim povratnim tipom.

5.3.7. Umetnute funkcije

Kada se funkcija definira prevoditelj kreira skup instrukcija u memoriji. Prilikom poziva funkcije, program skače na mjesto u memoriji gdje funkcija počinje. Ako se funkcija poziva 10 puta u programu izvršava se 10 skokova na isto mjesto u memoriji,

funkcija postoji na samo jednom mjestu. No ponekad ovi skokovi mogu usporiti program, ako se funkcija mora poziva veliki broj puta (u nekoj petlji). Zbog toga se koristi umetnuta funkcija. Kod nje se tijelo funkcija kopira na svako mjesto gdje se poziva. Ovo rezultira većim izvršnim kodom, ali se dobije na brzini izvršavanja.

Ključna riječ kojoj se prevoditelju sugerira da je funkcija umetnuta je *inline* ispred deklaracije funkcije . Primjer:

```
inline int Double(int target);

...

int Double(int target)
{
        return 2*target;
}
```

5.3.8. Rekurzija

Rekurzija se zove kada funkcija poziva samu sebe. Prilikom rekurzivnog poziva funkcije kreira se nova kopija funkcije (njenih lokalnih varijabli), te one ne mogu utjecati na varijable funkcije pozivatelja.

5.3.9. Funkcija ispod “haube”

Pisanje programa u nekom od viših programskih jezika u većini slučajeva ne zahtijeva duboki ulazak u organizaciju računala. Programeri, ovisno o vrsti problema koji rješavaju, uglavnom nemaju potrebe zalaziti u područja registara procesora, ukoliko to problem ne zahtijeva, te razumijevanje načina kako se na strojnom nivou obavljaju instrukcije.

Na niskom nivou prilikom startanja program se učitava u radnu memoriju računala (RAM). Prilikom učitavanja i pokretanja programa, u dio memorije se učitava izvršni kod programa, a drugi dio se koristi za pohranjivanje podataka.

Dio za pohranjivanje podataka podijeljen je na tri dijela: podatkovni segment, hrpu *(engl. "heap")* i stog *(engl. "stack")*.

Podatkovni segment je područje memorije u kojem su smješteni globalni i statički podaci.

Hrpa je dio memorije u kojem se pohranjuju dinamički alocirani podaci.

Stog je dio memorije na koji se pohranjuju lokalni podaci (lokalne varijable) dostupni samo funkciji koja se trenutno izvodi. Na stog se još spremaju i podaci potrebni prilikom poziva funkcija (argumenti funkcija), povratna adresa u programu kada funkcija završi sa izvođenjem, te povratna vrijednost funkcije.

Procesor (CPU) interpretira naredbe strojnog koda, te na osnovu njih izvodi određene operacije: dohvaća podatke iz memorije, obavlja nad njima operacije i vraća ih natrag u memoriju.

Procesor ima nekoliko registara, međutim mogu se izdvojiti dva bitna: *instruction pointer* i *stack pointer*. *Istruction pointer* pokazuje na adresu u memoriji gdje se nalazi sljedeća instrukcija. Instrukcija (sadržaj memorije) na koji pokazuje *instruction pointer* učita se u registar procesora i izvrši. Nakon toga *ip* se poveća tako da pokazuje na sljedeću instrukciju.

Ako procesor naiđe na instrukciju koja predstavlja skok u na neku novu adresu u programu, procesor puni *ip* sa tom novom adresom te učitava instrukciju sa nove adrese.

Prilikom poziva funkcije mora se zapamtiti adresa odakle je funkcija pozvana da bi završetkom izvođenja funkcije program mogao nastaviti od mjesta odakle je funkcija pozvana.

Sadržaj *ip* registra prvo se poveća tako da pokazuje na sljedeću instrukciju. Zatim se ta adresa u *ip* registru postavi na stog. Stog je LIFO tipa (engl. "last in first out") što znači posljednji podatak koji se postavi na stog prvi se sa njega skida.

Nakon što se podatak o povratnoj adresi postavi na stog, puni se instrukcijski registar procesora adresom funkcije.

Na stogu se rezervira mjesto u koje će funkcija prilikom završetka rada upisati povratnu vrijednost. Ako je povratna vrijednost tip *int,* na stogu se rezerviraju 4 byte-a.

Argumenti koji se prenose funkcije također se pohranjuju na stog iza mjesta rezerviranog za povratnu vrijednost funkcije.
Veličina rezervirana za argumente ovisi o broju parametara koji se prenose funkciji.

Npr. ako funkcija ima jedan *int* tip podatka i jedan *char*,* ukupno rezervirano područje je 8 byte-ova, jer *int* tip podatka zauzima 4, a pokazivač na niz karaktera također zauzima 4 byte-a.

Argumenti funkcije kopirani na stogu predstavljaju lokalne varijable. Ako se prenosi podatak preko pokazivača, kopira se adresa varijable, te joj se može pristupiti preko adrese, koja je lokalna varijabla.

Prilikom ulaska programa u tijelo funkcije sve se lokalne varijable kreiraju postavljajući ih na stog redoslijedom kreiranja.

Pokazivač stoga se automatski inkrementira kako se podaci na njega postavljaju.

Završetkom rada funkcije vrijednost koju funkcija treba vratiti sprema se u područje stoga koje je rezervirano za povratnu vrijednost.

Povratna vrijednost funkcije stavlja se u registar procesora.

Stack pointer se postavlja na poziciju povratne adrese funkcije. *Instruction pointer* puni se tom adresom, te je sljedeća instrukcija koja se izvodi instrukcija koja slijedi iza funkcije. Npr. ako je sljedeća instrukcija bila izvršavanje operacije pridruživanja lijevo od funkcije, vrijednost podatka povratne vrijednosti funkcije iz registra kopira se u lokacija varijable smještene s lijeve strane operatora jednakosti.

5.4. Još o toku programa - p–tlje

5.4.1. goto

Naredba *goto* omogućava skok bilo gdje u programu. Sastoji se od *goto* naredbe i labele. Labela je ime koje završava sa karakterom ':' *(engl."colon").* Primjer:

```
...
loop: counter ++;      // top of the loop

cout << "counter: " << counter << "\n";

if (counter < 5)       // test the value
goto loop;     // jump to the top
...
```

5.4.2. while

Petlja sa while naredbom vrši ponavljanje skupa naredbi sve dok je uvjet koji slijedi iza naredbe ispunjen, tj. da je rezultat ispitivanja istinit. Primjer:

```
...
int counter = 0;

while( counter < 5 ) // test condition still true
{
        counter++; // body of the loop
        cout << "counter: " << counter << "\n";
}
...
```

Kada *counter* postane jednak 5 program izlazi iz petlje. Ako prilikom prvog ispitivanja uvjet iza naredbe *while* nije ispunjen, program uopće ne uđe u petlju.

5.4.3. do while

Naredba je slična *while* naredbi, ali u ovom slučaju program uđe bar jednom u blok naredbi unutar petlje. Izvođenje bloka naredbi se ponavlja sve dok je uvjet iza *while* ispunjen. Primjer:

```
...
int counter = 10;
```

```
do
{
        cout << "Hello\n";
        counter--;
} while (counter >0 );
...
```

Napomena: Uvjet unutar naredbe *while* i *do while* je ispunjen sve dok je izraz unutar zagrada veći od nule. To znači da rezultat može biti *true,* ili broj 1, 45, .. . Ako je rezultat 0 ili *false,* uvjet više nije ispunjen.

5.4.4. Naredba continue

Ponekad je potrebno unutar bloka naredbi petlje prekinuti izvođenje naredbi do kraja bloka, te prijeći ponovno na početak petlje. To se izvodi naredbom *continue.* Primjer:

```
while (small < large && large > 0 && small < 65535)
{
        small++;

          if (small % skip == 0)  // skip the decrement?
          {
                cout << "skipping on " << small << endl;
                continue;
          }

          if (large == target)    // exact match for the target?
          {
                cout << "Target reached!";
                break;
          }

        large-=2;
}                   // end of while loop
```

5.4.5. Naredba break

Naredba koja unutar petlje izaziva bezuvjetni izlaz iz petlje je *break* naredba. Primjer:

```
while (condition)
{
    if (condition2)
        break;
    // statements;
}
```

5.4.6. For petlja

Za izvođenje određenog bloka naredbi točno određeni broj puta koristi se *for* naredba. Njena sintaksa je sljedeća:

for (inicijalizacija; testiranje uvjeta; izvršavanje akcije)
        naredba;

Primjer:

```
for (counter = 0; counter < 5; counter++)
    cout << "Looping! ";
```

5.4.7. Switch case naredba

Naredba *switch case* omogućava grananje programa ovisno o početnom uvjetu. Iako se isti rezultat može postići korištenjem *if-else* naredbi, korištenje *switch case* naredbi je dosta preglednije. Primjer:

```
switch (expression)
{
case valueOne:
        statement;
        break;

case valueTwo:
        statement;
        break;
....

case valueN:
        statement;
        break;

default:
        statement;
}
```

Switch naredba ispituje izraz unutar zagrada, te ovisno o rezultatu skoči na dio koda koji obrađuje taj rezultat (*case).* Program se izvodi do naredbe *break,* te nailaskom na nju program iskače iz *switch-case* bloka i nastavlja sa radom.

Ukoliko ni jedna *case* vrijednost nije jednaka rezultatu *switch* izraza program izvršava skup naredbi iza *"default:"* naredbe. Ukoliko nema *default* naredbe, ni jedan *case* blok naredbi se neće izvršiti.

6. Pokazivači i reference

6.1. pokazivači

Pokazivači su objekti (varijable) koji pokazuju na drugi objekt. Sam pokazivač sadrži memorijsku adresu na kojeg pokazuje.

Pokazivač se može deklarirati tako da pokazuje na bilo koji tip podatka. U deklaraciji pokazivača se navodi tip podatka, ime pokazivača, ispred kojeg se stavlja '*' znak koji označava da je u pitanju pokazivačka varijabla.

Na primjer, neka je kreirana i inicijalizirana cjelobrojna varijabla *iBrzinaAutomobila*, te neka je deklariran i inicijaliziran pokazivač na tu varijablu imena *pBrzina*. Uzimanje adrese varijable vrši se korištenjem '&' tj. ampersand operatora.

```
int iBrzinaAutomobila = 15;
int *pBrzina = &iBrzinaAutomobila;
```

Ako se pokazivač ne inicijalizira prilikom deklaracije na neku konkretnu vrijednost, preporuka je da se postavi na vrijednost NULL. Ukoliko se ne postavi na neku konkretnu vrijednost, pokazivač ima slučajnu vrijednost, koja se u vrijeme dodjele memorijskog prostora za pokazivačku varijablu zatekla u njenim lokacijama. Korištenje pokazivača s takvim vrijednostima je veoma opasno jer može dovesti do nepravilnog rada programa ili do njegovog rušenja.

Pristupanje varijabli preko pokazivača zove se indirekcija – odnosno pristup varijabli na adresi na koju pokazuje pokazivač. Pristup vrijednosti varijable preko pokazivača vrši se navođenjem '*' znaka ispred imena pokazivača.

```
*pBrzina = 3;
cout << *pBrzina;
```

Ispis:

```
3
```

6.1.1. Tipovi pokazivača

Postoje tri različite vrste pokazivača: *pokazivači na konstantu, konstantni pokazivači na varijablu i konstantni pokazivači na konstantu*. Razlika se vidi prilikom deklaracije:

`const int *pOne`	pokazivač na konstantu
`int * const pTwo`	konstantni pokazivač na varijablu

`const int *const pThree`	konstantni pokazivač na konstantu

6.1.2. Viseći pokazivači

Jedan od velikih problema prilikom pisanja koda su viseći pokazivači (eng. “stray or dangling pointers”).
Viseći pokazivači nastaju kad se pokuša pristupiti memoriji koja je oslobođena operatorom *delete*. Pokazivač i dalje pokazuje na nepostojeću memoriju. Pokušaj pristupa dealociranoj memoriji obično završava rušenjem programa. Ukoliko ne završi rušenjem programa, uzrokuje greške koje se jako teško otkrivaju.

```
USHORT *pInt = new USHORT;
*pInt = 10;
...
delete pInt;
...
pInt = 0;
...
*pInt = 20;
...
```

6.2. reference

Reference su slične pokazivačima. One predstavljaju drugo ime za objekt ili varijablu određenog tipa.

Kada pokazivač nije inicijaliziran preporuka je postaviti mu vrijednost NULL. Za razliku od pokazivača referenca ne može imati vrijednost NULL, jer se mora inicijalizirati na konkretnu vrijednost prilikom deklaracije, te se u toku rada programa ne može mijenjati njena vrijednost.

Deklaracija reference vrši se tako da se između tipa reference i imena reference nalazi '&' (ampersand operator) koji ne predstavlja adresu, već kaže da je riječ o referenci.

Referenca se mora inicijalizirati prilikom deklaracije.

Također, program sa referencom inicijaliziranom na 0 javlja grešku prilikom prevođenja.

Primjer:

```
#include <iostream.h>

int main()
{
 int iBrzinaAutomobila = 15;
 int& rBrzinaAutomobila = iBrzinaAutomobila;

cout << "Brzina automobila preko iznosi: " << iBrzinaAutomobila << endl;
cout << "Brzina automobila preko iznosi: " << rBrzinaAutomobila << endl;

   return 0;
```

```
}
```

Ispis za obe varijable je isti:

```
Brzina automobila preko iznosi: 15
Brzina automobila preko iznosi: 15
```

Promjenom reference mijenja se vrijednost izvorne varijable:

```
rBrzinaAutomobila = 3;

cout << "Sada je brzina automobila: " << iBrzinaAutomobila << endl;
```

Ispis za prethodnu liniju koda izgleda ovako:

```
Sada je brzina automobila: 3
```

Na početku našeg programa se definira i inicijalizira cjelobrojna varijabla *iBrzinaAutomobila*, te se inicijalizira na vrijednost 15. Odmah iza nje deklarira se i inicijalizira referenca na tu istu varijablu imena rBrzinaAutomobila. S tim je kreirano novo ime za varijablu iBrzinaAutomobila.
Interno referenca pokazuje na varijablu *iBrzinaAutomobila*. Pristupom preko reference direktno se pristupa varijabli *iBrzinaAutomobila.* Za razliku od pokazivača za pristup podatku koji je vezan sa referencom nije potrebno koristiti operator za dereferenciranje '*'.

6.2.1. Proslijeđivanje reference funkciji

Prilikom prosljeđivanja reference funkciji ne stvara se lokalna kopija varijable koja je vezana preko reference, već se unutar funkcije barata direktno s originalnom varijablom.

Primjer:

```
void Swap(int c, int d)
{
   int temp;
   temp = c;
   c = d;
   d = temp;
}

int a= 4, b=10;
....
Swap ( a, b )

printf ( "a = %d, b=%d", a, b );
```

Ispis programa će biti:

```
a=4, b=10
```

Funkcija vrši zamjenu vrijednosti dviju varijabli proslijeđenih funkciji. Pošto se funkciji varijable prosljeđuju po vrijednosti, te se kreiraju lokalne kopije, vrši se zamjena vrijednosti lokalnih kopija. Na kraju, funkcije lokalne varijable se uništavaju.

Po izlasku programa iz funkcije, vrijednosti originalnih varijabli ostaju nepromijenjene.

Naprotiv, korištenjem referenci ne kreiraju se lokalne kopije već se zamjena vrijednosti vrši direktno na proslijeđenim varijablama.

```
void Swap(int& c, int& d)
{
   int temp;
   temp = c;
   c = d;
   d = temp;
}

int a= 4, b=10;
....
Swap ( a, b );

printf ( "a = %d, b=%d", a, b );
```

Ispis programa će biti:

```
a=10, b=4
```

U ovom slučaju vrši se zamjena vrijednosti originalnih varijabli.

Za razliku od pokazivača, prilikom korištenja referenci varijable ne treba dereferencirati odnosno ne treba izvlačiti podatak sa adrese na koju pokazivač pokazuje, već se referenca koristi kao i obična varijabla. Ona je kao što je već rečeno "alias" odnosno zamjensko ime za varijablu.

6.3. Rukovanje memorijom operatorima new i delete

Pronovimo iz C jezika:

U C programskom jeziku rezerviranje i oslobađanje memorije vršilo se funkcijama *malloc* i *free*.
Funkciji *malloc* proslijeđivala se veličina memorije koja se željela rezervirati. Kao povratnu vrijednost funkcija je vraćala pokazivač tipa *void* na rezerviranu memoriju. Daljnje rukovanje tom memorijom vršilo se preko pokazivača. Pokazivač se mogao konvertirati u neki drugi tip, na primjer u pokazivač na integer, ili pokazivač na korisnički definiranu strukturu.
Kad memorija koja se rezervirala više ne bi bila potrebna, morala se osloboditi, odnosno vratiti operativnom sistemu. Tu je operaciju vršila funkcija *free* kojoj se kao parametar proslijeđivao pokazivač na alociranu memoriju.
Ukoliko se memorija ne bi oslobodila, ili bi se u toku rada programa izgubio pokazivač na nju, ta bi alocirana memorija bila izgubljena za program. Ta situacija se zove curenje memorije *(engl. "memory leak")*.

Primjer za alociranje i oslobađanje memorije:

```
void *pMem = NULL;
```

```
int *pInt = NULL;
pMem = malloc (sizeof(int)*10);

if (pMem == NULL )
exit(1);

pInt = (int *) pMem;

pInt[0] = 0;
pInt[1] = 1;
...
printf ( "\n %d %d", pInt[0], pInt[1] );

free (pMem);
```

6.3.1. Operator new

Iako se u C++ može koristiti sve što je naslijeđeno iz C jezika pa tako i prethodne dvije funkcije, C++ jezik ima svoj mehanizam za alokaciju memorije. To je operator *new*

Operator *new* kao ulazni parametar također prima količinu memorije koju treba rezervirati, a kao povratni podatak vraća pokazivač na prvi *byte* alocirane memorije. Također, ukoliko zbog nekog razloga nije u mogućnosti rezervirati memoriju (na primjer zbog toga što su memorijski resursi računala potrošeni), kao povratnu vrijednost vraća NULL.

Primjer:

```
int *pIntBroj = NULL;
int *pIntPolje = NULL;
char *pCharPolje = NULL;

pIntBroj = new int;
pIntPolje = new int[10];
pCharPolje = new char[10];
```

Spremanje vrijednosti u lokaciju na koju pokazuje pokazivač *pIntPolje* vrši se preko samog pokazivača.

```
*pIntBroj = 12;
pIntPolje[2] = 3;
```

itd.

6.3.2. operator delete

Memorija se dealocira, odnosno oslobađa upotrebnom operatora *delete*, kojem se prosljeđuje pokazivač na alociranu memoriju.

Primjer:

```
delete pIntBroj;
```

Međutim postoji razlika između oslobađanja jednog podatka i polja podataka. Prilikom oslobađanja polja podataka između operatora *delete* i pokazivača na memoriju potrebno je navesti da se radi o polju, navodeći "[]".

Primjer:

```
delete [] pIntPolje;
delete [] pCharPolje;
```

Ukoliko se za oslobađanje polja navede samo:

```
delete pIntPolje;
```

oslobodit će se samo prvi element polja dok će ostali elementi ostati u memoriji, te se na taj način stvara curenje memorije.

To može predstavljati probleme prilikom intenzivnog rezerviranja i oslobađanja memorije, jer će računalo vrlo brzo ostati bez memorijskih resursa.

7. Osnovne klase

7.1. Klasa

Klasa predstavlja skup kombinacije varijabli često različitih tipova, sa njima pridruženim funkcijama. Ona predstavlja tip podatka.

Primjer klase je automobil.
S jedne strane automobil se može promatrati kao skup dijelova: karoserija, motor, kotači, mjenjač, itd. S druge strane se gleda što automobil može učiniti: ubrzavati, usporavati, kočiti, okretati upravljač lijevo i desno, itd.

Klase omogućavaju *enkapsulaciju*, tj. povezivanje podataka i funkcija u jednu cjelinu koja se zove objekt.

Enkapsulacija omogućava da sve što znamo o npr. automobilu, postavimo na jedno mjesto. To olakšava manipulaciju podacima. Korisnik klase, tj. jedan dio programa može koristiti objekt bez brige o tome kako on radi. Jednostavni primjer je kada vozač pritisne papučicu gasa automobil počinje ubrzavati, iako vozač ne treba ništa znati o tome kako to funkcionira motor ispod haube automobila.

7.2. Članovi klase

Klasa se sastoji od varijabli, koje mogu biti ugrađeni tipovi, korisnički tipovi, i druge klase. Varijable unutar klase se nazivaju *podatkovni članovi (engl. "data members")*.

Funkcije unutar klase manipuliraju podacima u klasi. Funkcije unutar klase zovu se *funkcijski članovi* ili *metode ("methodes").*
Funkcijski članovi određuju što objekt može učiniti.

7.2.1. Deklaracija klase

Deklaracija klase počinje ključnom riječi *class,* te imenom klase iza koje slijedi otvorena vitičasta zagrada. Nakon toga slijedi lista podatkovnih članova i funkcijskih metoda klase.
Deklaracija završava zatvorenom vitičastom zagradom iza koje slijedi *točka-zarez*.
Primjer deklaracije klase CAutomobil:

```
class Cautomobil
{
int iBoja;
int iBrzina;
BOOL UpaliMotor ( void );
BOOL UgasiMotor ( void );
};
```

iBoja i *iBrzina* predstavljaju varijable unutar klase koje opisuju dijelove i karakteristike općeg automobila.

Deklaracija klase predstavlja novi tip podatka, kao što integer predstavlja tip podatka, u ovom slučaju opći automobil. Da bismo kreirali konkretni automobil u memoriji računala, moramo definirati objekt. Objekt se definira navođenjem imena klase iza koje slijedi ime objekta koje završava znakom ';'. Primjer:

```
int             iBrzina;
CAutomobil      topolino;
```

Definicijom objekta prevoditelj odvaja u memoriji računala prostor za njegov smještaj. Veličina koju zauzima objekt u memoriji definirana je klasom. Objekt klase automobil će u ovom slučaju zauzeti 4 (8) byte-ova.

Kreiranjem objekta iz klase kaže se da je kreirana *instanca (engl."instance")* klase.

7.2.2. podatkovni članovi

Podatkovni članovi klase mogu biti ugrađeni tipovi podataka *(int, char, long, ...)* ili korisnički definirani tipovi podataka kao npr. strukture ili druge klase.

Kada se kreira instanca klase tj. objekt podatkovni članovi predstavljaju stanja objekta.

Podatkovni članovi klase imaju sličnosti sa elementima struktura, s tom razlikom da se struktura deklarira sa ključnom riječju *struct*, dok se klasa deklarira sa *class.*

Podatkovnim članovima klase može se uglavnom pristupiti na isti način kao i članovima strukture, preko točka operatora. Naravno, u sljedećim poglavljima dolaze na vidjelo i prava pristupa elementima klase.

7.2.3. funkcijski članovi

Funkcijski članovi predstavljaju akcije koje se vrše nad podatkovnim članovima objekta. Funkcijski članovi još se zovu i *metode (engl. "methods").*

Pristup funkcijskim članovima preko objekta opisan je u sljedećem poglavlju.

7.3. Pristup članovima klase

Pristupanje podatkovnim i funkcijskim članovima objekta vrši se preko *(.)* operatora. Primjer:

```
topolino.iBoja = COLOR_CRVENA;
topolino.UpaliMotor();
topolino.iBrzina = 50;
```

7.4. Vidljivost podataka u klasi

Objekt neka svojstva izlaže vanjskom programu, dok neka zadržava za sebe.
Unutar deklaracije klase koriste se ključne riječi za dodjelu prava pristupa podacima i funkcijama. Te ključne riječi su: *public, private* i *protected.*

Ključne riječi za dodjelu prava pristupa se koriste tako da se navedu unutar deklaracije, te se ne završavaju karakterom ':'. Sve varijable i funkcije koje slijede iza navedene ključne riječi imaju navedeno pravo pristupa, do kraja deklaracije ili do početka nove ključne riječi za pravo pristupa.

```
class CAutomobil
{
public:
int iBoja;
int iBrzina;
BOOL StartajMotor (void);
BOOL IsključiMotor (void);

BOOL OtvoriProzor (void);
BOOL ZatvoriProzor (void);
BOOL StanjeMotora (void);

protected:
BOOL bStanjeMotora;
```

```
private:
BOOL bProzorOtvoren;
}

CAutomobil topolino;

topolino.StartajMotor();
topolino.iBoja = CRVENA;

BOOL b = topolino.bProzorOtvoren;     // ovo nije ispravno
BOOL b = topolino.bStanjeMotora;      // također nije ispravno

if ( topolino.StanjeMotora() )
{
       // motor radi, obavi neku funkciju
}
```

7.4.1. public

Podacima unutar objekta s *public* pravom pristupa, moguće je pristupiti preko funkcijskih metoda klase objekta, a dostupni su i iz vanjskog programa.

7.4.2. private

Podacima označenim s *private* pravom pristupa moguće je pristupiti samo unutar klase, preko funkcijskih članova klase. Obično se varijable unutar klase u deklaraciji navedu s *private* pravom pristupa, te im se iz vanjskih dijelova programa može pristupiti preko javnih funkcija. Primjer je varijabla *bStanjeMotora* kojoj se ne može pristupiti direktno, već uz pomoć javnog funkcijskog člana *StanjeMotora*.

Prilikom deklaracije klase ako se ne navede pravo pristupa, podrazumijeva se privatno pravo pristupa. U principu takva klasa je beskorisna, jer se ni jednom članu objekta te klase ne može pristupiti iz vanjskog programa.

7.4.3. protected

protected pravo pristupa slično je kao i *private,* ali ima važnu ulogu prilikom *nasljeđivanja*, koje ćemo kasnije obraditi.

7.5. Interface i implementacija

Funkcije za pristup podacima *(engl.“accessor function”)* predstavljaju javni *interface* prema privatnim podacima klase.
Svaki od funkcijskih članova za pristup mora imati implementaciju. Ta se implementacija zove definicija funkcijskog člana.

Definicija funkcijskog člana klase počinje imenom klase iza kojeg slijede karakteri ‘::’, ime funkcije, te njeni parametri.

Također, ukoliko je tijelo funkcijskog člana klase malo, npr. ako samo vraća vrijednost podatkovnog člana klase, definicija funkcije se može pisati unutar deklaracije klase.

```
class CAutomobil
{
public:
...
BOOL StanjeMotora (void);

// primjer definicije funkcije unutar klase
BOOL OtvoriProzor (void) { bProzorOtvoren = TRUE;};
BOOL ZatvoriProzor (void){ bProzorOtvoren = FALSE;};
...
private:
BOOL bStanjeMotora;
BOOL bProzorOtvoren;
...
}

// primjer definicije funkcije izvan klase
BOOL CAutomobil::StanjeMotora (void)
{
        return bStanjeMotora;
}
```

7.6. Prijatelji klase

Pristup iz vanjskog programa podacima tipa *private* nije dozvoljen direktno. Pristup se može vršiti preko javnih funkcijskih članova, ili unutar klase preko funkcijskih članova klase.

Postoji međutim mogućnost pristupa privatnim i zaštićenim članovima klase preko vanjske funkcije ili druge klase, ako se ta funkcija ili vanjska klasa proglasi prijateljem (*"friend"*) klase kojoj se pristupa podacima i funkcijama.

Da bismo neku funkciju napravili prijateljem određene klase i time joj omogućili pristup svim podacima te klase, unutar deklaracije klase moramo navesti deklaraciju funkcije prethođenu ključnom riječi *friend.* Primjer:

```
class CAutomobil
{
        friend BOOL PopraviMotor (CAutomobil &auto);
public:
...
int bMotorIspravan;
}

friend PopraviMotor (CAutomobil &auto)
{
        auto.PopraviMotor();
}

BOOL PopraviMotor (CAutomobil &auto)
{
        if ( !auto.bMotorIspravan )
                auto.bMotorIspravan = TRUE;
```

```
return TRUE;
}

...

CAutomobil topolino;

if ( !bMotorIspravan )
{
        PopraviMotor ();
}
```

Funkciju ili klasu prijatelja klase nije potrebno prethodno deklarirati prije deklaracije klase kojoj su prijatelji.

Funkcija prijatelj se može definirati i unutar klase. Tada će ona postati umetnuta, kao i ostale funkcije definirane unutar deklaracije klase.

Više o funkcijama i klasama prijateljima u jednom od sljedećih poglavlja.

7.7. this ključna riječ

Prilikom poziva svakog funkcijskog člana klase, kao skriveni parametar prosljeđuje se pokazivač na objekt kojemu taj član pripada. Pokazivaču se može pristupiti pomoću ključne riječi *this*.

```
void CAutomobil::OtvoriProzor(void)
{
   // bProzorOtvoren = TRUE; // ili
   this->bProzorOtvoren = TRUE;
}
```

7.8. Konstruktor i destruktor

Varijable ugrađenih tipova podataka možemo definirati i nakon toga im pridijeliti vrijednost. Isto tako možemo im dodijeliti vrijednost prilikom definiranja. Dodjela vrijednosti prilikom definicije zove se inicijalizacija.

```
int a;          // definicija
a=5;            //
```

ili

```
int b = 5;      // inicijalizacija
```

7.8.1. konstruktor

Podatkovni članovi klase se također mogu inicijalizirati prilikom definiranja objekta. Klasa ima u tu svrhu posebni funkcijski član nazvan konstruktor. Konstruktor ima isto ime kao i klasa, može primati parametre po potrebi, te nema povratnu vrijednost.

Pretpostavimo da imamo klasu CPoint koja sadrži koordinate točke u kartezijevom koordinatnom sustavu. Definirajmo također nekoliko objekata klase CPoint:

```
class CPoint
{
private:
        int iX, iY;
public:
        CPoint ();
        CPoint ( int x, int y );

        int GetX (void);
        int GetY (void);

        void SetX ( int x ) { iX = x }
        void SetY ( int y ) { iY = y }
}

CPoint::CPoint ()
{
        iX = 0;
        iY = 0;
}
CPoint::CPoint ( int x, int y )
{
        iX = x;
        iY = y;
}

...

CPoint tocka1;
CPoint tocka2 ( 10, 15 );
CPoint *pokTocka2 = &tocka2;
CPoint &refTocka2 = tocka2;
```

- generirani objekt *tocka1* inicijalizira se preko konstruktora bez parametara na vrijednost (0,0)
- vrijednosti objekta tocka2 se inicijaliziraju na iX = 10, iY = 15.
- pokazivaču pokTocka2 prilikom inicijalizacije pridjeljuje se adresa objekta tocka2
- refTocka2 referenca na objekt tocka2.

Poziv konstruktora bez parametara mora biti bez zagrada kao npr. za definiranje objekta *tocka1*. Kada bi se stavile zagrade prevoditelj bi to shvatio kao deklaraciju funkcije koja vraća CPoint objekt.

Konstruktor dolazi do izražaja u slučaju kada se koristi dinamička alokacija memorije. Npr. ako je potrebno alocirati polje objekata. Pretpostavimo da nam treba klasa koja sadrži polje točaka u koordinatnom sustavu, s tim da nije fiksna veličina polja. Možemo deklarirati klasu *CPoljeTocaka*. Prilikom kreiranja objekta iz ove klase specificiramo konstruktoru koliko je točaka potrebno. Primjer:

```
class CPoljeTocaka
{
private:
        int     iBrojTocaka;
        CPoint *pokPoljeTocaka;

public:
        CPoljeTocaka ( ) :
                iBrojTocaka(5),
                pokPoljeTocaka ( new CPoint[5]  )
        {
```

```
        }

        CPoljeTocaka ( int brojTocaka ) :
                iBrojTocaka ( brojTocaka ),
                pokPoljeTocaka ( new CPoint[brojTocaka] )
        {
        }
}
```

Konstruktor ne alocira memoriju za smještanje objekta, već to radi prevoditelj. Konstruktor inicijalizira alociranu memoriju.

Konstruktor se kao funkcija razlikuje od običnih funkcija, jer prije ulaska u tijelo funkcije konstruktora se mogu inicijalizirati neki od podatkovnih članova.
Treba paziti da inicijalizacija članova ne ide po redu kako su navedeni u inicijalizacijskoj listi, nego po redoslijedu navođenja elemenata u deklaraciji klase.

Ako klasa sadrži objekte druge klase, te se inicijalizacija tih objekata obavlja u konstruktoru, on će pozvati navedeni konstruktor objekta te klase, a ako konstruktor ne postoji, pozvat će se podrazumijevani konstruktor (opisan u slijedećem poglavlju).

7.8.1.1. podrazumijevani konstruktor

Ako se ne definira konstruktor za klasu, prevoditelj ga generira sam. Podrazumijevani *(engl."default")* konstruktor ne prima nikakve podatke, te ne vraća nikakvu vrijednost.
Podrazumijevani konstruktor ostavlja ugrađene tipove podataka neinicijalizirane, a ako su unutar klase definirani objekti, klasa poziva njihov podrazumijevani konstruktor.

Za klase koje kao članove imaju reference i konstantne objekte, prevoditelj ne može generirati podrazumijevani konstruktor, jer se oni moraju inicijalizirati u inicijalizacijskoj listi. Konstantni članovi se ne mogu mijenjati u toku rada programa, a referenca se ne može definirati bez pridruživanja nekom elementu, ona sama za sebe ne egzistira.

7.8.1.2. konstruktor kopije

Ako je potrebno stvoriti objekt koji je po svojim članovima jednak nekom već postojećem objektu, koristi se *konstruktor kopije*.
Konstruktor kopije prima samo jedan parametar, referencu na već postojeći objekt.

```
CPoint::CPoint ( CPoint &tocka ):
iX( tocka.iX ),
iY( tocka.iY )
{}

CPoint tocka1( 5, 6 );
CPoint tocka2 ( tocka1 );
CPoint tocka3 = tocka1;
```

Ako konstruktor kopije nije korisnički definiran, prevoditelj ga generira sam. Takav konstruktor ugrađene tipove podataka jednostavno kopira u drugi objekt, dok za objekte koje klasa može sadržavati poziva njihove konstruktore kopije.

Za slučaj da se koriste pokazivači i alokacija memorije, korištenje automatskog konstruktora kopije može stvoriti probleme, jer se vrijednost pokazivača jednostavno kopira u novi objekt, te se ne alocira nova memorija. Za takve slučajeve potrebno je definirati korisnički konstruktor kopije.

7.8.1.3. inicijalizacija referenci i konstantnih članova

Klase mogu sadržavati konstantne članove i reference na druge objekte. Konstantni članovi i reference ne mogu mijenjati vrijednosti u toku rada programa, pa ih se mora inicijalizirati u inicijalizacijskoj listi konstruktora.

Primjer inicijalizacije konstantnog člana je u objektu klase koja sadrži polje točaka. Kada se u početku postavi veličina polja, ona se više ne mijenja. Svaki pokušaj promjene u kodu rezultirao bi prijavom greške od strane prevoditelja.

```
class CPoljeTocaka
{
...
const int iBrojTocaka;
CPoint *pokPoljeTocaka;
...
CPoljeTocaka ( int brojTocaka ):
        iBrojTocaka ( brojTocaka )
        pokPoljeTocaka ( new CPoint[iBrojTocaka] )
{
}
...
}
```

Reference ne možemo ostaviti neinicijalizirane. Moramo ih inicijalizirati u konstruktoru. Klasu *CLinija*, koja ima reference na dva objekta klase CPoint, možemo inicijalizirati u konstruktoru.

```
class CLinija
{
CPoint &pocTocka, &krajnjaTocka;

CLinija ( CPoint &t1, Cpoint &t2 ):
        pocTocka ( t1 ),
        krajnjaTocka ( t2 )
{
}

}
```

Kod korištenja ulaznih objekata u konstruktoru t1 i t2 mora se uz njih nalaziti operator '&' koji prosljeđuje referencu na objekte, jer ako ga ne bi bilo uzele bi se reference kopija tih objekata, koje bi se uništile prilikom izlaska iz tijela konstruktora.

7.8.1.4. konstruktori i prava pristupa

Konstruktori kao i ostali članovi klase podliježu pravima pristupa, pa se treba paziti koje im se pravo dodjeljuje *(public)*.

7.8.2. destruktor

Na kraju "života" objekta, lokalnog na kraju funkcije, ili globalnog na kraju programa, postoji potreba da se počisti memorija koju je objekt alocirao, recimo ako se koristila dinamička alokacija memorije. U tu svrhu se koristi također posebni funkcijski član klase nazvan *destruktor*. *Destruktor* također ima isto ime kao i *konstruktor*, s tim što mu se ispred imena funkcije nalazi tilda '~' karakter. *Destruktor* nema nikakvih parametara, te ne daje nikakvu povratnu vrijednost.

```
class CLinija
{

...
~CLinija ();

}

CLinija::~CLinija ()
{
        delete [] pokPoljeTocaka;
}
```

Iza operatora *delete* navedeno je '[]' koje znači da je riječ o polju objekata tipa CPoint. Kada se to ne bi navelo *delete* bi uništilo samo prvi objekt, a ostali dio memorije bio bi trajno izgubljen za program.

7.9. const funkcije

Konstantni funkcijski članovi osiguravaju da njihov poziv neće promijeniti podatkovne članove objekta. Ukoliko ih funkcija mijenja, a proglašena je tipom *const*, prevoditelj će javiti grešku prilikom prevođenja. Primjer funkcije:

```
class CPoint
{
...
CPoint DajPocTocku ( void ) const;    // funkcija ne mijenja objekt, samo vraća
                                      // vrijednost pocetne tocke

}
```

Ključna riječ *const* navodi se na kraju funkcijskog potpisa.

7.10. volatile

Objekti klase mogu se deklarirati koristeći ključnu riječ *volatile*. Pristup takvim objektima unutar klase moguć je samo preko funkcijskih članova također tipa volatile. Korištenje funkcijskih članova koji nisu tipa *volatile* prevoditelj će prijaviti kao grešku. Primjer:

```
class CLinija
{
...
public:
        CPoint pocTocka;
        CPoint krajnjaTocka;

        CPoint  DajPocTocku () volatile;
        CPoint DuljinaLinije();
...
}

main ()
{
volatile CLinija linija1;

...

linija1.DajPocTocku ();       // OK
linija1.DuljinaLinije();      // greška

...

}
```

7.11. statički članovi klase

Statički član klase je zajednički svim objektima klase. On nije sadržan u svakom objektu, već je globalan za cijelu klasu.
Deklaracija statičkog člana vrši se tako da se ispred tipa člana stavi ključna riječ *static*.
Primjer:

```
class CPoint
{
...
static int iBrojKreiranihTocaka;
...
}
```

Inicijalizacija statičkog člana mora se eksplicitno obaviti navođenjem potpunog imena statičkog člana.

```
int CPoint::iBrojKreiranihTocaka = 0;
```

Pristup statičkom članu moguć je preko imena klase i preko imena objekta:

```
CPoint tocka1;

a = CPoint::iBrojKreiranihTocaka;
tocka1.iBrojKreiranihTocaka++;
```

Statički članovi klase mogu se navesti kao podrazumijevani parametri funkcija.

```
CPoint::UzimaStaticClan ( int = iBrojKreiranihTocaka );
```

Ovo se ne može primijeniti na obične članove klase.

7.11.1.1. Statički funkcijski članovi

Klasa može sadržavati statičke funkcijske članove. Oni mogu pristupati samo statičkim članovima klase. Statički funkcijski član u sebi ne sadrži skriveni *this* pokazivač pa je preko statičke funkcije nemoguće pristupiti ostalim članovima objekta klase. Prednost ovakvih funkcija je u tome da one mogu pristupiti statičkim članovima klase čak i kada ni jedan objekt te klase nije definiran.

7.12. umetnute funkcije

Funkcije se mogu definirati unutar deklaracije klase, u slučaju da njihovo tijelo stane u par crta. Takve funkcije su automatski umetnute *(engl. "inline")* funkcije.
Ukoliko funkcija ima veće tijelo, njena definicija može biti izvan deklaracije klase, s tim da unutar deklaracije klase prije deklaracije funkcijskog člana mora stajati ključna riječ *inline*.

Ovakve funkcije rezultiraju bržim kodom, jer se kod funkcije kopira direktno na mjesto poziva funkcije i ne treba vršiti operacije skoka do nje.

```
class CKlasa
{
   ...
   inline int GetValue();
   ...
};
```

7.13. područje razlučivanja klase

Objekt klase je vidljiv u području *(engl. "scope")* u kojem je definiran. Dva objekta ili varijable istog imena ne mogu egzistirati u istom području definicije.

Unutar funkcija varijable su vidljive od mjesta na kojem su definirane. Ako je varijabla definirana unutar bloka unutar funkcije, tada je ona vidljiva samo unutar tog bloka.

Ako postoje dvije varijable istog imena, jedna definirana unutar tijela funkcije a druga unutar bloka koje je unutar tijela funkcije, tada je unutar bloka unutar tijela funkcije vidljiva ova druga varijable, te ona skriva prvu varijablu.

Kada funkcija izađe iz bloka, ponovno je vidljiva prva varijabla.

Slična je situacija kada imamo globalnu i lokalnu varijablu unutar funkcije istog imena. Unutar funkcije lokalna varijabla skriva globalnu. Da bi se pristupilo globalnoj varijabli mora se koristiti *operator za razlučivanje područja*, '::' ispred imena varijable. Tako se prevoditelju daje na znanje da se radi o globalnoj varijabli.

Ista je situacija i unutar funkcijskog člana objekta klase. Funkcijski članovi klase imaju pristup članovima u području klase.

Redoslijed određivanja područja kojem varijabla pripada je sljedeći:

- pretraživanje tekućeg bloka, ako je definirana varijabla, on se uzima, ako ne pretražuje se blok iznad
- funkcijski član, pretražuje se da li postoji lokalna varijabla tog imena
- područje klase – da li postoji podatkovni član klase
- globalno područje – postoji li varijabla u globalnom području

Ako ni u jednom od ovih područja varijabla nije pronađena prevoditelj javlja grešku prilikom prevođenja.

7.14. pokazivači i klase

7.14.1. pokazivači na podatkovne članove

Pokazivači na podatkovne članove klase, za razliku od običnih pokazivača, predstavljaju podatak o relativnoj udaljenosti podatkovnog člana od početka objekta.

Deklaracija pokazivača na određeni tip podatka koji je član npr. klase CLinija ima oblik:

```
int CLinija::*
```

Ovo je pokazivač integer tip podataka klase *CLinija*.

Za točno određeni član klase, npr. X1, koordinatu prve točke linije, može se pisati:

```
int CLinija::*pokX1 = &CLinija::X1;
```

Karakteristike pokazivača na podatkovne članove klase:

- pokazivač pokX1 ne pokazuje na neku konkretnu vrijednost, već daje relativni pomak od početka objekta klase.
- pokazivači na jedan tip podatkovnog člana neke klase ne mogu se pridružiti adresi pokazivača na drugi tip podatkovnog člana te iste klase (ne može se miješati pokazivač na integer i float tip podatkovnog člana u istoj klasi)

- pokazivači na podatkovne članove objekta jedne klase ne mogu se pridruživati podatkovnim članovima objekta druge klase.
- pokazivači se ne mogu konvertirati u *void**
- kod pokazivača nema implicitne konverzije u aritmetički tip (npr. `if (pokNesto)` - ovo je neispravno)
- može se definirati pokazivač na član klase
- pokazivač na član klase moguće je proslijediti kao parametar funkcijama

Pristup podatkovnim članovima objekta klase može se vršiti na dva načina: preko objekta operatorom ".*" ili preko pokazivača na objekt operatorom "->*". Primjer:

```
class CLinija
{
public:
int X1, Y1, X2, Y2;
...
}

CLinija linija, *pokLinija = &linija;

main()
{
int CLinija::*pokClan = &CLinija::X1;

linija.*pokClan = 32;

pokClan = &CLinija::X2;

pokLinija->*pokClan = 76;

...

}
```

7.14.2. pokazivači na funkcijske članove

Pokazivači na funkcijske članove klase ne određuju direktno adresu funkcije na koju pokazuju. Oni identificiraju određeni funkcijski član klase.

Poziv funkcijskog člana pomoću pokazivača vrši se navođenjem objekta ili pokazivača na objekt. Primjer:

```
class CLinija
{
...

void Translatiraj ( int dx, int dy);
void Rotiraj ( int kut );

void ObradiLiniju ( (CLinija::*funkcija) ( int, int ) );
}
```

Deklaracija pokazivača na funkciju se vrši navođenjem tipa povratne vrijednosti, klase funkcije te njenih parametara.

Parametri funkcijskog člana navode se u zagradama:

```
void ( CLinija::*pokNaFunkciju ) ( int, int );
```

Dodjela vrijednosti pokazivaču vrši se:

```
pokNaFunkciju = CLinija::Translatiraj;
```

Poziv funkcije preko objekta ili pokazivača na objekt vrši se na sljedeći način:

```
CLinija linija2, *pokNaLin2 = &linija2;
...
(linija2.*pokNaFunkciju) ( 15, 4 );


(pokNaLin2->*pokNaFunkciju) (12,3);
...
```

Upotreba zagrada oko naziva funkcije i objekta je obavezna.

Poziv funkcije koja kao parametar prenosi pokazivač na funkciju:

```
ObradiLiniju ( pokNaFunkciju ( 15, 4 ) );
ili
ObradiLiniju ( CLinija::Translatiraj ( 15, 4 ) );
```

Operacije dozvoljene s pokazivačima na funkcijske članove su iste kao i s pokazivačima na podatkovne članove. Pokazivači na funkcijske članove su ovisni o potpisu funkcije (tipovima ulaznih i izlaznih parametara), tj. mogu se dodijeliti samo pokazivačima na funkcijske članove koji imaju istu povratnu vrijednost i iste parametre.

7.15. privremeni objekti

Prevoditelj stvara i uništava *neimenovane privremene objekte*, koji služe za pohranjivanje vrijednosti.

7.15.1. privremeni objekti kod prijenosa parametara u funkciju

Objekti se kao argumenti funkcije mogu prenositi preko reference, pokazivača i preko vrijednosti. Prilikom prijenosa objekta preko reference ili pokazivača, svaka operacija odražava se direktno na originalnom objektu, te se zadržava izlaskom programa iz funkcije.

Prilikom prijenosa objekta funkciji preko vrijednosti kreira se privremeni objekt. Sve operacije nad tim privremenim objektom neće se odraziti nad originalnim objektom.

Privremeni objekt koji je vidljiv unutar funkcije kreira se kao kopija originalnog objekta *copy konstruktorom*.

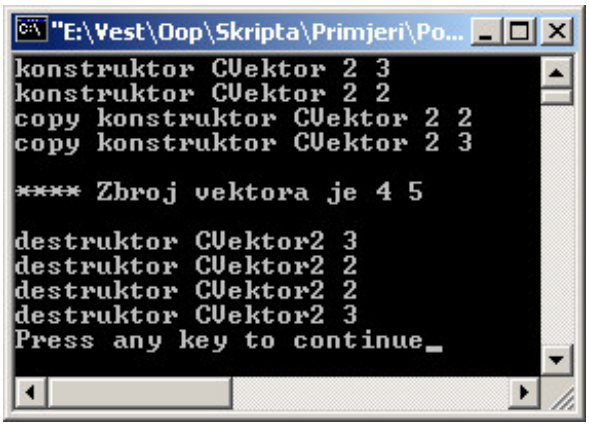
Život ovakvog privremenog objekta je unutar funkcije. Na kraju funkcije poziva se destruktor koji uništava objekt.

Primjer prosljeđivanja objekta funkciji preko vrijednosti.

Kreirana je klasa *CVektor* koja predstavlja vektor sa svojom x i y komponentom. U programu koji koristi klasu *CVektor* kreirana je funkcija *IspisiZbroj* kojoj se prenose po vrijednosti dva objekta klase *CVektor*.

```
class CVektor
{
public:
        CVektor():dX(0),dY(0)
        {
                cout << "konstruktor CVektor " << dX << " " << dY << endl;
        };

        CVektor(CVektor &vektor)
        {
                dX = vektor.DajX();
                dY = vektor.DajY();

                cout << "copy konstruktor CVektor " << dX << " " << dY << endl;
        };

        CVektor(double x, double y):dX(x), dY(y)
        {
                cout << "konstruktor CVektor " << dX << " " << dY << endl;
        }

        ~CVektor()
        {
                cout << "destruktor CVektor" << dX << " " << dY << endl;
        }
        void PostaviX(double x){ dX =x; }
        void PostaviY(double y){dY =y; }
        double DajX(void){return dX; }
        double DajY(void){return dY; }
private:
        double dX;
        double dY;
};

void IspisiZbroj ( CVektor vektor1, CVektor vektor2 )
{
        cout << endl << "**** Zbroj vektora je " << vektor1.DajX() + vektor2.DajX() <<
" " << vektor1.DajY() + vektor2.DajY() << endl << endl;
}

int main()
{
        CVektor a(2, 3), b(2, 2);

        IspisiZbroj( a, b );

        return 0;
}
```

Ispis programa je sljedeći:



Opis:

U programu su kreirana dva *CVektor* objekta. Ta dva objekta proslijeđena su funkciji *IspisiZbroj* po vrijednosti. Prilikom prosljeđivanja od ta dva objekta kreirane su lokalne kopije pozivom copy konstruktora.
Te kopije kao lokalni objekti korišteni su prilikom ispisa zbroja vektora.
Izlaskom programa iz funkcije, pozivaju se destruktori za dva lokalna objekta.

Izlaskom iz funkcije ujedno se dolazi do kraja programa, te se pozivaju destruktori dvaju objekata *a* i *b*.

Promjena vrijednosti lokalnih objekata unutar funkcije ni na koji način ne utječe na objekte *a* i *b*.

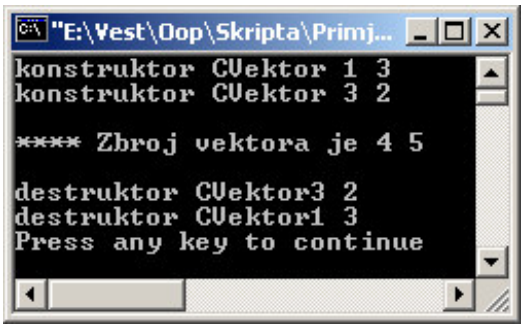
Druga mogućnost prosljeđivanja objekta funkciji je kreiranje bezimenog privremenog objekta.

Prethodni primjer se može promijeniti tako da se funkciji proslijede dva privremena bezimena objekta koja se kreiraju pomoću konstruktora klase *CVektor* kojem se navedu argumenti.
Razlika u odnosu na prethodni kod je samo u *main* funkciji:

```
int main()
{
        IspisiZbroj( CVektor(3,2), CVektor(1,3) );

        return 0;
}
```

Umjesto da se funkciji proslijede objekti prosljeđuje se konstruktor *CVektor (3,2)* i *CVektor(1,3)*. To rezultira kreiranjem lokalnih objekata unutar funkcije *IspisiZbroj*.

Program startanjem daje sljedeći rezultat:



Vidi se razlika u broju kreiranih objekata u odnosu na prethodni primjer. Dva objekta su kreirana lokalno unutar funkcije. Na kraju funkcije poziva se destruktor te se objekti uništavaju.

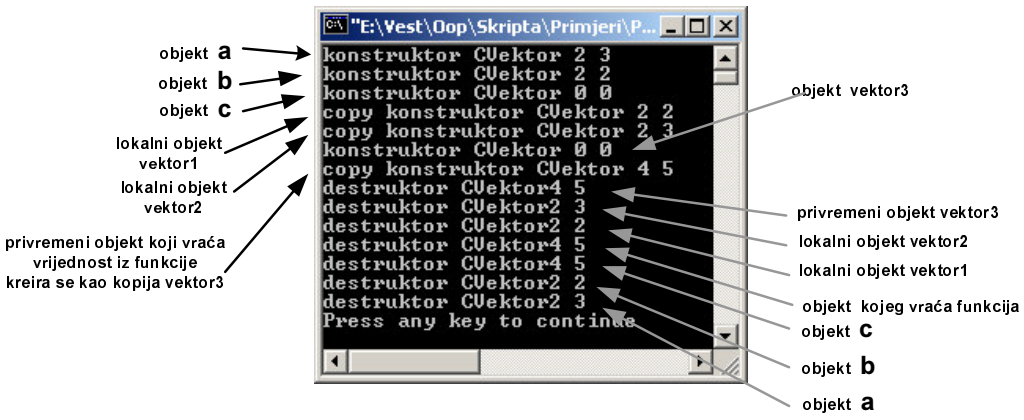
7.15.2. privremeni objekti kod vraćanja vrijednosti

Kada funkcija kao povratnu vrijednost vraća privremeni objekt, moguće je to realizirati na dva načina.

1. Kreiranjem privremenog objekta

```
CVektor ZbrojiVektore ( CVektor vektor1, CVektor vektor2 )
{
        CVektor vektor3;

        vektor3.PostaviX(vektor1.DajX() + vektor2.DajX());
        vektor3.PostaviY(vektor1.DajY() + vektor2.DajY());

        return vektor3;
}

int main()
{
        CVektor a(2, 3), b(2, 2), c;

        c = ZbrojiVektore( a, b );

        return 0;
}
```

Program daje sljedeći rezultat:



Uz ispis programa naznačeno je kojem objektu pripadaju koji konstruktori i destruktori. Redoslijed destrukcije objekata obrnut je od redoslijeda konstrukcije.

Prvo se kreiraju objekti a, b i c. Nakon toga proslijede se po vrijednosti objekti a i b funkciji *ZbrojiVektore*. Pri tome se sa copy konstruktorima kreiraju njihove lokalne kopije *vektor1* i *vektor2*.
Lokalni objekt *vektor3* nema nikakvih parametara prilikom konstrukcije, te je inicijaliziran na vrijednost 0,0.

Unutar funkcije postavljaju se vrijednosti objekta *vektor3,* kao zbroj pojedinih vrijednosti podatkovnih članova objekata *vektor1* i *vektor2*.

Objekt *vektor3* se vraća kao povratna vrijednost funkcije. Međutim on se ne vraća direktno već prevoditelj kreira privremeni objekt, kojeg inicijalizira copy konstruktorom. Taj privremeni objekt se pridružuje objektu *c* sa lijeve strane jednakosti kada program izađe iz funkcije.

Nakon što je kreiran bezimeni privremeni objekt, uništavaju se lokalni objekti funkcije obrnutim redoslijedom od njihovog kreiranja *vektor3, vektor2, vektor1*.

Funkcija je vratila kao rezultat privremeni bezimeni objekt sa vrijednostima zbroja dvaju vektora. Taj objekt se sada nalazi s desne strane jednakosti, te se pridružuje objektu *c* koji se nalazi s lijeve strane jednakosti.

Nakon pridruživanja privremeni objekt se uništava, pozivanjem destruktora.

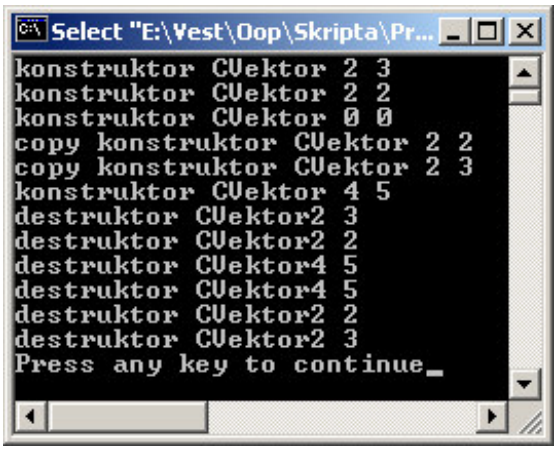
Dolaskom do kraja *main* funkcije, pozivaju se destruktori objekata *c, b* i *a.*

Druga mogućnost prilikom vraćanja objekta kao povratne vrijednosti funkcije je vraćanje bezimenog objekta. U *return* izrazu se kreira bezimeni objekt sa inicijalnim vrijednostima zbroja dvaju vektora. Time se preskače kreiranje lokalnog objekta *vektor3.*

Primjer:

```
CVektor ZbrojiVektore ( CVektor vektor1, CVektor vektor2 )
{
        return CVektor(vektor1.DajX() + vektor2.DajX(), vektor1.DajY() + vektor2.DajY()
);
}

int main()
{
        CVektor a(2, 3), b(2, 2), c;

        c = ZbrojiVektore( a, b );

        return 0;
}
```

Ispis je sljedeći:



Ispis je sličan prethodnom osim što sada nema lokalnog objekta kojeg je predstavljao *vektor3* objekt, te nema copy konstruktora kojim se on kopirao u bezimeni privremeni objekt, već se direktno kreira bezimeni objekt sa vrijednostima zbroja dvaju vektora.

8. Funkcijski članovi i operatori

8.1. Preopterećenje operatora

C++ ima u sebi ugrađene tipove: *char, int, double*, itd. Svaki od ovih tipova ima nekoliko ugrađenih operatora, npr. '+' i '-'.

C++ omogućava da se korisničkim klasama dodaju osnovni operatori. Osim operatora koji su ugrađeni u sam jezik nije moguće definirati nove operatore.

8.1.1. Preopterećenje operatora +

Ako je potrebno zbrojiti dva cjelobrojna broja, te rezultat pridijeliti trećem,

```
int a=3;
int b=4;
int c;

c = a + b;
```

Ispis rezultata daje vrijednost 7.

Međutim da bi se obavilo zbrajanje dvaju objekata neke klase treba za tu klasu definirati nad kojim će članovima klase izvršiti operacija.

Kao primjer klase nad kojom treba izvršiti neke matematičke operacije može biti klasa *CVektor*. Ova klasa može predstavljati dvodimenzionalni vektor.

```
class CVektor
{
public:
        CVektor ():dX(0), dY(0){}
        CVektor ( double x, double y):dX(x), dY(y){}
        void PostaviX(int x){ dX = x; }
        void PostaviY(int Y){ dY = Y; }
        double DajX(void){ return dX; }
        double DajY(void){     return dY; }

private:
        double dX;
        double dY;
};
```

Mogu se definirati tri objekta klase *CVektor*.

```
CVektor a(2, 3), b(2, 2), c;
```

Potrebno je zbrojiti vektore a i b te rezultat spremiti u c. To se može obaviti na nekoliko načina.

Prvo se može kreirati globalna funkcija koja zbraja dva vektora i vraća privremeni objekt koji predstavlja zbroj dvaju vektora.

```
CVektor ZbrojiVektore( CVektor vektor1, CVektor vektor2 )
{
   CVektor temp ( vektor1.DajX() + vektor2.DajY(), vektor1.DajY() + vektor2.DajY() );
return temp;
}
```

Zbrajanje dva vektora vršilo bi se na sljedeći način:

```
c = ZbrojiVektore ( a, b );
```

Zbrajanje dvaju vektora vezano je za klasu CVektor, te bi operacija zbrajanja trebala biti vezana uz funkcijski član klase CVektor. Može se kreirati funkcijski član klase imena Zbroji()

```
CVektor::CVektor Zbroji ( CVektor &rhs )
{
        CVektor temp ( dX+rhs.DajX(), dY + rhs.DajY());
        return temp;
}
```

Zbrajanje bi sada izgledalo ovako:

```
c = a.Zbroji(b);
```

Ipak najelegantnije bi bilo zbrajanje dvaju objekata klase vektor izvesti na sljedeći način:

```
c = a + b;
```

Da bi ovakav način zbrajanja dvaju objekata bio moguć, potrebno je preopteretiti operator zbrajanja unutar klase *CVektor*.

```
CVektor CVektor::operator+(CVektor &rhs);
```

Klasi je dodan funkcijski član *operator*+, te ona sada izgleda ovako:

```
class CVektor
{
public:
        CVektor():dX(0),dY(0){};
        CVektor(double x, double y):dX(x), dY(y){};
        void PostaviX(double x){ dX =x; }
        void PostaviY(double y){dY =y; }
        double DajX(void){return dX; }
        double DajY(void){return dY; }
        CVektor operator+(CVektor &);
private:
        double dX;
        double dY;
};

CVektor CVektor::operator+(CVektor &rhs)
{
        CVektor temp ( dX + rhs.DajX(), dY + rhs.DajY());
        return temp;
}
```

Operatorska funkcija kao parametar ima referencu na objekt klase *CVektor*. To je objekt koji se nalazi desno od operatorske funkcije. Lijevo od operatorske funkcije nalazi se objekt čija se operatorska funkcija koristi.

Primjer korištenja operatorske funkcije je sljedeći:

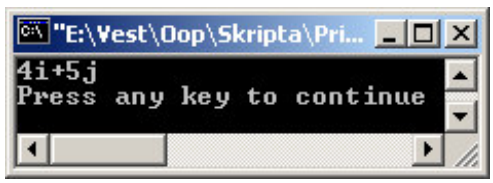
```
void Ispisi(CVektor &vektor)
{
        cout << vektor.DajX() << "i+"<< vektor.DajY() << "j" << endl;
}

int main()
{
        CVektor a(2, 3), b(2, 2), c;

        c = a + b;

        Ispisi(c);

        return 0;
}
```

Umjesto c = a + b moguće je pisati:

```
c = a.operator+(b);
```

U principu ovakav način pisanja je isti kao i gornji, ali očito je koji je način očitiji i razumljiviji.

Ispis je u oba slučaja isti:



8.1.2. Preopterećenje inkrement operatora

Preopterećenje inkrement operatora ovisi o tome da li se koristi prefiks ili postfiks inkrement operator.

Kao primjer može se uzeti objekt klase *CBrojac*:

```
class CBrojac
{
public:
        CBrojac():iCounts(0){};
        int DajStanje(){ return iCounts; }
        int operator++(int){ return iCounts++; }
        int operator++(){ return ++iCounts; }
private:
        int iCounts;
};

int main()
{
        CBrojac brojac;

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

Klasa *CBrojac* po default-u nema ugrađeni inkrement operator, te se on definira kao funkcijski član.

Postoje dvije verzije inkrement operatora: prefix i postfiks.
Prilikom korištenja kod ugrađenih tipova podataka (char, short, int i long) prefiks inkrement operator prvo povećava vrijednost varijable, te onda pridružuje vrijednost varijable, dok postfiks inkrement operator prvo pridružuje vrijednost varijable, a tek onda povećava vrijednost varijable.

Primjer:

```
int a = 5;
int b = ++a;
int c = a++;
```

Slično je i sa '- -' operaotrom

Za klasu *CBrojac* definirano je da ++ operator povećava vrijednost podatkovnog člana *iCounts* za jedan.

Da bi prevoditelj prepoznao razliku između prefix i postfix operatora, za postfix operator uvedeno je pravilo da se prilikom deklaracije treba kao parametar navesti *int* tip podatka iako je ta operatorska funkcija unarna, te ne prima nikakve parametre. S time je definirano da je riječ o postfix operatoru.

8.1.3. Preopterećenje operatora pridruživanja =

Ovaj operator se poziva uvijek kada se jedan objekt pridružuje drugom. Na primjer:

```
CVektor a(2, 4), b;
a = b;
```

Rezultat operacije pridruživanja objekta *b* objektu *a* je pridruživanje vrijednosti svih podatkovnih članova objekta b objektu a. U slučaju vektora kopirale bi se vrijednosti varijabli dX i dY objekta b u dX i dY objekta a.

Ugrađeni operator pridruživanja funkcionira ukoliko objekt klase sadrži podatkovne članove koji su varijable određenih tipova podataka. Međutim problem nastaje kad jedan od podatkovnih članova predstavlja pokazivač na dinamički alociranu memoriju.

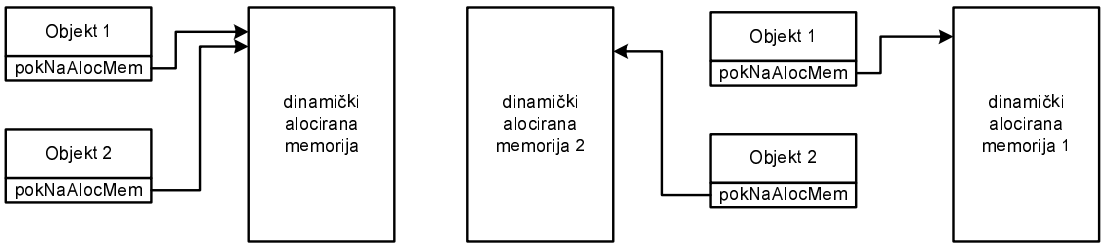
Prilikom operacije pridruživanja objekt s lijeve strane operatora sada ima pokazivač čija vrijednost pokazuje na isto memorijsko područje kao i objekt čiji su podatkovni članovi kopirani.

Problem nastaje kada se jedan od ova dva objekta uništi. Naime prilikom uništavanja objekta zove se destruktor u kojem je vjerojatno dio koda za oslobađanje dinamički alocirane memorije. Sada objekt koji nije uništen sadrži pokazivač na dio memorije koji više ne postoji. Pristup drugog objekta nepostojećoj memoriji može uzrokovati rušenje programa.

Kreirana kopija objekta korištenjem ugrađenog operatora pridruživanja zove *plitka kopija (engl. "shallow copy").*

Za razliku od plitke kopije može se kreirati *duboka kopija (engl "deep copy")*, preopterećenjem ugrađenog operatora pridruživanja. Kreiranjem duboke kopije (npr. kod pokazivača na alociranu memoriju) može se alocirati memorija, kopirati njen sadržaj i pokazivaču u novom objektu pridružiti adresa na kopiju alocirane memorije.

Na slici je prikazan primjer kopiranja objekta koji sadrži pokazivač na dinamički alociranu memoriju: a) plitkom kopijom, b) dubokom kopijom.



Također moguće je preopteretiti operator ispitivanja jednakosti, te se može izdefinirati što je to jednakost dvaju objekata. Npr. dva vektora su jednaka samo ako su im x i y komponente jednake.

8.2. Operatori konverzije

Kako prevoditelj reagira kada se neki od ugrađenih tipova podataka pokuša pridružiti nekom objektu.
Kako prevoditelj reagira ako se objekt određene klase pridruži nekom od ugrađenih tipova podataka, npr. tipu *int*?

Ako se kao primjer uzme prethodno definirana *CBrojac* klasa, cjelobrojni tip podatka pridruži objekt klase *CBrojac*.

```
...
int pocBrojTikova = 15;

CBrojac brojac = pocBrojTikova;
```

Ako klasa ima samo *defualt* konstruktor prevoditelj će javiti grešku prilikom prevođenja.

Međutim ako je definiran konstruktor koji prima jedan cjelobrojni parametar (u slučaju gornjeg primjera), tada je moguće pridruživanje operatorom jednakosti. Unutar konstruktora sa jednim cjelobrojnim parametrom definirano je kako će se pridruživanje obaviti.

U primjeru klase *CBrojac* potrebno je kreirati konstruktor koji prima jedan cjelobrojni parametar.

```
CBrojac::CBrojac ( int startVrijednost )
{
        iCounts = startVrijednost;
}
```

Pridruživanje varijable *pocBrojTikova* objektu *brojac* interno kreira privremeni objekt kome je *iCounts* inicijaliziran na vrijednost varijable *pocBrojTikova*.
Tako kreirani privremeni objekt klase *CBrojac* se pridružuje objektu *brojac*, tj. vrši se kopiranje podatkovnih članova privremenog objekta u podatkovne članove objekta *brojac*.

Pridruživanje objekta nekom od ugrađenih tipova podataka npr. varijabli tipa *int* nije automatski definirano već je potrebno definirati kako će se konverzija izvoditi. Prevoditelj ne zna kako konvertirati objekt koji može sadržavati više podatkovnih članova u jednu varijablu.

Kao primjer može se ponovno uzeti objekt klase *CBrojac*, te ga se može pridijeliti cjelobrojnoj varijabli.

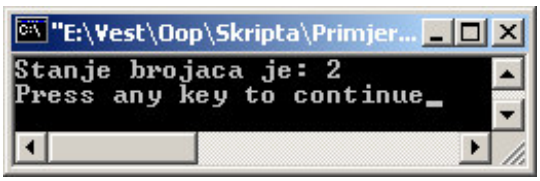
```
CBrojac brojac;
...
int stanjeBrojaca;

stanjeBrojaca = brojac;
```

Da bi se definiralo kako izvršiti konverziju iz objekta klase *CBrojac* u cjelobrojnu varijablu potrebno je kreirati konverzijski operator kao član klase. Konverzijski operator nema povratnog tipa, tj. ne specificira se povratna vrijednost, iako on vraća konvertiranu vrijednost.

```
class CBrojac
{
public:
...
operator int();
...
};

CBrojac::operator int()
{
   return  iCounts;
}
```

Primjer:

```
class CBrojac
{
public:
        CBrojac(int count=0):iCounts(count){};
        int DajStanje(){ return iCounts; }
        int operator++(int){ return iCounts++; }
        int operator++(){ return ++iCounts; }
        operator int(){return iCounts;}
private:
        int iCounts;
};

int main()
{
        CBrojac brojac;

        brojac++;
        brojac++;

        int stanjeBrojaca = brojac;

        cout << "Stanje brojaca je: " << stanjeBrojaca << endl;

        return 0;
}
```



9. Polja

9.1. Osnovno o poljima

Polje je skup podataka koje sadrže isti tip podatka.

Deklaracija polja vrši se navođenjem tipa polja, imena i broja elemenata unutar zagrada:

```
int intPolje[25];
```

U primjeru je kreirano polje od 25 elemenata tipa *int.*

Pristup elementu polja, odnosno memorijskoj lokaciji na kojoj je smješten podatak vrši se navođenjem imena polja, te udaljenosti elementa polja od početka polja unutar zagrada.

```
intPolje[10] = 15;
```

Broj koji se navodi u zagradama još se zove i indeks polja. Indeks polja počinje od nule i može ići do n-1 elementa. Npr. ako je kreirano polje *intPolje* od 25 elemenata, prvi element polja je

```
intPolje[0]
```

dok je posljednji element polja

```
intPolje[24];
```

Što se događa ako se pokuša pridružiti vrijednost elementu polja sa indeksom 25? Iako intPolje[25] nije element polja, vrijednost se uredno zapisuje, međutim na tom mjestu može biti pohranjena vrijednost neke druge varijable koja nema veze sa poljem. To može uzrokovati nepravilan rad programa.

Pristup npr. nepostojećem elementu

```
intPolje[150]
```

uredno prolazi prilikom prevođenja jer u C/C++ jezik nije ugrađen automatski zaštitni mehanizam koji to omogućava, tako da je na programeru da pazi kojim elementima pristupa. Zaštita od ovakvih situacija je moguća korištenjem *iznimki (engl. exceptions)* opisanih pri kraju ove skripte.

Ime polja bez indeksa predstavlja *konstantni pokazivač* na prvi element polja.

Inicijalizacija polja vrši se na sljedeći način:

Iza znaka jednakosti iza deklaracije polja navode se inicijalne vrijednosti elemenata polja u vitičastim zagradama. Elementi u zagradama odvojeni su zarezom.

Ako se inicijalizira polje od 25 elemenata, navede se 25 cjelobrojnih vrijednosti. Ako je potrebno inicijalizirati manji broj vrijednosti od veličine polja, mogu se navesti samo prvih nekoliko vrijednosti. Prevoditelj inicijalizira polje od 25 elemenata, te inicijalizira počevši od indeksa 0, onoliko elemenata koliko ih je navedeno u inicijalizacijskoj listi.

```
int intPolje[25] = { 0, 1, 2, 3, 4, 5 };
```

Kada se navede tip i ime polja, bez navođenja broja elemenata, te inicijalizacijska lista, na osnovu broja elemenata koje treba inicijalizirati na određene vrijednosti, prevoditelj automatski kreira polje od ukupnog broja elemenata koliko ih je navedeno u inicijalizacijskoj listi.

```
int intPolje[] = { 0, 1, 2, 3, 4, 5};
```

Moguće je kreirati dvodimenzionalno polje, kao u sljedećem primjeru. Popunjavanje polja vrši se da se prvo popuni desni indeks prije nego se poveća lijevi: [0][0] [0][1] [1][0] [1][1], itd.

```
int intPolje[3][2] = { 1, 2, 3, 4, 5, 6 };
```

Popunjavanje dvodimenzionalnih (ili višedimenzionalnih) polja ispravan je na prethodni način, međutim nije pregledan. Slijedeći primjer je inicijalizacija polja od 3 reda i dvije kolone, s tim da je svaki red izdvojen u posebne zagrade, pa time i preglednij.

```
int intPolje[3][2] = {
{1, 2},
{3, 4},
{5, 6}
        };
```

9.2. Polja i objekti

Kao i ugrađeni tipovi podataka i objekti se mogu spremiti u polja. Pristup pojedinom objektu kao elementu polja vrši se preko indeksnog operatora ([]). Pristup pojedinim članovima objekta vrši se preko točka operatora ('.').

Kao primjer može se uzeti polje objekata klase *CZrakoplov,* koja u sebi sadrži podatke o brzini, visini, smjeru i koordinatama zrakoplova u zračnom prostoru. Kreiranje polja objekata vrši se na isti način kao i polja ugrađenih tipova podataka. Prvo se navodi tip, zatim ime polja, te po potrebi inicijalizacija članova polja. Prilikom inicijalizacije navodi se konstruktor sa parametrima za one članove koji imaju inicijalne vrijednosti objekta.
Članovi polja koji se ne trebaju inicijalizirati ne navode se. Prva dva objekta u polju u donjem primjeru su inicijalizirana, dok sljedeća tri nisu.

Ako u inicijalizacijskoj listi nije navedena inicijalizacija svih objekata, nenavedeni objekti inicijaliziraju se sa *default* konstruktorom. Prema tome *default* konstruktor mora biti kreiran unutar klase. U protivnom prevoditelj će javiti grešku prilikom prevođenja.

```
CZrakoplov zrakoplov[5] = { CZrakoplov(500, 100, 100, 1000, 1000),
CZrakoplov (300, 120, 100, 1500, 1000)};
```

Pristup pojedinim elementima polja objekata vrši se navođenjem indeksa polja, te navođenjem člana objekta:

```
cout << zrakoplov[1].DajVisinu();
```

U primjeru se ispisuje visina drugog zrakoplova u polju.

9.3. Polja pokazivača

Objekte možemo kreirati dinamički u tzv. *"free store"* memoriji pomoću *new* operatora. Kao povratnu vrijednost operator *new* vraća pokazivač na alociranu memoriju.

```
CZrakoplov *pZrakoplov = new CZrakoplov ( 1000, 100, 120, 1000, 1000 );
```

Da bi se imala kontrola nad nekoliko objekata dinamički kreiranih u memoriji može se kreirati polje pokazivača na objekte određene klase.

Može se kreirati primjer 5 fiktivnih zrakoplova sa istim brzinama, visinama i smjerom, ali na različitim pozicijama.

```
CZrakoplov *pZrakoplov[5];

for ( int i=0;i<5;i++)
{
   pZrakoplov[i] = new CZrakoplov (300, 120, 180, 300*i, 300*i);
}
```

9.4. Pokazivač na polje

Moguće je kreirati čitavo polje objekata u *"free store"* memoriji, te izvući pokazivač na polje objekata.

```
// kreiranje 5 objekata CZrakoplov sa default konstruktorom
CZrakoplov pZrak = new CZrakoplov[5];

//Dio koda za inicijalizaciju polja od 5 zrakoplova
pZrak[0] = CZrakoplov(100, 200, 120, 1000, 1000);
...

CZrakoplov *pZrakoplov = pZrak;

cout << (*pZrakoplov).DajVisinu();
pZrakoplov++;
cout << (*pZrakoplov).DajVisinu();
```

pZrak sadrži adresu polja objekata u *free store* memoriji. Pokazivačem *pZrakoplov* se inicijalizira na vrijednost pZrak, tj. na prvi element polja objekata.

pZrakoplov je pokazivač na jedan element polja tipa CZrakoplov.

Kod pokazivača na objekt vrijedi pokazivačka aritmetika (inkrement i dekrement operator).
Ako pZrakoplov pokazuje na prvi element polja objekata tada inkrementiranjem

```
pZrakoplov++;
```

pokazivač sada pokazuje na drugi element polja. To znači da pokazivač sada pokazuje na memorijsku adresu uvećanu za veličinu objekta klase CZrakoplov.

Uspoređivanje pokazivača na polje i polja pokazivača

```
CZrakoplov zrakoplov[5];
CZrakoplov *pZrakoplov_1[5];
CZrakoplov *pZrakoplov_2  = new CZrakoplov[5];
```

Polje objekata *zrakoplov*, ne navodeći indeks je konstantni pokazivač na prvi član polja.

pZrakoplov_1 predstavlja polje pokazivača na objekte tipa CZrakoplov. *pZrakoplov_1* bez navođenja polja je pokazivač na prvi element polja. Pošto su elementi polja pokazivači na objekt, to znači da je *pZrakoplov_1* pokazivač na pokazivač na objekt klase *CZrakoplov.*

Pokazivač *pZrakoplov_2* predstavlja pokazivač na prvi element polja objekata dinamički alociranih u memoriji.

Po prethodnom opisu vidi se da *pZrakoplov_2* i *zrakoplov* imaju isto značenje. Oboje su pokazivači na prvi element polja, s tom razlikom da je jedan od njih pokazivač a drugi konstantni pokazivač na prvi element polja. To što je jedno polje statički (na *stack-u*) a drugo dinamički alocirano operatorom *new* ne igra nikakvu ulogu prilikom pristupanja pojedinom objektu kao elementu polja.

9.5. Brisanje polja u free store memoriji

Memorija, dinamički alocirana operatorom *new,* briše se operatorom *delete*. Međutim kada je potrebno osloboditi polje elemenata, bilo da je riječ o ugrađenim tipovima podataka, bilo da je riječ o korisnički definiranim tipovima ili klasama, potrebno je naznačiti da je riječ o polju elemenata.

```
CZrakoplov *pZrakoplov = newCZrakoplov[5];
int i;
CZrakoplov *pZrak;

for ( i=0;i<5;i++)
{
```

```
pZrak = new CZrakoplov;
...
pZrakoplov[i] = *pZrak;
delete pZrak;
}
...
delete [] pZrakoplov;
```

"[]" operator kaže prevoditelju da se polje elemenata treba brisati. Ako se navedeni operator ne stavi izbrisat će se samo prvi element polja te se stvara curenje memorije *("memory leak").*

10. Nasljeđivanje

Nasljeđivanje omogućava kreiranje klasa s postupnim uobličavanjem.

10.1. Hijerarhija nasljeđivanja

Hijerarhija nasljeđivanja uspostavlja tzv. "is-a relationship". Kao primjer može se uzeti:

- Toyota je vrsta automobila
- pas je vrsta sisavca
- airbus_319 je vrsta zrakoplova

Nova klasa koja nasljeđuje već postojeću klasu, automatski dobije sve karakteristike klase koju je naslijedila.

Za novu klasu izvedenu iz već postojeće klase kaže se da je izvedena klasa.

Za klasu iz koje se izvodi nova klasa kaže se da je osnovna klasa.

Deklaracija izvedene (naslijeđene) klase vrši se tako da se iza imena izvedene klase navedu imena klasa koje ona nasljeđuje, te da se navede vrsta nasljeđivanja.

Primjer:

Kao primjer može se uzeti program za radarsku kontrolu zračnog prometa. U zračnoj luci radar prati kretanje zrakoplova u zračnom prostoru. Za svaki objekt u zračnom prostoru prati se nekoliko veličina: brzina, smjer, visina, X i Y koordinate. Pretpostavka je da je radar u centru kartezijevog koordinatnog sustava. Podaci se prenose iz radarskog uređaja u PC računalo, te se kretanje prikazuje monitoru.

```
class CZrakoplov
{
public:
        CZrakoplov( double visina, double brzina, double smjer, double xKoord, double
yKoord );

        double DajVisinu(){return dVisina;}
        PostaviVisinu(double visina) {dVisina = visina;}
        double DajBrzinu () {return dBrzina;}
        PostaviBrzinu (double brzina){dBrzina = brzina;}
        double DajSmjer(){return dSmjer;}
        PostaviSmjer(double smjer){dSmjer = smjer;}
        double DajXKoord(){return dXKoord;}
        PostaviXKoord(double xKoord){dXKoord = xKoord;}
        double DajYKoord(){return dYKoord;}
        PostaviYKoord(double yKoord){dYKoord = yKoord;}
protected:
        double dVisina;
        double dBrzina;
        double dSmjer;
        double dXKoord;
        double dYKoord;
```

```
}

CZrakoplov::   CZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord ) :
dVisina(visina),
dBrzina(brzina),
dSmjer(smjer),
dXKoord(xKoord),
dYKoord(yKoord)
{
}
```

Klasa CZrakoplov predstavlja osnovnu klasu općeg zrakoplova. Iz nje se dalje izvode klase za svaki specifični zrakoplov. Npr. u zračnom prostoru mogu se naći razne vrste zrakoplova: putnički, teretni, vojni (lovac), mali laki sportski zrakoplov itd.

Iz osnovne klase izvodi se klasa koja predstavlja putnički zrakoplov (CPutnickiZrakoplov):

```
typedef enum
{
        ZRAK_PUTNICKI,
                ZRAK_VOJNI_LOVAC,
                ZRAK_TERETNI,
                ZRAK_LAKI_SPORTSKI
                ZRAK_MAX_ZRAK
}
enum_tip_zrakoplova;

class CPutnickiZrakoplov : public CZrakoplov
{
public:
CPutnickiZrakoplov( double visina, double brzina, double smjer, double xKoord, double
yKoord );
        enum_tip_zrakoplova DajTipZrakoplova(){return eTipZrakoplova;}
private:
const enum_tip_zrakoplova eTipZrakoplova;
};

CPutnickiZrakoplov::CPutnickiZrakoplov (double visina, double brzina, double smjer,
double xKoord, double yKoord):
CZrakoplov (double visina,
double brzina,
double smjer,
double xKoord,
double yKoord),
eTipZrakoplova = ZRAK_PUTNICKI
{
}
```

Klasa CPutnickiZrakoplov nasljeđuje osnovnu klasu CZrakoplov i time automatski poprima sve njene atribute i metode (podatkovne i funkcijske članove).

Prilikom specificiranja osnovne klase navodi se i način nasljeđivanja (u ovom primjeru je tipa *public*). Postoje još dva načina nasljeđivanja, koji će biti opisani kasnije.

U naslijeđenoj klasi podatkovni i funkcijski članovi definirani sa *public* tipom pristupa vidljivi su izvan izvedene klase (ostaju public).

Zbog nasljeđivanja uvodi se *protected* pravo pristupa. Članovi osnovne klase definirani s ovim tipom pristupa vidljivi su u izvedenoj klasi samo iz funkcijskih članova klase, te nisu vidljivi izvan klase.

Članovi osnovne klase definirani s *private* tipom pristupa nisu vidljivi u izvedenoj klasi, te im se iz izvedene klase ne može pristupiti direktno, već jedino preko pristupnih funkcija osnovne klase (defniranih s *public* ili protected pravom pristupa).

Primjer kreiranja objekta izvedene klase:

```
CPutnickiZrakoplov airbus_319(100, 200, 300, 1000, 2000);

cout << airbus_319.DajTipZrakoplova();

airbus_319.PostaviVisinu(500);
cout << airbus_319.DajBrzinu();
```

10.2. Konstruktor i destruktor

Prilikom kreiranja objekta osnovne klase na primjer *CZrakoplov*, prvo se poziva konstruktor *CZrakoplov()* s parametrima.

Prilikom kreiranja objekta izvedene klase program poziva konstruktor izvedene klase. U njemu se prvo poziva konstruktor osnovne klase, te kad on završi nastavlja se izvođenje koda unutar konstruktora izvedene klase.

To je i logično, pošto je klasa iz koje se kreira objekt kreirana postupnim nadograđivanjem. Inicijalizacija se radi od dna prema gore, na primjer, ako se u izvedenoj klasi koristi memorija alocirana u osnovnoj klasi.

Program u stvari prvo dolazi na početak konstruktora izvedene klase, ali ne ulazi u tijelo konstruktora, tj. ne starta izvršavanje inicijalizacijskog dijela i tijela konstruktora, već odmah starta konstruktor osnovne klase.

Obrnuta je situacija prilikom uništavanja objekta izvedene klase. Prvo se poziva destruktor izvedene klase, pa tek nakon toga se izvršava destruktor osnovne klase.

10.2.1. Prosljeđivanje argumenata konstruktoru osnovne klase

Ukoliko se ne navede unutar konstruktora izvedene klase konstruktor osnovne klase, automatski se prilikom inicijalizacije objekta izvedene klase poziva default konstruktor osnovne klase (bez ulaznih parametara).

Ako je potrebno da se prilikom inicijalizacije objekta osnovne klase trebaju proslijediti neki parametri koristi se preopterećeni konstruktor osnovne klase. On se eksplicitno sa parametrima navodi u inicijalizacijskom dijelu izvedene klase.

Primjer konstruktora:

```
CPutnickiZrakoplov( double visina, double brzina, double smjer, double x, double y ):
CZrakoplov ( visina, brzina, smjer, x, y ),
eTipZrakolova (ZRAK_PUTNICKI)
{
}
```

10.3. Overriding funkcija osnove klase

Overriding, odnosno nadjačavanje znači promjenu implementacije (tijela) funkcije osnovne klase u izvedenoj klasi.

Kreiranjem funkcijskog člana u izvedenoj klasi koji ima isti potpis u osnovnoj klasi (isto ime, ulazne parametre i povratni tip) skriva se funkcijski član osnovne klase.

Kada se kreira objekt izvedene klase, te pozove nadjačani *(engl. "override")* funkcijski član izvršava se kod definiran u izvedenoj klasi.

Primjer:

U osnovnoj klasi *CZrakoplov* postoji funkcijski član DajVisinu(), koji vraća visinu na kojoj se nalazi zrakoplov u metrima. U izvedenoj klasi postoji funkcijski član istog imena i potpisa, koji vrijednost visine prije nego je vrati konvertira u stope (1 feet = 0.30 m).

Kreiranjem objekta iz osnovne klase i pozivom funkcijskog člana DajVisinu() kao povratna vrijednost dobije se visina zrakoplova u metrima.

```
class CZrakoplov
{
public:
double DajVisinu() { return dVisina;}
...
};

class CPutnickiZrakoplov : public CZrakoplov
{
public:
       double DajVisinu() { return dVisina/0.30;}
...
};

CZrakoplov zrakoplov1(100, 500, 210, 1000, 1000);
cout << "Visina leta zrakoplova = " << zrakoplov1.DajVisinu() << endl;

CPutnickiZrakoplov airbus_319 (100, 500, 210, 1000, 1999);
cout << "Visina leta airbusa 319 croair = " << airbus_319.DajVisinu() << endl;

Vvisina leta zrakoplova = 500.000000
Visina leta airbusa 319 croair = 1666.666667
```

Kreiranjem objekta izvedene klase *CPutnickiZrakoplov* i pozivom istoimenog funkcijskog člana poziva se funkcijski član izvedene klase te se vraća vrijednost visine u stopama na kojoj se zrakoplov nalazi.

Da bi se pristupilo funkcijskom članu DajVisinu klase CZrakoplov potrebno je navesti potpuni put do funkcijskog člana osnovne klase (ime klase i funkcijskog člana):

```
double visinu_u_metrima = airbus_319.CZrakoplov::DajVisinu();
```

10.4. Skrivanje funkcija osnovne klase

Nadjačavanjem funkcijskog člana osnovne klase u izvedenoj klasi, svi preopterećeni funkcijski članovi osnovne klase više nisu dostupni u izvedenoj klasi direktno.
Da bi im se pristupilo potrebno je navesti potpuno ime funkcijskog člana navodeći ime klase.
Naravno moguće je kreirati funkcijski član istog potpisa u izvedenoj klasi koji poziva funkcijski član osnovne klase.

10.5. Virtualne funkcije

C++ jezik podržava polimorfizam (osobina tijela da poprimi više oblika) na način da se pokazivaču tipa osnovne klase može pridružiti objekt izvedene klase.

Primjer:

```
CZrakoplov pZrakoplov* = new CPutnickiZrakoplov;
```

U primjeru se operatorom *new* kreira na heap-u objekt izvedene klase *CPutnickiZrakoplov*, te operator kao rezultat kreiranja vraća pokazivač na kreirani objekt čija se vrijednost pridružuje pokazivaču *pZrakoplov* tipa osnovne klase *CZrakoplov.*

To je ispravno, jer putnički zrakoplov je tip zrakoplova.

Preko pokazivača tipa osnovne klase na objekt mogu se pozivati funkcijski članovi objekta izvedene klase.
Međutim, ako se u izvedenoj klasi nalaze nadjačane funkcije osnovne klase, potrebno je osigurati da se prave funkcije pozivaju preko pokazivača osnovne klase.
To je omogućeno korištenjem *virtualnih funkcija.*

Primjer:

```
class CZrakoplov
{
...
virtual int DajBrojPutnika()
{
cout << "opći tip zrakoplova nema podataka o putnicima" << endl;
return 0;
}
virtual void PostaviBrojPutnika( int brojPutnika )
{
cout << "općem tipu zrakoplova ne moze se postaviti podatak o putnicima" << endl;}
...
};

class CPutnickiZrakoplov : public CZrakoplov
{
...
int DajBrojPutnika(){ return iBrojPutnika; }
void PostaviBrojPutnika( int brojPutnika )
{
iBrojPutnika = brojPutnika;
cout << "broj putnika u putnickom zrakoplovu postavljen je na:" << brojPutnika <<
"putnika";
}
```

```
...
private:
iBrojPutnika;
...
};

...
CZrakoplov *pOpciZrakoplov = new CZrakoplov ( );
CZrakoplov *pPutnickiZrakoplov = new CPutnickiZrakoplov ( );

pOpciZrakoplov->PostaviBrojPutnika(15);
pPutnickiZrakoplov->PostaviBrojPutnika(15);

cout << "Broj putnika u općem zrakoplovu:" << pOpciZrakoplov->DajBrojPutnika();

cout << "Broj putnika u putničkom zrakoplovu: " << pPutnickiZrakoplov-
>DajBrojPutnika();

...
općem tipu zrakoplova ne moze se postaviti podatak o putnicima
broj putnika u putnickom zrakoplovu postavljen je na: 15 putnika;
Broj putnika u općem zrakoplovu: opći tip zrakoplova nema podataka o putnicima
Broj putnika u putničkom zrakoplovu: 15
```

Ako se neki funkcijski član klase proglasi virtualnim to znači da će se najvjerojatnije promijeniti njegova implementacija u nekoj od izvedenih klasa.

Funkcijski član se kreira kao virtualan tako da se ispred njegove deklaracije postavi ključna riječ *virtual.*
Ako u osnovnoj i izvedenoj klasi postoje funkcije istog imena i potpisa (npr. za manipuliranjem podacima o broju putnika *fn. DajBrojPutnika()*) s tim da nisu virtualne, te ako se funkcijskom članu pristupa preko pokazivača na objekt, ovisno o tipu pokazivača poziva se jedan od dva navedena funkcijska člana.

Ako pokazivač tipa osnovne klase pokazuje na objekt izvedene klase, pozivom funkcijskog člana preko pokazivača poziva se funkcijski član osnovne klase.

Naprotiv, ako je pokazivač tipa izvedene klase npr. CPutnickiZrakoplov i pokazuje na objekt izvedene klase, pozivom funkcijskog člana preko pokazivača poziva se funkcijski član izvedene klase.

Navodeći određenim funkcijskim članovima osnovne klase prefix *virtual*, način pozivanja funkcijskih članova se mijenja.

Rezultat poziva funkcijskog člana izvedene klase preko pokazivača tipa izvedene klase ostaje isti.

Međutim pozivom funkcijskog člana izvedene klase preko pokazivača tipa osnovne klase, ako funkcija osnovne klase ima prefix *virtual*, poziva se funkcija izvedene klase, naravno ako u izvedenoj klasi postoji funkcija istog potpisa.

U gornjem primjeru sa funkcijskim članovima za manipuliranje podacima o broju putnika u zrakoplovu vidi se da rezultat poziva funkcijskog člana nad konkretnim putničkim zrakoplovom daje podatak o broju putnika u zrakoplovu (poziva se funkcijski član izvedene klase, jer je funkcijski član osnovne klase bio kreiran kao *virtual*).

Drugi primjer korištenja virtualnih funkcija mogao bi se primijeniti prilikom iscrtavanja prozora (Windows OS). Dialog Box, Scroll Bar, Check Box, Buttons, itd. predstavljaju razne vrste prozora. Svi su oni izvedeni iz osnovne klase prozora (*CWindow).* Osnovna klasa ima virtualni funkcijski član *Draw()*, te svaka od izvedenih klasa ima svoju nadjačanu implementaciju funkcijskog člana *Draw(),* jer svaka vrsta navedenih prozora se iscrtava na drugačiji način.
Svakom od ovih prozora (objekata izvedenih klasa osnovne klase CWindow) može se pristupiti preko pokazivača tipa osnovne klase *CWindow*.
Preko pokazivača se poziva funkcijski član *Draw()*, koji je specifičan za svaku pojedinu vrstu prozora.

Primjer:

```
CWindow *pWindow[3];

pWindow[0] = new CDialog;
pWindow[1] = new CCheckBox;
pWindow[2] = new CButton;
...
for ( int i=0;i<3;i++)
 pWindow[i]->Draw();
```

Polimorfizam sa virtualnim funkcijama funkcionira na isti način sa pokazivačima i referencama, s tim da se funkcijskim članovima objekta ako se koristi referenca na objekt pristupa sa točka operatorom.

10.5.1. Prosljeđivanje objekta funkcijama

Polimorfizam prilikom prosljeđivanja objekta funkciji vrijedi samo ako se objekt prenosi preko pokazivača ili reference. Ako se objekt prenosi funkciji preko vrijednosti onemogućava se poziv virtualnih funkcija.

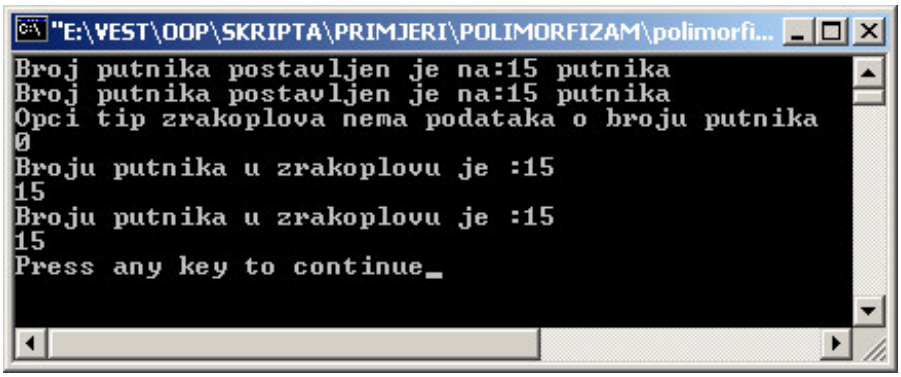
Primjer:

```
#include <iostream.h>

class CZrakoplov
{
public:
        CZrakoplov( double visina, double brzina, double smjer, double xKoord, double
yKoord );

        double DajVisinu(){return dVisina;}
        void PostaviVisinu(double visina) { dVisina = visina; }
        double DajBrzinu (void) {return dBrzina;}
        void PostaviBrzinu (double brzina){dBrzina = brzina;}
        double DajSmjer(){return dSmjer;}
        void PostaviSmjer(double smjer){dSmjer = smjer;}
        double DajXKoord(){return dXKoord;}
        void PostaviXKoord(double xKoord){dXKoord = xKoord;}
        double DajYKoord(){return dYKoord;}
        void PostaviYKoord(double yKoord){dYKoord = yKoord;}


        virtual void PostaviBrojPutnika(int brojPutnika)
        {
```

```
                cout << "Ne mogu postaviti podatak o broju putnika opcem zrakoplovu" <<
endl;
        }
        virtual int DajBrojPutnika()
        {
                cout << "Opci tip zrakoplova nema podataka o broju putnika" << endl;
return 0;
        }

private:
        double dVisina;
        double dBrzina;
        double dSmjer;
        double dXKoord;
        double dYKoord;
};

CZrakoplov::CZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord ):
        dVisina(visina),
        dBrzina(brzina),
        dSmjer(smjer),
        dXKoord(xKoord),
        dYKoord(yKoord)
{
}


typedef enum
{
        ZRAK_PUTNICKI,
                ZRAK_VOJNI_LOVAC,
                ZRAK_TERETNI,
                ZRAK_LAKI_SPORTSKI,
                ZRAK_MAX_ZRAK
}
enum_tip_zrakoplova;

class CPutnickiZrakoplov : public CZrakoplov
{
public:
        CPutnickiZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord );
        enum_tip_zrakoplova DajTipZrakoplova(){ return eTipZrakoplova; }
        void PostaviBrojPutnika(int brojPutnika)
        {
                iBrojPutnika = brojPutnika;
                cout << "Broj putnika postavljen je na:" << iBrojPutnika << " putnika"
<< endl;
        }
        int DajBrojPutnika()
        {
                cout << "Broju putnika u zrakoplovu je :" << iBrojPutnika << endl;
                return iBrojPutnika;
        }
private:
        const enum_tip_zrakoplova eTipZrakoplova;
        int iBrojPutnika;
};

CPutnickiZrakoplov::CPutnickiZrakoplov (double visina, double brzina, double smjer,
double xKoord, double yKoord):
CZrakoplov (visina, brzina, smjer, xKoord, yKoord),
                        eTipZrakoplova(ZRAK_PUTNICKI)
{
}

void VIspisiBrojPutnika( CZrakoplov zrakoplov );
void RIspisiBrojPutnika( CZrakoplov &rZrakoplov );
void PIspisiBrojPutnika( CZrakoplov *pZrakoplov );

int main()
{

        CZrakoplov *pZrakoplov = new CPutnickiZrakoplov(1000, 500, 120, 1000, 1000);
```

```
        CZrakoplov &rZrakoplov = *pZrakoplov;


        pZrakoplov->PostaviBrojPutnika (15);
        rZrakoplov.PostaviBrojPutnika (15);

        VIspisiBrojPutnika ( *pZrakoplov );
        RIspisiBrojPutnika ( rZrakoplov );
        PIspisiBrojPutnika ( pZrakoplov );

        return 0;
}

void VIspisiBrojPutnika( CZrakoplov zrakoplov )
{
        cout << zrakoplov.DajBrojPutnika() << endl;
}

void RIspisiBrojPutnika( CZrakoplov &rZrakoplov )
{
        cout << rZrakoplov.DajBrojPutnika () << endl;
}

void PIspisiBrojPutnika( CZrakoplov *pZrakoplov )
{
        cout << pZrakoplov->DajBrojPutnika() << endl;
}
```

Ispis je sljedeći:



U programu je na heap-u kreiran objekt klase *CPutnickiZrakoplov*, te je adresa objekta u memoriji pridijeljena pokazivaču tipa osnovnog objekta imena *pZrakoplov*. Također kreirana je referenca na taj isti objekt imena rZrakoplov.

Pozivom funkcijskog člana za postavljanje broja putnika i preko pokazivača i reference izvršava se kod u izvedenoj klasi, jer je funkcijski član osnovne kreiran sa prefixom *virtual.*
Da funkcijski član osnovne klase nije kreiran kao *virtual,* pristupom preko reference ili pokazivača tipa osnovne klase na objekt izvedene klase pozvao bi se funkcijski član osnovne klase (vratila bi se poruka da se ne može postaviti podatak o broju putnika, jer taj podatkovni član ne postoji u osnovnoj klasi).

U programu se pozivaju tri funkcije kojima se prosljeđuje objekt. Sve tri funkcije primaju podatak koji je tipa osnovne klase, s tim da jedna funkcija prima objekt po vrijednosti, druga po referenci i treća preko pokazivača.

Prilikom prenošenja objekta po vrijednosti onemogućava se poziv virtualnih funkcija, tako da iako je proslijeđen objekt klase *CPutnickiZrakoplov*, poziva se funkcijski član klase *CZrakoplov*, te ispisuje da ne može postaviti podatak o broju putnika (podatak koji funkcija prima je tipa osnovne klase).

Za razliku od prosljeđivanja objekta po vrijednosti prosljeđivanjem izvedene klase preko pokazivača ili reference pozivaju se funkcijski članovi izvedene klase, jer je funkcijski član istog potpisa osnovne klase kreiran kao *virtual.*

10.5.2. Virtualni destruktor

Moguće je proslijediti pokazivač na objekt izvedene klase tamo gdje se očekuje objekt osnovne klase.

Ako je potrebno taj proslijeđeni objekt uništiti, poziva se destruktor. Da bi se pozvao pravi destruktor potrebno je destruktor osnovne klase kreirati kao *virtual*.

Prilikom uništavanja objekta izvedene klase koji je proslijeđen preko pokazivača osnovne klase, ako objekt osnovne klase ima definiran destruktor kao *virtual,* poziva se destruktor izvedene klase, koji uredno uništava objekt.

Primjer:

```
class CZrakoplov
{
...
virtual ~CZrakoplov();
...
};

class CPutnickiZrakoplov : public CZrakoplov
{
...
~CPutnickiZrakoplov();
...
};

void UnistiObjekt ( CZrakoplov *pZrakoplov )
{
     delete pZrakoplov;
}

int main()
{
CZrakoplov *pZrakoplov = new CPutnickiZrakoplov (1000, 120, 270, 1000, 1000);
...
PUnistiObjekt ( pZrakoplov );
...
return 0;
}
```

Pozivom operatora *delete* uredno se poziva destruktor izvedene klase (*~CPutnickiZrakoplov()*) iako je pokazivač tipa osnovne klase *CZrakoplov.*

Napomena:

Ako se bar jedna funkcija osnovne klase definira kao *virtual*, preporuka je definirati i destruktor kao *virtual.*

Za razliku od destruktora, konstruktor se ne može biti kreirati kao *virtual.*

11. Polimorfizam

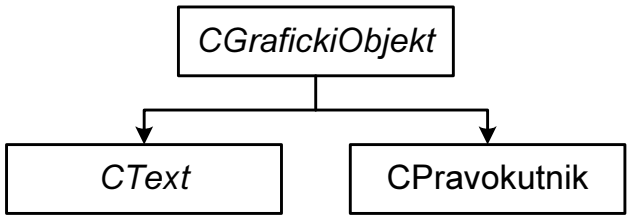
Podatke o poziciji zrakoplova u zračnom prostoru treba prikazati na monitoru računala (simulacija radarskog ekrana). Prikaz podataka na ekranu treba odgovarati stvarnom stanju u zračnom prostoru. Podaci o svakom zrakoplovu sadrže lokaciju zrakoplova, visinu, brzinu, tip, te su mogući još i dodatni parametri, kao npr. broj leta za putničke zrakoplove ili ID zrakoplova (broj na repu), za male privatne ili sportske zrakoplove. Svi ti podaci trebaju biti prikazani na ekranu.

Za početak neka podaci budu prikazani kao jedan uokvireni red teksta. Naravno poslije su moguće varijacije.

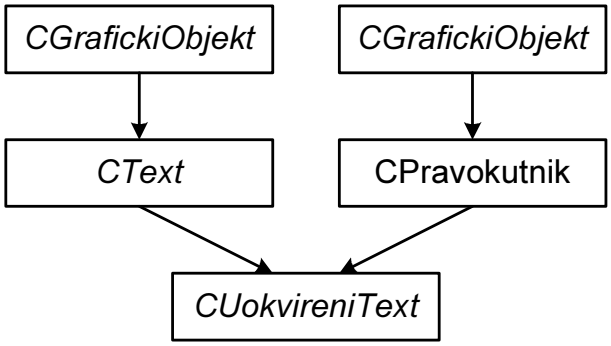
Primjer:

Iscrtavanje uokvirenog teksta može se promatrati kao skup od dva grafička objekta: pravokutnika koji uokviruje tekst i samog teksta. Oba grafička objekta imaju neke zajedničke karakteristike, a to su: boja, početne koordinate. Također, prilikom iscrtavanja ova dva grafička objekta mogu se za svaki objekt definirati funkcije *Crtaj()* koje se pozivaju da bi se grafički objekt iscrtao.

Prema tome može se definirati osnovna klasa koja sadrži zajedničke podatkovne i funkcijske članove koju će ova dva grafička objekta naslijediti. Klasu se može nazvati *CGrafickiObjekt*.
Iz osnovne klase *CGrafickiObjekt* dalje se izvode dvije nove klase *CText* i *CPravokutnik*.



Klasa *CUokvireniTekst* nasljeđuje dvije klase *CText* i *CPravokutnik*. Obje ove klase naslijeđene su iz iste klase *CGrafickiObjekt*. Prema tome postoje dvije zasebne osnovne klase *CGrafički objekt*.



11.1. Problem jednostrukog nasljeđivanja

Kreiranjem klase *CUokvireniText* nasljeđuju se svi podatkovni i funkcijski članovi osnovnih klasa. Pošto osnovne klase (u ovom slučaju CText i CPravokutnik) imaju iste svoje osnovne klase, to znači da postoje dvije kopije podatkovnih i funkcijskih članova klase *CGrafickiObjekt*.

Pristup preko objekta klase *CUokvireniText* npr. funkcijskom članu *PostaviBoju()* predstavlja problem prevoditelju, jer ne zna kojem funkcijskom članu da pristupi, da li naslijeđenom od klase *CText* ili klase *CPravokutnik.* Zbog toga pristup funkcijskom članu na ovaj način generira grešku prilikom prevođenja.

Na primjer:

```
CUokvireniText *pUokvireniText = new CUokvireniText ( "FWW1002, 120, 230", 400, 200,
100, 100 );

cout << pUokvireniText->DajBoju();
```

generira grešku.

Da bi se pristupilo nekom od funkcijskih članova potrebno je potpuno navesti put do njega:

```
pUokvireniText->CText::DajBoju();
```

S ovim je jednoznačno određen pristup do određenog funkcijskog člana.

Direktan pristup funkcijskom članu moguć je i da se u klasi *CUokvireniText* kreira nadjačana funkcija *DajBoju(),* koja jednoznačno poziva funkcijski član u jednoj od klasa *CGrafickiObjekt*.

Na primjer.

```
class CUokvireniText
{
...
int DajBoju () {return CText::iBoja;}
```

```
...
};
```

To međutim znači da je u naslijeđenoj klasi potrebno kreirati nadjačane funkcijske članove za svaku naslijeđenu funkciju. To predstavlja nepotreban posao.

11.2. Virtualno nasljeđivanje

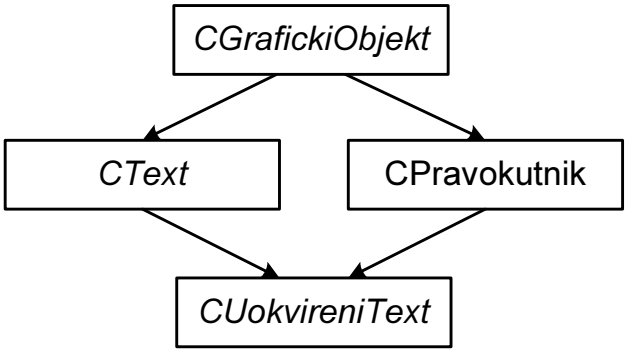
Nasljeđivanje dviju klasa koje imaju istu osnovnu klasu generira grešku prilikom prevođenja ako je potrebno pristupiti nekom od funkcijskih članova osnovne klase. Osim toga pitanje je koja je od osnovnih klasa ona prava, čiji su podatkovni i funkcijski članovi oni pravi.

C++ prevoditelju se može reći da se ne želi imati dvije osnovne klase (u ovom slučaju *CGrafickiObjekt*) nego se želi samo jedna osnovna klasa.

To se može postići tako da prilikom kreiranja klasa *CText* i *CPravokoutnik* klasa *CGrafickiObjekt* nasljedi kao *virtual.*

Na taj način klase *CText* i *CPravokutnik* dijele jednu zajedničku osnovnu klasu *CGrafickiObjekt*, za razliku od prethodnog slučaja gdje je svaka naslijeđena klasa imala svoju kopiju objekta klase *CGrafickiObjekt.*

Na slici se vidi kako izgledaju klase koje nasljeđuju osnovnu klasu CGrafickiObjekt kao *virtual*:



Primjer kreiranja klase *CUokvireniText* koristeći virtualno naslijeđenu osnovnu klasu *CGrafickiObjekt*:

```
class CGrafickiObjekt
{
...
};

class CText : virtual public CGrafickiObjekt
{
};
```

```
class CPravokutnik : virtual public CGrafickiObjekt
{
};

class CUokvireniText : public CPravokutnik, public CText
{
public:
        CUokvireniText ( char *text, int x, int y, int xSize, int ySize ):
          CText( text, x, y ),
                  CPravokutnik ( x, y, xSize, ySize ),
          CGrafickiObjekt(x, y){}

          virtual void Draw (){ cout << "Iscrtavanje uokvirenog texta" << endl;}
private:
};
```

Pošto konačna izvedena klasa ima samo jednu klasu *CGrafickiObjekt* pristup pojedinim funkcijskim i podatkovnim članovima te klase jednoznačno je određen. Prema tome nije potrebno navoditi potpuni put za pristup elementima klase iako je i to ispravno.

Redoslijed ključnih riječi *virtual* i *public* nije bitan.

Prilikom inicijalizacije klase *CUokvireniText* tj. poziva konstruktora u inicijalizacijskom dijelu se pozivaju konstruktori dviju osnovnih klasa. Po pravilu svaka klasa prilikom inicijalizacije može pozvati samo inicijalizaciju svojih podklasa. U ovom slučaju u inicijalizacijskom dijelu klase *CUokvireniText* vrši se inicijalizacija dviju osnovnih klasa *CPravokutnik* i *CText.*

Korištenjem virtualnog nasljeđivanja odstupa se od ovog pravila, te se u inicijalizacijskom dijelu finalne klase navodi i konstruktor zajedničke osnovne klase *CGrafickiObjekt*, kao što je navedeno u gornjem primjeru.

Inicijalizacija osnovne klase se može nalaziti i u konstruktorima klasa CText i CPravokutnik. To znači da bi se inicijalizacija osnovne klase CGrafickiObjekt mogla izvršiti tri puta. Da se to ne bi dogodilo, u C++ jezik uvedeno je pravilo da se virtualna klasa inicijalizira konstruktorom čiji je poziv naveden najdalje izvedenoj klasi. U ovom slučaju to je konstruktor naveden u inicijalizacijskom dijelu konstruktora izvedene klase *CUokvireniText.*

11.3. Apstraktni tipovi podataka

Prilikom kreiranja objekata često se radi hijerarhija objekata. Na primjer. u slučaju ispisa podataka na ekranu kreirane su klase *CText*, *CPravokutnik*, *CUokvireniText*, te se mogu kreirati i druge klase kao npr. *CLinija*, *CTocka* itd.

Sve navedene klase izvedene su direktno ili preko neke sljedeće izvedene klase od jedne osnovne klase, *CGrafickiObjekt*.

Klasa *CGrafickiObjekt* sadrži osnovne funkcijske i podatkovne članove za pozicioniranje objekta, te njegovu boju. Također ona sadrži i virtualne funkcije koje u

samoj klasi *CGrafickiObjekt* nemaju implementacije, jer nije definiran sam tip grafičkog objekta, već će se konkretizirati u izvedenim klasama.
Primjer takvog virtualnog funkcijskog člana je *Draw()* koji služi za iscrtavanje lika na ekranu, ali za svaku vrstu lika bilo da je riječ o tekstu ili iscrtavanju pravokutnika potreban je drugačiji algoritam.

Prema tome virtualni funkcijski član *Draw()* klase *CGrafickiObjekt* nema nikakvu drugu svrhu osim što postoji samo da bi se nadjačao u nekoj od izvedenih klasa.

Također nema smisla kreirati objekt klase *CGrafickiObjekt*, koji postoji da bi se naslijedio nekom od izvedenih klasa te kreirao konkretni grafički objekt (linija, pravokutnik ili red teksa).

11.4. Čiste virtualne funkcije

Apstraktni tip podataka *(engl. "abstract data type" ili ADT)* uvijek predstavlja baznu, tj. osnovnu klasu drugim klasama. Nije ispravno kreirati instancu ADT-a.

C++ jezik podržava kreiranje čistih virtualnih funkcija. Čista virtualna funkcija se kreira tako se izjednači sa nulom:

```
virtual void Draw() = 0;
```

Svaka klasa sa jednom ili više čistih virtualnih funkcija je apstraktni tip podatka (ADT). Iz ADT klasa nije dozvoljeno kreiranje instanci. Pokušaj kreiranja javlja grešku prilikom prevođenja.

Sve čiste virtualne funkcije potrebno je u naslijeđenim klasama nadjačati (override-ati) da bi se program uspješno preveo. To je i logično jer je čiste virtualne funkcije moguće kreirati bez implementacije funkcije. Kada se nadjača, potrebno je kreirati implementaciju.

Primjer:

```
class CGrafickiObjekt
{
public:
...
        virtual void Draw() = 0;
protected:
...
};

class CText : public CGrafickiObjekt
{
public:
        CText( char *text, int iX, int iY ):CGrafickiObjekt (iX, iY){ strcpy ( szText,
text); }
        void PostaviText ( char *text) { strcpy ( szText, text);    }
        char *DajText ( char *text) { strcpy ( text, szText ); return text; }
        virtual void Draw(){ cout << szText << endl;};
private:
        char szText[20];
};
```

U primjeru je u osnovnoj klasi kreirana čista virtualna funkcija *Draw()*. Kreirajući je kao čistu virtualnu funkciju, onemogućeno je kreiranje instance klase *CGrafickiObjekt*.

U klasi *CText* nadjačana je čista virtualna funkcija *Draw()*, te je iz klase *CText* moguće kreirati instancu.

12. Specijalne klase i funkcije

12.1. Statički podatkovni članovi

Podatkovni članovi objekta kreiranog iz neke klase jedinstveni su za taj objekt. Taj podatkovni član se ne dijeli između drugih instanci iste klase.

Ponekad je potrebno da neki od podatkovnih članova klase bude dostupan (da se dijeli među) svim instancama određene klase. Npr. za klasu *CPutnickiZrakoplov* može se pratiti ukupni broj kreiranih objekata, tj. koliko putničkih zrakoplova se nalazi u zračnom prostoru zračne luke.

Taj je podatak moguće spremiti u globalnu varijablu, ali ako se ne želi da podatak bude dostupan ostatku programa već samo instancama klase, tip podatka se kreira tipa *static* podatkovni član klase.

Primjer:

```
class CPutnickiZrakoplov : public CZrakoplov
{
public:
        CPutnickiZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord );
        ~CPutnickiZrakoplov();
...
        static int DajBrojPutnickihZrakoplova() { return iBrojPutnickihZrakoplova; }
private:
        static int iBrojPutnickihZrakoplova;
...
};
```

Da bi se podatkovni član definirao kao *static* potrebno je ispred tipa podatka staviti ključnu riječ *static.*

Inicijalizacija *static* podatkovnog člana vrši se izvan objekata klase.

Primjer inicijalizacije *static* podatkovnog člana:

```
int CPutnickiZrakoplov::iBrojPutnickihZrakoplova = 0;

main()
{
   CPutnickiZrakoplov *pZrakoplov = new CPutnickiZrakoplov (1000, 300, 120, 1000, 1000
);
...
}
```

12.2. Statički funkcijski članovi

Statički funkcijski članovi ne egzistiraju u području objekta već u području klase. Mogu se pozivati a da niti jedan objekt dotične klase nije definiran.

Statički funkcijski članovi kreiraju se tako da se ispred povratnog tipa funkcije navede ključna riječ *static*.

Poziv statičkog funkcijskog člana može se obaviti preko imena objekta ili preko imena klase.

Primjer:

Ako se kreira objekt klase *CPutnickiZrakoplov* iz prethodnog primjera, te inicijalizira statički podatkovni član na nulu, poziv statičkog funkcijskog člana može se vršiti preko klase ili preko objekta.

```
...
int CPutnickiZrakoplov::iBrojPutnickihZrakoplova = 0;

CPutnickiZrakoplov oPZrakoplov(1000, 180, 130, 1000, 1000);

// pristup preko objekta
cout << oPZrakoplov.DajBrojZrakoplova() << endl;

// pristup preko klase
cout << CPutnickiZrakoplov::DajBrojZrakoplova() << endl;
...
```

Statički funkcijski članovi nemaju skriveni *this* pokazivač. Pošto se interno pristup podatkovnim članovima funkcije vrši koristeći *this* pokazivač, kojeg statička funkcija nema, to znači da statički funkcijski član ne može pristupati nestatičkim podatkovnim članovima.

12.3. Pokazivači na funkcije

Kao što je ime polja konstantni pokazivač na prvi element polja, tako je i ime funkcije konstantni pokazivač na funkciju.

Moguće je deklarirati pokazivačku varijablu koja pokazuje na funkciju, te pozvati funkciju koristeći taj pokazivač.

Pokazivačka varijabla mora imati tip koji se poklapa s potpisom funkcije. Na primjer, ako funkcija prima dvije varijable tipa *int* i vraća *int,* potrebno je da tip varijable bude pokazivač na funkciju koja prima dva integera i vraća integer.

Primjer: Poziv funkcije koja zbraja dva broja preko pokazivača.

```
int Zbroji ( int a, int b )
{
   return a + b;
}

int (*pZbroji) (int, int);

int main()
{
   pZbroji = Zbroji;
   cout << pZbroji (2, 3);
   return 0;
}
```

pZbroji je pokazivačka varijabla na tip funkcije koja prima dvije cjelobrojne varijable, te vraća također cjelobrojni podatak. Bitno je da se ime pokazivačke varijable zajedno sa '*' karakterom stavi unutar zagrada, u protivnom bi to bila deklaracija funkcije koja prima dva integera i vraća integer.

U *main* funkciji prvo se adresa funkcije *Zbroji* pridjeljuje pokazivaču. Ime funkcije bez parametara predstavlja konstantni pokazivač na funkciju.

pZbroji(2, 3) – predstavlja poziv funkcije *Zbroji* preko pokazivača. Iza pokazivača navedeni su parametri funkcije, kao da je ona poziva.

Rezultat koji vraća funkcija ispisuje se pomoću *cout* objekta.

12.3.1. Prosljeđivanje pokazivača funkciji

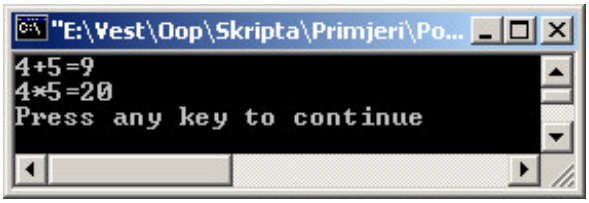
Pokazivač na funkciju se može proslijediti funkciji kao parametar. Tada se mora u deklaraciji funkcije koja taj pokazivač prima navesti tip pokazivača.

Primjer: Unutar funkcije *Operacija* može se pozvati funkcija koja vrši neku od matematičkih operacija (zbrajanje ili množenje) nad dva proslijeđena parametra funkciji.

```
#include <iostream.h>

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

int Mnozi ( int a, int b )
{
        return a * b;
}

int (*pFunc) (int, int);


int Operacija ( int (*pFunc)(int, int), int a, int b)
{
        return pFunc(a, b);
}

int main()
```

```
{
        pFunc = Zbroji;

        cout << "4+5="<< Operacija ( pFunc, 4, 5 ) << endl;

        pFunc = Mnozi;

        cout << "4*5="<< Operacija ( pFunc, 4, 5 ) << endl;

        return 0;
}
```

Ispis će biti sljedeći:



12.4. *Korištenje* typedef

Navođenje tipa pokazivača koji se prenosi funkciji je dosta nezgrapno:

```
int (*)(int, int);
```

te se umjesto njega može kreirati korisnički tip podatka koristeći *typedef*

```
typedef int (PFUNC*)(int, int);
```

Sada bi dio koda gdje je definirana funkcija izgledao ovako:

```
int Operacija (PFUNC  pFunc, int a, int b)
{
        return pFunc(a, b);
}
```

12.5. Pokazivači na funkcijske članove

Pokazivač se može kreirati na određeni funkcijski član klase. Prilikom deklaracije, osim imena pokazivača treba navesti ime klase funkcijskog člana.

Ako se želi uzeti pokazivač na funkcijski član *PostaviBoju()* klase *CGrafickiObjekt*, treba ga kreirati na sljedeći način:

```
void (CGrafickiObjekt::*pFunc) (int);
```

pFunc predstavlja pokazivač na neki funkcijski član klase *CGrafickiObjekt* koji prima jedan cjelobrojni podatak i ne vraća ništa.

Pridjeljivanje adrese funkcije pokazivaču vrši se na jedan od dva načina:

preko objekta

```
pFunc = pGrObjekt->PostaviBoju;
```

ili

preko klase

```
pFunc = CGrafickiObjekt::PostaviBoju;
```

Poziv funkcijskog člana preko pokazivača vrši se preko objekta na sljedeći način:

```
(pGrObjekt->*pFunc) (15);
```

Ispred pokazivača se obavezno treba staviti '*', te se pokazivač zajedno sa objektom stavlja u zagrade, zbog prioriteta.

U ovom primjeru koristio se pokazivač na objekt, ali ista je situacija s pristupom preko objekta, razlika je samo u operatoru pristupa. ('.' ili '->').

Primjer:

```
#include <iostream.h>
#include <string.h>

class CGrafickiObjekt
{
public:
        CGrafickiObjekt(int x, int y, int boja=0, int iKut=0){ iX = x; iY = y; };
        void PostaviBoju(int boja){ iBoja = boja;}
        int DajBoju(){return iBoja;}
        void PostaviKoordinate(int x, int y){ iX=x; iY=y;}
        int DajXKoord(){return iX;}
        int DajYKoord(){return iY;}
        virtual void Draw() {}
protected:
        int iBoja;
        int iX;
        int iY;
        int iKut;
};

class CText : public CGrafickiObjekt
{
public:
        CText( char *text, int iX, int iY ):CGrafickiObjekt (iX, iY){ strcpy ( szText,
text); }
        void PostaviText ( char *text) { strcpy ( szText, text);    }
        char *DajText ( char *text) { strcpy ( text, szText ); return text; }
        virtual void Draw(){ cout << szText << endl;};
private:
        char szText[20];
};

class CPravokutnik : public CGrafickiObjekt
{
public:
        CPravokutnik( int xstart, int yStart, int xSize, int ySize
):CGrafickiObjekt(xstart, yStart),
                iXSize(xSize), iYSize(ySize){}
        virtual void Draw () { cout << iX << "       " << iY << "   " << iXSize << "
        " << iYSize << endl; }
private:
```

```
        int iXSize;
        int iYSize;
};

class CUokvireniText : public CPravokutnik, public CText
{
public:
        CUokvireniText ( char *text, int x, int y, int xSize, int ySize ):
          CText( text, x, y ),
                  CPravokutnik ( x, y, xSize, ySize ){}
          virtual void Draw (){ cout << "Iscrtavanje uokvirenog texta" << endl;}
private:
};

int main()
{
        void (CGrafickiObjekt::*pFunc)(int) = 0;

        CGrafickiObjekt *pGrObjekt = new CText( "WTC 123.122", 0, 0 );

        pFunc = pGrObjekt->PostaviBoju;

        (pGrObjekt->*pFunc)(15);

        return 0;
};
```

12.6. Polja pokazivača na funkcijske članove

Ponekad je potrebno kreirati polje pokazivača na funkcijske članove. Pozivi funkcijskih članova vrše se preko elemenata polja. Kao primjer mogla bi biti klasa koja nad podatkom (podacima) koje sadrži objekt vrši matematičke operacije. Pozivi funkcija za zbrajanje, oduzimanje, množenje, dijeljenje itd mogu biti preko pokazivača kao elementa polja

Da bi se kreiralo polje pokazivača na funkcijske članove prvo se može kreirati pokazivač na funkcijski član klase kao korisnički tip podatka pomoću *typedef* te nakon toga polje pokazivača.

```
typedef int (CGrafickiObjekt::*pFUNC)();

pFUNC pFn[2]={0};
```

Pridruživanje adrese funkcije pojedinom elementu polja vrši se na sljedeći način:

```
pFn[1] = CGrafickiObjekt::DajYKoord;
```

Poziv funkcije vrši se navođenjem elementa polja i parametara funkcije:

```
(pGrObjekt->*pFn[1])();
```

Primjer: Izvlačenje početnih koordinata grafičkog objekta preko polja pokazivača na funkcijske članove.

```
#include <iostream.h>
#include <string.h>

class CGrafickiObjekt
{
```

```
public:
        CGrafickiObjekt(int x, int y, int boja=0, int iKut=0){ iX = x; iY = y; };
        void PostaviBoju(int boja){ iBoja = boja;}
        int DajBoju(){return iBoja;}
        void PostaviKoordinate(int x, int y){ iX=x; iY=y;}
        int DajXKoord(){return iX;}
        int DajYKoord(){return iY;}
        virtual void Draw() {}
protected:
        int iBoja;
        int iX;
        int iY;
        int iKut;
};

class CText : public CGrafickiObjekt
{
public:
        CText( char *text, int iX, int iY ):CGrafickiObjekt (iX, iY){ strcpy ( szText,
text); }
        void PostaviText ( char *text) { strcpy ( szText, text);    }
        char *DajText ( char *text) { strcpy ( text, szText ); return text; }
        virtual void Draw(){ cout << szText << endl;};
private:
        char szText[20];
};

class CPravokutnik : public CGrafickiObjekt
{
public:
        CPravokutnik( int xstart, int yStart, int xSize, int ySize
):CGrafickiObjekt(xstart, yStart),
                iXSize(xSize), iYSize(ySize){}
        virtual void Draw () { cout << iX << "       " << iY << "   " << iXSize << "
        " << iYSize << endl; }
private:
        int iXSize;
        int iYSize;
};

class CUokvireniText : public CPravokutnik, public CText
{
public:
        CUokvireniText ( char *text, int x, int y, int xSize, int ySize ):
          CText( text, x, y ),
                  CPravokutnik ( x, y, xSize, ySize ){}
          virtual void Draw (){ cout << "Iscrtavanje uokvirenog texta" << endl;}
private:
};

int main()
{
        typedef int (CGrafickiObjekt::*pFUNC)();

        pFUNC pFn[20]={0};

        CGrafickiObjekt *pGrObjekt = new CText( "ANC 123.122", 10, 20 );

        pFn[0] = CGrafickiObjekt::DajXKoord;
        pFn[1] = CGrafickiObjekt::DajYKoord;

        cout << (pGrObjekt->*pFn[0])() << endl;
        cout << (pGrObjekt->*pFn[1])() << endl;

        return 0;
};
```

13. Napredno nasljeđivanje

13.1. private i protected nasljeđivanje

Do sada su se kroz primjere osnovne klase nasljeđivale s *public* načinom. To znači da će funkcijski i podatkovni članovi koji su imali *public* pravo pristupa biti vidljivi izvan naslijeđene klase. *protected* članovi biti će vidljivi samo unutar naslijeđene klase. *private* članovi neće biti vidljivi ni u, ni iz naslijeđene klase, izvedena klasa nema nikakva posebna prava nad njima.

Kada se koristi *private* tip nasljeđivanja, svi public i protected članovi klase postaju privatni u izvedenoj klasi.

Kod *protected* načina nasljeđivanja svi *public* i *protected* članovi osnovne klase postaju *protected*, te su vidljivi samo unutar izvedene klase. *private* članovi osnovne klase i dalje nisu vidljivi u izvedenoj klasi, izvedena klasa na njih nema nikakve povlastice.

13.2. Klasa prijatelj "friend" class

Pristup pojedinom *private* ili protected funkcijskom ili podatkovnom članu klase izvan objekta nije direktno dopušten. Svaki takav pokušaj rezultira javljanjem greške prilikom prevođenja.

Ponekad je potrebno da klasa omogući pristup nekom svojim privatnim podacima i funkcijskim članovima objektima neke druge klase.

Da bi se to omogućilo potrebno je drugu klasu proglasiti tzv. klasom prijateljem *(engl. "friend class")*.
Prijateljska klasa sada ima pristup svim članovima klase kojoj je prijatelj.

Kao primjer prijateljske klase može se uzeti klasa *CVektor* (za početak ogoljena klasa). Klasa *CVektor* u sadrži podatkovne članove o trima parametrima vektora u prostoru (x, y, i z).
Osim funkcijskih članova za postavljanje i čitanje vrijednosti koordinata, može sadržavati preopterećene verzije operatora za zbrajanje, oduzimanje i množenje vektora.
Objekti ove klase mogu poslužiti prilikom matematičkih proračuna međusobne udaljenosti pojedinih zrakoplova, itd...

Prilikom kreiranja objekta klase *CVektor*, jedan od konstruktora prima objekt klase *CZrakoplov*. Iz objekta klase *CZrakoplov* izvlače se podaci o koordinatama zrakoplova. To se može obaviti preko pristupnih funkcija, ali u ovom slučaju može se koristiti direktni pristup podatkovnim članovima.

```
class CZrakoplov
{
public:
friend class CVektor;
...
};
```

...

```
class CVektor
{
public:
        CVektor ( CZrakoplov zrakoplov )
        {
                dX = zrakoplov.dXKoord;
                dY = zrakoplov.dYKoord;
                dZ = zrakoplov.dVisina;
        }

        CVektor( double x, double y, double z ): dX(x), dY(y), dZ(z){}
        double DajX(){return dX;}
        void PostaviX(double x){dX = x;}
        double DajY(){return dY;}
        void PostaviY(int y){dY = y;}
        double DajZ(){return dY;}
        void PostaviZ(int y){dY = y;}
private:
        double dX;
        double dY;
        double dZ;
};

void IspisKoordinate(void)
{
   // kod za ispis koordinata
}

int main()
{
        CZrakoplov *pZrak = new CPutnickiZrakoplov ( 1000, 300, 120, 1000, 1000);

        CVektor zrVektor(*pZrak);

        Ispisikoordinate();

        return 0;
}
```

Konstruktor klase *CVektor* direktno čita podatkovne članove proslijeđenog objekta klase *CZrakoplov*.

13.3. Funkcija prijatelj "friend" class

Ponekad za pristup privatnim podacima klase nije potrebna vanjska klasa već samo jedna funkcija. Npr. ako je potrebno formatirano ispisati trenutnu poziciju zrakoplova direktnim pristupom u podatkovne članove klase *CVektor*.

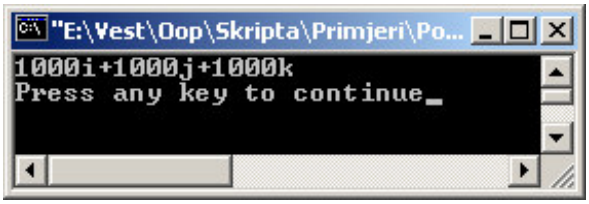
Za formatirani ispis kreira se funkcija:

```
void Pozicija ( class CVektor vektor )
{
        cout << vektor.dX << "i+" << vektor.dY << "j+" << vektor.dZ << "k" << endl;
}
```

Klasa *CVektor* ovu funkciju proglasi za funkciju prijatelja (friend funkcion)

```
class CVektor
{
public:
...
        friend void Pozicija ( class CVektor vektor );
...
private:
        double dX;
        double dY;
        double dZ;
};

int main()
{
        CZrakoplov *pZrak = new CPutnickiZrakoplov ( 1000, 300, 120, 1000, 1000);

        CVektor zrVektor(*pZrak);

        Pozicija (zrVektor);

        return 0;
}
```

Funkciji *Pozicija* prosljeđuje se objekt *zrVektor* klase *CVektor*. Funkcija direktno čita i formatirano ispisuje koordinate zrakoplova (vektora).



13.4. Preopterećeni operator "<<"

Za ispis podataka na konzolu koristi se objekt *cout*. On ispisuje tekst, varijable ugrađenih tipova podataka, kao što su *int, char,* itd. Za ulančavanje ispisa koristi se operator insertiranja "<<".

Međutim *cout* objekt ne zna kako da ispisuje podatke sadržane u korisnički definiranim klasama. Npr. za objekt klase CVektor nije definirano kako bi se ispisali podaci:

```
CVektor zrVektor(...);
cout << zrVektor;
```

Za drugu liniju koda prevoditelj bi javio grešku prilikom prevođenja.

C++ pruža mogućnost preopterećenja globalnog operatora "<<".

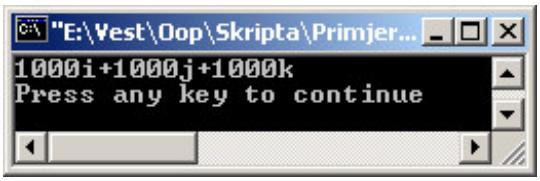
Preopterećeni globalni operator prima referencu na objekt klase ostream, te vraća referencu na isti objekt. Kao ulazni podatak još prima korisnički definiran podatak.

Kao korisnički definirani tip može se uzeti klasa CVektor, s tri koordinate.

Deklaracija preopterećenog globalnog operatora "<<":

```
ostream& operator<< ( ostream &theStream, CVektor &theVektor )
{
        return theStream << theVektor.dX << "i+" << theVektor.dY << "j+" <<
theVektor.dZ << "k" << endl;
}

int main()
{
        CZrakoplov *pZrak = new CPutnickiZrakoplov ( 1000, 300, 120, 1000, 1000);

        CVektor zrVektor(*pZrak);

        cout << zrVektor;

        return 0;
}
```

Ispis programa izgleda ovako:



Naravno, da bi preopterećeni globalni operator "<<" mogao pristupiti privatnim članovima klase klasa *CVektor*, ona mora proglasiti preopterećeni operator prijateljem.

```
class CVektor
{
...
    friend ostream& operator<< ( ostream& theStream, CVektor &theVektor );
...
};
```

14. Ulazno izlazni tokovi

14.1. Općenito o tokovima

C++ kao jezik ne definira kako se podaci zapisuju na ekran, upisuju u datoteku ili čitaju iz nje.
Standardna C++ biblioteka uključuje *iostream* biblioteku koja omogućava rad sa ulazom/izlazom.

To što su ulaz i izlaz izdvojeni od jezika i obrađeni u bibliotekama klasa, omogućava C++ jeziku neovisnost o platformi. Program napisan na jednoj platformi rekompiliran na drugoj uredno radi (bar teoretski).

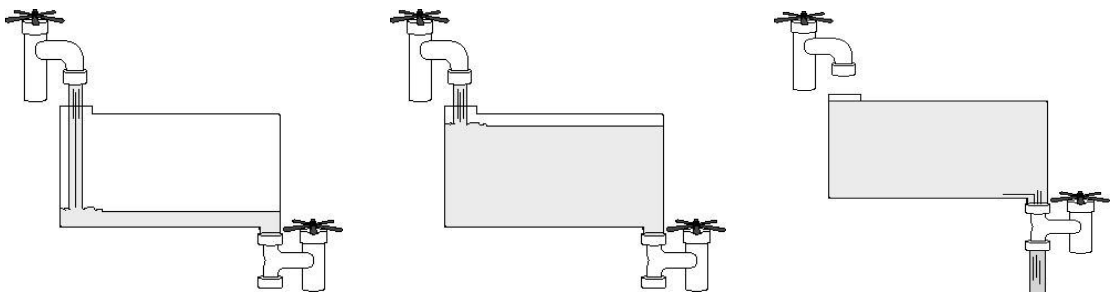
14.2. Ulazno-izlazni tokovi i međuspremnik (buffer)

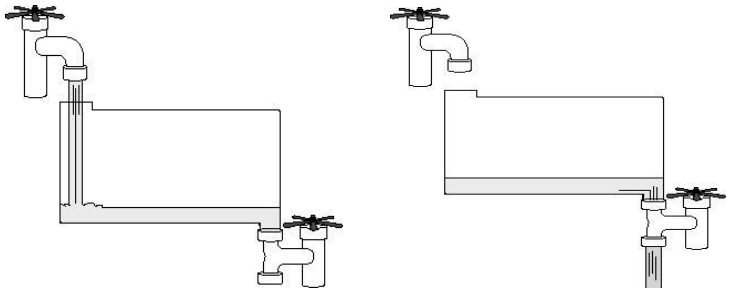
iostream vidi tok podataka programa byte po byte, bilo da je riječ o zapisivanju podataka (u datoteku, na ekran itd.) ili čitanju podataka (npr. iz datoteke ili tipkovnice).

Zapisivanje i čitanje podataka na disk ili ekran zahtijeva dosta vremena. U tom vremenu rad programa je blokiran.

Da bi se taj problem riješio *tokovi (streams)* koriste međuspremnike (*buffers).*
Prilikom zapisivanja podaci se spremaju u *buffer,* umjesto da se zapisuju direktno na disk.
Prilikom zapisivanja *buffer* se puni sve dok se ne napuni, i tek onda se podaci odjednom spreme na disk.

Kao primjer *buffera* može se uzeti spremnik za vodu sa dva ventila:





Slika 13-1

Kada je otvoren gornji ventil spremnik se puni vodom sve dok se ne napuni. Kad se napuni do vrha, otvara se donji ventil i spremnik se potpuno isprazni.

Moguća je i druga situacija: iako spremnik nije do vrha popunjen otvara se donji ventil, te se spremnik potpuno isprazni.
Ova operacija zove se pražnjenje *buffera (flushing the buffer).*

C++ koristi objektno orijentirani pristup implementaciji *tokova (streams)* i *buffera*.

Osnovna klasa je *streambuf*, koja rukuje sa bufferom, i njeni funkcijski članovi omogućavaju punjenje, pražnjenje (flush) i druge manipulacije sa bufferom.

Klasa *ios* je osnovna klasa izvedenih klasa za ulaz i izlaz. Klasa *ios* ima objekt *streambuf* klase kao podatkovni član

Klase *istream i ostream* izvedene su iz *ios* klase i specijalizirane za ulazne, odnosno izlazne tokove podataka.

Klasa *iostream* je izvedena od *istream i ostream* klasa te ima funkcijske članove za upis, te za ispis na ekran.

Klasa *fstream* koristi se za čitanje i pisanje u datoteke.

Postoji četiri standardna ulazno izlazna objekta: *cin, cout, cerr, clog*.

14.3. Objekt cin

cin predstavlja globalni objekt. Uključen je u program prilikom uključivanja datoteke *iostream.h*.

Objekt *cin ima preopterećeni operator redirekcije ">>"* , tako da prihvaća razne vrste parametara: int&, short&, long&, double&, float&, char&, char* itd.

Kada se napiše:

```
cin >> vVarijabla;
```

te ako je vVarijabla tipa *int*, poziva se sljedeća preopterećena operatorska funkcija:

```
istream& operator>> (int&);
```

14.4. Objekt cout

Ispis podataka pomoću objekta *cout* korištenjem "<<" operatora omogućava ispis raznih tipove podataka (char, int, double, char*, ...). Za svaki od ovih tipova podataka postoji jedan preopterećeni operator "<<".

Operator "<<" kao povratnu vrijednost vraća referencu na *ostream* objekt, te se zbog toga može ulančati više operatora prilikom ispisa:

```
cout << "Telefonski broj je: " << "552-345" << endl;
```

Korisni funkcijski članovi objekta *cout* su:

flush() – služi da bi se ispraznio *buffer* u koji se prilikom ispisa na konzolu spremaju podaci. Podaci ostaju u bufferu dok se on ne napuni, te se tada odjednom ispisuju. Ako se želi odmah neki podatak ispisati koristi se ovaj funkcijski član koji prazni buffer iako on nije napunjen.

endl() – je funkcijski član koji u izlazni string ubacuje oznaku za novi red i poziva flush funkcijski član koji prazni buffer, tako da se sadržaj buffera odmah ispisuje na ekran.

Ova dva funkcijska člana imaju svoje pandane u tzv. manipulatorima, koji se mogu ulančavati s ostalim podacima u ispisu preko *cout* objekta.

Primjer:

```
cout << "Ispis na ekran" << flush;
```

14.4.1. Funkcijski članovi za ispis teksta

put() – upisuje jedan karakter na izlazni uređaj. Ima jedan ulazni parametar, a to je karakter koji ispisuje. Funkcijski član kao povratnu vrijednost vraća referencu na *ostream* objekt.

```
cout.put ('H');

cout.put('H').put('e').put('l') .put('l') .put('o') .put('!');
```

write() – funkcijski član radi slično kao i "<<" operator. Razlika je u drugom parametru funkcijskog člana *write*, koji određuje maksimalni broj karaktera koji će biti ispisan.

```
cout.write ("tekst", 5);
```

Ako je maksimalni dopušteni broj karaktera veći od duljine stringa koji se ispisuje, tada se u taj "višak" ispisuje smeće.

Primjer ispisa dužeg od duljine stringa:

```
int main()
{
        cout.write("Ovaj tekst iza sebe ima smeće", 40 );

        cout << endl;

        return 0;
}
```

Ispis programa:

14.5. Stringovi

cin objekt rukuje sa pokazivačem na niz karaktera (char*), te se stoga može napisati:

```
char szIme[50];
cout << "Upisi ime: ";
cin >> szIme;
```

Ako se ime kao jedna riječ, npr: Marko, varijabla szIme biti će popunjena karakterima 'M', 'a', 'r', 'k', 'o', '\0'. Posljednji karakter je terminator, *cin* objekt ga automatski stavlja iza karaktera 'o'.

Međutim problem nastaje kada se npr. preko tastature unese ime i prezime. Razmak između (*whitespace)* imena i prezimena *cin* objekt shvaća kao separator te u *szIme* niz karaktera unese samo ime. Prezime ostaje i dalje u *bufferu*.

14.6. Operator >>

Povratna vrijednost *cin* objekta je referenca na istream objekt. Pošto je *cin* objekt također istream objekt povratna vrijednost jedne ekstrakcije može biti ulaz za drugu, tj operacije *cin* objekta mogu se ulančati.

Primjer:

```
int iBroj;
double dBroj;
char szText[20];

cin >> szText >> iBroj >> dBroj;
```

U istoj liniji koda učitava se niz karaktera, cjelobrojna vrijednost i double tip varijable.

Gornja linija koda ima isti učinak kao i sljedeća

```
( ( cin >> szText ) >> iBroj ) >> dBroj;
```

14.7. Funkcijski članovi objekta cin

Operator ">>" može se koristiti za učitavanje jednog karaktera.

Za učitavanje jednog karaktera može se također koristiti funkcijski član *get*. Može se koristiti na dva načina, sa i bez parametara.

Koristeći *get* bez parametara kao povratnu vrijednost vraća pročitani karakter.

```
char a = cin.get();
```

Funkcijski član *get* u ovom slučaju vraća podatak tipa *char*.

Drugi način je korištenje funkcijskog člana *get* koji kao parametar prima referencu na podatak tipa char u koji će upisati učitani karakter.

```
char a;
cin.get(a);
cout << a;
```

Prototip funkcijskog člana je sljedeći:

```
istream::get(char &);
```

Kao što se vidi, vraća istream objekt. Prema tome moguće je ulančavanje.

Primjer:

```
cin.get(a).get(b).get(c);
```

ili

```
cin.get(a) >> b;
```

Funkcijski član vraća i EOF karakter kada se prilikom čitanja dođe do kraja datoteke ili se karakter simulira prilikom učitavanja podataka da tastature sa Ctrl-Z.

14.7.1. Učitavanje stringova sa standardnog ulaza

Prilikom učitavanja stringa preko operatora redirekcije ">>" problem predstavlja "white space" karakter koji predstavlja terminaciju učitavanja iako niz učitanih karaktera može biti veći.

U takvom slučaju može se koristiti funkcijski član *getline*, koji prima tri parametra: pokazivač na polje karaktera gdje se sprema učitani niz karaktera, drugi parametar predstavlja veličinu polja karaktera odnosno maksimalni broj karaktera koji se u polje može učitati plus jedan za terminacijski karakter. Posljednji treći parametar predstavlja terminacijski karakter. Njegova pretpostavljena vrijednost je '\n'. Prema tome ako se treći parametar ne navede učitavanje niza karaktera preko tastature završava kada se pritisne *enter* ili se popuni buffer. Funkcijski član na kraju automatski uspisuje umjesto unesenog terminacijskog karaktera terminator stringa '\0'.

Deklaracija funkcijskog člana *getline* je sljedeća:

```
istream& istream::getline(signed char *, int, char='\n');
```

14.7.2. Funkcijski član *ignore*

Ponekad je potrebno izbrisati karaktere koji su ostali u *bufferu* prilikom unošenja sa tastature.
Funkcija ignore briše obređene broj karaktera iz *buffera.*

Funkcija ima dva parametra. Prvi je maksimalni broj karaktera koji treba izbrisati iz *buffera*, a drugi je terminacijski karakter.

```
istream& istream::ignore ( int, int );
```

Primjer:

```
char szString[20];
cin >> szString;
cin.ignore(80,'\n');
```

Ako je preko tastature uneseno npr. "Danas je lijep dan" te na kraju <enter>, u szString upisuje se *"Danas"*. Ostatak rečenice ostaje *bufferu*. *ignore* funkcijski član prazni *buffer*. Preostalih karaktera ima manje od 80, a na kraju je i '\n' terminacijski karakter.
Buffer je sada prazan.

Da se nije koristio funkcijski član *ignore* prilikom unošenja novog stringa u sljedećoj liniji koda unio bi se podatak koji je ostao u *bufferu,* a to je niz karaktera "je".

14.7.3. Funkcijski članovi *peek() i putback()*

Funkcijski član *peek* čita vrijednost sljedećeg karaktera u *bufferu,* ali ga ne izvlači.

```
char a;
a= cin.peek();
```

Za razliku od funkcijskog člana *peek*, funkcijski član *putback* insertira karakter u ulazni tok, tako da ga se sa sljedećim čitanjem npr. sa *get* čita.

Primjer:

```
cin.putback('$');
```

14.8. Funkcijski članovi objekta cout

Objekt *cout* omogućava ispis na standardni izlaz (konzolu). Pomoću preopterećenog operatora redirekcije ">>" prihvaća razne tipove varijabli, pokazivače i reference na podatke.

Ispis je moguće formatirati: numeričke varijable u decimalnom, heksadecimalno ili oktalnom sistemu, ispis teksta sa lijevim ili desnim poravnanjem itd.

Podaci proslijeđeni objektu *cout* ne ispisuju se istog trenutka, već se spremaju u izlazni *buffer*. Kada se buffer napuni podaci se ispisuju na konzolu.
Da bi se podaci odmah ispisali potrebno je isprazniti *buffer* pozivom funkcijskog člana *flush()* ili manipulatora *flush*.

Primjer:

```
cout.flush();
```

ili

```
cout << flush;
```

Umjesto operatora redirekcije "<<" za ispis mogu se koristiti dva funkcijska člana put() i write().

put() se koristi za ispis jednog jedinog karaktera na izlazni uređaj. *put*() funkcijski član vraća referencu na *ostream* objekt te se može povezati više naredbi jedna iza druge kao i kod redirekcijskog operatora.

Primjer:

```
cout.put('H').put('e').put('l').put('l').put('o').put('\n');
```

Vrijedi i sljedeće:

```
cout.put('H') << 'e';
```

write() funkcijski član radi na isti način kao "<<" operator redirekcije, samo ima dva parametra: string koji ispisuje i maksimalni broj karaktera stringa koji treba ispisati.

Primjer:

```
char szString[] = "Aerodrom Resnik";

cout.write( szString, strlen(szString)) << endl;
cout.write( szString, 8) << endl;
cout.write( szString, strlen(szString)+10) << endl;
```

Ispis će biti:

Aerodrom Resnik
Aerodrom
Aerodrom Resniksdfjkljdfa

Ako je maksimalni broj karaktera veći od broja duljine stringa, ispisuju se oni karakteri koji se nalaze iza kraja stringa, tj. terminator tu ne igra nikakvu ulogu.

Ako je maksimalni broj karaktera manji od veličine stringa za ispis ispisuje se dio stringa veličine maksimalnog broja karaktera.

14.9. Manipulatori, flagovi, formatiranje

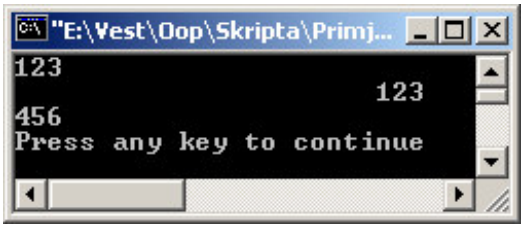
14.9.1. Funkcijski član *width()*

Za predefinirana širina ispisa je širine točno koliko je potrebno za ispis broja, karaktera ili stringa. Npr. širina ispisa 564.23 je točno 6 karaktera.
Širina ispisa se može promijeniti korištenjem *width()* funkcijskog člana objekta *cout*.

Primjer:

```
cout << 123 << '\n';

cout.width(25);
cout << 123 << '\n';
cout << 456 << '\n';
```

Ispis je sljedeći:



Funkcijski član *width()* djeluje samo na prvi sljedeći ispis.

14.9.2. Funkcijski član *fill()*

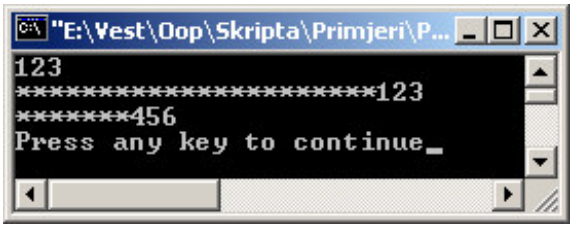
Ako je postavljena širina ispisa veća od samog ispisa prazni prostor ispunjen je po default-u white space karakterima tj. space karakterima.

Pomoću funkcijskog člana *fill()* može se postaviti kojim karakterom da se popunjavaju prazna mjesta prilikom formatiranog ispisa.

Primjer:

```
cout.fill('*');
cout << 123 << '\n';
cout.width(25);
cout << 123 << '\n';
cout.width(10);
cout <<456 << '\n';
```

Ispis je sljedeći:



Na ispisu se vidi da ispis prvog reda nema nikakvih promjena. Širina ispisa je ista kao i veličina broja karaktera.
Ispis u drugom redu ima širinu od 25 karaktera, te ima 23 prazna mjesta. Ta prazna mjesta popunjena su sa '*' karakterom. Slično je i sa ispisom trećeg reda.

14.9.3. Set flagovi

iostream objekt neka svoja stanja prati preko *flagova*. Ti se *flagovi* mogu postavljati pomoću funkcijskog člana *Setf()* objekta *cout*.

Set flagovi koji se mogu postavljati su:

ios::showpoint	ispisuje decimalnu točku
ios::showpos	ispisuje predznak ispred broja ('+')
ios::left	lijevo poravnanje prilikom ispisa
ios::right	desno poravnanje prilikom ispisa
ios::internal	interno poravnanje
ios::hex	ispis u hex brojnom sustavu
ios::dec	ispis u decimalnom brojnom sustavu
ios::oct	ispis u oktalnom brojnom sustavu

ios::showbase prilikom ispisa broja ispisuje se simbol brojnog sustava (0x za heksadecimalni sustav prije samog broja)

Primjer:

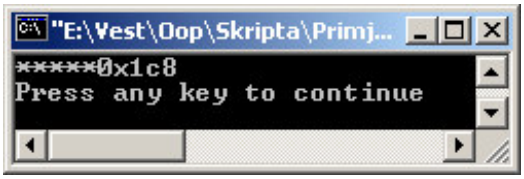
```
cout.setf ( ios::showpos );
cout << 456;
```

Ispis:

+456

```
cout.setf( ios::showpos );
cout.fill ( '*' );
cout.width (10);
cout.setf (ios::showbase);
cout.setf ( ios::hex );
cout << 456;
```

Ispis:



Opis:
Širina ispisa je 10 karaktera po default-u desno poravnana.
Prazna mjesta su popunjena sa '*' (fill funkcijski član).
Broj je ispisan u heksadecimanom formatu (ios::hex).
Ispred ispisa broja ispisuje se oznaka brojevnog sustava "0x".

14.10. Ulazi i izlazi datoteka

Ulazno izlazni tokovi omogučavaju rukovanje podacima na uniforman način, bilo da je riječ o ulazu podataka sa tastature ili iz datoteke sa hard diska, te ispisu podataka na ekran ili u datoteku na disku.
U oba slučaja koriste se operatori redirekcije "<<" i ">>".

Da bi se moglo rukovati sa datotekom potrebno je kreirati *ifstream* i *ofstream* objekte koji se koriste za učitavanje iz ili upisivanje u datoteku.

14.10.1. ofstream

Objekti koji se koriste za čitanje i pisanje u datoteke zovu se *fstream* objekti. Oni su izvedeni iz *iostream* klase.

Da bi se moglo pisati u datoteku potrebno je prvo kreirati *ofstream* objekt. Prije toga potrebno je učitati *fstream.h* datoteku:

```
#include <fstream.h>
```

Datoteka *iostream.h* se automatski uključuje kada se uključi *fstream.h* datoteka pa je ne treba posebno navoditi.

Objekt klase *iostream* sadrži flagove koji sadrže trenutna stanja ulaza i izlaza. Stanja flagova mogu se provjeriti pomoću nekoliko funkcija koje vraćaju BOOL tip podatka. Te funkcije su: *eof(), bad(), fail()* i *good().*

Funkcija *bad()* vraća TRUE ako se izvrši neka nevažeća operacija.

fail() vraća TRUE svaki put kada je bad() TRUE ili neka operacija ne uspije.

good() vraća TRUE svaki put ako dvije prethodne funkcije vrate FALSE.

14.11. Otvaranje datoteka za ulaz i izlaz

Da bi se otvorila datoteka imena *logFile.log* za pisanje potrebno je deklarirati instancu *ofstream* objekta i proslijediti ime datoteke kao parametar.

```
ofstream fout ( "e:\\logFile.log" );
```

Otvaranje iste datoteke za čitanje radi se na isti način, s tim da se koristi *ifstream* objekt.

```
ifstream fin ( "e:\\logFile.log" );
```

Ovim se kreiraju dva objekta koja služe za pisanje i čitanje datoteke.

Nakon završetka korištenja datoteke potrebno je pozvati funkcijski član ovih objekata za zatvaranje datoteka *close()*.

```
fin.close();

fout.close();
```

Zatvaranjem datoteke osigurava se da se svi podaci koji se trebaju upisati u datoteku, pohrane u nju, s obzirom da su neki možda još u *buffer-u* (flush).

Ispis u datoteku i čitanje iz nje radi se na isti način kao kod korištenje *cin* i *cout* objekata, tj. koriste se operatori redirekcije, s tim da se koriste imena objekata *fin* i *fout*.

Primjer:

```
#include <fstream.h>

int main()
{
        char szBuffer[255];
        ofstream fout ( "e:\\logFile.log" );
        fout << "Spremanje teksta u datoteku";
        fout.close();
        ifstream fin ( "e:\\logFile.log" );
        fin.get( szBuffer, 255 );
        fin.close();

        return 0;
}
```

Default postavke prilikom otvaranja datoteke su:
- kreiranje datoteke ako ne postoji
- ako postoji izbriše se sadržaj

Ako je potrebno promijeniti ove postavke potrebno je koristiti drugi parametar prilikom otvaranja datoteke.

Neke od vrijednosti drugog parametra su:

ios::app	- dodavanje podataka na kraj postojeće datoteke
ios::at	- početno postavljanje na kraj datoteke, ali moguće je zapisivanje bilo gdje u datoteci
ios::trun	- briše sadržaj postojeće datoteke prilikom otvaranja
ios::nocreat	- ako datoteka ne postoji, javlja se greška prilikom pokušaja otvaranja
ios::noreplac	- ako datoteka postoji javlja se greška prilikom pokušaja otvaranja

Primjer otvaranja datoteke korištenjem drugog parametra:

```
ofstream fout ( fileName, ios::app );
...
fout << buffer;
...
fout.close();
```

U gornjem primjeru datoteka se otvara tako da se pokazivač na sljedeći zapis postavlja na kraj datoteke. Početni podaci su zapisani.

14.12. Binarne i tekstualne datoteke

Prilikom rada sa objektima ili poljima objekata dolazi se do situacije kada je potrebno sačuvati kompletno stanje objekta.

Struktura objekta definirana je podatkovnim članovima koja predstavljaju trenutna stanja objekta i funkcijskim članovima kojim se vrše operacije nad objektom.

Da bi se sačuvalo trenutno stanje objekta potrebno je negdje spremiti sadržaj podatkovnih članova objekta.

Koristeći *fstream* objekte može se pohraniti trenutno stanje objekta, te učitati prilikom ponovnog startanja programa.

Za spremanje i čitanje u datoteku ovih podataka treba se koristiti binarni mod rada. U binarnom modu rada svaki byte se sprema kakav jest.

```
CZrakoplov zrakoplov_1(300, 200, 120, 1000), zrakoplov_2;
ofstream fout ( fileName, ios::binary );
fout.write ( (char *) &zrakoplov_1, sizeof (CZrakoplov) );
fout.close();
ifstream fin (fileName, ios::binary );
fin.read ( (char *) &zrakoplov_2, sizeof (CZrakoplov) );
fin.close ();
```

U prethodnom primjeru sadržaj objekta *zrakoplov_1* spremljen je u datoteku. Nakon toga iz te iste datoteke sadržaj je učitan u objekt *zrakoplov_2*.

Prilikom upisivanja datoteke u tekstualnom obliku, znak za novi red '\n' se konvertira u '\r\n' (oznaka za carriage return + oznaka za new line).

Prilikom učitavanja datoteke u tekstualnom obliku automatski se radi obrnuta konverzija.

14.13. Procesiranje komandne linije

Kada se iz komandne linije starta program, iza imena sa *exe* ekstenzijom mogu se navesti dodatni parametri.

U programu se ti parametri mogu koristiti, npr. program koji kopira datoteku iza imena navodi ime datoteke koja se kopira, te kao drugi parametar predstavlja odredište.

fileCopy datoteka_1 datoteka_2 <enter>

Funkcija *main()* može čitati parametre iz imena programa pomoću dva svoja parametra.

```
int main( int argc, char **argv )
```

argc – predstavlja broj parametara proslijeđenih programu plus ime programa

argv - predstavlja polje pokazivača na polja karaktera od kojih svako sadrži stringove koji predstavljaju jedan od parametara proslijeđenih programu.

argv[0] – pokazuje na ime samog programa ("fileCopy.exe")

15. Predlošci

15.1. Uvod

Podaci koji se koriste u programu (npr. podaci o objektima u nekom grafičkom programu) mogu se pohranjivati u memoriji računala preko liste objekata, struktura ili sl.

Svaki grafički objekt može biti pohranjen kao jedan element liste.

Također lista može sadržavati i objekte koji na primjer predstavljaju zrakoplove na radarskom ekranu.

Pristupanje listi vrši se preko pokazivača *pHead* koji pokazuje na prvi čvor elementa liste.
Svaki čvor je objekt klase *CNode* koji sadrži pokazivač na sljedeći čvor i pokazivač na objekt kao element liste.

Lista ima standardne funkcijske članove za insertiranje elemenata u listu, brisanje, izvlačenje iz liste, sortiranje itd. Ti funkcijski članovi u listi koja sadrži neke druge elemente npr. listi cjelobrojnih brojeva imaju istu funkciju, samo rukuju sa drugim tipovima podataka.

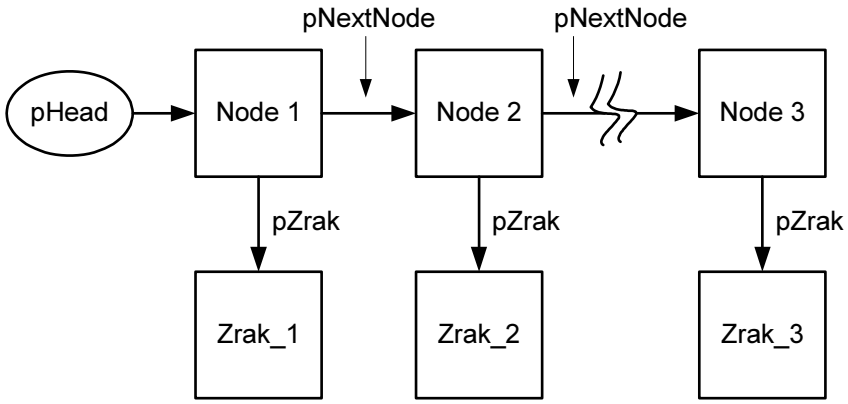
Ako imamo izvorni kod liste koja kao elemente sadrži objekte tipa *CZrakoplov*, a potrebno je kreirati listu koja kao elemente sadrži cjelobrojne brojeve, izvorni kod liste može se prepraviti "copy-paste" načinom da prihvati novi tip podataka.

Međutim ako se za npr. pretraživanje elemenata u listi uvede novi funkcijski član, potrebno je ponovno izvršiti "copy-paste" u izvorni kod liste za cjelobrojne brojeve.

Takav način je prilično dosadan i pogodan je za generiranje grešaka.

U C++ jeziku uvedeni su *predlošci ("templates"),* koji doskaču ovom problemu.

Predlošci omogučavaju kreiranje liste bilo kojeg tipa podataka umjesto da se kreira lista za jedan specifičan tip podatka. Predložak ima parametre, koji određuju za koji će se tip podatka kreirati lista.



Slika 14-1

Način na koji razlikujemo pojedine liste su tipovi podataka koje lista sadrži.

U predlošku, tip podatka koji lista sadrži postaje parametar prilikom definicije klase.

Postupak kreiranja specifičnog tipa iz predloška zove se instantacija. Kreirana klasa zove se *instanca predloška.*

15.2. Predlošci klasa

Za početak kreirana je klasa *CArray* koja sadrži polje određenog tipa podataka. Unutar klase kreirani su funkcijski članovi za rukovanje poljem unutar objekta, operator "[]" za pristup pojedinom elementu polja itd...

Primjer:

```
template< class T>
class CArray1
{
public:
	CArray1(int size = 10):iItsSize(size), pArray(new T[size]){};
	T& operator[] (int i)
	{
		return pArray[i];
	}
private:
	T *pArray;
	int iItsSize;
};

int main()
{
	CArray1<int> polje(3);

	polje[0] = 1;
	polje[1] = 2;
	polje[2] = 3;

	cout << polje[0] << endl << polje[1] << endl << polje[2] << endl ;

	return 0;
}
```

U programu je kreiran predložak (template) koji sadrži polje. Parametar predloška T predstavlja tip elemenata polja.

Kreiranje predloška počinje sa ključnom riječju *template* iza koje slijede "<>" zagrade u kojem se navode parametri predloška. Ispred parametra predloška obavezno dolazi ključna riječ *class*. T predstavlja parametar.
Prilikom instantacije kao parametar u programu naveden je *int*. Prevoditelj prilikom prevođenja u klasi *CArray1* zamjenjuje *T* sa *int.* Da je kao parametar navedena klasa *CZrakoplov* prevoditelj bi kreirao klasu koja bi sadržavala objekte tipa *CZrakoplov*.

Ispod reda u kojem se nalazi *template*, definira se klasa kao i obično, s tim da svi tipovi podataka koji ovise o parametru predloška umjesto tipa imaju oznaku *T*. *T* će prilikom instantacije biti zamijenjen sa vrijednošću parametra (*int* u primjeru).

Instanciranje predloška vrši se tako da se navede ime predloška te odmah iza u "<>" zagradama navede vrijednost parametra te konstruktor objekta koji se kreira.

```
CArray1<int> polje(3);
```

Bitno je da se '<' karakter nalazi odmah iza ključne riječi *template*, da se ne bi shvatio od strane prevoditelja kao operator '<' manje.

U primjeru je kreiran objekt koji sadrži polje od tri podatka tipa *int*.

Definicija konstruktora klase *CArray1* nalazila se unutar klase. Ako se konstruktor definira izvan klase tada on treba sadržavati još i *CArray<T>*.

```
template<class T>
CArray1<T>::CArray1 (int size):iItsSize(size), pArray1( new T[size] ))
{
   for(int i=0;i<size;i++)
        pArray1[i] = 0;
}
```

Primjer 2:

Kao sljedeći primjer može se uzeti lista. Lista se sastoji od čvorova (node) i elemenata liste (mogu biti ugrađeni tipovi podataka *int, char, double,* ili objekti neke klase, itd.)

Pojednostavljeni primjer liste ima funkcijske članove za insertiranje elementa na kraj liste *InsertAtEnd(), PrintData()*, te *DeleteList()* za ispis elemenata liste. Postoje i članovi se dobavljanje sljedećeg elementa liste, te postavljanje sljedećeg elementa liste.

Čvor liste ima pokazivače na jedan element liste, te pokazivač na sljedeći čvor liste. Pokazivač *pHead* sadrži adresu čvor prvog člana liste. Preko njega pristupamo listi.

U primjeru izvorni kod liste kreiran je preko *predloška*. Instanciranje liste vrši se sa cjelobrojnom *(int)* vrijednošću parametra T. Prema tome lista će se koristiti za pohranjivanje cjelobrojnih vrijednosti.

Kreiranje pokazivača na početak liste za cjelobrojne podatke vrši se na sljedeći način:

```
CNode<int> *pHead;
```

Insertiranje elementa na kraj liste vrši se na sljedeći način:

```
int a=5;
pHead->InsertAtEnd(&a);
```

Podatak će se pohraniti u novo kreiranom čvoru liste koji će se nalaziti na kraju liste. Ako lista nije postojala, pozivom ovog funkcijskog člana bit će kreiran čvor s prvim elementom liste.

Ispis svih elemenata liste vrši se sa funkcijskim članom *PrintData()*

```
pHead->PrintData();
```

Uništavanje liste vrši se pozivom funkcijskog člana *DeleteList()*.

```
pHead->DeleteList();
```

Izvorni kod predloška liste sa primjerom liste cjelih brojeva *(int)*:

```
#include <iostream.h>

template<class T>
class CNode
{
public:
        CNode(T&);
        ~CNode();
        SetNext(CNode *node);
        CNode<T>* InsertAtEnd (T*);
        CNode* GetNext(CNode &node);
        PrintData(void);
        DeleteList (void);
private:
        CNode   *pNext;
        T       *pItem;
};

template<class T>
CNode<T>::CNode( T& rItem)
{
        pItem = new T;
        *pItem = rItem;
        pNext = NULL;
}

template<class T>
CNode<T>::~CNode()
{
        CNode *pNode = this;

        delete this->pItem;

        pNode = pNode->pNext;

        if ( pNode )
                delete pNode;
}

template<class T>
CNode<T>::DeleteList (void)
{
        CNode<T> *pNode = this;

        delete this->pItem;
```

```
        pNode = pNode->pNext;

        if ( pNode )
                delete pNode;
}


template<class T>
CNode<T>* CNode<T>::GetNext(CNode &node)
{
        return node.pNext;
}

template<class T>
CNode<T>::SetNext(CNode *node)
{
        pNext = node;
}

template<class T>
CNode<T>* CNode<T>::InsertAtEnd(T *data)
{
        //Kreiranje novog clana
        CNode<T> *pNewItem = new CNode<T>(*data);

        // Nadji kraj liste ako postoji već neki element liste
        if (this)
        {
                CNode<T> *pNode = this;
                CNode<T> *pNextNode = this->pNext;

                while ( pNextNode )
                {
                        pNode = pNode->pNext;
                        pNextNode = pNode->pNext;
                }

                pNode->pNext = pNewItem;
        }

        return pNewItem;
}

template<class T>
CNode<T>::PrintData(void)
{
        CNode *pNode = this;

        do
        {
                cout << *(pNode->pItem) << endl;
                pNode = pNode->pNext;
        }
        while ( pNode != NULL );
}

CNode<int> *pHead;

int a = 1, b = 2;

int main()
{
        pHead = pHead->InsertAtEnd(&a);

        pHead->InsertAtEnd(&b);
        pHead->InsertAtEnd(&a);
        pHead->InsertAtEnd(&a);

        pHead->PrintData();

        pHead->DeleteList();

        return 0;
}
```

Ako je potrebno lista koja kao elemente ima npr. objekte klase *CZrakoplov*, prilikom instantacije u <> zagradama navodi se klasa *CZrakoplov* umjesto *int*.

```
CNode<CZrakoplov> *pHead;
```

Sve ostale operacije su iste.

Naravno, problem nastaje oko specifičnih stvari, a to su npr. ispis elemenata liste (za CZrakoplov bi trebalo definirati preopterećeni operator <<, da bi se mogli ispisati podaci bitni za objekt. Također prilikom insertiranja objekta klase *CZrakoplov* u listu potrebno je definirati konstruktor kopije ili će se koristiti podrazumijevani konstruktor kopije.

15.3. Predlošci funkcija

Predlošci klasa mogu se proslijeđivati funkcijama, na način da se proslijedi točno određena instanca polja.

Može se uzeti primjer klase koja sadrži polje brojeva sa početka poglavlja *CArray1*.

Objekt klase *CArray1* se prosljeđuje funkciji *PrintArrayData()* koja ispisuje sve članove objekta.

Potrebno je klasi *CArray1* dodati izostavljene članove za izvlačenje podatka o broju elemenata.

```
int CArray1<int>::GetArraySize(void)
{
   return iItsSize;
}
```

Definicija funkcije za ispis podataka kojoj se objekt prosljeđuje izgleda ovako:

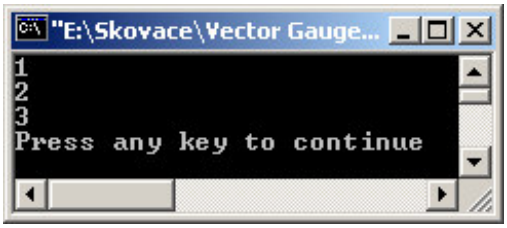
```
void PrintArrayData(CArray1<int>& arrData)
{
        for ( int i=0;i<arrData.GetArraySize();i++)
        {
           cout << arrData[i] << endl;
        }
}
```

Funkciji je proslijeđena referenca na objekt instanciran iz predloška sa cjelobrojnim vrijednostima.

Primjer i ispis programa koji koristi funkciju *PrintArrayData()*:

```
int main()
{
        CArray1<int> polje(3);

        polje[0] = 1;
        polje[1] = 2;
        polje[2] = 3;

        PrintData( polje );
```

```
        return 0;
}
```



Da bi se kreirala opća funkcija koja prihvaća sve predloške potrebno je kreirati predložak funkcije.

```
template<class T>
void PrintData( CArray1<T>& arrData )
{
        for ( int i=0; i<arrData.GetArraySize(); i++ )
        {
                cout << arrData[i] << endl;
        }
}
```

15.4. Predlošci i "friends"

Postoje tri vrste "friend" klasa i funkcija:

1. Friend klasa ili funkcija koja nije predložak
2. Opći predložak friend klase ili funkcije
3. Predložak klase ili funkcije specifične po tipu.

15.4.1. Friend klasa ili funkcija koja nije predložak

Svaka instanca klase u ovom slučaju tretirat će "prijatelja" kao da je deklaracija prijateljstva napravljena u svakoj pojedinoj instanci.

```
#include <iostream.h>

template< class T>
class CArray1
{
public:
        CArray1(int size = 10):iItsSize(size), pArray(new T[size]){};
        int GetArraySize () { return iItsSize;}
        T& operator[] (int i)
        {
                return pArray[i];
        }
        friend void gInitToZero(CArray1<int> &arr);
private:
        T *pArray;
        int iItsSize;
};

// prijateljska funkcija koja nije template, već joj se prosljeđuje instanca
// predloška sa int elementima sa vidi se u svim instancama, ali je validna samo u
// int polju.
// Pokušaj korištenja u objektu sa char poljem generira grešku prilikom
```

```
// prevođenja.
void gInitToZero(CArray1<int> &arr)
{
        for (int i=0;i<arr.iItsSize;i++)
        {
                arr.pArray[i] = 0;
        }
}

template<class T>
void PrintData( CArray1<T>& arrData )
{
        for ( int i=0; i<arrData.GetArraySize(); i++ )
        {
                cout << arrData[i] << endl;
        }
}

int main()
{
        CArray1<int> polje(3);
        CArray1<char> cPolje(3);

        gInitToZero(polje);
        //gInitToZero(cPolje); // greska

        PrintData( polje );

        return 0;
}
```

15.4.2. Opći predložak friend klase ili funkcije

Kao primjer može se definirati opći predložak operatora za prikazivanje sadržaja objekta klase *CArray1*, koji funkcionira za sve tipove klase *CArray1*.

Predložak preopterećenog globalnog operatora << kao *friend* funkcija prihvaća sve instance klase *CArray1*.

Definira se tako da se ispred definicije funkcije napiše:

```
template<class T>
```

te se kroz tijelo funkcije provlači *T* kao parametar.

Primjer:

```
#include <iostream.h>

template< class T>
class CArray1
{
public:
        CArray1(int size = 10):iItsSize(size), pArray(new T[size]){};
        int GetArraySize () { return iItsSize;}
        T& operator[] (int i)
        {
                return pArray[i];
        }
        friend ostream& operator<<(ostream&, CArray1<T> &);
private:
        T *pArray;
        int iItsSize;
};

template<class T>
ostream& operator<<(ostream& theStream, CArray1<T> &arr)
```

```
{
        for ( int i=0;i<arr.iItsSize;i++)
                theStream << arr.pArray[i] << endl;

        return theStream;
}


int main()
{
        CArray1<int> polje(3);
        CArray1<char> cPolje(3);

        polje[0] = 1;
        polje[1] = 2;
        polje[2] = 3;

        cPolje[0] = 'a';
        cPolje[1] = 'b';
        cPolje[2] = 'c';

        cout << polje;
        cout << cPolje;

        return 0;
}
```

15.4.3. Predložak klase ili funkcije specifične po tipu.

Prethodni primjer je dobar kada je riječ o ugrađenim tipovima podataka *(int, char, itd...)*. Međutim ako se kao vrijednost parametra proslijedi npr. klasa *CZrakoplov*, tada *friend* preopterećeni globalni operator neće ispravno raditi, jer *stream* objekt operatorskoj funkciji ne zna kako da ispise vrijednost objekta.

U tom slučaju ispred globalne operatorske funkcije potrebno je izbrisati:

```
template<class T>
```

te operatorsku funkciju prepraviti:

```
ostream& operator<<(ostream& theStream, CArray1<int> &arr)
{
        for ( int i=0;i<arr.iItsSize;i++)
                theStream << arr.pArray[i] << endl;

        return theStream;
}
```

Sada globalna operatorska funkcija << vrijedi samo za jednu određenu instancu predloška, u ovom slučaju samo sa predložak klase poljem cjelobrojnih *(int)* vrijednostima.
Za objekt sa *char tipom* polja podataka << operatorska funkcija više ne vrijedi.

15.5. Specijalizirane funkcije

Prilikom kreiranja predloška klase *CArray1* može se dogoditi da jedan od funkcijskih članova, recimo konstruktor ispravno radi za ugrađene tipove podataka, a ne radi ispravno kada se kao vrijednost parametra predloška postavi klasa *CZrakoplov*.

Jezik C++ omogućava tzv. specijalizaciju funkcije predloška za određeni tip podatka.

Definicija konstruktora predloška *CArray1* je sljedeća:

```
template<class T>
CArray1<T>::CArray1(int size):iItsSize(size), pArray(new T[size])
{
        iNumberOfObjects++;
};
```

Međutim ako se kreira predložak klase *CArray1* koji sadrži polje objekata klase *CZrakoplov*, te su tom predlošku potrebne neke dodatne inicijalizacije, može se kreirati konstruktor koji je specifičan samo za polje objekata klase *CZrakoplov*.

```
CArray1<CZrakoplov>::CArray1(int size):iItsSize(size), pArray(new CZrakoplov[size])
{
        iNumberOfObjects++;
        // dodatni dio inicijalizacije specijalizira za klasu CZrakoplov
        ...
};
```

Prilikom instantacije za sve vrijednosti parametra (tipove podataka) predloška biti će korišten prvi konstruktor, dok će za kreiranje sa tipom klase *CZrakoplov* biti korišten specijalizirani (drugi).

15.6. Statički članovi klase i predlošci

U predlošku se mogu deklarirati statički podatkovni članovi. Svaka instantacija predloška sadrži svoj skup statičkih podataka, jedan po klasi.

Kada se kreira statički podatkovni član klase npr. *CArray1*, koji sadrži podatak koliko je objekata klase *CArray1* kreirano, svaka instantacija predloška *CArray1* imat će svoj zasebni statički podatkovni član po tipu (jedan za objekte čiji su elementi cjeli brojevi, jedan za objekte čiji su elementi realni brojevi, te jedan za npr. objekte čiji su elementi objekti klase *CZrakoplov*.

Primjer korištenja statičkog člana klase:

```
template< class T>
class CArray1
{
public:
        CArray1(int size = 10):iItsSize(size), pArray(new T[size]){ iNumberOfObjects++;
};
        int GetArraySize () { return iItsSize;}
        T& operator[] (int i)
        {
                return pArray[i];
        }
        friend ostream& operator<<(ostream&, CArray1<T> &);
private:
        static int iNumberOfObjects;
        T *pArray;
        int iItsSize;
};
```

Statički podatkovni članovi moraju se deklarirati i inicijalizirati. Kod predložaka potrebno je navesti parametar. Prilikom prevođenja stvara se onoliko statičkih varijabli koliko se kreira instantacija predloška. To znači: ako se kreiraju dvije vrste instanci predloška. jedna sa vrijednosti *int* parametra *T*, a druga sa *CZrakoplov*, kreiraju se dva statička podatkovna člana *iNumberOfObjects*.

Inicijalizacija se vrši na sljedeći način:

```
template<class T>
int CArray1<T>::iNumberOfObjects = 0;
```

Ovom deklaracijom i inicijalizacijom predloška kreirane su i inicijalizirane dva statička podatkovna člana, jedan za klasu koja sadrži polje cjelobrojnih brojeva, jedan koji sadrži polje objekata klase *CZrakoplov.*

16. Iznimke

16.1. Uvod – što su iznimke

Prilikom razvoja software-a javljaju se razne vrste pogrešaka i bugova. Neke pogreške nalazi prevoditelj, neke su logičke prolaze kompajliranje, ali su odmah uočljive, jer sruše program svaki put prilikom pokretanja.

Pogreške koje su odmah vidljive najmanje i koštaju.

Međutim postoje pogreške koje se ne javljaju odmah, već nakon nekog vremena, koje ponekad sruše program bez nekog vidljivog razloga.

Također postoje pogreške koje se javljaju jer programer nije predvidio da će korisnik unijeti podatke koje program ne očekuje. Npr. unijeti tekst tamo gdje program očekuje numerički podatak.

Program također mora znati što napraviti ako ne bude u mogućnosti alocirati memoriju, ako disketa nije u uređaju ili sl.

Postoje dvije vrste pogrešaka koje je potrebno razlikovati, u jednu spadaju sintaksne pogreške i logičke pogreške zbog greške programera (krivo ukucanog koda i lošeg razumijevanja problema), te druga, iznimke koje nastaju zbog neobičnih ali predvidljivih situacija koje se mogu dogoditi (npr. program ostane bez memorije).

Iznimke *(engl. "exceptions")* su situacije koje se ne mogu eliminirati, već se program može pripremiti kako da reagira na njih. Ako program ostane bez memorijskih resursa može ga se pripremiti da reagira na jedan od sljedećih načina:

- sruši program
- obavijesti korisnika o problemu i terminira program
- obavijesti korisnika i omogući korisniku da pokuša popraviti problem
- pokuša riješiti problem bez uznemiravanja korisnika

16.2. Korištenje iznimki

Iznimka je objekt koji se prosljeđuje iz dijela koda gdje se problem pojavio u dio koda koji će taj problem obraditi.

Tip iznimke određuje koji će dio koda obraditi problem, te sadržaj bačenog objekta iznimke može se koristiti da proslijedi informaciju korisniku.

Osnovna ideja koja se krije iza iznimki je sljedeća:

Alokacija resursa (alokacija memorije, obrada datoteka itd.) radi se na vrlo niskom nivou programa.

Logika obrade kada neka operacija ne uspije, npr. alokacija memorije, pozicionirana je na višem nivou u programu, i vezana uz interakciju sa korisnikom.

16.2.1. try blok

try blok se postavlja oko dijela koda koji može biti problematičan. Npr. potrebno je kreirati objekt klase *CZrakoplov* s početnim parametrima (zrakoplov je ušao u područje osmatranja zračne luke).

```
try
{
        CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );
}
```

Npr. kreiranje objekta može biti sa nerealnim parametrima, recimo visina ne može biti negativna, brzina ne može biti preko neke vrijednosti (900 m/s).

Iza *try* bloka slijedi blok koji služi za hvatanje iznimki koje mogu biti bačene u *try* bloku.

```
try
{
        CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );
}
catch ( CZrakoplov::xBadAltitude )
{
        // obradi situaciju kada je nerealna visina
}
catch ( CZrakoplov::xBadSpeed )
{
        // obradi situaciju kada je nerealna brzina
}
```

Osnovni koraci prilikom korištenja iznimki su sljedeći:

1. Odrediti koja područja programa mogu podignuti iznimku, te ih postaviti unutar *try* bloka.

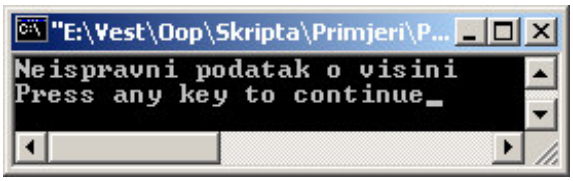
2. Kreirati *catch* blokove za hvatanje iznimki ako budu bačene, npr. za dealokaciju memorije i za obavještavanje korisnika.

Primjer kreiranja objekta klase *CZrakoplov* sa korištenjem iznimki:

```
#include <iostream.h>

class CZrakoplov
{
public:
        CZrakoplov():
                dVisina(0),
```

```
                dBrzina(0),
                dSmjer(0),
                dXKoord(0),
                dYKoord(0)
                {}

        CZrakoplov( double visina, double brzina, double smjer, double xKoord, double
yKoord );

        double DajVisinu(){return dVisina;}
        void PostaviVisinu(double visina) { dVisina = visina; }
        double DajBrzinu (void) {return dBrzina;}
        void PostaviBrzinu (double brzina){dBrzina = brzina;}
        double DajSmjer(){return dSmjer;}
        void PostaviSmjer(double smjer){dSmjer = smjer;}
        double DajXKoord(){return dXKoord;}
        void PostaviXKoord(double xKoord){dXKoord = xKoord;}
        double DajYKoord(){return dYKoord;}
        void PostaviYKoord(double yKoord){dYKoord = yKoord;}

        class xBadAltitude{};
        class xBadSpeed{};
private:
        double dVisina;
        double dBrzina;
        double dSmjer;
        double dXKoord;
        double dYKoord;
};

CZrakoplov::CZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord ):
        dVisina(visina),
        dBrzina(brzina),
        dSmjer(smjer),
        dXKoord(xKoord),
        dYKoord(yKoord)
{
        if ( dVisina < 0 || dVisina > 22000 )
                throw xBadAltitude();

        if ( dBrzina < 0 || dBrzina > 900 )
                throw xBadSpeed();
}

int main()
{
        try
        {
                CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );
        }

        catch ( CZrakoplov::xBadAltitude )
        {
                cout << "Neispravni podatak o visini" << endl;
        }
        catch ( CZrakoplov::xBadSpeed )
        {
                cout << "Neispravni podaci o brzini" << endl;
        }
        catch(...)
        {
                cout << "Neidentificirana iznimka" << endl;
        }

        return 0;
}
```

Ispis programa je sljedeći:



Opis:

Unutar klase *CZrakoplov* kreirane su dvije klase *xBadAltitude* i *xBadSpeed*. Ove dvije klase nemaju kreiranih podatkovnih i funkcijskih članova. Ipak imaju automatski generirane default konstruktor i destruktor, copy konstruktor i operator pridruživanja.

Ove dvije klase nemaju nikakvih posebnih prava u odnosu na klasu *CZrakoplov* niti klasa *CZrakoplov* ima neka posebna prava u odnosu na ove dvije klase.

Konstruktoru klase *CZrakoplov* dodan je dio koda koji ispituje ispravnost upisanih podataka visine i brzine zrakoplova.
Podaci o visini i brzini da bi bili realni moraju se nalaziti u neki dozvoljenim granicama.
U slučaju da neki od ova dva podatka nije validan baca se iznimka.

Iznimka se baca korištenjem ključne riječi *throw*, te navođenjem konstruktora iznimke (iznimka je objekt neke klase koji se prosljeđuje dijelu koda koji je obrađuje)

```
if ( dVisina < 0 || dVisina > 22000 )
        throw xBadAltitude();
```

Poziv konstruktora nalazi se unutar *try* bloka.

16.2.2. catch blok

Ispod *try* bloka mora postojati *catch* blok koji prima i obrađuje bačene iznimke. *catch* blok je niz izraza, od kojih svaki počinje sa ključnom riječju *catch* iza koje slijedi u zagradama tip iznimke.
Unutar vitičastih zagrada nalazi se kod koji obrađuje određenu bačenu iznimku.

```
catch ( CZrakoplov::xBadAltitude )
{
        // obrada
        ...
}
```

16.3. Hvatanje iznimki

Kada je iznimka bačena ispituje se *stack poziva. stack poziva* je lista funkcija kreiranih kada jedan dio programa pozove drugu funkciju.
Problematični dio koda postavljen je u *try* blok. Kada se baci iznimka, dio koda ograničen u *try* bloku se preskače (*stack-a poziva* se odmotava i sa njega se skidaju sve funkcije koje su pozvane unutar bloka).

Program se nastavlja na mjestu iza *try* bloka gdje se nalazi se jedan ili više *catch* izraza.

Ovisno o bačenoj iznimci izvršava se jedan od *catch* izraza.

```
        try
        {
                CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );
        }

        catch ( CZrakoplov::xBadAltitude )
        {
// obrada greške 1
        }
        catch ( CZrakoplov::xBadSpeed )
        {
                // obrada greške 2
        }
        catch(...)
        {
                // default-na obrada
                // ako nije pronađena ni jedna od gornjih iznimki
                // program se nastavlja na ovom mjestu
        }
```

Ako bačena iznimka nije ni jedna od navedenih iznimki u *catch* izrazima izvršava se *catch* izraz koji sadrži "..." unutar zagrada. Ovaj posljednji ima analogiju sa *default* ključnom riječju kod *switch case* izraza.

Ako ne postoji *catch(...)* izraz na kraju više *catch* izraza, te ni jedan od njih ne obrađuje bačenu iznimku, *stack poziva* se "odmotava" tako da skače više u hijerarhiji poziva, te traži sljedeće *try-catch* blokove u dijelu koda koji je pozvao funkciju koja je sadržavala *try-catch* kod. Ako ni tu ne pronađe dio za obradu *stack poziva* se dalje "odmotava" dok ne pronađe dio koda za obradu ili ne dođe do main() funkcije sa početka programa. Ako ni tu ne pronađe dio za obradu, program se terminira sa porukom o abnormalnoj terminaciji.

Primjer programa sa odmotavanjem *stack-a poziva*:

```
#include <iostream.h>

class CZrakoplov
{
public:
        CZrakoplov():
                dVisina(0),
                dBrzina(0),
                dSmjer(0),
                dXKoord(0),
                dYKoord(0)
```

```
                {}

        CZrakoplov( double visina, double brzina, double smjer, double xKoord, double
yKoord );

        double DajVisinu(){return dVisina;}
        void PostaviVisinu(double visina) { dVisina = visina; }
        double DajBrzinu (void) {return dBrzina;}
        void PostaviBrzinu (double brzina){dBrzina = brzina;}
        double DajSmjer(){return dSmjer;}
        void PostaviSmjer(double smjer){dSmjer = smjer;}
        double DajXKoord(){return dXKoord;}
        void PostaviXKoord(double xKoord){dXKoord = xKoord;}
        double DajYKoord(){return dYKoord;}
        void PostaviYKoord(double yKoord){dYKoord = yKoord;}

        class xBadAltitude{};
        class xBadSpeed{};
private:
        double dVisina;
        double dBrzina;
        double dSmjer;
        double dXKoord;
        double dYKoord;
};

CZrakoplov::CZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord ):
        dVisina(visina),
        dBrzina(brzina),
        dSmjer(smjer),
        dXKoord(xKoord),
        dYKoord(yKoord)
{
        if ( dVisina < 0 || dVisina > 22000 )
                throw xBadAltitude();

        if ( dBrzina < 0 || dBrzina > 900 )
                throw xBadSpeed();
}

void KreirajObjekt(void)
{
        try
        {
                CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );
        }
        catch ( CZrakoplov::xBadSpeed )
        {
                cout << "Neispravni podaci o brzini" << endl;
        }

        cout << "Ovo se nece ispisati" << endl;
}

int main()
{
        try
        {
                KreirajObjekt();

                cout << "Ni ovo se neće ispisati" << endl;
        }

        catch ( CZrakoplov::xBadAltitude )
        {
                cout << "Neispravni podatak o visini" << endl;
        }
        catch ( CZrakoplov::xBadSpeed )
        {
                cout << "Neispravni podaci o brzini" << endl;
        }
        catch(...)
        {
                cout << "Neidentificirana iznimka" << endl;
        }
```

```
        return 0;
}
```

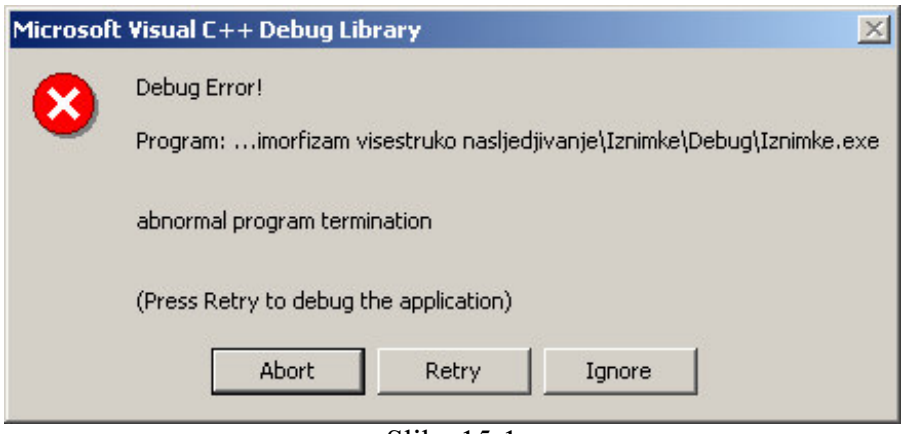
Ponovno je kreiran objekt klase *CZrakoplov*, međutim sada se kreira unutar funkcije *KreirajObjekt()*. Konstruktor klase može baciti dvije iznimke, ukoliko podaci uneseni prilikom kreiranja nisu validni.

Unutar funkcije *main* također poziv funkcije *KreirajObjekt()* ograničen je u *try* blok.

Funkcija *KreirajObjekt()* sadrži dio koda koji obrađuje iznimku vezanu za neispravnu vrijednost brzine, ali ne i visine. Prema tome ne može obraditi iznimku koja će se baciti.

Prilikom poziva funkcije *KreirajObjekt()*, unesena vrijednost visine nije ispravna, te konstruktor baca iznimku o krivoj visini. *catch* izraz unutar funkcije može samo obraditi iznimku vezanu za invalidnu vrijednost brzine.
Pošto *catch izraz* ne može obraditi iznimku izvođenje programa skače više u hijerarhiji (stack poziva se "odmotava"), u *try* blok main funkcije kojem se prosljeđuje iznimka, te se pokušava obraditi u njenim *catch* izrazima.
U ovom dijelu postoji *catch* izraz za obradu iznimke vezane uz brzinu. Taj dio koda će se izvršiti.

Da ni ovdje nije bilo *catch* izraza za obradu ove iznimke, te da nije postojao catch izraz koji hvata sve iznimke, program bi se terminirao te bi se javila slijedeća poruka.



Slika 15-1

16.4. Hijerarhija iznimki

Iznimke su klase, te se prema tome mogu nasljeđivati. U prethodnim primjerima definirane su iznimke koje se generiraju zbog neispravno unesenih podataka prilikom kreiranja objekta klase *CZrakoplov*.

Moglo bi se kreirati osnovnu klasu *xBadData*, te iz nje kreirati izvedene klase: *xBadAltitude, xBadSpeed, ...*

```
class CZrakoplov
{
public:
...
        class xBadData
{

};
        class xBadAltitude : public xBadData {};
        class xBadSpeed : public xBadData {};
private:
...
};
```

16.5. Podaci u iznimkama i dodjeljivanje imena iznimkama

Ponekad je potrebno znati više osim proslijeđenog tipa iznimke, kada je potrebno proslijediti neke podatke sa iznimkom. Pošto su iznimke klase, podaci se mogu inicijalizirati u konstruktoru, te tako proslijediti sa iznimkom.

```
class CZrakoplov
{
public:
...

        class xBadData
        {
        public:
                double GetDataValue(){return dDataValue;}
        protected:
                double dDataValue;
        };
        class xBadAltitude : public xBadData
        {
        public:
                xBadAltitude(double value)
                {
                        dDataValue = value;
                }
        };
        class xBadSpeed : public xBadData
        {
        public:
                xBadSpeed(double value)
                {
                        dDataValue = value;
                }
        };
private:
...
};

CZrakoplov::CZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord ):
        dVisina(visina),
```

```
        dBrzina(brzina),
        dSmjer(smjer),
        dXKoord(xKoord),
        dYKoord(yKoord)
{
        if ( dVisina < 0 || dVisina > 22000 )
                throw xBadAltitude(dVisina);

        if ( dBrzina < 0 || dBrzina > 900 )
                throw xBadSpeed(dVisina);
}

void KreirajObjekt(void)
{
        try
        {
                CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );
        }
        catch ( CZrakoplov::xBadSpeed theSpeed )
        {
                cout << "Neispravni podaci o brzini" << endl;
        }

        cout << "Ovo se nece ispisati" << endl;
}

int main()
{
        try
        {
                KreirajObjekt();

                cout << "Ni ovo se neće ispisati" << endl;
        }

        catch ( CZrakoplov::xBadAltitude theAltitude )
        {
                cout << "Neispravni podatak o visini" << endl;
                cout << "Uunesena visina: " << theAltitude.GetDataValue() << endl;
        }
        catch ( CZrakoplov::xBadSpeed theSpeed )
        {
                cout << "Neispravni podaci o brzini" << endl;
                cout << "Brzina: " << theSpeed.GetDataValue() << endl;
        }
        catch(...)
        {
                cout << "Neidentificirana iznimka" << endl;
        }

        return 0;
}
```

Napomena: dio programa je izbačen.

Kreirana je osnovna klasa *xBadData* koja sadrži osnovne podatkovne članove i funkcijski član za dobavljanje vrijednosti podatka.

Izvedene klase sadrže konstruktor kojem se prosljeđuje vrijednost podatka koji se treba proslijediti sa iznimkom (invalidni podatak o brzini ili visini, ovisno o klasi).

Bacanje iznimke sada se radi sa navođenjem vrijednosti parametra u konstruktoru iznimke:

```
throw xBadSpeed(dVisina);
```

Dio koda koji hvata iznimku kao argument ima objekt klase:

```
catch ( CZrakoplov::xBadSpeed theSpeed )
{
        cout << "Neispravni podaci o brzini" << endl;
        cout << "Brzina: " << theSpeed.GetDataValue() << endl;
}
```

Objekt *theSpeed* sadrži podatak o neispravnoj vrijednosti brzine. Sada taj podatak može poslužiti prilikom obrade greške.

16.6. Prosljeđivanje po referenci i upotreba virtualnih funkcija

Prilikom bacanja i hvatanja iznimki može se koristiti mehanizam polimorfizma. U osnovnoj klasi može se kreirati virtualna funkcija koja se uobliči (dobije izvršni kod specifičan za svaku izvedenu klasu) u izvedenim klasama.

Izrazu *catch* može se prosljeđivati objekt iznimke preko reference, pa ako se proslijedi referenca tipa objekta osnovne klase na objekt izvedene klase, vrijede pozivi virtualnih funkcija.

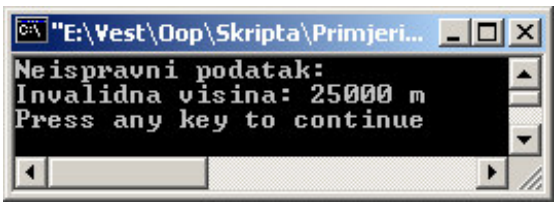
U slijedećem preprevljenom primjeru kreiranja objekta *CZrakoplov*, *catch* izrazu prosljeđuje se referenca na objekt osnovne klase *xBadData*, dok objekt iznimke može biti *xBadAltitude* ili *xBadSpeeed*. Definirana je i virtualna funkcija za ispis podatka o grešci.

catch izraz koji hvata iznimku preko reference na objekt osnovne klase točno zna koju funkciju *PrintData()* treba pozvati.

```
#include <iostream.h>

class CZrakoplov
{
public:
...
        class xBadData
        {
        public:
                double GetDataValue(){return dDataValue;}
                virtual void PrintData(void){}
        protected:
                double dDataValue;
        };

        class xBadAltitude : public xBadData
        {
        public:
                xBadAltitude(double value)
                {
                        dDataValue = value;
                }
                virtual void PrintData(void)
                {
                        cout << "Invalidna visina: " << dDataValue << " m" << endl;
                }
        };

        class xBadSpeed : public xBadData
```

```
        {
        public:
                xBadSpeed(double value)
                {
                        dDataValue = value;
                }
                virtual void PrintData(void)
                {
                        cout << "Invalidna brzina: " << dDataValue << " m/s" << endl;
                }

        };
private:
...
};

CZrakoplov::CZrakoplov( double visina, double brzina, double smjer, double xKoord,
double yKoord ):
        dVisina(visina),
        dBrzina(brzina),
        dSmjer(smjer),
        dXKoord(xKoord),
        dYKoord(yKoord)
{
        if ( dVisina < 0 || dVisina > 22000 )
                throw xBadAltitude(dVisina);

        if ( dBrzina < 0 || dBrzina > 900 )
                throw xBadSpeed(dVisina);
}

int main()
{
        try
        {
                CZrakoplov zrakoplov(25000, 200, 100, 1000, 1000 );

                cout << "Ni ovo se neće ispisati" << endl;
        }

        catch ( CZrakoplov::xBadData &theData )
        {
                cout << "Neispravni podatak: " << endl;
                theData.PrintData();
        }
        catch(...)
        {
                cout << "Neidentificirana iznimka" << endl;
        }

        return 0;
}
```

Ispis je sljedeći:



Napomena jedan dio koda je izostavljen.

16.7. Iznimke i predlošci

Iznimke se mogu koristiti zajedno sa predlošcima. Iznimke se mogu kreirati za svaku instancu predloška, ili se klasa koja predstavlja iznimku može deklarirati izvan deklaracije predloška.

U sljedećem primjeru kreiran je predložak *CArray1* koj sadrži polje podataka. U primjeru predložak je instanciran u dva objekta, jedan sadrži polje cjelobrojnih (int), a drugi polje char tipa podataka.

Osnovna klasa iznimke deklarirana je izvan klase i naslijeđena unutar klase.

Osnovna klasa je *xBadData* deklaracija kreirana izvan klase.

Naslijeđena klasa *xInvalidIndex* je iznimka koja se kreira kad je index za pristup elementu polja unutar objekta invalidan.

Definicija operatorske funkcije izmijenjena je, u nju je dodan dio koda koji testira valjanost proslijeđenog indeksa. Testira se da li je indeks u granicama veličine polja. Ako nije, baca se iznimka.

catch izraz se mora kreirati za svaku vrstu klase *CArray1* koja se instancira. Ovdje su kreirani dva objekta, jedan koji sadrži char tip polja, drugi koji sadrži int tip polja.

```
catch ( CArray1<int>::xInvalidIndex &indexExcpt )
{
   ...
}

catch ( CArray1<char>::xInvalidIndex &indexExcpt )
{
   ...
}
```

Ovisno o vrsti podatka u *try* bloku koji je invalidan obradit će se jedan od ova dva izraza

```
#include <iostream.h>

class xBadData{};

template< class T>
class CArray1
{
public:
        CArray1(int size = 10);
        int GetArraySize () { return iItsSize;}
        T& operator[] (int i);
        friend ostream& operator<<(ostream&, CArray1<T> &);

        class xInvalidIndex:public xBadData
        {
        public:
                xInvalidIndex(int i)
                {
                        iIndex = i;
                }
        void PrintData(void)
```

```
        {
                cout << "Uneseni index je: " << iIndex << endl;
        }
        private:
                int iIndex;
        };
private:
        static int iNumberOfObjects;
        T *pArray;
        int iItsSize;
};

template<class T>
CArray1<T>::CArray1(int size):iItsSize(size), pArray(new T[size])
{
        iNumberOfObjects++;
};


CArray1<char>::CArray1(int size):iItsSize(size), pArray(new char[size])
{
        iNumberOfObjects++;
};

template<class T>
T& CArray1<T>::operator[] (int i)
{
        if ( i<0 || i>= iItsSize )
                throw xInvalidIndex    (i);
        else
                return pArray[i];
}



template<class T>
int CArray1<T>::iNumberOfObjects = 0;

template<class T>
ostream& operator<<(ostream& theStream, CArray1<T> &arr)
{
        for ( int i=0;i<arr.iItsSize;i++)
                theStream << arr.pArray[i] << endl;

        return theStream;
}

ostream& operator<<(ostream& theStream, CArray1<int> &arr)
{
        for ( int i=0;i<arr.iItsSize;i++)
                theStream << arr.pArray[i] << endl;

        return theStream;
}


int main()
{
        CArray1<int> polje(3);
        CArray1<char> cPolje(3);

                cPolje[0] = 'a';
                cPolje[1] = 'b';
                cPolje[2] = 'c';

                polje[0] = 1;
                polje[1] = 2;
                polje[2] = 3;

                cout << flush;

        try
        {
                cPolje[3] = 2;
                polje[4] = 3;
        }
```

```
        catch ( CArray1<int>::xInvalidIndex &indexExcpt )
        {
                indexExcpt.PrintData();
        }

        catch ( CArray1<char>::xInvalidIndex &indexExcpt )
        {
                indexExcpt.PrintData();
        }

        cout << polje;
        cout << cPolje;

        return 0;
}
```

17. Literatura:

Boris Motik, Julijan Šribar: "Demistificirani C++", "Element", Zagreb, 1997

Jesse Liberty: "Teachyourself C++ in 21 days", "Sams Publishing"

Bruce Eckel: "Thinking in C++", www.cplusplus.com