TEHNIČKO VELEUČILIŠTE U ZAGREBU ZAGREB UNIVERSITY OF APPLIED SCIENCES

OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

dr. sc. Željko Kovačević
dr. sc. Miroslav Slamić
dr. sc. Aleksandar Stojanović

Zagreb, 2022

PRIRUČNICI TEHNIČKOG VELEUČILIŠTA U ZAGREBU
MANUALIA POLYTECHNICI STUDIORUM ZAGRABIENSIS

Željko Kovačević
Miroslav Slamić
Aleksandar Stojanović

OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Skripta na kolegiju 'Objektno orijentirano programiranje' koji se održava u sklopu nastave na dodiplomskim stručnim studijima informatike i računarstva Tehničkog veleučilišta u Zagrebu

Zagreb, 2022.

TEHNIČKO VELEUČILIŠTE U ZAGREBU ZAGREB UNIVERSITY OF APPLIED SCIENCES

izdavač	Tehničko veleučilište u Zagrebu Vrbik 8, Zagreb
za izdavača	mr. sc. Goran Malčić
autori	dr. sc. Željko Kovačević
	dr. sc. Miroslav Slamić
	dr. sc. Aleksandar Stojanović
recenzenti	Prof. dr. sc. Danijel Radošević, Fakultet organizacije i informatike
	Hrvoje Rončević, dipl. ing.
vrsta djela	skripta
	Objavljivanje je odobrilo Stručno vijeće Tehničkog veleučilišta u Zagrebu odlukom broj: 2832-1/18 od 23. listopada 2018.
ISBN	978-953-7048-77-8

SADRŽAJ

Uvodna riječ

Pred vama je skripta pisana za potrebe studenata dodiplomskih stručnih studija informatike i računarstva na Tehničkom veleučilištu u Zagrebu. Sadržaj skripte obuhvaća sadržaj kolegija "Objektno orijentirano programiranje", a namijenjen je svima koji žele učiti objektno orijentiranu programsku paradigmu korištenjem programskog jezika C++.

Skripta je pisana pod pretpostavkom da čitatelj već ima određena iskustva u programskom jeziku C ili drugim sličnim programskim jezicima, te odmah kreće sa uvodom u objektno orijentirano programiranje. Skripta je izrazito praktično orijentirana te sadrži mnoštvo primjera i zadataka za samostalnu vježbu. Za njihovo rješavanje preporuka je koristiti Microsoft Visual Studio ili neko drugo razvojno okruženje sa integriranim C++ prevoditeljem koji podržava minimalno C++11 standard.

Ovim putem autori se također žele zahvaliti i svima koji su doprinijeli kvaliteti sadržaja ove skripte bilo svojim prijedlozima ili primjedbama, a to su mnogobrojni kolege iz struke, recenzenti te studenti Tehničkog veleučilišta u Zagrebu.

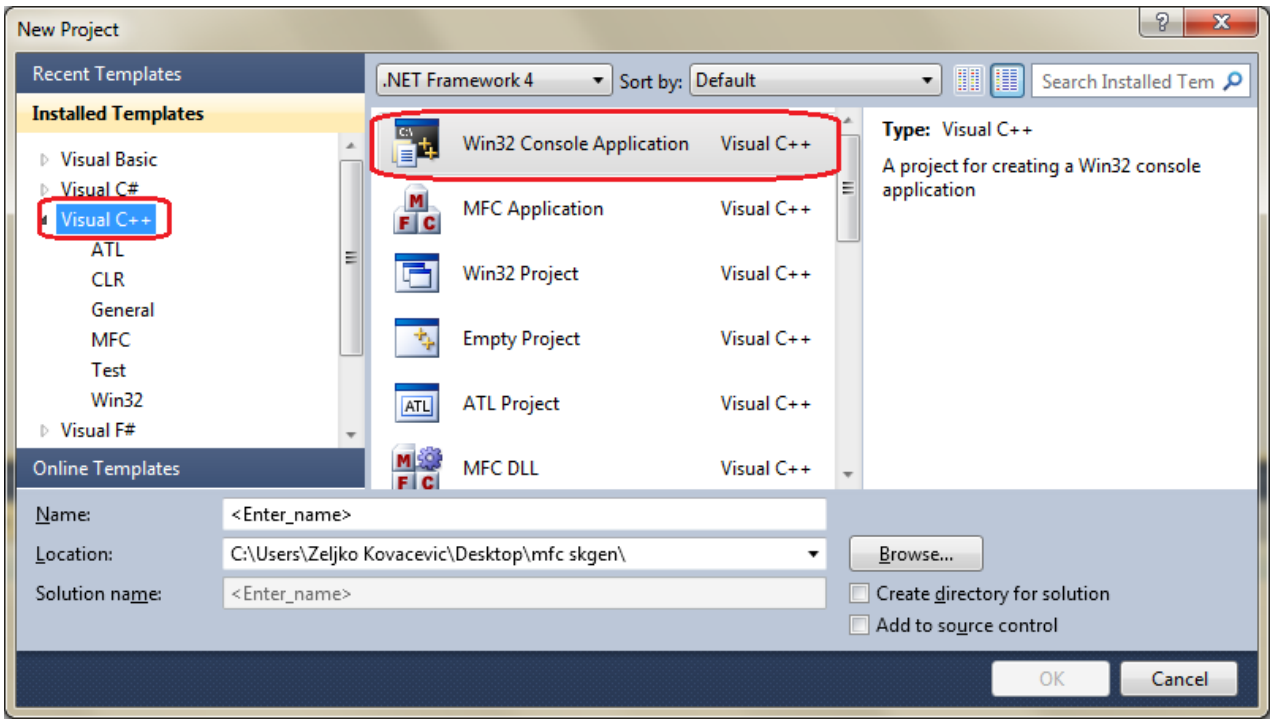
Autori
Tehničko veleučilište u Zagrebu, 2021.g.

1. Uvod u klase

1.1. Visual Studio

Za pisanje C++ programa koristiti ćemo MS Visual Studio, pa je stoga bitno da poznajemo barem osnove ovog razvojnog okruženja. Pod time se misli na kreiranje aplikacije komandne linije te organizaciju programskog koda u projektu.

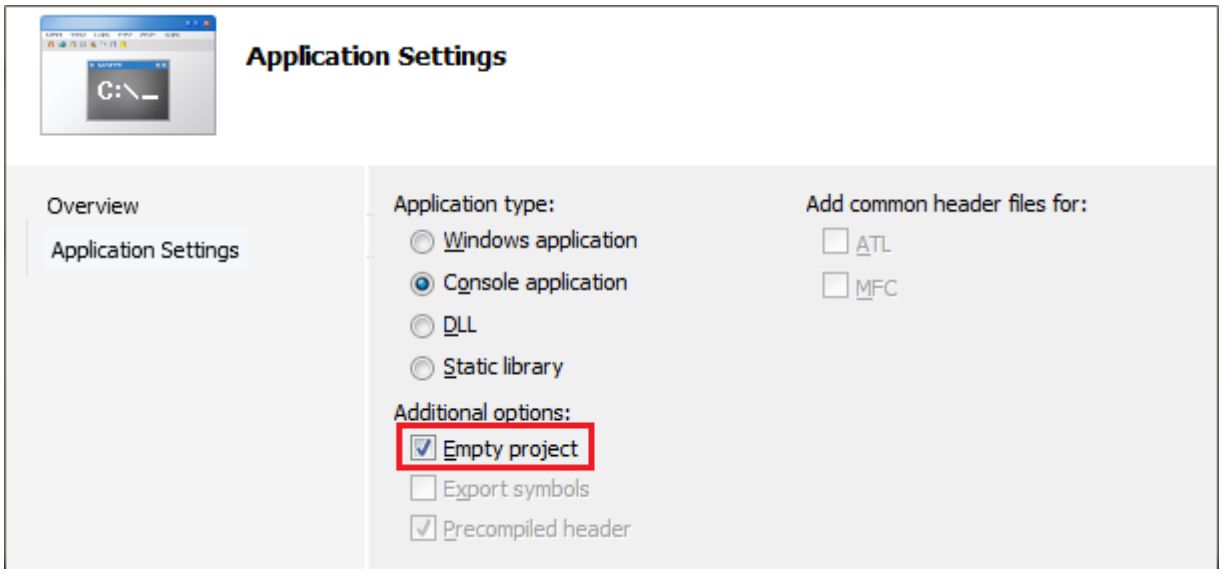
Da bi kreirali aplikaciju komandne linije potrebno je kreirati *Win32 Console Application* projekt. To radimo na način da odaberemo ***File/New/Project...***. Prikazuje se sljedeći dijalog (slika).



Slika 1.1.1. Kreiranje aplikacije komandne linije

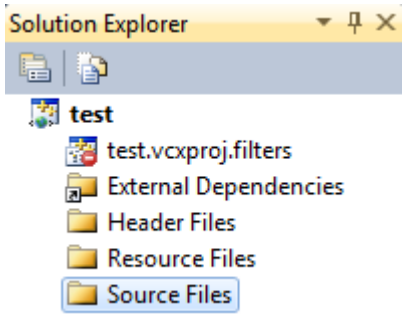
Odaberite postavke kao na slici te odredite ime projekta i njegovu lokaciju. Nakon toga kliknite OK. Pojaviti će vam se **Win32 Application Wizard** dijalog.

Kliknite **next**, nakon čega se pojavljuje sljedeći dijalog:



Slika 1.1.2. Dijalog „Application Settings“

Pri kreiranju aplikacije komandne linije želimo samostalno dodavati nove datoteke u projekt a ne koristiti predefinirane postavke Visual Studija, pa je stoga potrebno odabrati postavke kao na slici (kvačica na „Empty project“). Kliknite "Finish", nakon čega bi se trebao prikazati **Solution Explorer** kao na slici:



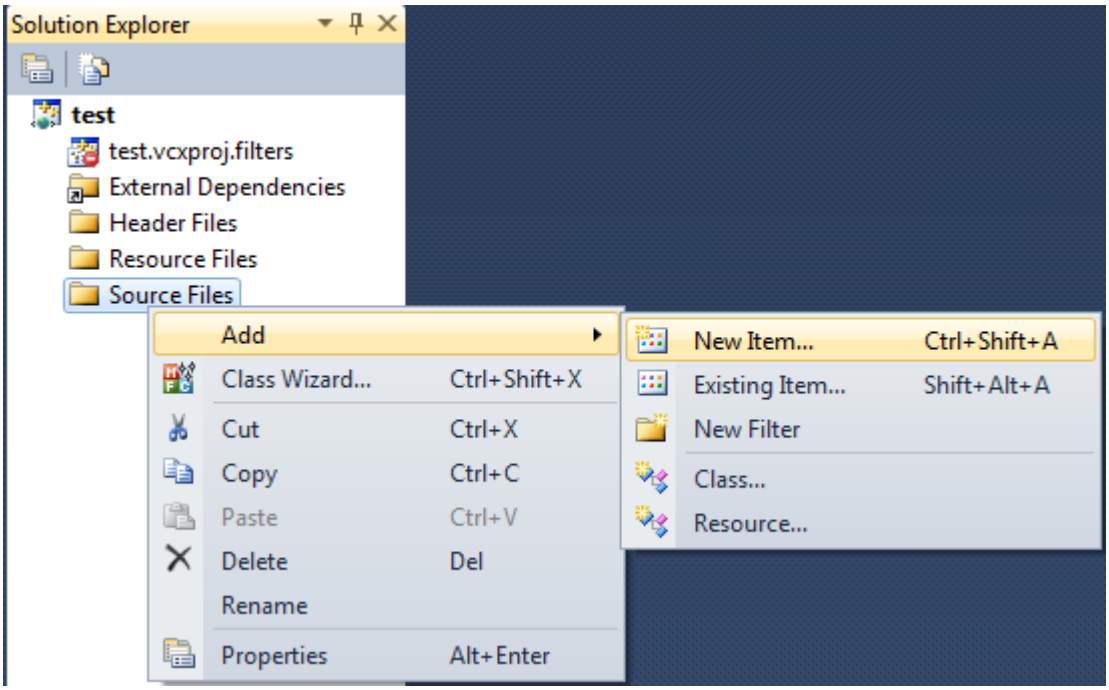
Slika 1.1.3. Solution Explorer

Pravilno organizirati projekt među-ostalim znači i imati pravilnu organizaciju datoteka unutar projekta. Zato u *Solution Explorer*-u imamo tri glavne skupine datoteka: *Source Files*, *Header Files* i *Resource Files*.

U skupini *Header files* trebaju se nalaziti datoteke zaglavlja (.h), a unutar njih definicije struktura i klasa te prototipi metoda i funkcija. Skupina *Source Files* podrazumijeva datoteke s .cpp ekstenzijom. U tim datotekama pišemo programsku implementaciju (tijela funkcija i metoda).

Projekt je uvijek potrebno organizirati na ovakav način, a ne pisati definiciju i implementaciju programskog koda u samo jednoj datoteci. To doprinosi nečitljivost i nepreglednosti projekta, a kasnije stvara i probleme pri prijenosu programskog koda u statičke i dinamičke biblioteke.

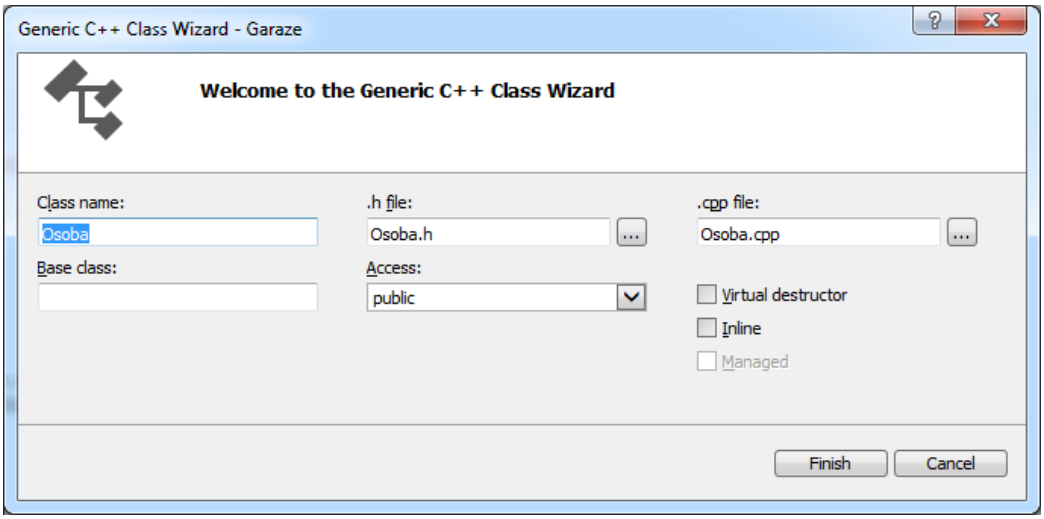
U bilo koju od navedenih skupina datoteka moguće je dodati već postojeću ili novu datoteku. To radimo na način da desnim klikom na željenu skupinu odabiremo stavku "Add" (Slika 1.1.4).



Slika 1.1.4. Dodavanje datoteka u projekt

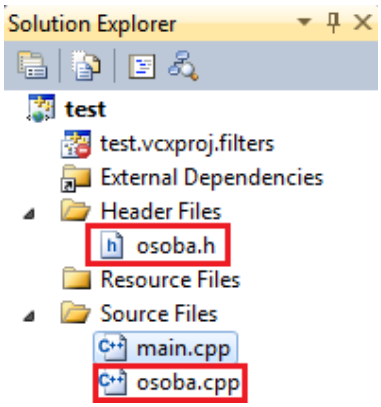
Odabirom stavke *Add / New Item…* možemo dodati novu datoteku u projekt. Kako svaki C++ program mora imati funkciju *main* za nju možemo na ovaj način kreirati datoteku *main.cpp*. Ona će tada automatski biti dodana kao dio projekta.

Stavku *Add* možemo iskoristiti i za dodavanje novih klasa (*Add / Class…*). Primjerice, recimo da želimo napisati klasu *Osoba*. Svaka klasa treba imati svoju datoteku zaglavlja (*osoba.h*) te implementacijsku datoteku (*osoba.cpp*). Umjesto da svaku od ovih datoteka kreiramo posebno korištenjem stavke *Add / New Item…* možemo obje datoteke kreirati automatski upotrebom *Add / Class…*



Slika 1.1.5. Dodavanje nove klase u projekt

Primijetite da se na slici 1.1.5 odmah nakon unosa imena klase *Osoba* automatski za nju generiraju datoteke *Osoba.h* i *Osoba.cpp*. Sukladno tome u projektu sada imamo 3 datoteke (Slika 1.1.6).



Slika 1.1.6. Sadržaj projekta nakon dodavanja klase Osoba

U datoteci *osoba.h* napisati ćemo definiciju klase, njenih atributa i metoda, a u datoteci *osoba.cpp* implementaciju metoda (njihova tijela).

1.2. Osnove klasa

Prethodno smo kreirali klasu *Osoba*, no pitanje je što klasa općenito predstavlja. Ukratko, klasa predstavlja opis novog tipa podatka, a taj opis se nalazi u datoteci zaglavlja klase (.h datoteci).

Osoba.h (definicija klase i prototipi metoda)

```
class Osoba{
public:
	// podatkovni članovi (atributi)
	int starost;
	char ime[20];
	char prezime[20];

	// metoda
	void ispis();
};
```

Deklaracija klase može vrlo lako podsjećati na deklaraciju strukture u programskom jeziku C. Međutim, klase i strukture se razlikuju po dvije stvari. Klase podrazumijevano koriste privatno pravo pristupa svojim članovima, dok C strukture javno pravo pristupa. Stoga, unutar klase *Osoba* smo eksplicitno ključnom riječi *public* definirali javno pravo pristupa za članove naše klase. Također, C strukture ne mogu kao svoje članove imati funkcije (metode). To je tek moguće u C++u.

U definiciji klase moguće je vidjeti od čega se ona sastoji tj. od podatkovnih članova (atributa) te metoda (funkcija). U datoteci zaglavlja nalaze se tek prototipi metoda, dok njihova tijela (implementaciju) pišemo u .cpp datoteci.

Osoba.cpp (implementacija)

```
#include <iostream>
#include "osoba.h" // zaglavlje klase Osoba

void Osoba::ispis(){
	std::cout << ime << " " << prezime << " ima " << starost << " godina";
}
```

Da bi napisali implementaciju metode *ispis* prevoditelj prvo mora znati gdje se nalazi njen prototip. Zato je prethodno potrebno uključiti datoteku zaglavlja klase *Osoba* pretprocesorskom naredbom *include*.

Primijetite kako je napisana metoda *ispis*. Prvo je naveden njen povratni tip te naziv klase kojoj pripada. Tek nakon operatora dosega „::“ navodi se ime metode i popis parametara.

```
retVal ClassName::MethodName(params…){
	// function body
}
```

Iako smo rekli da je pravilno razdvajati definiciju klase i njenu implementaciju kroz .h i .cpp datoteke isto tako je sve moguće napisati u .h datoteci. Npr.

Osoba.h (klasa sadrži inline metodu)

```
class Osoba{
public:
	// podatkovni članovi (atributi)
	int starost;
	char ime[20];
	char prezime[20];

	// inline metoda!
	void ispis(){
		std::cout << ime << " " << prezime << " ima " << starost << " godina";
	}
};
```

Ukoliko klasa u svojoj definiciji sadrži i tijelo metode onda se ta metoda implicitno deklarira kao tipa *inline*. Metode ovog tipa se kopiraju na mjesto gdje se treba dogoditi njihov poziv. Time se štedi na vremenu jer su pozivi metoda brži. Međutim, zahtjevi za memorijom su veći jer ukoliko 100 puta pozovemo *inline* metodu tada će postojati i 100 njenih kopija. Zato se preporučuje da *inline* metode budu jako male, a u krajnjem slučaju i sam prevoditelj može odbiti *inline* specifikaciju ukoliko smatra da je riječ o prevelikoj metodi.

Sada kada konačno imamo deklaraciju i implementaciju klase *Osoba* možemo prikazati primjer upotrebe.

Main.cpp (glavni program)

```
#include <iostream>
#include "osoba.h" // zaglavlje klase Osoba
using namespace std;

int main(){
	// deklaracija objekta (instance)
	Osoba Ante;

	// inicijalizacija podatkovnih članova
	strcpy(Ante.ime, "Ante");
	strcpy(Ante.prezime, "Antic");
	Ante.starost = 20;

	// poziv metode ispis
	Ante.ispis();
	return 0;
}
```

Kao i u slučaju rada s C strukturama potrebno je kreirati objekt tipa *Osoba*. Da bi to mogli opet moramo upotrebom pretprocesorske naredbe *include* prevoditelju reći gdje se nalazi definicija tog objekta (klase). Tek nakon toga u funkciji *main* možemo napraviti jedan objekt (instancu) tog tipa.

```
// deklaracija objekta (instance)
Osoba Ante;
```

Ovdje treba razlikovati klasu (*Osoba*) od objekta (*Ante*) jer klasa predstavlja tek opis objekta (ne zauzima mjesto u memoriji), dok objekt je fizička tvorevina u memoriji.

Članovima klase pristupamo na isti način kao i članovima strukture (pomoću točke):

```
Ante.starost = 20;  // pristup atributu
Ante.ispis();       // poziv metode
```

Međutim, ukoliko je objekt *Ante* deklariran kao pokazivač onda bi pristupali pomoću strelice.

```
Ante->starost = 20;       // pristup atributu
Ante->ispis();            // poziv metode
```

Više o statičkim i dinamički alociranim objektima će biti u sljedećoj temi.

2. Funkcije, metode i pokazivači

2.1. Konstruktor i destruktor klase

Prilikom kreiranja objekta neke klase poziva se njegov konstruktor. To je specijalna metoda koja ima isto ime kao i klasa, nema povratnu vrijednost te služi za inicijalizaciju objekta. Također, konstruktorom možemo definirati da li su za kreiranje objekta potrebni ulazni parametri ili ne. Primjerice,

```
class Osoba{
public:
	// …
};
…
Osoba Ante; // Ante je inicijaliziran podrazumijevanim konstruktorom
```

Svaka klasa uvijek ima barem jedan konstruktor, a on se zove *podrazumijevani konstruktor* (eng. d*efault constructor*). Za klasu *Osoba* on bi glasio na sljedeći način.

```
Osoba(){}; // podrazumijevani konstruktor
```

Podrazumijevani konstruktor postoji samo u slučaju ako nismo definirali niti jedan drugi konstruktor. Tada je on automatski implicitno dodan za svaku takvu klasu. On omogućuje da se objekt može kreirati bez ikakvih dodatnih parametara. Međutim, ukoliko sami napišemo neki drugi konstruktor tada podrazumijevani konstruktor više ne postoji.

```
class Osoba{
public:
    string ime;
    string prezime;
    // konstruktor s parametrima
    Osoba(string _ime, string _prezime){
        ime = _ime;
        prezime = _prezime;
    }
};

…

Osoba Ivan("Ivan", "Ivanov"); // konstruktor s parametrima
Osoba Ante; // greška! podrazumijevani konstruktor više ne postoji!
```

Sada klasa *Osoba* sadrži konstruktor s dva parametra. Zato prevoditelj neće implicitno dodati podrazumijevani konstruktor već će zahtijevati da se svaki objekt tipa *Osoba* kreira pomoću dodatna dva parametra (ime i prezime). Rezultat toga je pogreška prevoditelja pri kreiranju objekta *Ante*. Da bi mogli kreirati objekt *Ante* moramo mu dodati ta dva parametra ili sami eksplicitno napisati podrazumijevani konstruktor klase *Osoba*. Tada bi klasa *Osoba* imala dva konstruktora tj. preopterećenje (više u nastavku).

Treba napomenuti da smo tijelo konstruktora mogli napisati na više načina. Prethodno smo napisali

```
Osoba(string _ime, string _prezime){
	ime = _ime;
	prezime = _prezime;
}
```

Ulazni parametri *_ime* i *_prezime* se kopiraju u članske varijable *ime* i *prezime*. Ulazni parametri i članske varijable klase *Osoba* imaju različito ime, pa ih je lako razlikovati. No, ovaj konstruktor smo mogli napisati i na sljedeći način.

```
Osoba(string ime, string prezime){
	this->ime = ime;
	this->prezime = prezime;
}
```

Sada ulazni parametri konstruktora imaju isto ime kao i članske varijable klase *Osoba*. Stoga, koristimo pokazivač *this* koji će naglasiti kada koristimo članske varijable *ime* i *prezime*, a kada parametre konstruktora s tim imenom. Primjerice,

```
this->ime = ime;
```

Izrazom *this->ime* se adresiramo na člansku varijablu *ime*, dok s desne strane operatora pridruživanja je riječ o parametru konstruktora s nazivom *ime*. Da bi napisali prethodni konstruktor s parametrima možemo koristiti i inicijalizacijsku listu.

```
Osoba(string _ime, string _prezime) : ime(_ime), prezime(_prezime){}
```

Sada se inicijalizacija članova klase izvršava prije ulaska u samo tijelo konstruktora.

Konstruktori se izvršavaju pri kreiranju objekta, dok se pri njihovom uništenju izvršavaju destruktori. To su također specijalne funkcije koje nemaju povratnu vrijednost ali niti ulazne parametre. Imaju isto ime kao i klasa s znakom, ali s znakom „~“ ispred imena. Primjerice, destruktor za klasu *Osoba* bi glasio *~Osoba()*.

2.2. Preopterećenje funkcije

Još jedno poboljšanje u odnosu na programski jezik C se vidi u mogućnosti preopterećenja funkcija (metoda). U programskom jeziku C++ preopterećenje funkcije je slučaj kada postoji više funkcija s istim imenom ali različitim setom parametara. Ti parametri su različitog tipa ili različitog broja. Npr:

```
class Clock{
private:
	int m_hours;
	int m_mins;
	int m_secs;
public:
	// preopterećenja metode setTime
	void setTime( int hours, int mins, int secs );
	void setTime( int hours, int mins);
	void setTime( int hours);
};
```

Primijetite kako se metoda *setTime* može pozvati na tri različita načina tj. na način da joj se predaju samo sati, sati i minute, ili sve. Također, i konstruktor se može preopteretiti;

```
class Clock{
private:
	int m_hours;
	int m_mins;
	int m_secs;
public:
	// (podrazumijevani) konstruktor bez parametara
	Clock();
	//Konstruktor s parametrima
	Clock(int hours , int mins , int secs);
};
```

Kao i u slučaju s prethodnom klasom *Osoba*, čim smo napisali konstruktor s parametrima podrazumijevani konstruktor (bez parametara) više ne postoji. Stoga, da bi mogli kreirati objekt tipa *Clock* bez parametara moramo eksplicitno napisati podrazumijevani konstruktor. Na ovaj način konstruktor možemo proizvoljno puta preopteretiti.

Destruktor klase nije moguće preopteretiti jer je on jedinstven (nikada nema povratnu vrijednost niti ulazne parametre). Pogledajmo još jedan primjer preopterećenja konstruktora:

```
class Tocka{
public:
	float x, y;
	Tocka(float a, float b) { x = a; y = b; }
};
class Pravac{
public:
	float k, l;
	// preopterećenje konstruktora
	Pravac(float koeficijent, float odsjecak);
	Pravac(Tocka A, Tocka B);
	void Jednadzba();
};
Pravac::Pravac(float koeficijent, float odsjecak){
	k = koeficijent;
	l = odsjecak;
}
Pravac::Pravac(Tocka A, Tocka B){
	k = (B.y - A.y) / (B.x - A.x);
	l = -k * A.x + A.y;
}
void Pravac::Jednadzba(){
	cout << "Y = " << k << "x + " << l << endl;
}
int main(){
	Tocka A(1, 1), B(2, 8);
	Pravac PrviPravac(1, 4);  // koeficijent smjera i odsjecak na osi x
	Pravac DrugiPravac(A, B); // pravac definiran dvjema tockama u ravnini
	PrviPravac.Jednadzba();   // y = 1x + 4
	DrugiPravac.Jednadzba();  // y = 7x - 6
	return 0;
}
```

Navedeni program može kreirati objekte tipa *Pravac* u dva slučaja:

- Ako je poznat koeficijent smjera pravca i odsječak na koordinatnoj osi x (prvi oblik konstruktora klase *Pravac*).
- Ako su poznate dvije točke kojima prolazi pravac (drugi oblik konstruktora klase *Pravac*).

Zbog drugog slučaja smo kreirali klasu *Tocka* da bi mogli kreirati dva objekta koja predstavljaju dvije točke u koordinatnom sustavu. Njihove podatkovne članove *x* i *y* smo u drugom obliku konstruktora klase *Pravac* upotrijebili kako bi izračunali koeficijent smjera pravca i odsječak na koordinatnoj osi x.

2.3. Statički i dinamički objekti

Konstruktor i destruktor objekta se izvršavaju za svaki kreirani objekt. Ipak, postoje dva osnovna tipa objekata tj. statički i dinamički alocirani. Statički se nalaze na stog-u (eng. *stack*). Stog je memorija malog kapaciteta pogodna za manje privremene objekte. Štoviše, ukoliko pokušate preveliki objekt staviti na stog dobiti ćete grešku poput „stack overflow“.

```
int p[99999999]; //  statičko polje p
```

Ukoliko pokušate kreirati statičko polje ove veličine to vjerojatno nećete uspjeti upravo zbog ograničenosti stog-a. Da bi to napravili potrebno je *p* deklarirati kao pokazivač.

```
int *p; // pokazivač na polje
if ((p = new int[99999999]) == NULL){
	cout << "Nema dovoljno memorije!";
		return -1;
}
...
delete[] p; // dealokacija niza
```

Objekt *p* sada nije statičko polje već pokazivač koji će naknadno biti dinamički alociran upotrebom operatora *new*. Dinamički alocirani objekti se moraju ručno dealocirati nakon što više nisu u upotrebi. To radimo upotrebom operatora *delete*.

Dinamički objekti se nalaze u memorijskoj strukturi *heap*. Veličina te memorijske strukture je mnogo veća i najčešće ovisi o dostupnoj količini RAM memorije u računalu. Zato je preporuka sve veće objekte stavljati upravo na *heap*.

Da bi demonstrirali rad sa statički i dinamički alociranim objektima pogledajmo sljedeći primjer klase *Test*.

```
class Test{
public:
	Test(){
		cout << "Objekt kreiran\n";
	}
	~Test(){
		cout << "Objekt uništen\n";
	}
};
```

Kreirajmo statički i dinamički alocirani objekt ove klase.

```
Test Obj1; // statički objekt
Test *pObj2 = new Test; // dinamički alocirani objekt
// ...
delete pObj2; // dealokacija dinamički alociranog objekta
```

Ispis programa:

Objekt kreiran
Objekt kreiran
Objekt uništen
Objekt uništen

Klasa *Test* ima konstruktor koje se pokreće pri kreiranju objekta i destruktor koji se izvršava pri njegovom uništenju. Kako imamo dva objekta (jedan statički i drugi dinamički alocirani) jasno je zašto imamo četiri ispisa. Međutim, što se točno dogodilo pri izvršenju ovog programa?

```
Test Obj1; // statički objekt
```

Prvo smo kreirali statički objekt. Prilikom kreiranja ovog objekta poziva se njegov konstruktor pa se stoga ispisuje „Objekt kreiran“.

```
Test *pObj2 = new Test; // dinamički alocirani objekt
```

U drugoj liniji smo kreirali dinamički objekt tj. napravili smo njegovu dinamičku alokaciju. Tek nakon što je objekt uspješno alociran operatorom *new* poziva se konstruktor tog objekta te se drugi put ispisuje „Objekt kreiran“.

```
delete pObj2; // dealokacija dinamički alociranog objekta
```

Dinamički alocirane objekte programer uvijek mora sam obrisati (dealocirati) operatorom *delete* ili bi se u protivnom dogodilo curenje memorije (eng. *memory leak*). Stoga, pri izvršenju ove naredbe poziva se destruktor objekta *pObj2* te se ispisuje „Objekt uništen“.

Prilikom izlaska iz (pot)programa aplikacija će sama počistiti sve statičke objekte sa stoga. Zato se pri brisanju statičkog objekta *Obj1* pokreće njegov destruktor te se opet ispisuje „Objekt uništen“.

Iz ovog primjera možemo zaključiti nekoliko stvari;

- Statički objekti se u pravilu kreiraju prvi a uništavaju zadnji
- Dinamički alocirani objekti se kreiraju naknadno, a uništavaju prije statičkih

2.4. Pametni pokazivači

Dugo je vremena sakupljač smeća (eng. *garbage collector*) bio jedna od prednosti programskih jezika poput C# i Jave. Svojevremeno, i C++ je razvijao svoju alternativu u smislu pametnih pokazivača. Oni bi nakon izlaska iz dosega automatski dealocirali zauzetu memoriju, čak i u slučajevima kada bi se dogodile iznimke. Sada programer više ne mora misliti o dealokaciji dinamički alociranih objekata što uvelike olakšava pisanje aplikacija. Da bismo razumjeli pametne pokazivače prvo je potrebno shvatiti koncept vlasništva jer ovisno o tome C++11 nudi više različitih tipova pametnih pokazivača.

```
int *p = new int;
…
delete p;
```

Za dinamičku alokaciju memorije do sada smo najčešće koristili ovakav oblik koda pomoću operatora *new* i *delete*. Ovdje programer mora voditi brigu o dealokaciji memorije novog objekta te se stoga može reći da je upravo programer njegov vlasnik. Ali ako smo za dinamičku alokaciju objekta koristili pametni pokazivač, tada programer više nije vlasnik tog novog objekta jer brigu o njegovoj dealokaciji preuzima upravo taj pametni pokazivač.

```
#include <memory>
std::auto_ptr<int> p(new int); // p postaje vlasnik novog objekta
```

Prvi pokušaj realizacije bio je pomoću *auto_ptr* pametnog pokazivača. Svaki pametni pokazivač tog tipa je bio vlasnik jednog dinamički alociranog objekta o kojemu je vodio brigu. Štoviše, nije se moglo dogoditi da dva pametna pokazivača imaju vlasništvo nad istim objektom jer se tada ne bi znalo tko je zadužen za njegovu dealokaciju. Primjerice,

```
auto_ptr<int> p(new int);
auto_ptr<int> p2 = p; // p2 sada postaje vlasnik objekta!
*p = 1; // nedefinirano ponašanje!
```

Pokušate li prevesti ovaj dio programskog koda vaš prevoditelj neće javiti grešku, dok će neki prevoditelji javiti tek upozorenje da je korištenje *auto_ptr* pametnog pokazivača zastarjelo. O čemu se zapravo radi?

Prvi pametni pokazivač *p* dinamički je alocirao objekt tipa *int* te je on njegov isključivi vlasnik. Već u drugoj liniji koda kreira se drugi pametni pokazivač *p2* koji koristi semantiku kopiranja kako bi preuzeo vlasništvo prethodno alociranog objekta. Čim je *p2* preuzeo vlasništvo nad tim objektom, zadnja linija koda predstavlja nedefinirano ponašanje jer *p* nije više vlasnik nikakvog objekta.

Ovdje vidimo koliko je ovaj kod opasan i nesiguran, a pogotovo pri radu s kontejnerima koji bi interno izvršavali operacije kopiranja elemenata radi sortiranja, umetanja itd. ukoliko bi elementi bili upravo *auto_ptr* pametni pokazivači. Izvršavanjem tih operacija pomoću semantike kopiranja kontejneri bi implicitno mijenjali i vlasnike objekata čime bi kod postao potpuno nepredvidiv i nesiguran.

Sve ovo događa se jer je pametni pokazivač *auto_ptr* pisan u vrijeme kada C++ nije imao semantiku prijenosa te je koristio isključivo semantiku kopiranja za prijenos vlasništva. Danas C++11 donosi i semantiku prijenosa te novi pametni pokazivač koji u potpunosti zamjenjuje *auto_ptr*. Riječ je o *unique_ptr* pametnom pokazivaču. Nakon njegovog uvođenja korištenje *auto_ptr* pametnog pokazivača smatra se zastarjelim te se više ne preporučuje njegova upotreba.

```
unique_ptr<int> p(new int);
unique_ptr<int> p2 = p; // greška prevoditelja!
```

Iz ovog primjera vidljivo je kako *unique_ptr* uopće ne dopušta korištenje semantike kopiranja za prijenos vlasništva, već da bi to bilo moguće mora se koristiti semantika prijenosa pomoću funkcije *move*.

```
unique_ptr<int> p2 = move(p); // ispravan prijenos vlasništva!
```

Stoga, korištenje *unique_ptr* pametnog pokazivača puno je sigurnije jer će prevoditelj pri ovakvom pokušaju javiti grešku te neće dopustiti nedefinirana ponašanja kao u slučaju korištenja *auto_ptr* pametnog pokazivača.

Ovaj pametni pokazivač ima i druge pogodnosti, poput dopuštanja dinamičke alokacije niza objekata. To prethodno nije bilo moguće upotrebom *auto_ptr* pametnog pokazivača jer bi on implicitno uvijek pozivao operator *delete* umjesto *delete[]* kada bi to bilo potrebno.

```
unique_ptr<int[]> niz(new int[10]);
```

Korištenjem *unique_ptr-a* sigurni smo da će se pri dealokaciji osloboditi cijeli niz, tj. pozvati *delete[]*. Naravno, ukoliko je alociran tek jedan objekt automatski će biti pozvan samo *delete*.

I sama dealokacija može se dodatno specificirati korištenjem *deleter* izraza. Naime, *unique_ptr* pametni pokazivači podržavaju mogućnost da programer sam definira funkcijski objekt koji bi bio zadužen za dealokaciju objekta. Primjerice,

```
class MojDeleteObjekt{
public:
	template <class T>
	void operator()(T* p) {
		delete p;
		cout << "Objekt je dealociran!\n";
	}
};
…
unique_ptr<int, MojDeleteObjekt> niz(new int);
```

ili

```
MojDeleteObjekt del;
unique_ptr<int, MojDeleteObjekt&> niz(new int, del);
```

U oba ova slučaja koristimo vlastite funkcijske objekte koji će osim dealokacije objekta moći obaviti i neke dodatne operacije ukoliko je to potrebno.

Također treba napomenuti da unatoč pogodnostima pametnih pokazivača većina ugrađenih funkcija koristi upravo obične pokazivače. Stoga se često javlja potreba da se iz pametnog pokazivača dohvati njegov interni (obični) pokazivač na alocirani objekt.

```
void inkrement(int* p){
	(*p)++;
}
```

Pitanje je kako ovoj funkciji predati pametni pokazivač, tj. njegov interni pokazivač koji pokazuje na alocirani objekt. To možemo postići na sljedeći način:

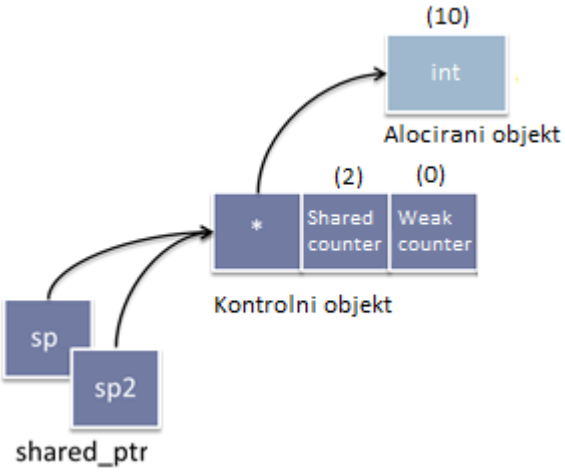
```
unique_ptr<int> n(new int(1));
inkrement(n.get()); // get() vraća interni pokazivač na objekt
cout << *n; // 2  (dereferenciranje pametnog pokazivača)
```

Metodom *get()* iz pametnog pokazivača *n* dohvaćamo njegov interni (obični) pokazivač na alocirani objekt. Taj interni pokazivač onda prosljeđujemo funkciji *inkrement*. Ipak, u ovakvim slučajevima treba biti na oprezu ukoliko nismo sigurni što dotična funkcija radi. Primjerice, takva funkcija može dealocirati memoriju na koju pokazuje pokazivač te time učiniti pametni pokazivač beskorisnim.

Iako se *unique_ptr* danas smatra najpogodnijim za korištenje postoji i druga alternativa. Primjerice, *unique_ptr* garantira da postoji samo jedan vlasnik alociranog objekta. Dozvoljeno je tek prenositi to vlasništvo na drugi *unique_ptr* pokazivač upotrebom semantike prijenosa. No postoje i situacije gdje nas to u potpunosti ne zadovoljava, odnosno upravo kada trebamo mogućnost da jedan objekt ima više vlasnika. Zbog toga C++11 nudi i *shared_ptr* pametni pokazivač.

```
shared_ptr<int> sp(new int(10));
shared_ptr<int> sp2 = sp; // sp2 postaje još jedan vlasnik objekta!
cout << *sp2; // 10
```

Alocirani objekt sada ima dva vlasnika. Ovakva situacija ne bi bila dopuštena upotrebom *unique_ptr* pokazivača jer bi prevoditelj zahtijevao prijenos vlasništva funkcijom *move*.



Slika 2.4.1. Približni memorijski raspored za prethodni primjer

Proces započinje dinamičkom alokacijom željenog objekta nakon čega se pokreće konstruktor *shared_ptr* objekta *sp*. Njime se stvara kontrolni objekt koji pokazuje na alocirani objekt. Kontrolni objekt interno prati koliko drugih *shared_ptr* i *weak_ptr* pokazivača koristi alocirani objekt te sukladno tome povećava ili smanjuje njihove brojače.

Tako se kreiranjem pametnog pokazivača *sp* povećava brojač *shared_ptr* objekata (*shared counter*) za 1. Nakon kreiranja drugog pametnog pokazivača *sp2* koji ima vlasništvo nad istim alociranim objektom taj brojač se opet povećava za 1 te trenutno iznosi 2. Ali kada jedan od tih pametnih pokazivača izađe iz dosega tada se i njihov brojač umanjuje za 1. Konačno, kada brojač *shared counter* poprimi vrijednost 0 (svi *shared_ptr*-i su izašli iz dosega) događa se uništenje alociranog objekta (dealokacija).

Treba napomenuti da uništenje alociranog objekta ne znači nužno i uništenje kontrolnog objekta. On će i dalje postojati sve dok brojač *weak counter* također ne poprimi vrijednost 0. Bez obzira na uništenje alociranog objekta, kontrolni objekt treba pružiti informaciju *weak_ptr* pokazivačima o postojanosti alociranog objekta. Zato, sve dok postoji barem jedan *weak_ptr* pokazivač koji je bio vezan za jedan od prethodnih *shared_ptr* pokazivača kontrolni objekt će postojati. O *weak_ptr* pokazivačima bit će riječ u nastavku.

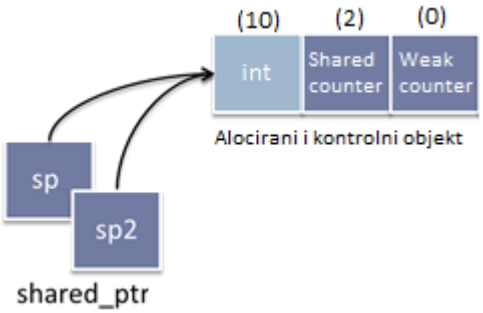
U prethodnom primjeru vidjeli smo što se događa nakon izvršenja sljedeće linije koda:

```
shared_ptr<int> sp(new int(10)); // dvostruka alokacija!
```

Dogodila se dvostruka alokacija memorije. Prva alokacija je pri pozivanju operatora *new*, a druga pri pozivanju *shared_ptr* konstruktora koji će kreirati kontrolni objekt (pogledati sliku 3). Da bismo to izbjegli možemo koristiti *make_shared* funkciju.

```
shared_ptr<int> sp(make_shared<int>(10)); // samo jedna alokacija!
```

Funkcija *make_shared* samo će u jednoj alokaciji kreirati traženi objekt i njegov kontrolni objekt. U memoriji bi to izgledalo na sljedeći način:



Slika 2.4.2. Korištenje make_shared funkcije

Upravo iz razloga što dinamičke alokacije memorije najduže traju, preporuča se korištenje *make_shared* funkcije kao dodatnog načina optimizacije.

Broj objekata	shared_ptr (ms)	make_shared (ms)
100.000	46	31
1.000.000	422	390
2.000.000	843	702
5.000.000	2012	1701
10.000.000	4068	3432
20.000.000	7921	6443

Tablica 2.4.1. Rezultati testiranja pri kreiranju shared_ptr objekata

Tablicom su prikazani rezultati testiranja pri kreiranju određenog broja objekata upotrebom *shared_ptr* predloška i *make_shared* funkcije. Rezultati mogu varirati u ovisnosti o konfiguraciji na kojoj se testiranje izvodi, no u ovom slučaju mogu se primijetiti i preko 20% bolje performanse upotrebom *make_shared* metode. Jasno, razlog boljim performansama metode *make_shared* je samo jedna memorijska alokacija. Za testiranje je korišten Clang (64 bit) prevoditelj te C++ Builder XE6 razvojno okruženje.

Jedini ozbiljniji problem koji se može javiti pri korištenju *shared_ptr* pametnih pokazivača cirkularne su ovisnosti. Primjerice, dva ili više *shared_ptr* pokazivača mogu jedan drugog „održavati na životu“ te time u konačnici uzrokovati curenje memorije (eng. *memory leak*).

```
class Osoba{
public:
	string ime;
	shared_ptr<Osoba> prijatelj;
	// konstruktor
	Osoba(string _ime) : ime(_ime){
		cout << ime << " je stvoren!\n";
	}
	// destruktor
	~Osoba(){
		cout << ime << " je unisten!\n";
	}
	// dodaj novog prijatelja (shared_ptr)
	void dodajPrijatelja(shared_ptr<Osoba> _prijatelj){
		prijatelj = _prijatelj;
	}
};
```

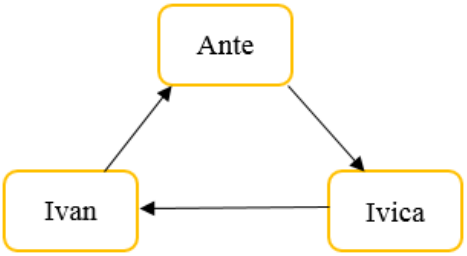
Za demonstraciju cirkularne ovisnosti kreirat ćemo 3 osobe pomoću *shared_ptr* pametnih pokazivača.

```
shared_ptr<Osoba> Ante(make_shared<Osoba>("Ante"));
shared_ptr<Osoba> Ivica(make_shared<Osoba>("Ivica"));
shared_ptr<Osoba> Ivan(make_shared<Osoba>("Ivan"));
```

Svaka od kreiranih osoba može biti prijatelj s drugom osobom. Primjerice,

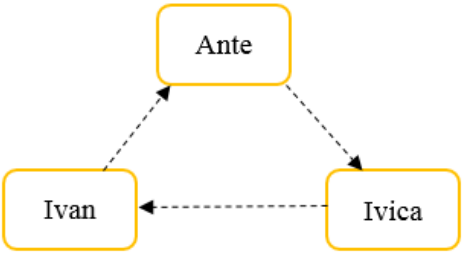
```
// cirkularne reference
Ante->dodajPrijatelja(Ivica); // Ante -> Ivica
Ivica->dodajPrijatelja(Ivan); // Ivica -> Ivan
Ivan->dodajPrijatelja(Ante);  // Ivan -> Ante
```

Izvršavanjem ovog programskog koda nikada se neće pokrenuti niti jedan od destruktora za 3 alocirana objekta. To je upravo zato jer jedan drugog „održavaju na životu“ zbog međusobne čvrste veze *shared_ptr* pokazivačima.



Slika 2.4.3. Cirkularna ovisnost pomoću shared_ptr pokazivača

Zbog ovakvih situacija uvedeni su *weak_ptr* pokazivači. Njima se definira „slaba veza“ između *shared_ptr* pokazivača, odnosno prekida se cirkularna ovisnost. Za *weak_ptr* pokazivače možemo reći da tek promatraju objekt ne utječući na njegov životni vijek. Sukladno tome, ukoliko podatkovni član *shared_ptr<Osoba> prijatelj* deklariramo kao *weak_ptr<Osoba> prijatelj* imat ćemo sljedeću situaciju:



Slika 2.4.4. Veze pomoću weak_ptr pokazivača

Nakon što sada izvršimo gornji programski kod uredno će se izvršiti destruktori za sve 3 kreirane osobe.

S obzirom da *weak_ptr* pokazivačima ne definiramo čvrstu vezu među *shared_ptr* objektima nikada ne znamo je li objekt na koji *weak_ptr* pokazuje još uvijek živ. Ipak, to možemo provjeriti metodom *lock()*. Ova metoda stvara privremeni *shared_ptr* objekt. Ukoliko objekt na koji *weak_ptr* pokazuje nije živ tada je privremeni *shared_ptr* objekt prazan.

```
void Promatraj(std::weak_ptr<int> weak){
    std::shared_ptr<int> pom(weak.lock());
    if (pom) {
        std::cout << "Objekt postoji. Vrijednost je " << *pom << "\n";
    } else {
        std::cout << "Objekt vise ne postoji!\n";
    }
}
```

Funkcija *Promatraj* prima *weak_ptr* objekt, a zatim metodom *lock()* ispituje je li objekt na koji on pokazuje još uvijek živ.

```
int main(){
    std::weak_ptr<int> weak;
    {
        std::shared_ptr<int> shared(new int(10));
        weak = shared; // stvara se "slaba veza"
        Promatraj(weak); // shared objekt postoji!
    }
    Promatraj(weak); // shared objekt više ne postoji!
}
```

Treba primijetiti da *shared_ptr* objekt *shared* postoji u zasebnom bloku. Ukoliko ga u tom bloku promatramo pomoću *weak_ptr* pokazivača neće biti problema. Tek nakon izlaska iz tog bloka objekt *shared* izlazi iz dosega te je automatski dealociran, a to se detektira zadnjim pozivom funkcije *Promatraj*.

3. Enkapsulacija i imenovani prostor

3.1. Enkapsulacija i const metode

Enkapsulacija je jedno od četiri najvažnija svojstva programskog jezika C++. Njegovom upotrebom smo u mogućnosti zabraniti direktni pristup određenim podatkovnim članovima (atributima) klase. To se realizira na način da te podatkovne članove sakrijemo pod *private* pravom pristupa. Umjesto direktnog pristupa tom podatkovnom članu omogućava se indirektni pristup pomoću javnih *get* i *set* metoda.

Zašto bi to bilo potrebno? Naime, ukoliko je podatkovni član potrebno provjeriti (validirati) prije inicijalizacije ili ukoliko njegova inicijalizacija indirektno može utjecati na druge podatkovne članove onda se ne smije dozvoliti njegov direktni pristup. Primjerice,

```
class Clock{
public:
	int hour, min, sec;
};
```

Klasa *Clock* ima tri podatkovna člana s javnim pravom pristupa. Pogledajmo što se događa sa sljedećim kodom:

```
Clock budilica;
budilica.hour = 7;
budilica.min = 59;
budilica.sec = 60; // ??
```

Ovdje je problem jasno vidljiv jer sekunde su inicijalizirane na vrijednost 60. Umjesto toga sekunde bi trebalo postaviti na 0, a minute uvećati za 1. S obzirom da i minute tada imaju vrijednost 60 i njih je potrebno postaviti na 0, a sate uvećati za 1. Konačno vrijeme bi onda bilo točno 8:0:0.

Inicijalizacija podatkovnog člana *sec* se nije mogla prethodno provjeriti (validirati), te su stoga dopuštene ovakve logičke greške. Ukoliko upotrijebimo enkapsulaciju možemo ih spriječiti.

```
class Clock{
private:
```

```
	int hour, min, sec;
public:
	Clock(int h, int m, int s) : hour(h), min(m), sec(s){}
	// postavi vrijednost podatkovnog člana min
	void setMin(int newMin) {
		if (newMin >= 60){
			hour = hour + newMin / 60;
			hour = hour % 24;
		}
		min = newMin % 60;
	}
	// vrati vrijednost podatkovnog člana min
	int getMin() const{
		return min;
	}
	// ... slično za ostale atribute klase
};
```

Klasa *Clock* sada ne dopušta direktan pristup svojim podatkovnim članovima izvana, te u tu svrhu moramo koristiti njihove *set* i *get* metode. U ovom primjeru je napisana *set* i *get* metoda samo za podatkovni član *min*, no i za ostale je slično. Upotrijebimo sljedeći primjer programskog koda.

```
Clock budilica(8, 0, 0);
// budilica.min = 95; - min je sada nedostupan (privatan)!
budilica.setMin(95); // vrijeme je 9:35:0
```

Umjesto da direktno definiramo vrijednost podatkovnog člana *min* te time eventualno napravimo logičku pogrešku pozvana je metoda *setMin*. Ona će se pobrinuti da podatkovni član *min* ima vrijednost u intervalu 0-59, te ukoliko predani broj minuta prelazi taj interval da se i broj sati adekvatno poveća. Ovaj pristup uvelike olakšava posao programeru prilikom svake inicijalizacije podatkovnog člana *min*, a slično se može napraviti i za ostale podatkovne članove.

Metode tipa *set* su uvijek tipa *void* jer nikada ne trebaju ništa vratiti kao povratnu vrijednosti. Njihov posao je postaviti novu vrijednost (koja je predana kao argument metodi) u podatkovni član, pa stoga uvijek imaju ulazni parametar (novu vrijednost).

Metode tipa *get* nikada nisu tipa *void* jer one vraćaju trenutnu vrijednost podatkovnog člana. Iz istog razloga *get* metode nemaju potrebu za ulaznim parametrima. Ipak, treba primijetiti deklaraciju *const* prije tijela metode *getMin*.

```
int getMin() const
```

Deklaracijom *const* prije tijela metode *getMin* naglašavamo da konstantni objekti tipa *Clock* mogu pozvati metodu *getMin* jer ona neće mijenjati stanje tog objekta. Metode koje nemaju deklaraciju *const* se neće moći pozvati nad konstantnim objektima te klase iako možda u stvarnosti niti ne mijenjaju stanje objekta.

```
const Clock budilica(8, 0, 0);
cout << budilica.getMin();  // 0
budilica.setMin(45); // greška! objekt je konstantan!
```

Prevoditelj će javiti grešku pri prevođenju zadnje linije koda jer metoda *setMin* nema deklaraciju *const*. Netko se može dosjetiti pa napisati

```
void setMin(int newMin) const{
```

Niti ovaj trik neće uspjeti jer će prevoditelj primijetiti da se unutar metode *setMin* mijenjaju vrijednosti podatkovnih članova *min* i *hour*. Ukoliko je u nekim situacijama ipak potrebno da se mijenja podatkovni član konstantnog objekta onda taj podatkovni član mora biti deklariran kao *mutable*.

3.2. Prijenos argumenata

Još iz C jezika je poznato da funkciji možemo prenositi argumente na dva načina. Prvi je prijenos po vrijednosti (eng. *call by value*). Ovim načinom pri pozivu funkcije predajemo vrijednosti varijabli. Pozivajuća funkcija zatim kreira pomoćne varijable u koje kopira te vrijednosti, te ih na taj način dalje koristi u ostatku funkcije.

```
int suma(int a, int b){
	return a + b;
}
int main(){
	int x = 1, y = 2;
	int rez = suma(x, y); // 3
	return 0;
}
```

Varijable *x* i *y* predajemo funkciji *suma* koja njihove vrijednosti kopira u pomoćne varijable *a* i *b*. Programski kod je ispravan, no po pitanju optimizacije ovo i nije najbolje rješenje. Naime, zamislimo da se funkciji *suma* predaju dva velika objekta (ili više njih). Zbog predaje

po vrijednosti funkcija *suma* bi svaki put trebala kreirati velike pomoćne objekte u koje bi zatim kopirala predane vrijednosti. To oduzima mnogo procesorskog vremena, pa je prikladnije koristiti predaju po adresi (eng. *call by reference*).

```
int suma(int *a, int *b){
	return *a + *b; // dereferenciranje pokazivača
}
int main(){
	int x = 1, y = 2;
	int rez = suma(&x, &y); // 3
	return 0;
}
```

Umjesto da funkciji *suma* predajemo vrijednosti varijabli *x* i *y* mi joj predajemo njene adrese. Na taj način izbjegavamo kopiranje varijabli *x* i *y* u funkciju *suma* jer običnim dereferenciranjem iz funkcije *suma* se mogu doznati vrijednosti varijabli *x* i *y* . Ovaj pristup predstavlja ogromnu uštedu vremena jer osim izbjegavanja kopiranja izbjegava se i kreiranje pomoćnih objekata.

U C++u se osim pokazivača mogu koristiti i reference. Reference uvode još jedno ime za već postojeći objekt (varijablu). Primjerice,

```
int n;
int &r = n;  // referenca r je varijabla n
r = 15;      // isto kao n = 15;
cout << n;   // 15
```

Reference se moraju inicijalizirati odmah po deklaraciji, a naknadno ih nije moguće preusmjeriti na neku drugu varijablu. U ovom slučaju smo pomoću reference *r* kreirali još jedno ime za već postojeću varijablu *n*. Zato je svejedno da li napisali *r=15* ili *n=15*. Ovo svojstvo reference možemo iskoristiti i pri radu s funkcijama.

```
int suma(const int &a, const int &b){
	return a + b;
}
int main(){
	int x = 1, y = 2;
	int rez = suma(x, y); // 3
	return 0;
}
```

Pozivanjem funkcije *suma* kreiraju sve i inicijaliziraju dvije reference (*a* i *b*). Prva će se referencirati na varijablu *x* dok druga na varijablu *y*. Ovdje se ne događa prijenos po vrijednosti jer reference *a* i *b* nisu kopije objekata *x* i *y*, već tek njihova druga imena. Zato se dobije gotovo isti efekt kao i u slučaju prijenosa po adresi (eng. *call by reference*), pa je i poziv funkcije puno brži.

Kako su reference *a* i *b* zapravo varijable *x* i *y* treba biti oprezan pa u funkciji *suma* ne napisati nešto poput

```
a = 10;
```

Programer se lako može zabuniti misleći da je *a* lokalna varijabla funkcije, dok zapravo, riječ je o referenci na varijablu koja je u sasvim drugoj funkciji. Da se ovakve greške ne bi događale reference se često deklariraju kao konstantni parametri.

```
int suma(const int &a, const int &b)…
```

Sada nije moguće nehotice promijeniti vrijednosti varijabli *a* (*x*) i *b* (*y*) jer su one deklarirane kao konstante.

Reference su mnogima zgodnije za koristiti nego li pokazivači jer se izbjegava potreba za dereferenciranjem. Ipak, interno, reference su realizirane kao pokazivači i zauzimaju isto toliko memorije.

3.3. Imenovani prostor

Projekt može biti sastavljen od više datoteka izvornog koda. Svaka od njih može biti pisana od strane drugog autora, pa neke od klasa, funkcija ili varijabli u tim datotekama mogu imati ista imena. U takvim slučajevima prevoditelj neće moći razlikovati imena koja se preklapaju, pa ih je zato potrebno smjestiti u različite imenovane prostore (imenike).

```
namespace student{
	int godina;
	int semestar;
}
namespace datum_rodenja{
```

```
        int godina;
        int mjesec;
        int dan;
}
```

Imenici se deklariraju pomoću ključne riječi *namespace*, nakon čega se navodi njihov naziv. Deklaracija imenika može postojati samo u globalnom prostoru i u prostoru nekog drugog imenika, a na kraju njegove deklaracije se ne navodi točka-zarez. Također, imenik ne mora nužno imati ime, ali je u tom slučaju vidljiv samo u datoteci gdje se nalazi njegova deklaracija.

Imenici *student* i *datum_rodenja* sprječavaju konflikt pri korištenju varijable *godina*. U prvom slučaju ona predstavlja godinu studija studenta, a u drugom godinu rođenja osobe. U prošlosti dok imenici još nisu postojali programeri bi za ovakav slučaj koristili dugačka imena varijabli, ali uvođenjem imenika to više nije potrebno.

```
student::godina = 2;
datum_rodenja::godina = 1990;
```

Za pristup varijabli *godina* sada je potrebno navesti i naziv imenika u kojemu se ona nalazi. Ovime prevoditelj nedvojbeno zna na koju varijablu se misli. Ipak, i ovaj pristup se može skratiti korištenjem deklaracije *using*.

```
using namespace student;
godina = 2; // student::godina
datum_rodenja::godina = 1990;
```

Zbog deklaracije `using namespace student;` pri svakom korištenju varijable *godina* prevoditelj će pretpostaviti da je riječ o varijabli iz imenika *student*. Ukoliko želimo koristiti varijablu *godina* iz drugog imenika onda opet moramo navesti ime tog imenika.

```
using namespace student;
using namespace datum_rodenja;
godina = 2; // ??
```

C++ ne ograničava programera na korištenje samo jednog podrazumijevanog imenika. Ipak, u ovakvoj situaciji prevoditelj opet neće znati na koju varijablu *godina* se misli jer je ona sadržana u oba imenika. Stoga će ovakav programski kod rezultirati greškom u prevođenju.

Deklaraciju *using* je moguće koristiti i pri podrazumijevanom korištenju dijela imenika tj. pojedinih klasa, funkcija i varijabli. Primjerice,

```
namespace prvi{
	int x = 1;
	int y = 2;
}
namespace drugi{
	double x = 1.5;
	double y = 2.5;
}
int main() {
	using prvi::x;
	using drugi::y;
	cout << x << endl;  // 1
	cout << y << endl;  // 2.5
	cout << prvi::y << endl;  // 2
	cout << drugi::x << endl; // 1.5
	return 0;
}
```

Za varijablu *x* se podrazumijeva da je iz imenika *prvi*, dok za varijablu *y* da je iz imenika *drugi*. Za sve druge varijable *x* i *y* je potrebno navesti i imenike u kojima se nalaze.

Standardna C++ biblioteka već koristi imenike za svoje klase i funkcije, a gotovo uvijek smo do sada koristili imenik *std* za pristup tokovima ostream (cout) i istream (cin). U imeniku *std* se također nalaze i predlošci klasa za kontejnere, razni operatori i drugi tipovi podataka poput *string*, *regex* itd.

4. Kopiranje i prijenos

4.1. Kopirni konstruktor

Postoji poseban oblik konstruktora koji se naziva konstruktor kopije (eng. *copy constructor*). On se koristi kada vršimo kreiranje i inicijalizaciju novog objekta pomoću već postojećeg objekta tog tipa. Tada se vrši kopiranje predanog objekta u novi objekt. Svaka klasa već ima implicitni kopirni konstruktor, no on radi plitku kopiju objekta (eng. *shallow copy*), što može biti problem ukoliko klasa sadrži pokazivače.

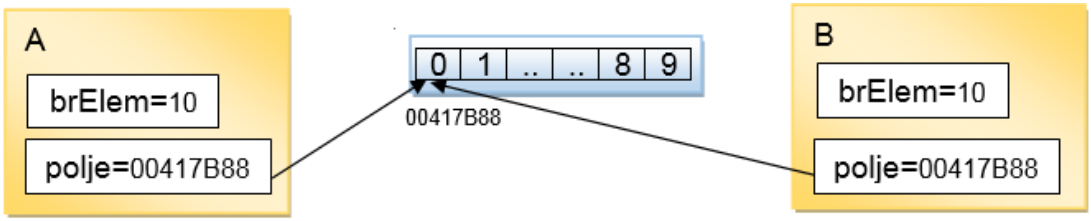
```
class IntPolje{
public:
	int *polje;
	int brElem;
	IntPolje(){}
	IntPolje(int n) : brElem(n){
		polje = new int[n];
	}
	~IntPolje(){
		delete[] polje;
	}
};
```

Klasa *IntPolje* predstavlja polje tipa *int*. Polje se dinamički alocira u konstruktoru klase pomoću pokazivača, a na kraju dealocira u destruktoru klase. U funkciji *main* napišimo sljedeći programski kod.

```
IntPolje A(10);
IntPolje B = A; // ili IntPolje B(A) - poziv kopirnog konstruktora
cout << A.polje << endl;  // 00417B88
cout << B.polje << endl;  // 00417B88 ??
```

Treba primijetiti da pokazivači *A.polje* i *B.polje* pokazuju na istu memorijsku adresu. To znači da objekti *A* i *B* dijele isti dinamički alocirani niz. To svakako nije poželjno jer ukoliko jedan od ovih objekata bude mijenjao niz te promijene će se reflektirati i pri radu s drugim objektom.

Isto tako, čim jedan od objekata bude uništen on će dealocirati dijeljeni niz, te time ga učiniti nedostupnim za drugi objekt. Sve je ovo posljedica izvršavanja implicitnog kopirnog konstruktora koji je napravio plitko kopiranje (kopiranje po vrijednosti).



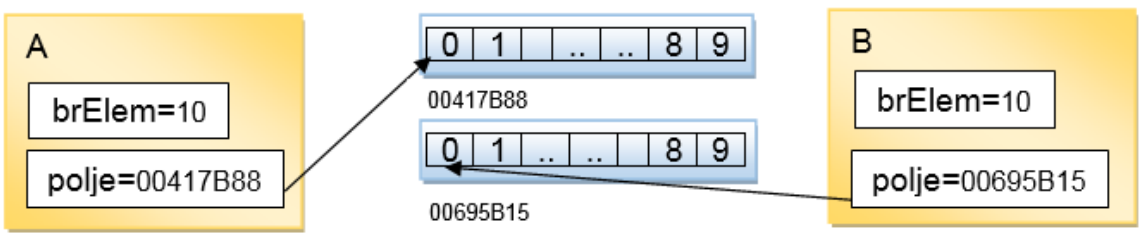
Slika 4.1.1. Plitko kopiranje

Naš je cilj da se prilikom poziva kopirnog konstruktora za svaki novi objekt dinamički alocira novi niz. Tako bi izbjegli dijeljenje memorijskih adresa među pokazivačima, pa i sve komplikacije koje iz toga proizlaze. Da bi to realizirali moramo napisati vlastitu implementaciju kopirnog konstruktora koja radi duboko kopiranje (eng. *deep copy*).

```
IntPolje(const IntPolje &X){
	polje = new int[X.brElem];  // duboko kopiranje!
	for (int i = 0; i < X.brElem; i++)
		polje[i] = X.polje[i];
	brElem = X.brElem;
}
```

Kopirni konstruktor kao parametar prima konstantnu referencu objekta svoje klase. Konstanta se koristi kako bi se spriječilo mijenjanje predanog objekta unutar kopirnog konstruktora, a referenca za ubrzanje poziva.

Naš kopirni konstruktor je napravio duboko kopiranje za pokazivač *polje*. On neće više pokazivati na istu memorijsku adresu kao i pokazivač predanog objekta, već se preusmjerava na novu memorijsku adresu tj. novi dinamički alocirani niz. Sada oba objekta imaju svoje vlastite, a ne dijeljene nizove.



Slika 4.1.2. Duboko kopiranje

Implicitni kopirni konstruktor će u najvećem broju slučajeva biti dovoljan. Ipak, kada klasa sadrži pokazivače uvijek treba provjeriti da li treba pisati i vlastitu implementaciju kopirnog konstruktora kako bi se izbjeglo plitko kopiranje.

4.2. Operator pridruživanja

Problem plitkog kopiranja kod implicitnog kopirnog konstruktora je prisutan i kod implicitnog operatora pridruživanja. Zato je najčešće slučaj da se u pri implementaciji kopirnog konstruktora s dubokim kopiranjem implementira i operator pridruživanja s dubokim kopiranjem.

```
IntPolje A(10), B;
B = A; // (implicitni) operator pridruživanja
cout << &A.polje[0] << endl; // 00707B88
cout << &B.polje[0] << endl; // 00707B88
```

Objekt *B* je početno inicijaliziran konstruktorom bez parametara, a u sljedećoj liniji mu kopiramo vrijednosti objekta *A*. Ipak, implicitni operator pridruživanja je napravio plitko kopiranje pa pokazivači *A.polje* i *B.polje* pokazuju na istu memorijsku adresu. Kao i u slučaju implicitnog kopirnog konstruktora tako i ovdje moramo implementirati operator pridruživanja s dubokim kopiranjem.

```
IntPolje& operator = (const IntPolje& X){
	if (this != &X){ // ne dopusti da se objekt pridružuje samom sebi
		delete[] polje; // dealociraj prethodnu memorijsku lokaciju
		polje = new int[X.brElem]; // duboko kopiranje!
		for (int i = 0; i < X.brElem; i++)
			polje[i] = X.polje[i];
		brElem = X.brElem;
	}
	return *this;
}
```

Operatorske funkcije ćemo detaljnije obrađivati u kasnijim laboratorijskim vježbama, a za sada možemo reći da postoje dva tipa: članske i ne-članske. Ovaj ovo je članska implementacija operatora pridruživanja jer ima samo jedan parametar tj. vrijednost koja se treba nalaziti s desne strane operatora pridruživanja (*const IntPolje& X*). S lijeve strane operatora se podrazumijevano nalazi objekt tipa *IntPolje*.

Parametar ove operatorske funkcije je isti kao i u slučaju kopirnog konstruktora, no povratna vrijednost je referenca na objekt koji se nalazi s lijeve strane operatora (*IntPolje&*). Upravo zbog toga je ovaj operator moguće jednostavno ulančavati jer referencom prosljeđujemo već taj postojeći objekt a ne njegovu kopiju.

```
IntPolje A(10), B, C, D, E;
B = C = D = E = A; // ulančavanje operatorom pridruživanja
```

Zadnja linija operatorske funkcije vraća **this*. Općenito, *this* je pokazivač na objekt u kojemu se nalazi a **this* je vrijednost tog objekta. Ovaj pokazivač se često koristi upravo pri radu s operatorskim funkcijama.

4.3. Prijenosni konstruktor i operator pridruživanja

Jedan od noviteta koji su došli s objavom C++11 standarda su konstruktor prijenosa (eng. *move constructor*) i operator pridruživanja sa semantikom prijenosa. Pomoću njih je moguće uvelike doprinijeti optimizaciji C++ aplikacija, pogotovo pri kopiranju velikih privremenih objekata. Uzmimo klasu *IntPolje* iz prethodnih primjera.

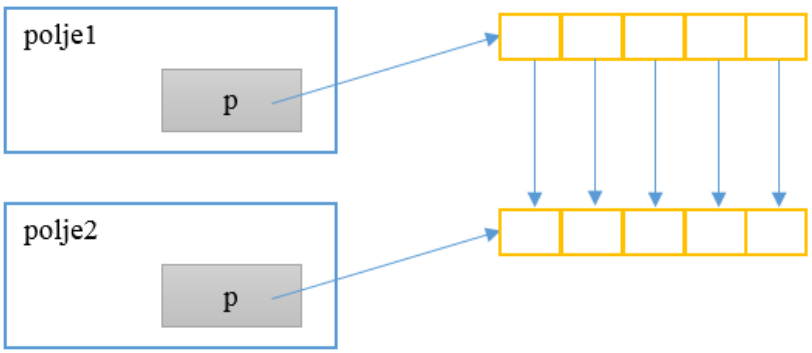
```
class IntPolje{
public:
	int* p;
	int size;
	IntPolje(int n) : size(n){
		p = new int[n];
	}
	// Kopirni konstruktor (duboko kopiranje)
	IntPolje(const IntPolje& X){
		p = new int[X.size];
		for (int i = 0; i < X.size; i++)
			p[i] = X.p[i];
		size = X.size;
	}
	~IntPolje(){
		delete[] p;
	}
};
```

Već nam je poznato čemu ova klasa služi. Ona interno sadrži dinamički alocirano polje tipa *int*, te omogućava duboko kopiranje upotrebom kopirnog konstruktora. Ipak, pogledajmo sljedeću situaciju.

```
IntPolje polje1(10); // Konstruktor!
IntPolje polje2 = polje1; // Kopirni konstruktor!
IntPolje polje3(IntPolje(10)); // konstruktor pa kopirni konstruktor ili RVO?
```

U prve dvije linije koda jasno je što se događa. Međutim, problematična je zadnja linija čiji rezultat ovisi o prevoditelju. Ono što bismo očekivali jest da se zbog privremenog objekta *IntPolje(10)* prvo pokrene konstruktor s parametrima, a nakon toga kopirni konstruktor koji bi svojstva tog privremenog objekta kopirao u objekt *polje3*. Međutim, ukoliko vaš prevoditelj koristi RVO (eng. *Return Value Optimization*) onda bi se zadnja linija koda interpretirala kao *IntPolje polje3(10)*. Ovime prevoditelj izbjegava kopirni konstruktor i izravno stvara objekt na osnovu parametara koji su proslijeđeni privremenom objektu.

Ipak, RVO optimizacija nije karakteristika svih prevoditelja te takav programski kod može rezultirati velikim kopiranjima zbog prethodnog pozivanja kopirnog konstruktora.



Slika 4.3.1. Duboko kopiranje

Kao što je vidljivo iz slike korištenjem semantike kopiranja (kopirni konstruktor) oba objekta dobila su nove memorijske lokacije za svoje pokazivače. Međutim, nakon toga još je potrebno izvršiti kopiranje podataka iz jednog dinamički alociranog niza u drugi. Sve ovo je uvijek nužno potrebno ukoliko je riječ o kopiranju iz jednog već postojećeg objekta (*lvalue* vrijednosti) u drugi objekt.

Ono što se dodatno optimiziralo kroz C++11 kopiranje je *rvalue* vrijednosti, tj. privremenog objekta u novi objekt. U tu svrhu sada se može koristi semantika prijenosa tj. prijenosni konstruktor i operator pridruživanja sa semantikom prijenosa.

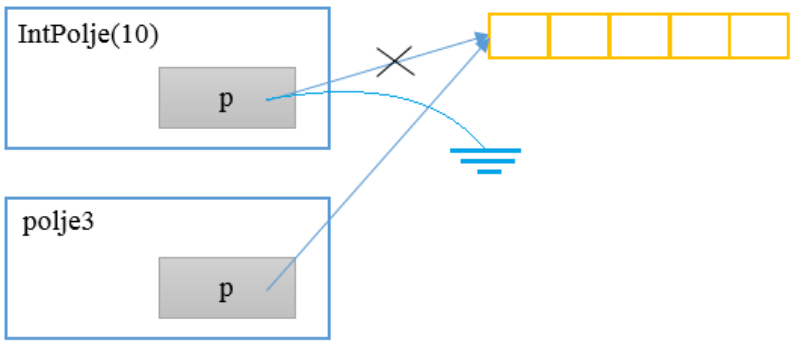
U prethodnom primjeru demonstrirali smo oba poziva:

```
IntPolje polje2 = polje1;
IntPolje polje3(IntPolje(10));
```

Problem s drugom naredbom jest da je privremeni objekt *IntPolje(10)* već dinamički alocirao niz od 10 elemenata te će ga uništiti odmah nakon izvršenja kopirnog konstruktora objekta *polje3*. Pitanje je onda zašto uopće raditi kopiju privremenog objekta i time gubiti procesorsko vrijeme ako već postojeći niz privremenog objekta njemu ionako više neće trebati? Stoga, umjesto da radimo kopiju privremenog objekta možemo prenijeti njegova postojeća svojstva (dinamički alocirani niz) u novi objekt. Upravo to je semantika prijenosa.

```
// Konstruktor prijenosa s rvalue referencom
IntPolje(IntPolje&& privremeni){
	p = privremeni.p; // Preusmjeri pokazivač
	privremeni.p = NULL; // privremeni objekt više nije vlasnik pokazivača
	size = privremeni.size;
}
```

Ovako bi izgledao prijenosni konstruktor za gore opisanu klasu. Kao parametar prima *rvalue* referencu i umjesto rezerviranja nove memorijske lokacije i kopiranja (kopirni konstruktor) jednostavno preuzima pokazivač iz privremenog objekta.



Slika 4.3.2. Semantika prijenosa (prijenosni konstruktor)

Što se točno događa u kodu vidljivo je iz slike. Objekt *polje3* pomoću prijenosnog konstruktora preuzima pokazivač iz privremenog objekta, a zatim se pokazivač privremenog objekta postavlja na vrijednost NULL. To je potrebno iz razloga što će se prilikom uništenja privremenog objekta pokrenuti njegov destruktor koji bi taj pokazivač

dealocirao, a to ne smijemo dopustiti budući da tu memorijsku lokaciju (niz) sada koristi novi objekt.

Napomena: Ukoliko vaš prevoditelj koristi RVO optimizaciju pri izvršenja izraza `IntPolje polje3(IntPolje(10));` uvijek za demonstraciju možete koristiti funkciju *move* koja forsira semantiku prijenosa. Primjerice,

```
IntPolje polje3 = move(IntPolje(10)); // izbjegava RVO, poziva prijenosni konstruktor
```

Kao i u slučaju kopirnog konstruktora i operatora pridruživanja s dubokim kopiranjem, tako isto možemo definirati i operator pridruživanja sa semantikom prijenosa. Za gornju klasu on bi izgledao na sljedeći način.

```
IntPolje& operator = (IntPolje&& privremeni){
	delete[] p; // dealociraj prethodnu memorijsku lokaciju
	p = privremeni.p;
	privremeni.p = NULL;
	size = privremeni.size;
	return *this;
}
```

Ukoliko pretpostavimo da naša klasa ima podrazumijevani konstruktor, primjer upotrebe bio bi sljedeći.

```
IntPolje polje4;
polje4 = IntPolje(10); // operator pridruživanja sa semantikom prijenosa
```

Upotrebom semantike prijenosa C++ postaje još brži programski jezik. Ova značajka je također integrirana i u kontejnere koji za svoje interne operacije koriste privremene objekte (*rvalue* vrijednosti).

5. Statički članovi klase i iznimke

5.1. Deklaracija friend

Pravom pristupa *private* definiramo da se određeni podatkovni član ili metoda ne može koristiti izvan klase. Ipak, nekada se može učiniti potrebnim napraviti iznimku ovog pravila te eksplicitno dopustiti nekoj drugoj funkciji ili klasi da koristi privatne članove naše klase. Da bi to omogućili koristimo deklaraciju *friend*.

```
class Tocka{
private:
	int x, y;
public:
	Tocka(int a, int b) : x(a), y(b){}
};
int getX(Tocka A){
	return A.x; // greška! x je nedostupan!
}
int getY(Tocka A){
	return A.y; // greška! y je nedostupan!
}
```

U privatnom dijelu klase *Tocka* se nalaze dva podatkovna člana (*x* i *y*). Oni predstavljaju koordinate točke, dok sa dvije globalne funkcije *getX* i *getY* želimo vratiti te koordinate. Problem je što funkcije *getX* i *getY* nemaju pristup privatnim članovima klase *Tocka*, te tako ne mogu vratiti vrijednosti *x* i *y*. Stoga, u klasi *Tocka* moramo eksplicitno deklarirati te dvije funkcije kao prijatelje klase *Tocka*.

```
class Tocka{
private:
	int x, y;
	friend int getX(Tocka A);
	friend int getY(Tocka A);
public:
	Tocka(int a, int b) : x(a), y(b){}
};
```

Funkcije *getX* i *getY* sada imaju pristup svim članovima klase *Tocka*, neovisno o pravima pristupa koja su definirana u klasi. Deklaracije prijateljskih funkcija *getX* i *getY* su u ovom slučaju navedene u privatnom dijelu klase *Tocka*, no one se mogu nalaziti bilo gdje u klasi. Također, mogli smo i tijela tih funkcija napisati unutar klase, no svejedno treba imati na umu da su to i dalje vanjske funkcije i da nisu članice klase.

```
class Tocka{
private:
	int x, y;
	double UdaljenostOdIshodista(){
		return pow(x*x + y*y, 0.5);
	}
public:
	friend class Ravnina;  // klasi Ravnina je dopušten puni pristup!
	Tocka(int a, int b) : x(a), y(b){}
};
class Ravnina{
public:
	vector<Tocka> tocka;
	// koja je točka u ravnini najudaljenija od ishodišta?
	int NajudaljenijaTocka(){
		int indeks = -1;
		double najdalja = -1;
		for (int i = 0; i < tocka.size(); i++){
			int pom = tocka[i].UdaljenostOdIshodista(); // privatna metoda!
			if (pom > najdalja){
				najdalja = pom;
				indeks = i;
			}
		}
		return indeks;
	}
};
```

Kao prijatelji se osim funkcija mogu deklarirati i klase. Primjerice, klasa *Ravnina* predstavlja skup točaka koje su interno spremljene u vektoru. Da bi njena metoda *NajudaljenijaTocka* izračunala koja od točki u ravnini (vektoru) je najudaljenija od ishodišta ona za svaku točku treba pozvati njenu metodu *UdaljenostOdIshodišta*. Problem je u tome što je ta metoda privatna unutar klase *Tocka*, te da bi joj klasa *Ravnina* imala pristup ona mora postati prijatelj klase *Tocka*.

```
friend class Ravnina;  // klasi Ravnina je dopušten puni pristup!
```

U prvom primjeru smo vidjeli kako se deklaracijom *friend* koristimo kako bi iz globalnih funkcija *getX* i *getY* pristupili privatnom podatkovnom članu klase *Tocka*. U ovom primjeru pak imamo slučaj da iz klase *Ravnina* pristupamo privatnoj metodi koja se nalazi u drugoj klasi (*Tocka*).

Sve ovo narušava dobre principe programiranja i enkapsulacije, te je općenito preporuka izbjegavati korištenje deklaracije *friend*.

5.2. Statički podatkovni članovi klase

Deklaracijom klase se definira novi tip podatka, a članovi klase definiraju od čega se on sastoji. Svi objekti (instance) imaju svoju kopiju tih podataka i spremaju ih u svoj memorijski prostor. Ipak, u klasi je moguće definirati i podatkovne članove koji se međusobno dijele između svih instanci. To su statički članovi klase.

```
class A{
public:
	static int n;
};
int A::n = 0; // inicijalizacija statičkog člana

int main(){
	A obj1, obj2;

	cout << obj1.n << " " << obj2.n << endl; // 0 0
	obj1.n = 1;
	cout << obj1.n << " " << obj2.n << endl; // 1 1
	obj2.n = 2;
	cout << obj1.n << " " << obj2.n << endl; // 2 2
	return 0;
}
```

Kada se koriste statički članovi klase potrebno ih je prije početka programa obavezno inicijalizirati. Nadasve, nećete moći prevesti programski kod ukoliko to ne napravite. Inicijalizacija statičkog člana se vrši izvan klase, i to na sljedeći način.

```
int A::n = 0; // inicijalizacija statičkog člana
```

Umjesto da koristimo neku od instanci klase *A* statičkom članu smo pristupili preko operatora „::“ (eng. *scope operator*). Tek nakon inicijalizacije statičkog člana možemo u funkciji *main* kreirati objekte *obj1* i *obj2*.

```
cout << obj1.n << " " << obj2.n << endl; // 0 0
```

Pošto je riječ o statičkom članu klase koji je u početku inicijaliziran na vrijednost 0, sve instance klase *A* će vrijednost tog člana pročitati kao 0. Zbog toga se pri ispisu *obj1.n* i *obje2.n* u oba slučaja ispisuje 0. Isti ispis bi dobili i za `cout << A::n`, pa je na programeru da izabere koji pristup mu bolje odgovara.

```
obj1.n = 1;
cout << obj1.n << " " << obj2.n << endl; // 1 1
```

U drugoj naredbi smo preko instance *obj1* promijenili vrijednost statičkog člana *n* u 1 pa se pri ispisu pojavljuje *1 1*. Isti slučaj se događa i u zadnjem ispisu gdje smo ovaj put preko instance *obj2* promijenili vrijednost statičkog člana, pa se ispisuje *2 2*.

```
class A{
public:
	static int kol;
	int rb;
	A(){
		rb = ++kol;
	}
};
int A::kol = 0;

int main(){
	A obj1, obj2, objX[5];

	cout << "Kreirano je " << A::kol << " instanci tipa A\n"; // A::kol = 7
	// ispis rednih brojeva instanci
	cout << obj1.rb << obj2.rb; // 12
	for (int i = 0; i < 5; i++)
		cout << objX[i].rb;  // 34567
	return 0;
}
```

Statički podatkovni članovi klase mogu poslužiti i u slučaju kada želimo brojati i numerirati instance neke klase. Statički se član *kol* automatski uvećava za 1 svaki put kada se kreira nova instanca klase *A*. Trenutno stanje tog brojača se sprema i u podatkovni član *rb* kako bi se spremio redni broj instance.

5.3. Statičke metode klase

Osim statičkih podatkovnih članova postoje i statičke metode klase. One prvenstveno služe za komunikaciju sa statičkim podatkovnim članovima ukoliko se u klasi koristi enkapsulacija (skrivanje podataka). Primjerice,

```
class A{
private:
	static int N;
public:
	static int getN(){
		return N;
```

```
	}
	static void setN(int x){
		N = x;
	}
};
int A::N = 0;

int main(){
	cout << A::getN();  // 0
	A::setN(5);
	cout << A::getN();  // 5
	return 0;
}
```

Statički podatkovni član *N* se nalazi u privatnom dijelu, te mu se ne može direktno pristupiti izvan klase. Zato su za njega napisane njegove javne *get* i *set* metode koje također imaju deklaraciju *static*. Zbog toga se pri pozivu ovih metoda ne moramo adresirati na neku postojeću instancu, već tek na ime klase. Npr. `cout << A::getN();`.

Statičke metode klase u svom tijelu mogu komunicirati samo sa statičkim podatkovnim članovima. Pokušaj pristupa podatkovnim članovima koji nisu statički će uzrokovati grešku prevoditelja.

```
class A{
private:
	static int N;
	int nijeStaticka;
public:
	...
	static int getN(){
		return N + nijeStaticka;  // greška!
	}
};
```

Razlog ovom ograničenju je vrlo jednostavan. Statički članovi klase su dijeljeni među svim instancama klase, te se u tom kontekstu iz statičke metode ne može komunicirati sa podatkovnim članovima koji pripadaju pojedinim instancama.

5.4. Iznimke

Iznimka je situacija kada se tijekom rada programa zbog nepredviđenih okolnosti dogodi greška. Takav tip greške može uzrokovati prekid rada programa, pa je od velike važnosti znati kako predvidjeti takve situacije i kako obrađivati iznimke. U tu svrhu C++

nudi mehanizam za rukovanje iznimkama (eng. *exception handling*) pomoću ključnih riječi *try*, *throw* i *catch*.

```
try{
	// blok pokušaja - problematični dio koda
	if (problem)
		throw Iznimka; // baci iznimku
}
catch (parametar){
	// blok hvatanja - obrada iznimke određenog tipa
}
catch (...){
	// obrada ostalih tipova iznimki
}
```

U bloku pokušaja (*try*) se navode problematične operacije koje bi mogle uzrokovati iznimku. Sama iznimka se baca (generira) ključnom riječi *throw*, a u ovisnosti o tipu iznimke izvršava se odgovarajući blok hvatanja. Ukoliko se bačena iznimka nije obradila u niti jednom prethodnom bloku hvatanja onda će se obraditi u bloku hvatanja koji kao parametre ima 3 točke – *catch(…)*.

```
double a, b, c;
cout << "Unesite dva broja: ";
cin >> a >> b;
try {
	if (b == 0)
		throw "b je jednak nuli!";
	c = a / b;
	cout << "a/b = " << c;
}
catch (const char* Iznimka){
	cout << "Iznimka: " << Iznimka;
}
```

```
Unesite dva broja: 5 0
Iznimka: b je jednak nuli!
```

Jedna od kritičnih operacija je dijeljenje dva broja. Da bi uspješno izveli tu operaciju broj s kojim dijelimo ne smije biti jednak nuli. Međutim, ukoliko se to ipak dogodi program će generirati iznimku i vrlo vjerojatno prekinuti izvođenje ostatka programa. Pošto smo sada predvidjeli takvu mogućnost unutar bloka pokušaja provjeravamo djelitelj te sami bacamo iznimku ukoliko je on jednak nuli.

```
if (b == 0)
	throw "b je jednak nuli!";
```

Nakon bacanja (generiranja) iznimke preskaču sve daljnje naredbe u bloku pokušaja (pa tako i problematično dijeljenje) te se traži adekvatni izraz hvatanja. Prvo što moramo primijetiti jest da se poslije ključne riječi *throw* nalazi znakovni niz (*const char**). Zato postoji i blok hvatanja iznimke tog tipa.

```
catch (const char* Iznimka){
	cout << "Iznimka: " << Iznimka;
}
```

Ključna riječ *catch* označava početak koda koji služi za hvatanje i obradu nekog tipa iznimke. Kao i blok pokušaja uvijek mora biti omeđen vitičastim zagradama bez obzira da li se unutra nalazi jedna ili više naredbi.

Nekad će se u bloku pokušaja nalaziti više problematičnih operacija. Tada je moguć i slučaj da se baca jedna od više iznimki koje nisu nužno istog tipa. U tom slučaju poželjno je za svaki tip iznimke napisati i adekvatni izraz hvatanja koji ju obrađuje.

```
double a, b, c;
char operacija;
cout << "Unesite dva broja: ";
cin >> a >> b;
cout << "Operacija (+,-,*,/): ";
cin >> operacija;

try{
	if (operacija != '+' && operacija != '-' &&
	    operacija != '*' && operacija != '/')
	      throw operacija;  // nedozvoljena operacija!
	if (operacija == '/' && b == 0)
	      throw 0;  // pokušaj dijeljenja s nulom!
	// c = ....
}
catch (char operacija){
	cout << "Iznimka: nedozvoljena operacija " << operacija << endl;
}
catch (int n){
	cout << "Iznimka: dijeljenje s " << n << " nije dozvoljeno!" << endl;
}
catch (...){
	cout << "Nepoznata iznimka...";
}
```

Prikazanim programskim odsječkom pokušavamo izvršiti nekakvu matematičku operaciju nad dva broja. U kodu smo predvidjeli dvije moguće greške: Ako je riječ o nedozvoljenoj operaciji tada se baca iznimka tipa *char* tj. nedozvoljena operacija. Sukladno tome, pokreće se blok hvatanja `catch (char operacija)` koji korisniku ispisu poruku o grešci.

```
Unesite dva broja: 2 3
Operacija (+,-,*,/): #
Iznimka: nedozvoljena operacija #
```

Druga moguća greška je ukoliko korisnik želi dijeliti s nulom. Tada se kao iznimka baca vrijednost 0 tj. iznimka tipa *int*. Ona će biti obrađena u bloku hvatanja `catch (int n)`.

```
Unesite dva broja: 3 0
Operacija (+,-,*,/): /
Iznimka: dijeljenje s 0 nije dozvoljeno!
```

Ukoliko bi se u ovom programskom kodu dogodila iznimka nekog drugog tipa onda bi se ona obradila u bloku hvatanja `catch (...)`. Ovaj blok hvatanja se uvijek stavlja kao zadnji. Time se daje prednost prethodnim blokovima hvatanja koji će bačenu iznimku možda prepoznati kao iznimku točno određenog tipa.

5.5. Iznimke korisničkih i standardnih tipova

Iznimke ne moraju nužno biti samo primitivnog tipa (*int, char, double…*). Pomoću klasa programer može definirati i iznimke korisničkih tipova. Prednost ovakvih tipova iznimki je u puno detaljnijoj povratnoj informaciji. Umjesto samo jednog podatka moguće je proslijediti cijeli set informacija o grešci.

```
class Iznimka_Dijeljenje{
public:
	// brojevi koji su se koristili pri dijeljenju
	double broj1, broj2;
	Iznimka_Dijeljenje(double x, double y) :broj1(x), broj2(y) {}

	// ispis poruke o grešci
	void ispis(){
		cout << "Iznimka_Dijeljenje: broj " << broj1 <<
			" se ne moze podijeliti s " << broj2 << "!";
	}
};
```

Napisana klasa se može iskoristiti prilikom obrade iznimki nastalih dijeljenjem s nulom.

```
double a, b, c;
try{
	cout << "Unesite dva broja: ";
```

```
	cin >> a >> b;
	if (b == 0)
		throw Iznimka_Dijeljenje(a, b);
	c = a / b;
}
catch (Iznimka_Dijeljenje iznimka){
	iznimka.ispis();
}
```

```
Unesi dva broja: 4 0
Iznimka_Dijeljenje: broj 4 se ne moze podijeliti s 0!
```

Blok hvatanja je koristio metodu *ispis* kako bi korisniku ispisao poruku o grešci. Osim pristupa ovoj metodi unutar bloka hvatanja možemo pristupiti i samim brojevima koji su korišteni u dijeljenju (*broj1* i *broj2*).

U C++u već postoji skup klasa definiranih standardom koje služe za obradu iznimki. Sve one nastaju nasljeđivanjem klase *exception*. Tim klasama su pokriveni mogu tipovi iznimki, počevši od onih logičkog tipa do iznimki u vrijeme izvođenja. Štoviše, objavom C++11 standarda broj klasa za rad s iznimkama se povećao, te sada standard obuhvaća i iznimke za rad s regularnim izrazima, pametnim pokazivačima itd. Kompletan popis klasa je moguće vidjeti na http://en.cppreference.com/w/cpp/error/exception.

```
vector<int> v = { 1, 2, 3, 4, 5 };
try {
	cout << v.at(5); // baca iznimku std::out_of_range
}
catch (const std::out_of_range& e) {
	std::cout << "Iznimka: " << e.what() << endl;
}
```

Metoda *at* klase *vector* baca iznimku tipa *out_of_range* u slučaju da pokušamo pristupiti elementu vektora koji ne postoji. U prikazanom slučaju 5-i element vektora ne postoji pa će se baciti i obraditi navedena iznimka. Također, još jedan česti primjer upotrebe iznimki iz standardne biblioteke je pri dinamičkoj alokaciji memorije.

```
int *p;
try {
	// while(1)
	p = new int[99999999];
}
catch (const std::bad_alloc& e) {
	std::cout << "Dinamicka alokacija nije uspjela: " << e.what() << endl;
}
```

Iznimka tipa *bad_alloc* će biti bačena ukoliko nije moguće dinamički alocirati željenu količinu memorije.

Na sličan način je moguće koristiti i ostale iznimke iz standardne biblioteke, a za C++14 standard su najavljena i nova proširenja klase *exception*. Primjerice, biti će moguće koristiti i iznimke tipka *bad_array_length*.

5.6. Ugniježđene i proslijeđene iznimke

Ugnježđivanje iznimke je slučaj kada se jedan blok pokušaja nalazi unutar drugog bloka pokušaja. U slučaju bacanja iznimke u unutrašnjem bloku ona se može obraditi čak više puta tj. i kroz vanjske blokove hvatanja.

```
class Iznimka_Matematika{
public:
	void ispis(){
		cout << "Iznimka_Matematika!" << endl;
	}
};
class Iznimka_Dijeljenje : public Iznimka_Matematika{
public:
	double broj1, broj2;
	Iznimka_Dijeljenje(double x, double y) :broj1(x), broj2(y) {}
	void ispis(){
		cout << "Iznimka_Dijeljenje: broj " << broj1 <<
			" se ne moze podijeliti s " << broj2 << "!" << endl;
	}
};
```

Klasa *Iznimka_Dijeljenje* nasljeđuje klasu *Iznimka_Matematika*. To znači da se iznimka tipa *Iznimka_Dijeljenje* može obraditi u dva različita bloka hvatanja. Primjerice,

```
double a, b, c;
try{
	try{   // ugniježđeni blok pokušaja
		cout << "Unesite dva broja: ";
		cin >> a >> b;
		if (b == 0)
			throw Iznimka_Dijeljenje(a, b);
		c = a / b;
	}
	catch (Iznimka_Dijeljenje iznimka){
		iznimka.ispis();
		throw; // proslijedi iznimku vanjskom bloku hvatanja
	}
```

```
}
catch (Iznimka_Matematika iznimka){
	iznimka.ispis();
}
```

```
Unesi dva broja: 5 0
Iznimka_Dijeljenje: broj 5 se ne moze podijeliti s 0!
Iznimka_Matematika!
```

U unutrašnjem bloku pokušaja se baca iznimka tipa *Iznimka_Dijeljenje*. Ona će se uhvatiti i obraditi čak dva puta tj. u unutrašnjem i vanjskom bloku hvatanja. Nakon hvatanja i obrade iznimke u unutrašnjem bloku ona će se naredbom *throw* proslijediti vanjskom bloku hvatanja, gdje će biti obrađena kao iznimka tipa *Iznimka_Matematika*.

Iznimke se ne moraju nužno prosljeđivati. Ukoliko bismo izostavili naredbu *throw* u unutrašnjem bloku hvatanja bačena iznimka bi se obradila samo jedanput.

5.7. Neprihvaćene iznimke

Programer treba osigurati prihvat i obradu svih bačenih iznimki. Ukoliko se nepažnjom dogodi da se zaboravi napisati prihvat nekog tipa iznimke i ako ne postoji catch(...) onda je to neprihvaćena iznimka. Program će u tom slučaju pozvati funkciju na koju pokazuje funkcijski pokazivač *terminate*, a to je funkcija *abort*.

Da bi spriječili direktno pozivanje podrazumijevane funkcije *abort*, a time i nasilni završetak programa, moguće je preusmjeriti pokazivač *terminate* na neku drugu funkciju tipa *void*. To radimo funkcijom *set_terminate*.

```
void Neprihvacena() {
	cout << "Dogodila se neprihvacena iznimka!" << endl;
	exit(-1);
}
int main(){
	double a, b;
	set_terminate(Neprihvacena);

	try{
		cout << "Unesite dva broja: ";
		cin >> a >> b;
		if (b == 0)
			throw "Dijeljenje s nulom!";
		cout << a / b;
```

```
	}
	catch (int n){
		cout << "ID greske: " << n << endl;
	}
	return 0;
}
```

U glavnom programu se baca iznimka tipa *const char**, no nema niti jednog bloka hvatanja koji bi tu iznimku uhvatio i obradio. Obično bi se u ovakvoj situaciji pozvala funkcija *abort* koja bi nasilno završila program. Umjesto nje, poziva se funkcija *Neprihvacena* koja korisniku ispisuje odgovarajuću poruku, a zatim uredno prekida izvršavanje programa.

Kada se dogodi neprihvaćena iznimka program uvijek mora završiti s radom. Samo je pitanje da li će to biti pozivanjem podrazumijevane funkcije *abort*, ili pozivanjem neke druge funkcije (*Neprihvaćena*) koja u konačnici opet završava program pozivanjem funkcije *exit*. Kraj programa se ne može spriječiti niti izostavljanjem poziva funkcije *exit* jer bi se umjesto nje opet automatski izvršila funkcija *abort*.

Ipak, prednost pri izvršavanju funkcije *Neprihvaćena* je i u tome da program može u datoteke zapisati logove o izvršavanju koji kasnije mogu biti korisni pri analizi grešaka.

5.8. Specifikacija iznimki u deklaraciji funkcije

C++ dozvoljava specifikaciju iznimki pri deklaraciji funkcija. Takva specifikacija služi tek kao napomena programeru da navedena funkcija može baciti i proslijediti iznimke određenog tipa. Zato bi programer pri pozivu takve funkcije trebao imati blokove hvatanja kojima bi mogao uhvatiti i obraditi sve iznimke koje se proslijede iz te funkcije.

```
void f() throw();  // ne prosljeđuje nikakve iznimke
void f2(int a) throw(int, char); // prosljeđuje iznimke tipa int i char
```

Funkcija *f* nema parametara, a u zagradama nakon ključne riječi *throw* se ne nalazi ništa. Stoga, ova funkcija ne prosljeđuje nikakve iznimke. Međutim, u drugoj funkciji je moguće prosljeđivanje dvaju tipova iznimki (*int* i *char*).

```
double dijeli(double a, double b) throw(int){
	if (b == 0)
		throw 0;
```

```
	return a / b;
}
```

Funkcija *dijeli* može proslijediti iznimku tipa *int*. Iz koda je vidljivo da se bačena iznimka ne hvata i ne obrađuje unutar same funkcije, pa ju je potrebno obraditi negdje drugdje.

```
int main(){
	float a, b;
	try{
		cout << "Unesite dva broja: ";
		cin >> a >> b;
		dijeli(a, b); // može proslijediti iznimku tipa int!
	}
	catch (int n){ // hvatamo iznimku iz funkcije 'dijeli'!
		cout << "Iznimka: Dijeljenje s nulom!" << endl;
	}
	return 0;
}
```

Ako se unutar funkcije *dijeli* ipak baci iznimka koja nije navedena u specifikacijama tada se izvršava funkcija *unexpected*. Kao i u slučaju neprihvaćene iznimke riječ je o funkciji koja se definira pomoću pokazivača na funkciju tipa *void*. I taj pokazivač je također moguće preusmjeriti na neku drugu funkciju tipa *void*.

```
set_unexpected(Neocekivana_iznimka);
```

U praksi se najčešće izbjegava specificiranje iznimki pri deklaraciji funkcija, dok neki prevoditelji ovu specifikaciju čak i u potpunosti ignoriraju. Razlog je u tome što prevoditelj ne može prisiliti programera da u nekoj funkciji baca samo one iznimke koje se nalaze u specifikaciji, pa tako niti može garantirati da se neće baciti iznimke nekog drugog tipa.

6. Operatori

6.1. Članske operatorske funkcije

U klasi je moguće osim metoda i atributa definirati i operatore. Pomoću operatora možemo definirati razne operacije između objekata naše klase i ostalih tipova podataka. Realiziramo ih pomoću operatorskih funkcija, koje po implementaciji mogu biti članske ili ne-članske. U ovisnosti o tome operatorske funkcije mogu imati jedan ili dva ulazna parametra.

```
class Kompleksni{
public:
	double re, im;
	Kompleksni(){}
	Kompleksni(double x, double y) : re(x), im(y) {}
	Kompleksni operator +(const Kompleksni &Z); // kompleksni + kompleksni
	Kompleksni& operator =(double re); // kompleksni = double
};
Kompleksni Kompleksni::operator +(const Kompleksni &Z){
	return Kompleksni(re + Z.re, im + Z.im);
}
Kompleksni& Kompleksni::operator =(double re){
	this->re = re;
	this->im = 0;
	return *this;
}
```

Objekti klase *Kompleksni* sada imaju mogućnost međusobnog zbrajanja te inicijalizacije realnim brojem. Primjerice,

```
Kompleksni A(2.5, 6), B(4, 1.5), C, D;
C = A + B; // 6.5 + 7.5i
D = 10;    // 10 + 0i
```

Da bi uspješno napisali operatorsku funkciju moramo znati tri informacije:

1) Koji tip podatka se nalazi sa lijeve strane operatora
2) Koji tip podatka se nalazi sa desne strane operatora
3) Koji je rezultat operacije (operatora)

U naredbi `C = A + B;` sa lijeve i desne strane operatora + se nalaze objekti tipa *Kompleksni* (*A* i *B*). Zato je i rezultat zbrajanja također objekt tipa *Kompleksni* (*C*).

```
// kompleksni = kompleksni + kompleksni
Kompleksni Kompleksni::operator +(const Kompleksni &Z){
	return Kompleksni(re + Z.re, im + Z.im); // vraćamo privremeni objekt!
}
```

Ovaj operator je je realiziran kao članska operatorska funkcija, deklarirana unutar klase *Kompleksni*. Među ostalim, zato smo i naredbu `C = A + B;` mogli napisati i kao `C=A.operator+(B);`.

Pošto je operator + realiziran kao članska funkcija klase *Kompleksni* automatski se podrazumijeva da se sa lijeve strane tog operatora treba nalaziti objekt tipa *Kompleksni*. Pitanje je tek što se nalazi sa desne strane operatora, a to je objekt koji je predan kao parametar operatoru tj. nekakav drugi objekt tipa *Kompleksni*. Zato članske operatorske funkcije imaju samo jedan parametar jer je objekt sa lijeve strane operatora podrazumijevanog tipa.

Povratna vrijednost operatorske funkcije + je privremeni objekt koji se inicijalizira konstruktorom klase *Kompleksni*. Taj objekt postoji samo u trenutku kada funkcija mora vratiti povratnu vrijednost, a nakon toga se automatski uništava. Ipak, u drugom operatoru povratna vrijednost je drukčija.

```
Kompleksni& Kompleksni::operator =(double re){
	this->re = re;
	this->im = 0;
	return *this;
}
```

Ovaj operator pridruživanja koristimo kada kompleksni broj želimo inicijalizirati realnim brojem. Primjerice, pri naredbi `D = 10;` realni dio kompleksnog broja *D* treba imati vrijednost 10, a imaginarni dio vrijednost 0. Zato je parametar operatorske funkcije tek realni broj *re*.

U ovom slučaju nije potrebno vratiti privremeni objekt jer se za operaciju pridruživanja koristi samo jedan objekt klase *Kompleksni* (*D*). Nakon što njegov realni i imaginarni dio inicijaliziramo na potrebne vrijednosti upotrebom naredbe `return *this;` vraćamo upravo objekt *D*.

6.2. Ne-članske operatorske funkcije

Problemi pri pisanju članskih operatorskih funkcija nastaju kada su lijevi i desni operand različitih tipova. Tada članske operatorske funkcije nisu dobro rješenje upravo zbog činjenice što je u tom slučaju lijevi operand uvijek podrazumijevanog tipa. Uzmimo za primjer sljedeću klasu koja implementira operator +.

```
class Kompleksni{
public:
	double re, im;
	Kompleksni(){}
	Kompleksni(double x, double y) : re(x), im(y) {}
	Kompleksni operator +(double re); // kompleksni + double
};
Kompleksni Kompleksni::operator +(double re){
	return Kompleksni(this->re + re, im);
}
```

Operatorska funkcija + je deklarirana kao članska, te je zato moguće izvršiti sljedeći programski odsječak.

```
Kompleksni A(2.5, 6), B;
B = A + 4.5; // B = 7 + 6i
```

Primijetimo da je pri pozivu operatora + lijevi operand tipa *Kompleksni*, a desni tipa *double*. Zanimljivo je da ukoliko redoslijed operanada zamijenimo prevoditelj neće znati kako zbrojiti ta dva objekta.

```
B = 4.5 + A; // greška!
```

Da bi prevoditelj znao izračunati vrijednost izraza *4.5 + A* moramo napisati ne-člansku operatorsku funkciju. Kod njih lijevi operand nije podrazumijevanog tipa te zato one imaju dva ulazna parametra (lijevi i desni operand).

```
// kompleksni = double + kompleksni
Kompleksni operator +(double re, Kompleksni Z){
	return Z + re; // već definirano članskom operatorskom funkcijom
}
```

U ovisnosti o redoslijedu parametara prevoditelj zna kojeg tipa mora biti lijevi operand a kojeg tipa desni. Tako ovo preopterećenje operatora zbraja *double + Kompleksni*. Ova operacija je identična kao i operacija *Kompleksni + double*, pa kao povratnu vrijednost

možemo postaviti *Z + re*. Prevoditelj već zna zbrojiti ovaj izraz zbog članske implementacije operatora +.

Ukoliko bi klasa *Kompleksni* koristila privatno pravo pristupa za članove *re* i *im*, tada bi bilo potrebno koristiti deklaraciju *friend* unutar klase *Kompleksni*.

```
friend Kompleksni operator +(double re, Kompleksni Z);
```

Općenito je preporuka koristiti članske operatorske funkcije u slučajevima gdje su lijevi i desni operand istog tipa. Tada redoslijed predanih objekata pri pozivu operatora nije bitan. Ipak, u slučajevima gdje su lijevi i desni operand različiti i gdje su mogući pozivi operatora s različitim redoslijedom operanada potrebno je koristiti ne-članske operatorske funkcije.

6.3. Preopterećenja drugih operatora

U prethodnim primjera vidjeli smo preopterećenja nekih binarnih aritmetičkih operatora. To su oni za čije djelovanje su potrebna dva objekta. Osim njih, moguće je preopteretiti i unarne aritmetičke operatore poput ++ i --.

```
class Kompleksni{
public:
	double re, im;
	Kompleksni(){}
	Kompleksni(double x, double y) : re(x), im(y) {}

	Kompleksni& operator ++(); // prefiks ++
	Kompleksni operator ++(int); // postiks ++
};
// prefiks operator ++
Kompleksni& Kompleksni::operator ++(){
	++re;
	++im;
	return *this;
}
// sufiks operator ++
Kompleksni Kompleksni::operator ++(int){
	Kompleksni Pom = *this;
	re++;
	im++;
	return Pom;
}
```

Sufiks operator ++ ima parametar tipa *int*. On služi samo za to da bi se moglo razlikovati prefiks od sufiks operatora, te nema nikakvo dublje značenje.

```
Kompleksni A(1, 1), B, C;
// sufix operator ++
B = A++;
cout << B.re << " " << B.im << endl; // 1 1
cout << A.re << " " << A.im << endl; // 2 2
// prefix operator ++
C = ++B;
cout << C.re << " " << C.im << endl; // 2 2
cout << B.re << " " << B.im << endl; // 2 2
```

Također, moguće je preopteretiti operatore za ispis i unos podataka (<< i >>). Time bi omogućili ispis i unos kompleksnog broja navođenjem imena kompleksnog broja uz operator.

```
// preopterećenje operator za unos kompleksnog broja
istream& operator >>(istream& ulaz, Kompleksni& Z){
	ulaz >> Z.re >> Z.im;
	return ulaz;
}
// preopterećenje operator za ispis kompleksnog broja
ostream& operator <<(ostream& izlaz, Kompleksni Z){
	izlaz << Z.re << " + " << Z.im << "i";
	return izlaz;
}
```

Oba operatora se mogu ulančavati pa kao lijeve operande primamo referencu na *ostream* tj. *istream* objekt. Konačno, taj isti objekt vraćamo kao povratnu vrijednost operatora. Primjer poziva bi bio sljedeći:

```
Kompleksni A;
cin >> A;     // 1 1
cout << A;    // 1 + 1i
```

Kada program dođe do naredbe *cin* čekati će unos dva realna broja (realni i imaginarni dio kompleksnog broja *A*), a pri ispisu će ih ispisati u formatu *re* + *imj*.

I operatore pretvorbe je moguće preopteretiti.

```
class Kompleksni{
public:
	double re, im;
	Kompleksni(double x, double y) : re(x), im(y) {}
	// pretvorba Kompleksni -> double
```

```
	operator double(){
		return re;
	}
};
```

Operator *double* sada omogućuje pretvorbu kompleksnog broja u realni broj. Pri pretvorbi smo definirali da se vraća samo realni dio tog kompleksnog broja.

```
Kompleksni A(2.5, 10);
double re = (double)A; // re = 2.5
```

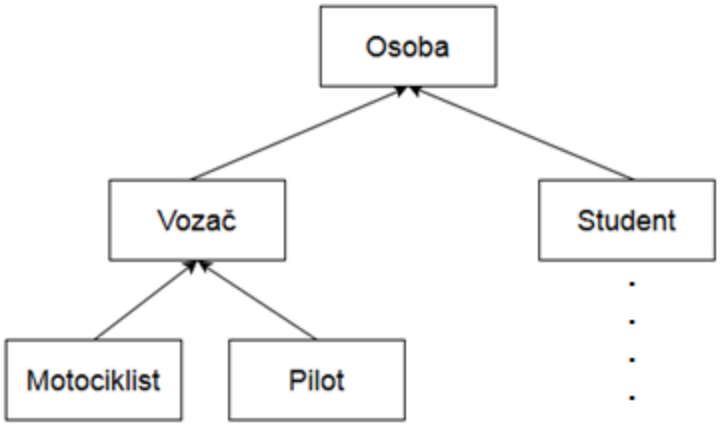
Moguće je preopteretiti i mnoštvo drugih operatora poput operatora usporedbe ==, !=, >, <, >=, <=, operatora +=, -=, *=, /= itd. U svim ovim slučajevima je potrebno slijediti prethodno opisane principe rada s članskim i ne-članskim operatorskim funkcijama, vodeći brigu i o mogućim ulančanim pozivima tih operatora.

7. Nasljeđivanje

7.1. Nasljeđivanje

Programski jezik C++ omogućuje deklaraciju nove klase na način da ona nasljeđuje neku već postojeću klasu. Prilikom nasljeđivanja nova klasa (izvedenica, derivacija) kopira članove bazne klase. Osim naslijeđenih članova nova klasa obično sadrži i svoje vlastite članove. Zato možemo reći i da je nova klasa (izvedenica) tek nadogradnja bazne klase.

Nasljeđivanjem smanjujemo količinu programskog koda pa je i vrijeme prevođenja aplikacije kraće. Štoviše, bez svojstva nasljeđivanja bilo bi gotovo nezamislivo pisati složenije aplikacije.



Slika 7.1.1. Hijerarhija nasljeđivanja

Nasljeđivanje je uvijek najlakše vizualizirati UML (eng. *Unified Modeling Language*) dijagramom. Upotrebom generalizacije (strelica) jasno je vidljivo koja je izvedena klasa (ona iz koje strelica počinje), a koja je bazna klasa (ona u kojoj strelica završava). Primjerice, u ovom dijagramu klase *Vozač* i *Student* su izvedenice klase *Osoba*. Također, klase *Motociklist* i *Pilot* su izvedenice klase *Vozač*.

Nasljeđivanjem se definira „is-a“ veza između izvedene i bazne klase. Primjerice, klasa *Osoba* predstavlja općenito neku osobu. Međutim, *Vozač* i *Student* su konkretniji tipovi osoba, dok *Motociklist* i *Pilot* su konkretniji tipovi vozača tj. još konkretniji tipovi osoba.

Osoba.h (zaglavlje bazne klase *Osoba*)

```
class Osoba{
protected:
	string JMBG;
public:
	string ime;
	string prezime;
	void Radi();
	void Odmori();
};
```

Prvo se uvijek deklarira bazna klasa. Ona definira osnovni set članova koji se nalazi u svakoj izvedenoj klasi koja je ispod nje u hijerarhiji nasljeđivanja. Primjerice, u ovom slučaju, bez obzira o kakvom tipu osobe da je riječ ta osoba uvijek ima svoje *ime* i *prezime*, zaštićeni član *JMBG* te javne metode *Radi* i *Odmori*.

Podatkovni član *JMBG* je deklariran kao tipa *protected*. Naime, privatni članovi klase osiguravaju skrivanje podataka te ne dopuštaju direktan pristup izvana. No isto tako, privatni članovi se ne mogu nasljeđivati. Umjesto toga, korištenjem prava pristupa *protected* ti članovi klase su i dalje nedostupni za direktan pristup izvana, ali moguće ih je nasljeđivati.

Student.h (zaglavlje klase *Student*)

```
class Student : public Osoba{
public:
	string JMBAG;
	string NazivStudija;
	int semestar;
	void Uci();
};
```

Klasa *Student* nasljeđuje klasu *Osoba*. To znači da je klasa *Student* ništa drugo već konkretnija inačica klase *Osobe*. Kao takva, klasa *Student* sada ima sve članove klase *Osoba* kao i set svojih vlastitih članova.

```
Student Ante;
Ante.ime = "Ante";
Ante.prezime = "Antic";
Ante.JMBAG = "024600112233";
Ante.NazivStudija = "TVZ";
Ante.semestar = 3;
Ante.Radi();
Ante.Uci();
Ante.Odmori();
```

Korištenjem objekta *Ante* se vidi kojim sve članovima raspolažu objekti tipa *Student*. Tu su podatkovni članovi i metode klase *Osoba*, kao i klase *Student*. Objekt *Ante* raspolaže i s članom *JMBG*, no pošto je on zaštićen (nedostupan izvana) ne može ga se direktno koristiti već tek iz klase *Student*.

Važno je napomenuti da izvedena klasa ipak ne nasljeđuje sve iz bazne klase. Osim privatnih članova izvedena klasa neće naslijediti konstruktore, destruktor, funkcije tipa *friend* te operator pridruživanja. Svaka od ovih stavki je specifična za pojedinu klasu, te ih kao takve nije moguće nasljeđivati.

Ostaje pitanje na koji način su naslijeđeni podatkovni članovi i metode iz klase *Osoba*. To smo definirali pri samoj deklaraciji klase. Primjerice,

```
class Student : public Osoba
```

Oznaka *public* definira da će prava pristupa naslijeđenih članova klase *Osoba* biti nepromijenjena u izvedenoj klasi *Student*. Nasljeđivanje se još može definirati i kao zaštićeno i privatno. U tom slučaju vrijede sljedeća pravila.

Nasljeđivanje	Značenje
public	Javni članovi bazne klase postaju javni članovi izvedene klase. Zaštićeni članovi bazne klase postaju zaštićeni članovi izvedene klase. Privatni članovi bazne klase nisu dostupni izvedenoj klasi
private	Javni i zaštićeni članovi bazne klase postaju privatni članovi izvedene klase. Privatni članovi bazne klase nisu dostupni izvedenoj klasi.
protected	Javni i zaštićeni članovi bazne klase postaju zaštićeni članovi izvedene klase. Privatni članovi bazne klase nisu dostupni izvedenoj klasi.

Slika 7.1.1. Tipovi nasljeđivanja

U praksi se najčešće koristi javni tip nasljeđivanja jer je iznimno rijetko potrebno mijenjati prava pristupa članova bazne klase unutar izvedenih klasa.

C++ podržava i višestruko nasljeđivanje tj. mogućnost da izvedena klasa nasljeđuje više baznih klasa.

```
class A{
public:
	// …
};
class B{
public:
	// …
};
class C : public A, public B{
public:
	// …
};
```

Iako ova mogućnost može biti korisna poželjno ju je izbjegavati zbog mogućeg problema dijamantne hijerarhijske strukture nasljeđivanja. Zbog tog i sličnih problema višestruko nasljeđivanje nije moguće u programskim jezicima poput Jave i C#-a.

7.2. Nadređenje metode i koncept „prikrivanja“ imena

U postupku nasljeđivanja izvedena klasa kopira sve dostupne članove bazne klase. Međutim, ukoliko se u izvedenoj klasi već nalazi metoda s istim imenom i parametrima kao i u baznoj klasi tada je riječ o nadređenju (eng. *overriding*) metode. U tom slučaju će pri pozivu te metode biti će izvršena njena implementacija iz izvedene klase, dok će se implementaciji iz bazne klase moći pristupiti samo pomoću operatora „::“.

```
class A{
public:
	void f(){
		cout << "A::f()" << endl;
	}
};
class B : public A{
public:
	// nadređenje metode void f()
	void f(){
		cout << "B::f()" << endl;
	}
};
```

…

```
B obj;
obj.f();	// B::f()
obj.A::f();	// A::f()
```

Nakon što se nad objektom *obj* pozove metoda *f* ispisuje se *B::f()*. Tek nakon upotrebe operatora „::“ možemo pozvati metodu *f* klase *A*. Ovo se događa iz razloga jer prevoditelj koristi koncept prikrivanja imena (eng. *name hiding*). Nakon nadređenja metode *f*

prevoditelj prikrije sve metode iz bazne klase s imenom *f*, te se kao jedina vidljiva izvrši metoda *f* iz izvedene klase. Zato treba biti oprezan i pri sljedećoj situaciji.

```
class A{
public:
	void f(){
		cout << "A::f()" << endl;
	}
	// preopterećenje metode f
	void f(int n){
		//...
	}
};
class B : public A{
public:
	// nadređenje metode void f()
	void f(){
		cout << "B::f()" << endl;
	}
};
```

Sljedeći programski kod uzrokuje grešku prevoditelja.

```
B obj;
obj.f();  // B::f()
obj.f(5); // greška prevoditelja!
```

U klasi *A* postoji preopterećena inačica metode *f* tj. metoda *void f(int n)*. Ipak, zbog nadređenja metode *void f()* automatski se primjenjuje koncept prikrivanja imena. Zbog toga metoda *void f(int)* neće biti vidljiva, što uzrokuje grešku prevoditelja.

Nadređenje metode ne treba miješati sa preopterećenjem metode. Ukoliko su potpisi metoda (liste parametara) u baznoj i izvedenoj klasi različiti tek tada je riječ o preopterećenju kod kojeg se ne primjenjuje koncept prikrivanja imena.

7.3. Kreiranje i uništavanje objekata izvedene klase

Da bi se kreirao i inicijalizirao objekt izvedene klase prvo se moraju izvršiti konstruktori baznih klasa. Tek nakon njih se izvršava konstruktor izvedene klase. Sve je poprilično jednostavno ukoliko bazne i izvedena klasa koriste samo podrazumijevane konstruktore tj. kada je objekt izvedene klase moguće kreirati bez parametara.

```
class Osoba{
```

```
public:
	Osoba(){
		cout << "Osoba konstruktor\n";
	}
	~Osoba(){
		cout << "Osoba destruktor\n";
	}
};
class Student : public Osoba{
public:
	Student(){
		cout << "Student konstruktor\n";
	}
	~Student(){
		cout << "Student destruktor\n";
	}
};
int main(){
	Student Ante;
	return 0;
}
```

Izvršavanje programa:

```
Osoba konstruktor
Student konstruktor
Student destruktor
Osoba destruktor
```

Prilikom kreiranja prvo se izvršavaju konstruktori bazne klase. Ukoliko izvedena klasa nasljeđuje više klasa tada će se prvo pokrenuti svi njihovi konstruktori pa tek onda konstruktor izvedene klase. Uništavanje se odvija obrnutim redoslijedom tj. prvo se poziva destruktor izvedene klase a zatim destruktori baznih klasa. Tako je i u sljedećoj situaciji.

```
class Student : public Osoba, public AkademskiGradanin
```

Kreiranjem objekta tipa *Student* redoslijed konstruktora i destruktora bi bio sljedeći:

```
Osoba konstruktor
AkademskiGradanin konstruktor
Student konstruktor
Student destruktor
AkademskiGradanin destruktor
Osoba destruktor
```

Ukoliko neke od baznih klasa sadrže konstruktor s parametrima izvedena klasa mora proslijediti tražene parametre tim konstruktorima. To se radi na način da konstruktor izvedene klase primi, a zatim proslijedi tražene parametre.

```
class Osoba{
public:
	string ime;
	string prezime;
	Osoba(string _ime, string _prezime) : ime(_ime), prezime(_prezime){
		//...
	}
};
class Student : public Osoba{
public:
	// prosljeđivanje parametara baznom konstruktoru
	Student(string ime, string prezime) : Osoba(ime, prezime){
		//...
	}
};
```

Iako klasa *Student* nema vlastite podatkovne članove *ime* i *prezime* ona te podatke treba predati konstruktoru klase *Osoba*. Upravo zato i klasa *Student* ima konstruktor s parametrima *ime* i *prezime*.

```
Student(string ime, string prezime) : Osoba(ime, prezime){}
```

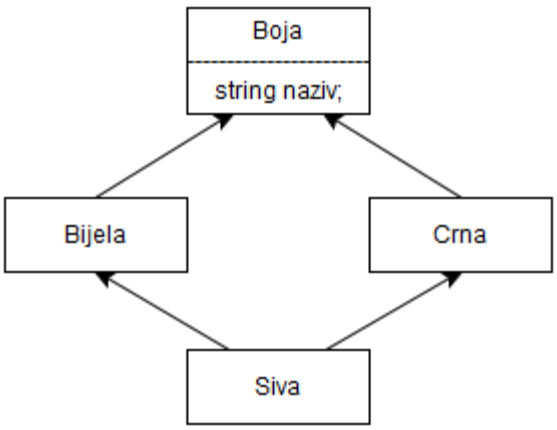
Slično je i u slučaju kada izvedena klasa nasljeđuje više klasa.

```
class Osoba{
public:
	string ime;
	string prezime;
	Osoba(string _ime, string _prezime) : ime(_ime), prezime(_prezime){
		//...
	}
};
class Student{
public:
	string JMBAG;
	Student(string jmbag) : JMBAG(jmbag){
		//...
	}
};
class Akademik : public Osoba, public Student{
public:
	// prosljeđivanje parametara konstruktorima baznih klasa
	Akademik(string ime, string prez, string jmbg):Osoba(ime, prez), Student(jmbg){
		//...
	}
};
```

Klasa *Akademik* nasljeđuje dvije klase. Prva bazna klasa *Osoba* ima konstruktor s dva parametra (*ime* i *prezime*), dok druga bazna klasa *Student* ima konstruktor s jednim parametrom (*jmbag*). Zato izvedena klasa *Akademik* mora imati konstruktor sa tri parametra. Prva dva će se proslijediti konstruktoru klase *Osoba*, a zadnji konstruktoru klase *Student*.

7.4. Virtualno nasljeđivanje

Jedan od mogućih problema s višestrukim nasljeđivanjem klasa je dijamantna hijerarhijska struktura. Jedna izvedena klasa (*Siva*) može nasljeđivati od više drugih klasa (*Bijela* i *Crna*), koje su pak također naslijeđene od neke treće bazne klase (*Boja*). U ovakvoj situaciji klasa *Siva* dva puta nasljeđuje iste članove. Prikažimo dijagramom.



Slika 7.4.1. Dijamantna struktura nasljeđivanja

Klasa *Boja* ima podatkovni član *naziv*. Taj podatkovni član se zatim nasljeđuje u klasama *Bijela* i *Crna*. Zatim, klasa *Siva* nasljeđuje klase *Bijela* i *Crna*, a time i podatkovni član *naziv* dva puta. Problem je u tome što klasa *Siva* ne zna koji od ova dva naslijeđena člana da koristi, a taj problem rješavamo virtualnim nasljeđivanjem.

```
class Boja{
public:
	string naziv;
};
```

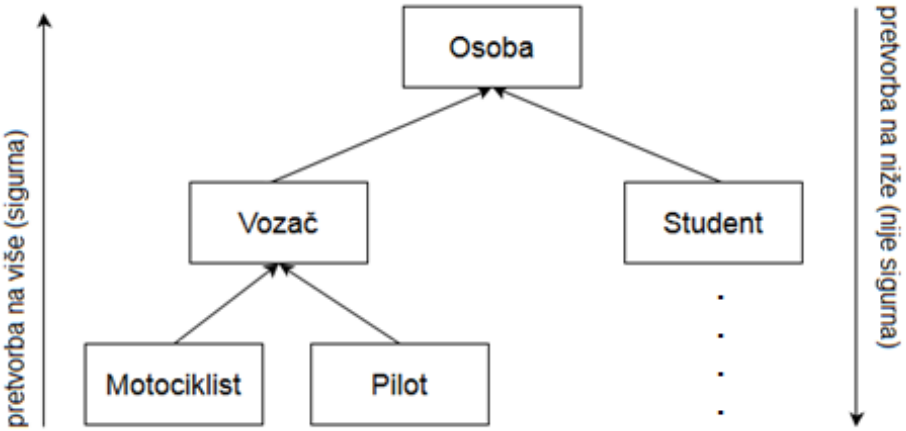
```
class Bijela : virtual public Boja{
public:
	//...
};
class Crna : virtual public Boja{
public:
	//...
};
class Siva : public Bijela, public Crna{
public:
	//...
};
```

Klase *Bijela* i *Crna* koriste virtualno nasljeđivanje klase *Boja*. Ovim načinom nasljeđivanja se definira da će klasa koja nasljeđuje bijelu i crnu (klasa *Siva*) imati samo jednu kopiju članova iz klase *Boja*. Zbog toga klasa *Siva* neće dva puta naslijediti član *naziv*, već samo jedanput. Unatoč ovoj mogućnosti višestruko nasljeđivanje se ne preporučuje, a u novijim programskim jezicima nije niti podržano.

8. Polimorfizam

8.1. Pretvorbe na više i niže

U hijerarhiji nasljeđivanja moguće su pretvorbe na više (eng. *upcast*) i pretvorbe na niže (eng. *downcast*). Pretvorba na više znači da se objekt izvedene klase može predstaviti pokazivačem bazne klase zbog „is-a“ veze. Primjerice, svaki objekt tipa *Student* se može predstaviti pokazivačem tipa *Osoba*, jer klasa *Student* nasljeđuje klasu *Osoba*. Pretvorba na više se smatra sigurnom. Prevoditelj ju izvodi implicitno a moguće ju je i eksplicitno izvršiti korištenjem *static_cast* pretvorbe.



Slika 8.1.1. Pretvorba na više i niže

Pretvorba na niže je obrnuta situacija tj. kada objekt bazne klase predstavljamo pokazivačem izvedene klase. Ova pretvorba nije sigurna te se provjerava u vrijeme izvršavanja programa upotrebom *dynamic_cast* pretvorbe.

```
Student Ante("Ante", "Antic");
Osoba* netko = &Ante;  // pretvorba na više (upcasting)
```

Vidimo primjer pretvorbe na više. Prevoditelj će implicitno provjeriti da li je pretvorba moguća te ukoliko nije javiti grešku. Iako, to smo mogli i mi eksplicitno provjeriti upotrebom *static_cast* pretvorbe naredbom `Osoba* netko = static_cast<Osoba*>(&Ante);`.

Pretvorba *static_cast* je identična standardnoj pretvorbi korištenoj u programskom jeziku C. Primjerice, sljedeće pretvorbe su identične.

```
float f = 5.25;
int n;
n = f;       // implicitna pretvorba
n = (int)f;  // eksplicitna pretvorba
n = static_cast<int>(f); // statička pretvorba
```

Upotreba pretvorbe na više (eng. *upcast*) je najčešća pri radu s funkcijama. Umjesto da radimo preopterećenje neke globalne funkcije za svaku izvedenu klasu u hijerarhiji dovoljno je napraviti samo inačicu koja koristi pokazivač bazne klase.

```
void ispisOsobe(Osoba* netko){
	cout << netko->ime << " " << netko->prezime << endl;
}
int main(){
	Student Ante("Ante", "Antic");
	ispisOsobe(&Ante);
	return 0;
}
```

Funkcija *ispisOsobe* prima adresu objekta tipa *Osoba*. Ipak, pri njenom pozivu predana je adresa objekta tipa *Student*. Prevoditelj će uspješno izvršiti implicitnu pretvorbu na više jer klasa *Student* nasljeđuje klasu *Osoba*. Točnije, klasa *Student* je naslijedila članove *ime* i *prezime* koji se koriste u funkciji *ispisOsobe*. Na ovaj način funkcija *ispisOsobe* može primiti adresu i bilo kojeg drugog objekta koji je u hijerarhiji ispod klase *Osoba*.

Pretvorba na niže (eng. *downcast*) nije sigurna te se drukčije provjerava. Naime, pretvorba na više se provjerava prilikom prevođenja programa, dok pretvorba na niže tokom rada programa. To se događa iz razloga jer se direktno ne zna tko niže u hijerarhiji trenutno predstavlja objekt bazne klase. Primjerice, da li je objekt tipa *Osoba* predstavljen pokazivačem tipa *Student* ili pokazivačem tipa *Vozac*? Ili možda pokazivačem tipa koji je još niže u hijerarhiji nasljeđivanja?

```
Osoba Ante("Ante", "Antic");
// pokušaj pretvorbe na niže
Student *pStudent = dynamic_cast<Student*>(&Ante);
if (pStudent == NULL)
	cout << "pretvorba nije uspjela!";
else
	cout << "pretvorba je uspjela!";
```

Pretvorba *dynamic_cast* će vratiti *NULL* ukoliko pretvorba nije moguća. To je tokom rada programa moguće provjeriti te sukladno tome izvršiti potrebne radnje. U ovom primjeru pretvorba neće biti uspješna jer stvarni objekt *Ante* nije objekt tipa *Student*. U protivnom bi se moglo dogoditi sljedeće:

```
pStudent->NazivStudija = "TVZ";
```

Pokazivač *pStudent* zapravo pokazuje na objekt tipa *Osoba*, a on nema podatkovni član *NazivStudija*. Program bi se zbog toga srušio. Upravo zbog ovakvih situacija pretvorbe na niže nisu sigurne i potrebno ih je provjeravati.

```
Osoba* Ante = new Student("Ante", "Antic");
// pokušaj pretvorbe na niže
Student *pStudent = dynamic_cast<Student*>(Ante);
if (pStudent == NULL)
	cout << "pretvorba nije uspjela!";
else
	cout << "pretvorba je uspjela!";
```

Za razliku od prethodne, ovaj pokušaj pretvorbe na niže će uspjeti jer je stvarni objekt na kojeg pokazuje pokazivač *Ante* tipa *Student*.

Važno je napomenuti da pretvorba *dynamic_cast* zahtjeva da je bazna klasa polimorfna (ima barem jednu virtualnu funkciju). O tome će više biti riječ u nastavku, no za sada je dovoljno reći da u baznoj klasi *Osoba* treba dodati sljedeću liniju da bi mogli koristiti *dynamic_cast* pretvorbu.

```
virtual void prazna(){};
```

Osim pretvorbi *static_cast* i *dynamic_cast* C++ omogućuje korištenje *const_cast* i *reinterpret_cast* pretvorbi. Pretvorbom *const_cast* smo u mogućnosti nekom objektu privremeno dodati ili oduzeti svojstvo konstantnosti.

```
class A{
public:
	int vrijednost;
	A(int n) : vrijednost(n){}
};
int main(){
	const A obj(0);
	cout << obj.vrijednost << endl;  // 0
```

```
	// privremeno oduzimanje svojstva konstantnosti
	const_cast<A&>(obj).vrijednost = 5;
	cout << obj.vrijednost << endl;  // 5
	return 0;
}
```

Iako je objekt tipa *A* deklariran kao konstantan privremenim oduzimanjem tog svojstva smo mu uspjeli promijeniti podatkovni član *vrijednost*. Bez upotrebe *const_cast* pretvorbe ovo ne bi bilo moguće te bi prevoditelj pri prevođenju javio grešku.

Sa upotrebom *const_cast* pretvorbe treba biti oprezan jer u nekim slučajevima se može dogoditi nedefinirano ponašanje.

```
const int a = 0;
const int *pa = &a;

int *p = const_cast<int*>(pa);
*p = 1;  // nedefinirano ponašanje!
cout << a << *p;  // ??
```

Upotrebom „prljavih trikova“ uspješno smo uspjeli pristupiti memorijskoj adresi varijable *a* pomoću ne-konstantnog pokazivača. Štoviše, u sljedećoj liniji smo taj pokazivač iskoristili kako bi na memorijsku lokaciju varijable *a* zapisali vrijednost 1. Što se točno događa pri izvršavanju izraza `*p = 1;` je u ovom slučaju nedefinirano, pa je tako i ispis u zadnjoj liniji koda upitan.

```
const int a = 0;
const_cast<int&>(a) = 1; // nedefinirano ponašanje!
```

Prethodni primjer smo napisati i na ovakav način koji također predstavlja nedefinirano ponašanje. I ovdje također nije jasno definirano što se događa u zadnjoj liniji koda, pa i ovakve slučajeve poželjno izbjegavati.

Posljednja, *reinterpret_cast* pretvorba, radi na način da reinterpretacijom bitova omogućuju sve pretvorbe među pokazivačima bilo kojih tipova. Pokazivač bilo kojeg tipa je moguće pretvoriti u pokazivač bilo kojeg tipa. No zato adresiranje podataka nakon pretvorbe gotovo uvijek može predstavljati grešku. Od svih pretvorba ova je najopasnija, te se općenito preporučuje izbjegavati njeno korištenje.

8.2. Polimorfizam

Nasljeđivanjem se definiraju sličnosti među klasama zbog „is-a“ veze. Objekti izvedene i bazne klase imaju sličan sastav ali i slično ponašanje. Ova se činjenica može iskoristiti prilikom pretvorbe na više kada objekt izvedene klase predstavljamo pokazivačem bazne klase.

Primjerice, recimo da imamo klase *Vozac* i *Profesor*. Objekti ovih tipova se uvijek mogu predstaviti pokazivačem njihove bazne klase (*Osoba*). Korištenjem ovog pokazivača uvijek će se izvršiti metode iz bazne klase, bez obzira što taj pokazivač pokazuje na objekt izvedene klase. Da bi izvršili ispravnu inačicu pozvane metode (u ovisnosti o tome gdje trenutno pokazuje pokazivač bazne klase) koristimo svojstvo polimorfizma tj. virtualne metode.

```
class Osoba{
public:
	string ime;
	Osoba(string _ime) : ime(_ime){}
	// virtualna metoda opisPosla!
	virtual void opisPosla(){
		cout << ime << " nesto radi!" << endl;
	}
};
class Vozac : public Osoba{
public:
	Vozac(string _ime) : Osoba(_ime){}
	// implementacija metode opisPosla u klasi Vozac!
	void opisPosla(){
		cout << ime << " je vozac!" << endl;
	}
};
class Profesor : public Osoba{
public:
	Profesor(string _ime) : Osoba(_ime){}
	// implementacija metode opisPosla u klasi Profesor!
	void opisPosla(){
		cout << ime << " je profesor!" << endl;
	}
};
```

Virtualne metode bazne klase predstavljaju metode čija implementacija treba biti različita u svakoj izvedenoj klasi. Tako sada u baznoj klasi *Osoba* imamo virtualnu metodu *opisPosla*. Ova metoda ispisuje opis posla neke osobe. Međutim, nemaju svi tipovi osoba isti posao. Zato klase *Vozac* i *Profesor* imaju svoje vlastite implementacije te metode.

Pogledajmo sljedeći programski odsječak:

```
Osoba* osoba;
Vozac vozac("Ante");
Profesor profesor("Marko");

osoba = &vozac;
osoba->opisPosla(); // 'Ante je vozac!'
osoba = &profesor;
osoba->opisPosla(); // 'Marko je profesor!'
```

Naredbom `osoba = &vozac;` se izvršava implicitna pretvorba na više. Pokazivač bazne klase *Osoba* pokazuje na objekt izvedene klase tipa *Vozac*. U sljedećoj naredbi se pomoću pokazivača *osoba* poziva metoda *opisPosla*. Obično bi se u ovoj situaciji pozvala metoda *opisPosla* iz bazne klase, ali pošto je ta metoda u baznoj klasi deklarirana kao virtualna izvršiti će njena inačica iz klase *Vozac*.

Što se točno događa pri pozivu virtualne funkcije? U memoriji postoji tablica virtualnih funkcija zvana *vtable*, a svaki objekt sadrži skriveni pokazivač na tu tablicu zvan *vptr*. Pozivanjem virtualne funkcije prvo se dohvaća adresa te tablice, a zatim se u njoj pronalazi adresa ispravne inačice funkcije koju treba pozvati. Tek nakon toga se događa njen poziv. Zbog ovoga je pozivanje virtualnih funkcija sporije, no to nikako ne može zamijeniti prednosti koje pruža svojstvo polimorfizma.

Primjere polimorfizma možemo vidjeti i pri radu s globalnim funkcijama:

```
void ispis(Osoba* netko){
	netko->opisPosla();
}
```

Funkcija *ispis* kao parametar prima adresu nekog objekta tipa *Osoba*. Ipak, mi toj funkciji možemo predati i adrese objekata tipa *Vozac* i *Profesor* zbog implicitne pretvorbe na više. Tako možemo napravili i sljedeće pozive funkcija.

```
Vozac vozac("Ante");
Profesor profesor("Marko");

ispis(&vozac);    // 'Ante je vozac!'
ispis(&profesor); // 'Marko je profesor!'
```

Rezultat izvršavanja je očekivan jer se u funkciji *ispis* nakon implicitne pretvorbe na više poziva virtualna metoda *opisPosla*.

Potrebno je spomenuti da u klasi jedino konstruktor ne može biti virtualan, dok destruktor to može biti. Nadasve, to je nekada i nužno.

```
class Bazna{
public:
	virtual ~Bazna(){
		cout << "Destruktor bazne klase...\n";
	}
};
class Izvedena : public Bazna{
private:
	int* polje;
public:
	Izvedena(int n){
		polje = new int[n];
	}
	~Izvedena(){
		delete[] polje;
		cout << "Polje je dealocirano! (destruktor izvedene klase...)\n";
	}
};
```

Izvedena klasa nasljeduje baznu klasu. Objekt izvedene klase pri svom uništenju mora osloboditi zauzetu memoriju i zato je nužno da se destruktor izvedene klase izvrši. No, zbog eventualne pretvorbe na više moguća je ovakva situacija:

```
Bazna* bazna = new Izvedena(10);
delete bazna;
```

Po definiciji (deklaraciji) *bazna* je pokazivač koji pokazuje na objekt bazne klase. No, on po inicijalizaciji zapravo pokazuje na objekt izvedene klase. Ipak, zbog svoje definicije, prilikom oslobađanja memorije poziva se samo destruktor bazne klase.

Kada bi se to dogodilo imali bi curenje memorije (eng. *memory leak*) jer zauzeta memorija u objektu izvedene klase ne bi bila oslobođena. Zbog toga, destruktor bazne klase moramo deklarirati kao virtualan. Ako je destruktor bazne klase virtualan program će prvo pokrenuti destruktor stvarnog objekta, pa tek onda destruktor bazne klase.

8.3. Ključne riječi override i final

Izlaskom C++11 standarda nadređene virtualne metode u izvedenim klasama moguće je dodatno označiti ključnom riječi *override*. Prevoditelj će u tom slučaju napraviti dodanu provjeru da li se u izvedenoj klasi implementira ispravna inačica virtualne metode bazne klase te time spriječiti eventualne greške programera.

```
class Zivotinja{
public:
	virtual void pricaj(){
		cout << "Zivotinja prica" << endl;
	}
};
class Pas : public Zivotinja{
public:
	void pricaj(string nesto) override { // greška prevoditelja!
		cout << nesto << endl;
	}
};
```

Klasa *Zivotinja* sadrži virtualnu metodu *pricaj*, dok se njena drukčija implementacija očekuje u izvedenoj klasi *Pas*. Pošto je prilikom implementacije te metode u izvedenoj klasi korištena ključna riječ *override* prevoditelj će dodatno provjeriti da li se virtualna metoda s tim imenom i parametrima nalazi u baznoj klasi. U ovom slučaju prevoditelj će javiti grešku jer virtualna metoda *pricaj* bazne klase *Zivotinja* nema parametar *string nesto*, te je pretpostavka da je programer napravio grešku prilikom njenog nadređenja u izvedenoj klasi *Pas*.

Da ključna riječ *override* nije navedena prevoditelj ne bi javio grešku, no programer bi umjesto nadređenja virtualne metode u izvedenoj klasi dodao tek njeno preopterećenje s dodatnim parametrom. Ključna riječ *override* nije obavezna, no iz prikazanih razloga svakako ju je preporučljivo koristiti.

Osim ključne riječi *override*, prilikom implementacije polimorfizma moguće je koristiti i ključnu riječ *final*. Njome se označava finalna (konačna) implementacija virtualne metode koja se ne može dalje nadrediti u nekoj od izvedenih klasa.

```
class Zivotinja{
public:
	// različiti tipovi životinja mogu "pričati" na različite načina
	virtual void pricaj(){
```

```
		cout << "Zivotinja prica" << endl;
	}
};
class Pas : public Zivotinja{
public:
	// final – svi psi "pričaju" na isti način
	virtual void pricaj() override final {
		cout << "Woof!" << endl;
	}
};
class Terijer : public Pas{
public:
	void pricaj() override { // greška prevoditelja!
		cout << "Woof!?" << endl;
	}
};
```

Iako različiti tipovi životinja mogu "pričati" na različite načine, svi psi "pričaju" na isti način (laju). Zbog toga je u klasi *Pas* prilikom implementacije njene virtualne metode *pricaj* korištena ključna riječ *final*. Njome smo osigurali da ova virtualna metoda ne može imati daljnja nadređenja u svojim izvedenim klasama tj. da svi tipovi pasa "pričaju" na isti način.

8.4. Apstraktne klase

Virtualne funkcije (metode) ne moraju imati tijelo. U tom slučaju se one zovu čiste virtualne funkcije (eng. *pure virtual functions*). Klasa u kojoj se nalazi barem jedna takva funkcija se naziva apstraktna klasa. Moguće je kreirati pokazivače (za potrebe pretvorbe na više), no ne i objekte apstraktnih klasa jer su one zamišljene tek kao nacrti budućih izvedenih klasa.

```
// Apstraktna klasa
class Osoba{
public:
	// čista virtualna funkcija
	virtual void opisPosla() = 0;
};
class Vozac : public Osoba{
public:
	// implementacija metode opisPosla u klasi Vozac!
	void opisPosla(){
		cout << "Vozac!" << endl;
	}
};
```

Čistoj virtualnoj funkciji se na kraju deklaracije dodjeljuje vrijednost 0. Tada prevoditelj zna da ona neće imati tijelo, a time i klasa u kojoj se ona nalazi automatski postaje apstraktna.

Kada koristimo obične virtualne funkcije koje imaju tijelo onda izvedena klasa ne mora nužno imati vlastitu implementaciju te virtualne funkcije. Međutim, to ne vrijedi u slučaju korištenja čiste virtualne funkcije. Tada izvedena klasa obavezno mora imati njenu implementaciju.

9. Predlošci

9.1. Predlošci funkcija

Funkcije su obično pisane na način da koriste unaprijed definirane tipove podataka. Pri svakom pozivu takve funkcije njeni ulazni parametri i povratna vrijednost su istog (unaprijed definiranog) tipa. Ukoliko bi htjeli takvoj funkciji omogućiti da radi isti posao, ali s objektima drugih tipova najčešće bi bili prisiljeni pisati njeno preopterećenje za svaki novi tip. Primjerice,

```
void zamjeni(int &a, int &b){
	int pom = a;
	a = b;
	b = pom;
}
```

Funkcija *zamjeni* se može koristiti samo ukoliko je potrebno zamijeniti sadržaj dvaju varijabli tipa *int*. Da bi zamijenili varijable tipa *double* morali bi napisati preopterećenje ove funkcije koje svugdje umjesto *int* koristi *double* tipove podataka. Za svaki sljedeći tip bi opet trebalo pisati preopterećenje funkcije, što predstavlja ogroman i bespotreban posao. Umjesto toga možemo napisati predložak funkcije *zamjeni*.

```
template<class T>
void zamjeni(T &a, T &b){
	T pom = a;
	a = b;
	b = pom;
}
```

Predložak funkcije definira dio programskog koda koja obavlja isti posao neovisno o tipovima podataka s kojima radi. Njegova deklaracija započinje ključnom riječi `template`, te navođenjem liste generičkih parametara `<class T>`. Tek nakon instantacije predloška (poziva funkcije i predaje argumenata) prevoditelj zamjenjuje generičke tipove pravima te generira i poziva funkciju.

```
int x1 = 5, y1 = 10;
double x2 = 5, y2 = 10;

zamjeni(x1, y1);    // void zamjeni(int, int);
zamjeni(x2, y2);    // void zamjeni(double, double);
```

Prvim pozivom funkcije *zamjeni* su predane varijable *x1* i *y1* tj. objekti tipa *int*. Zbog toga će prevoditelj generirati sljedeću funkciju:

```
void zamjeni(int &a, int &b){
	int pom = a;
	a = b;
	b = pom;
}
```

Svako navođenje generičkog parametra T je zamijenjeno tipom *int*. Štoviše, i pomoćna varijabla unutar tijela funkcije je tipa *int*. Odmah nakon generiranja ove funkcije njoj će biti predani objekti *x1* i *y1* nakon čega slijedi njeno izvršavanje. Slično se događa i u drugom pozivu funkcije *zamjeni* kada se joj predaju objekti tipa *double*.

Predložak funkcije može imati više različitih generičkih parametara.

```
template <class T1, class T2>
T1 suma(T1 a, T2 b){
	return a + b;
}
…
cout << suma(4.3, 5) << endl; // 9.3 [double suma(double a, int b)]
cout << suma(5, 4.3) << endl; // 9   [int suma(int a, double b)]
```

Predložak funkcije *suma* radi s dva generička tipa (*T1* i *T2*), a kao povratnu vrijednost vraća *T1*. Logički gledano, u oba poziva funkcije *suma* rezultat bi trebao biti 9.3, no to nije tako jer redoslijed predanih objekata nije isti.

U prvom pozivu funkcije *suma* prvi generički parametar (*T1*) je tipa *double* pa je i povratna vrijednost tipa *double (9.3)*. U drugom slučaju T1 je tipa *int*, pa se i povratna vrijednost predstavlja tipom *int (9)*. Da bi se izbjeglo ovakve komplikacije C++11 standard omogućuje drukčiju definiciju predloška funkcije:

```
template <class T1, class T2>
auto suma(T1 a, T2 b) ->decltype(a + b){
	return a + b;
}
```

U ovoj deklaraciji predloška definirali smo da se povratna vrijednost funkcije dobije tek nakon evaluacije izraza *a + b*. Nakon što se izračuna rezultat tog izraza, njegov tip možemo

doznati korištenjem ključne riječi *decltype*, nakon čega će funkcija *suma* imati upravo taj tip kao svoju povratnu vrijednost. Zbog toga će prethodni pozivi funkcije *suma* dati identični rezultat (9.3).

9.2. Predlošci klasa

Osim predložaka funkcija moguće je pisati i predloške klasa. Upotrebom liste generičkih parametara predloškom klase opisujemo buduće stvarne klase, njihove atribute i metode. Primjere već možemo naći u kontejnerima poput vektora, liste, mape itd., a pomoću predložaka klasa možemo pisati i vlastite kontejnere.

```
template<class T>
class MojKontejner{
public:
	T* polje;
	int BrojElemenata;
	MojKontejner(int n);
	~MojKontejner();
	T& operator [](int indeks);
};
template <class T>
MojKontejner<T>::MojKontejner(int n){
	BrojElemenata = n;
	polje = new T[n];
}
template <class T>
T& MojKontejner<T>::operator[](int indeks){
	return polje[indeks];
}
template <class T>
MojKontejner<T>::~MojKontejner(){
	delete[] polje;
}
```

Predložak klase *MojKontejner* predstavlja nekakvu buduću stvarnu klasu koja će biti kontejner objekata proizvoljnog tipa. Konstruktorom klase se alocira interno polje za objekte veličine *n*, a taj podatak se sprema u podatkovni član *BrojElemenata*. Pri uništenju svake instance ovog kontejnera poziva se destruktor koji dealocira to interno polje, a za jednostavniji pristup svakom objektu unutar kontejnera je implementiran operator *[ ]*.

```
template <class T> T& MojKontejner<T>::operator[](int indeks){
	return polje[indeks];
}
```

Tijelo metode predloška klase se piše na vrlo sličan način kao i kod obične klase. Prvo se navodi ključna riječ *template* i lista generičkih parametara. Tek tada se navodi povratna vrijednost metode, klasa kojoj pripada, njen naziv i parametri.

Napisani predložak klase možemo instancirati na sljedeći način:

```
MojKontejner<int> intPolje(10);
```

Ovom deklaracijom objekt *intPolje* predstavlja kontejner objekata tipa *int*. Točnije, svugdje gdje se u predlošku klase *MojKontejner* koristio generički tip *T* sada se koristi stvarni tip *int*. Konstruktorom smo definirali da taj kontejner sadrži 10 objekata, a svima im možemo jednostavno pristupiti pomoću napisanog operatora *[ ]*.

```
for (int i = 0; i < intPolje.BrojElemenata; i++){
	cin >> intPolje[i];  // intPolje.polje[i]...
	// ...
}
```

Korištenje kontejnera *intPolje* podsjeća na korištenje običnog polja. Međutim, za razliku od običnog polja ovaj kontejner ima i podatkovni član *BrojElemenata*. Također, predložak ove klase možemo proširiti na način da napišemo metode za dodavanje, brisanje, pretragu članova u kontejneru, proizvoljna preopterećenja raznih drugih operatora itd.

Predložak klase *MojKontejner* smo mogli napisati i puno jednostavnije proširenjem liste parametara.

```
template<class T, unsigned int size>
class MojKontejner{
public:
	T polje[size];
	int BrojElemenata;
	MojKontejner() : BrojElemenata(size){}
};
```

Broj elemenata više nije definiran konstruktorom klase već parametrom predloška *size*. Zato se instanciranje ovog predloška može izvršiti na sljedeći način:

```
MojKontejner<int, 10> X;
```

Ovaj način je mnogo bolji jer se veličina polja zna prije samog instanciranja predloška. Zbog toga se može u potpunosti izbjeći dinamička alokacija u konstruktoru i dealokacija u destruktoru predloška klase.

Također, instancu predloška jedne klase se može kreirati pomoću predloška druge klase.

```
template<class T>
class A{
public:
	//...
};
…
vector <A <int> > V; // instanciranje vektora V pomoću predloška klase A
```

Vektor *V* je instanca predloška klase *vector* koja je instancirana pomoću predloška klase *A*. Ovdje je potrebno paziti da se prilikom instanciranja napravi razmak između zadnje dvije „>“ zagrade. Ukoliko bi instanciranje bilo napisano kao `vector <A <int>> V;` prevoditelj bi zadnje dvije zagrade mogao shvatiti kao operator >> te javiti grešku pri prevođenju.

Predložak klase se može nalaziti i kao dio obične klase. Štoviše, moguće je napisati predložak klase unutar predloška klase.

```
template<class T1>
class A{
public:
	template<class T2>
	class B{
	public:
		void f(T2 x);
	};
};
// implementacija metode f
template<class T1>template<class T2>
void A<T1>::B<T2>::f(T2 x){
	//...
}
```

Instanciranje ovakvog predloška bi mogli napraviti deklaracijom `A<int>::B<float> obj;`, a nad instancom *obj* je sada moguće pozvati metodu *f*.

Predloške klasa je također moguće koristiti pri nasljeđivanju. Moguće je nekoliko različitih scenarija:

a) Predložak klase je bazna klasa nekoj običnoj klasi. U ovom slučaju predložak klase mora biti konkretiziran (ne koriste se generički tipovi parametara).

```
class Ravnina : public Kontejner<Tocka>
```

b) Predložak klase nasljeđuje običnu klasu. Predložak klase koristi generičke tipove parametara.

```
template <class T> class Cjepanica : public Drvo
```

c) Predložak klase nasljeđuje predložak klase. Bazna klasa mora biti konkretizirana tj. bez generičkih parametara.

```
template <class T> class Bicikl : public Vozilo<DvaKotaca>
```

Od ostalih mogućnosti predložaka klasa mogu se spomenuti podrazumijevani tipovi generičkih parametara. Ukoliko oni postoje programer nije dužan navoditi stvarni tip prilikom instanciranja predloška klase. Primjerice,

```
template<class T = double, unsigned int size = 10>
class MojKontejner{
public:
	T polje[size];
	int BrojElemenata;
	MojKontejner() : BrojElemenata(size){}
};
…
MojKontejner<> X; // instanciranje predloška
cout << typeid(X).name() << endl;  // class MojKontejner<double, 10>
cout << X.BrojElemenata << endl;   // 10
```

Predložak klase *MojKontejner* podrazumijevano koristi tip *double* za generički parametar *T*. Osim njega, u listi parametara se nalazi i *size* s podrazumijevanom vrijednosti 10. Zbog toga je ovaj predložak klase dovoljno instancirati korištenjem praznih zagrada <>. Napomenimo i da parametar *size* nije generički, već stvarni (konkretni) parametar. Zato se pri instanciranju ovog predloška treba navesti njegova vrijednost ukoliko je ona različita od podrazumijevane (10). Primjerice,

```
MojKontejner<char, 25> X;
```

Podrazumijevani tipovi parametara su jako dugo vremena bili ograničeni samo na predloške klasa. Tek uvođenjem C++0X tj. C++11 standarda moguće ih je koristiti i pri radu s predlošcima funkcija.

9.3. Specijalizacija predloška

U većini slučajeva programski kod definiran predloškom radi zadovoljavajuće sa svim tipovima podataka. Ipak, moguće su situacije kada je za određeni tip podatka potrebno napraviti modifikaciju inicijalnog predloška. Tada to zovemo specijalizacijom predloška za zadani tip.

Specijalizirati se mogu predlošci funkcija i predlošci klasa. Podrazumijevano, unutar predloška klase je moguće specijalizirati njene metode, statičke podatkovne članove, druge klase unutar predloška klase itd.

```
// inicijalni predložak funkcije Kvadrat
template <class T>
T Kvadrat(T n){
	return n * n;
}
// specijalizacija predloška funkcije Kvadrat za tip Kompleksni
template <>
Kompleksni Kvadrat(Kompleksni Z){
	return Kompleksni(Z.re * Z.re - Z.im * Z.im, 2 * Z.re * Z.im);
}
…
cout << Kvadrat(10) << endl;	// 100
cout << Kvadrat(5.5) << endl;	// 30.25
Kompleksni Z = Kvadrat(Kompleksni(1, 2));
cout << Z.re << " + " << Z.im << "i\n"; // -3 + 4i
```

Inicijalni predložak funkcije *Kvadrat* zadovoljava kada je riječ o većini tipova podataka. Ipak, kvadrat kompleksnog broja se računa drukčije nego li za ostale tipove podataka. Stoga je u tu svrhu napisana specijalizacija predloška funkcije *Kvadrat* za tip *Kompleksni*.

Specijalizacija započinje izrazom `template <>`. Unutar zagrada <> se ne navode generički parametri jer se specijalizacija vrši za točno određeni tip podatka (*Kompleksni*). U nastavku specijalizacije predloška funkcije umjesto generičkog tipa *T* koristimo stvarni tip *Kompleksni*, te pišemo drukčiju inačicu tijela funkcije.

Specijalizacija funkcije *Kvadrat* se mogla izbjeći preopterećenjem operatora množenja (*) unutar klase *Kompleksni*. Međutim, ukoliko nemamo pristup deklaraciji i implementaciji ove klase to onda nije moguće.

Specijalizacija predloška klase se piše na vrlo sličan način.

```
// inicijalni predložak klase
template <class T>
class MojKontejner{
public:
	T *polje;
	int brElem;
	MojKontejner(int n) : brElem(n){ /* alociraj polje.. */ }
	~MojKontejner() { /* dealociraj polje.. */ }
};
// specijalizacija predloška klase MojKontejner za tip double
template <>
class MojKontejner <double>{
public:
	double *polje;
	int brElem;
	MojKontejner(int n) : brElem(n){ /* alociraj polje.. */ }
	~MojKontejner() { /* dealociraj polje.. */ }
	double suma();
	double prosjek();
};
…
MojKontejner<int> intKontejner(10);
MojKontejner<double> doubleKontejner(10);
intKontejner.suma();	// greška! Ovaj tip kontejnera nema metodu suma.
doubleKontejner.suma();	// ispravno!
```

Predložak klase *MojKontejner* ima specijalizaciju za tip *double*, pa će svaki kontejner tog tipa moći koristiti dodatne metode *suma* i *prosjek*. Ukoliko pokušamo pristupiti jednoj od tih metoda iz kontejnera koji nije tipa *double* prevoditelj uredno javlja grešku.

Specijalizacija predložaka ne mora biti potpuna. Neki parametri se mogu konkretizirati u specijalizaciji predloška, dok se ostali parametri mogu i dalje ostaviti kao nepoznati. U tom slučaju je riječ o tek djelomičnoj (parcijalnoj) specijalizaciji predloška.

```
// inicijalni predložak klase
template <class T, int n>
class MojKontejner{
public:
	T polje[n];
	int brElem;
	MojKontejner() : brElem(n){}
};
```

Ovaj predložak klase ima dva parametra. Jedan je generičkog a drugi konkretnog tipa. Mogući primjeri specijalizacije su:

```
template <>
```

```
class MojKontejner < double, 4 > { }; // potpuna specijalizacija

template <int n>
class MojKontejner < double, n > { }; // djelomična specijalizacija
```

Djelomična specijalizacija je moguća samo kod predložaka klasa, dok njeno korištenje kod predloška funkcija trenutno nije podržano standardom.

9.4. Predlošci standardne biblioteke

Standardna biblioteka (STL) sadrži nekoliko skupina predložaka. Jedni od najčešće korištenih su kontejneri tj. predlošci klasa za spremanje podataka. Pri radu s kontejnerima često se koriste i algoritmi tj. predlošci funkcija te iteratori koji omogućuju pristup podacima unutar različitih tipova kontejnera.

Kontejnera postoji nekoliko vrsta, a razlikuju se po načinu organizacije podataka. Primjerice, kontejner *vector* je sekvencijalni kontejner koji se najčešće koristiti kao zamjena za statički i dinamički alocirani niz. Svaki vektor je dinamički proširiv te sadrži podatkovne članove i metode pomoću kojih se jednostavnije radi s podacima u samom vektoru.

```
vector<int> niz1;          // prazan vektor
vector<int> niz2(10);      // 10 elemenata, svi imaju vrijednost 0
vector<int> niz3(10, 1);   // 10 elemenata, svi imaju vrijednost 1
vector<int> niz4 = { 1, 2, 3, 4, 5 };   // 5 elemenata s vrijednostima 1-5
```

Vektore je je moguće kreirati i inicijalizirati na više načina. Vektor *niz1* je prazan vektor. Da bi u njega stavili neku vrijednost koristimo metodu *push_back*.

```
niz1.push_back(1); // dodaj vrijednost 1 u vektor
```

Ova poziv metode *push_back* će dinamički proširiti vektor za još jedan član. On se dodaje na kraj vektora te inicijalizira s vrijednosti 1. U suprotnome, ukoliko je potrebno izbrisati zadnji član vektora može se koristiti metoda *pop_back*.

Svaka promjena veličine vektora može uzrokovati i njegovu realokaciju. To je postupak koji u ovisnosti o veličini vektora oduzima dosta procesorskog vremena. Zato je za vektore

uvijek poželjno unaprijed rezervirati određenu količinu elemenata. To je moguće korištenjem metode *reserve*.

```
niz1.reserve(20);
niz1.push_back(1);
cout << niz1.capacity() << endl; // 20
cout << niz1.size() << endl;     // 1
```

Broj rezerviranih elemenata ne utječe na trenutnu veličinu vektora, a tek kada ona pređe trenutni kapacitet dogoditi će se realokacija vektora. Ukoliko već unaprijed znamo koliko nam elemenata treba vektore možemo kreirati kao u primjerima za *niz2* i *niz3*.

```
vector<int> niz2(10);     // 10 elemenata, svi imaju vrijednost 0
vector<int> niz3(10, 1);  // 10 elemenata, svi imaju vrijednost 1
```

U ove dvije deklaracije razlika je tek u početnoj inicijalizaciji svih elemenata vektora. Vektor *niz2* će podrazumijevano inicijalizirati sve elemente na vrijednost 0, dok *niz3* na vrijednost 1.

```
vector<int> niz4 = { 1, 2, 3, 4, 5 };  // 5 elemenata s vrijednostima 1-5
```

Tek od usvajanja C++11 standarda vektori se mogu kreirati i inicijalizirati pomoću inicijalizacijskog popisa. Vektor *niz4* ima točno 5 elemenata a njihove vrijednosti su redom od 1 do 5.

Vektori imaju implementiran i operator *[ ]*. Njime se omogućuje da na isti način pristupamo elementima kao i u slučaju statičkog polja.

```
for(int i = 0; i < niz4.size(); i++)
	cout << niz4[i] << endl;
```

Pristup preko indeksa je preporučljiv samo u slučaju rada s *vector* klasom. U ostalim kontejnerima se zbog drukčije organizacije podataka preporučuje korištenje iteratora.

```
vector<int> ::iterator it;
for (it = niz4.begin(); it != niz4.end(); it++)
	cout << *it; // 12345
```

Iteratori su pokazivači na elemente kontejnera. Svaki iterator je točno određenog tipa, te se može koristiti samo s kontejnerom tog tipa. Primjerice, kreirani iterator *it* se može koristiti samo pri radu s vektorom koji sadrži *int* tipove podataka.

Iteratori mogu biti i reverzni, time omogućujući prelazak vektora u obrnutom redoslijedu (od kraja prema početku).

```
vector<int> ::reverse_iterator it;
for (it = niz4.rbegin(); it != niz4.rend(); it++)
	cout << *it << endl; // 54321
```

Pomoću standardne biblioteke se mogu koristiti i algoritmi u kombinaciji s kontejnerima i iteratorima. Moguće je izvršiti operacije poput podjele kontejnera na dijelove, sortiranje, permutacije elemenata, binarno pretraživanje itd.

```
bool veci(int a, int b){
	return a > b;
}

…
// slučajno izmiješaj elemente u vektoru
random_shuffle(niz4.begin(), niz4.end());
// sortiraj elemente u vektoru (od manje vrijednosti prema većoj)
sort(niz4.begin(), niz4.end());
// sortiraj elemente u vektoru (od veće vrijednosti prema manjoj)
sort(niz4.begin(), niz4.end(), veci);
```

Ovo su tek neki od primjera algoritama koje je moguće koristiti pri radu s kontejnerima. Detaljniji popis moguće je pronaći na http://www.cplusplus.com/reference/algorithm/.

Svi ostali kontejneri se razlikuju tek po načinu organizacije podataka. Sukladno njoj imaju dodatne metode koje omogućuju lakši rad s pojedinim kontejnerom.

```
list<int> lista = { 1, 2, 3, 4, 5 };
```

Kontejner *list* predstavlja dvostruko povezanu listu. Ovom kontejneru je moguće dodati elemente na početak i na kraj liste metodama *push_front* i *push_back*. Sukladno tome, postoje i metode *pop_front* i *pop_back* za brisanje elemenata. Ovaj kontejner se može prelaziti isključivo iteratorom. Ukoliko vam je potrebna lista samo u jednom smjeru moguće je koristiti kontejner *forward_list*, koji je podržan C++11 standardom.

Od ostalih kontejnera potrebno je spomenuti asocijativne kontejnere *map* i *set*. Kontejner *map* se sastoji od kolekcije parova (ključ i vrijednost) gdje se ključ može pojaviti samo jedanput.

```
map<string, int> grad;     // naziv grada i broj stanovnika
grad["Zagreb"] = 800000;
grad["Split"] = 200000;
grad["Osijek"] = 120000;

cout << grad["Zagreb"];    // 800000
```

Mapa je kontejner koji sve elemente automatski sortira po ključu. Pojedinom elementu se može pristupiti pomoću operatora *[ ]*, nakon čega mu je moguće dodijeliti ili pročitati vrijednosti. Ukoliko želimo proći kroz cijeli sadržaj mape koristimo iteratore. Iteratori mapa imaju dva posebna podatkovna člana. Prvi (*first*) predstavlja ključ, a drugi (*second*) vrijednosti elementa.

```
map<string, int> ::iterator it;
for (it = grad.begin(); it != grad.end(); it++)
      cout << it->first << " ima " << it->second << " stanovnika!" << endl;
```

```
Osijek ima 120000 stanovnika!
Split ima 200000 stanovnika!
Zagreb ima 800000 stanovnika!
```

Ispis je sortiran po nazivu grada upravo zbog automatskog sortiranja mape po ključu.

Vrlo sličan kontejneru *map* je kontejner *set*. On također sadrži sortirane podatke, no osnovna razlika je što se podaci u *set*-u ne mogu mijenjati nakon što ih se jednom postavi. Moguće ih je tek izbrisati pa ponovno umetnuti. Usto, vrijednost elementa u *set* kontejneru služi i za njegovu identifikaciju.

Oba kontejnera postoje i u varijanti kada dopuštaju ponavljanje istih ključeva. Tada je riječ o *multimap* i *multiset* kontejnerima. Pojavom C++11 standarda uvedeni su i ne-sortirani kontejneri ovog tipa pomoću kojih se dobiva na brzini. Puni pregled svih kontejnera je moguće vidjeti na http://www.cplusplus.com/reference/stl/.

9.5. Metoda emplace_back vs. push_back

Izlaskom C++11 standarda kontejneri su dobili mnoge dodatne optimizacije, a jedna od njih je i mogućnost direktnog kreiranja i inicijalizacije objekta unutar kontejnera. Za korištenje ove mogućnosti potrebno je koristiti metodu *emplace* tj. *emplace_back*. Primjerice, pri radu s vektorima koristili bi sljedeći programski isječak.

```
class Student{
public:
	string ime;
	string jmbag;
	Student(string _ime, string _jmbag) : ime(_ime), jmbag(_jmbag) {}
};
// ...
vector<Student> ObjektnoProgramiranje;
ObjektnoProgramiranje.push_back(Student("Ante", "024600112233"));
```

Pri pozivu metode *push_back* događaju se tri zasebne operacije.

1) Kreira se privremeni objekt tipa *Student.*
2) U vektoru *ObjektnoProgramiranje* se kreira objekt u koji će se spremiti prethodni privremeni objekt.
3) Vrši se kopiranje sadržaja iz privremenog objekta u kontejner

Da bi ovaj postupak optimizirali možemo koristiti metodu *emplace_back* koja ove tri operacije svodi tek samo na jednu. Primjerice;

```
ObjektnoProgramiranje.emplace_back("Ante", "024600112233");
```

Metoda *emplace_back* kao argument ne prima objekt koji treba staviti u kontejner. Umjesto toga prima vrijednosti parametara za konstruktor tog objekta tako da se taj objekt stvori i inicijalizira u samom kontejneru. Na ovaj način se izbjeglo kreiranje privremenog objekta a i kasnije kopiranje (prethodne operacije 1 i 3).

9.6. Predlošci s neograničenim brojem argumenata

Sve do pojave C++11 standarda programeri su bili ograničeni na predloške funkcija i klasa s fiksnim brojem argumenata. Ukoliko bi predlošku trebalo proslijedili drukčiji broj

argumenata obično bi se vršilo njegovo preopterećenje ili bi se kreirao predložak s određenim brojem podrazumijevanih tipova argumenata. Ipak niti ta rješenja u potpunosti ne zadovoljavaju jer stvarni broj predanih argumenata ne mora odgovarati niti jednom od predviđenih slučajeva. Upravo zbog toga C++11 donosi podršku za predloške funkcija i klasa s proizvoljnim brojem argumenata (*eng. variadic templates*).

```
template <class ...T> void ispis(T ...argumenti);
```

Kako bi se predlošku proslijedio proizvoljan broj argumenata koristiti se poseban parametar s tri točke. Parametar *…T* predstavlja paket svih predanih argumenata, a njih može biti 0 ili neodređen broj. Broj predanih argumenata se može doznati i upotrebom operatora *sizeof* unutar tijela predloška na sljedeći način.

```
cout << sizeof...(argumenti);  // broj predanih argumenata
```

Na koji način napisani predložak funkcije dohvaća i obrađuje sve argumente? Da bi to bilo moguće prethodni predložak je potrebno modificirati na način da se iz njega naprave dva predloška. Prvi od njih opisuje ponašanje funkcije kada joj se preda najmanji broj potrebnih argumenata, te drugi koji prihvaća i obrađuje sve ostale predane argumente.

Primjerice, predlošku funkcije *ispis* treba predati minimalno 1 podatak (argument) jer inače neće imati što ispisati. Stoga, prvo pišemo predložak koji obrađuje taj slučaj.

```
// predložak funkcije ispis s minimalnim brojem potrebnih argumenata (1)
template <class T> void
ispis(T argument){
	cout << argument;
}
```

Nakon što je napisan predložak s minimalnim brojem potrebnih argumenata piše se predložak koji obrađuje sve ostale slučajeve.

```
// predložak funkcije ispis s proizvoljnim brojem argumenata
template <class T, class... T2>
void ispis(T prvi, T2... ostali){
	cout << prvi;// ispis prvog argumenta
	ispis(ostali...);   // prosljeđivanje ostalih argumenata rekurzijom
}
```

Drugi predložak otkriva na koji način funkcioniraju predlošci s neograničenim brojem argumenata. Naime, nizom rekurzivnih poziva funkcije *ispis* prosljeđuje se jedan po jedan argument sve dok se ne dođe do zadnjeg. Upravo zbog zadnjeg poziva funkcije *ispis* tj. kada joj se predaje samo 1 argument je napisan prvi predložak jer će se njegovim pozivom prekinuti rekurzija. Demonstrirajmo na sljedećem pozivu funkcije *ispis*.

```
ispis(1,2,3); // ispisuje 1, poziva ispis(2, 3)
    // ispis(2, 3) - ispisuje 2, poziva ispis(3)
        // ispis(3) - ispisuje 3, prekid rekurzije
```

Pri pozivu funkcije *ispis(1,2,3)* izvršava se tijelo drugog predloška tj. parametar *prvi* ima vrijednost 1, dok su argumenti 2 i 3 spremljeni u parametar *…ostali*. Ispisuje se prvi argument tj. vrijednost 1, dok se ostali argumenti prosljeđuju sljedećem rekurzivnom pozivu *ispis (2, 3)*. I u ovom slučaju se izvršava tijelo drugog predloška koje ispisuje vrijednost 2 a zatim poziva *ispis(3)*. Sada se izvršava tijelo prvog predloška i rekurzija završava.

```
// predložak funkcije s minimalnim brojem potrebnih argumenata (2)
template<class T1, class T2>
bool Razliciti(T1 prvi, T2 drugi){
	return prvi != drugi;
}
// predložak funkcije s proizvoljnim brojem argumenata
template<class T1, class T2, class ...T3>
bool Razliciti(T1 prvi, T2 drugi, T3 ...ostali){
	return prvi != drugi && Razliciti(prvi, ostali...) && Razliciti(drugi, ostali...);
}
```

Ovaj primjer prikazuje situaciju kada pokušavamo napisati predložak funkcije *Razliciti* koja vraća da li su svi predani argumenti različiti jedan od drugoga. Sada je broj minimalnih argumenata jednak 2 jer minimalno možemo usporediti dva podatka (argumenta). Zbog toga prvi predložak ima dva parametra, a drugi predložak uz dva parametra i paket parametara kako bi se moglo usporediti proizvoljno veliki broj argumenata. Primjer poziva funkcije *Razliciti* bi bio sljedeći.

```
if (Razliciti(1, -1, 3, 2, 5))
	cout << "Svi brojevi su razliciti!\n";
else
	cout << "Postoje dva ili vise istih brojeva!\n";
```

U oba prikazana primjera povratna vrijednost predloška funkcije je bila unaprijed definiranog tipa (*void, bool*). Ipak, u nekim slučajevima povratni tip je potrebno definirati

tek naknadno. U tu svrhu možemo upotrijebiti nekoliko različitih pristupa koje nudi C++11 standard, a jedan od njih smo već vidjeli pri pisanju običnog predloška funkcije *suma*.

```
template <class T1, class T2>
auto suma(T1 a, T2 b) ->decltype(a + b);
```

Povratna vrijednost ovog predloška se automatski određuje nakon izračunavanja izraza *a + b*. Ovo smo upotrebom C++11 standarda mogli napisati i na sljedeći način.

```
template <class T1, class T2>
typename std::common_type<T1, T2>::type suma(T1 a, T2 b);
```

Umjesto automatske dedukcije tipa koristili smo izraz *common_type* koji će u vrijeme instanciranja predloška odrediti zajednički od svih korištenih tipova podataka. Proširimo ovaj predložak na način da se mogu sumirati neodređene količine brojeva.

```
// suma minimalno dva podatka
template <class T1, class T2>
typename std::common_type<T1, T2>::type suma(T1 a, T2 b){
	return a + b;
}
// ukoliko treba sumirati više od dva podatka
template <class T1, class T2, class... T3>
typename std::common_type<T1, T2, T3...>::type suma(T1 a, T2 b, T3... ostali){
	return a + suma(b, ostali...);
}
```

Slijedeći logiku iz prethodnih primjera zaključujemo da predložak funkcije *suma* treba imati minimalno 2 parametra jer mora zbrojiti minimalno dva podatka. Stoga su napisana dva predloška koji odgovaraju tim zahtjevima. Treba napomenuti da smo drugi predložak možda htjeli napisati upotrebom automatske dedukcije tipa. To bi bilo na sljedeći način.

```
// podržano u C++14 standardu
template <class T1, class T2, class... T3>
auto suma(T1 a, T2 b, T3... ostali){
	return a + suma(b, ostali...);
}
```

C++17 standard omogućuje da korištenjem sažetih (eng. *fold*) izraza postojeća dva predloška napišemo kraće tj. korištenjem samo jednog predloška:

```
// podržano u C++17 standardu
template <class... T>
auto suma(const T&... argumenti){
	return (argumenti + ...);
```

```
}
```

Napisani predložak funkcije *suma* također prima neograničen broj argumenata te vraća njihovu sumu. Ipak, za prevođenjeg ovog koda potreban je noviji compiler.

Predloške s neodređenim brojem argumenata je moguće koristiti i pri pisanju klasa. Njima se omogućava pisanje klasa koje osim generičkih tipova podržavaju i neodređen broj članova.

```
// predložak klase Skup s neodređenim brojem članova
template<class T, class ...TArgs>
class Skup{
private:
	T prvi;
	Skup<TArgs...> ostali;
public:
	// konstruktor za inicijalizaciju svakog člana
	Skup(T prvi, TArgs ...ostali) : prvi(prvi), ostali(ostali...){}
};
// parcijalna specijalizacija klase Skup za tip T
template<class T>
class Skup<T>{
private:
	T tip;
public:
	Skup(T t) : tip(t){}
};
```

Rekurzivno instanciranje predloška se događa i pri pisanju klasa s neodređenim brojem argumenata. Zato i u ovom slučaju moramo imati predložak koji će prekinuti rekurziju tj. specijalizaciju za samo jedan tip. Predložak klase *Skup* je sada moguće instancirati na sljedeći način.

```
Skup<int, double> par(1, 3.14);
```

Podatkovni članovi instance *par* su *int* i *double* tipa, a pomoću konstruktora su odmah inicijalizirani na svoje početne vrijednosti 1 i 3.14. Ovi podatkovni članovi nemaju svoje ime, već da bi im se kasnije moglo pristupiti treba napisati vlastite predloške *get* metoda koje ovim podatkovnim članovima pristupaju pomoću njihovog indeksa.

Da bi se izbjegle sve te komplikacije pri pisanju ovakvih predložaka klasa programeri mogu koristiti već pripremljeni predložak klase *tuple* (zaglavlje *tuple*). Njega je dovoljno tek instancirati, nakon čega se za pristup članovima može koristiti već postojeća funkcija *std::get*.

```
#include <tuple>
...
tuple<int, double> skup = std::make_tuple(1, 3.14);
cout << get<0>(skup) << endl; // 1
cout << get<1>(skup) << endl; // 3.14
```

Klasa *tuple* jedinstveno predstavlja svaki predložak klase s neodređenim brojem argumenata (članova), dok funkcija *get* predstavlja svaki član u toj klasi pomoću njegovog indeksa. Svakom članu se može pročitati ali i mijenjati vrijednosti. Primjerice,

```
get<0>(skup) = 10; // zapiši vrijednost 10 u član s indeksom 0 (int)
```

Kao parametar ovakvoj instanci klase možemo predati i funkcije.

```
#include <tuple>
#include <functional>
...
tuple<int, double, function<double(double, double)>>
skup = std::make_tuple(1, 3.14, [](double a, double b){return a + b;});

cout << get<0>(skup) << endl; // 1
cout << get<1>(skup) << endl; // 3.14
cout << get<2>(skup)(get<0>(skup), get<1>(skup)); // 4.14
```

Osim dva podatkovna člana koji su tipa *int* i *double* instanca *skup* sadrži i funkciju koja ima povratnu vrijednost *double*, te dva parametra tipa *double*. Pri inicijalizaciji u konstruktoru za taj član treba predati ime postojeće funkcije, ili kao u gornjem slučaju napisati tijelo anonimne lambda funkcije. Pristup toj funkciji je i dalje pomoću funkcije *get* (pogledati zadnju liniju).

10. Funkcijski objekti i lambda funkcija

10.1. Funkcijski objekti

Ukoliko klasa sadrži preopterećenje funkcijskog operatora () tada je takvu klasu moguće koristiti za kreiranje funkcijskih objekata. Oni su zamišljeni kao objekti koje je moguće koristiti tamo gdje je potreban poziv funkcije. Najčešće ih se koristi kao argumente pri pozivu drugih funkcija.

```
class Veci{
public:
	bool operator()(int a, int b){
		return a > b;
	}
};
```

Klasa *Veci* sadrži preopterećenje funkcijskog operatora (). On kao parametre prima dva cijela broja te vraća da li je prvi broj veći od drugoga. Primjenu funkcijskog objekta ove klase možemo prikazati pri pozivu metode *sort* (zaglavlje *algorithm*).

```
vector<int> v = { 1, 2, 3, 4, 5 };
sort(v.begin(), v.end(), Veci());

for (int i = 0; i < v.size(); i++)
	cout << v[i];  // 54321
```

Vektor *v* sadrži 5 elemenata. Pozivom metode *sort* želimo sortirati taj vektor od veće vrijednosti prema manjoj. Zato kao zadnji argument navodimo privremeni funkcijski objekt klase *Veci*. Naime, metoda *sort* za zadnji parametar želi ime druge funkcije koja definira kriterij za sortiranje. Takva funkcija mora imati dva ulazna parametra kako bi se naizmjenice mogli usporediti dva po dva podatka.

Privremeni funkcijski objekt *Veci* zadovoljava taj kriterij jer njegov funkcijski operator () ima dva ulazna parametra. Pri svakom pozivu funkcijskog objekta *Veci* metoda *sort* će njegovom funkcijskom operatoru predati dva podatka koja treba usporediti, a on će vratiti koji od njih je veći.

Funkcijski objekti još uvijek imaju čestu primjenu u praksi, ali postupnim uvođenjem C++11 standarda sve češće se zamjenjuju lambda funkcijama.

10.2. Lambda funkcije

Jedna od najboljih mogućnosti novog C++11 standarda podrška je za lambda funkcije. Njima se omogućuje umetanje funkcijskog bloka tamo gdje bi inače trebao biti poziv funkcije. Najčešće se koriste u slučajevima gdje parametar jedne funkcije treba biti druga funkcija. Lambda funkcije nemaju identifikator pa se često zovu i anonimne funkcije. Promotrimo sljedeći primjer.

```
int sumaIzabranih(const vector<int> &v, function<bool(int)>kriterij){
	int s = 0;
	for (int i = 0; i < v.size(); i++) {
		if(kriterij(v[i])) // ukoliko v[i] zadovoljava kriterij
			s += v[i];
	}
	return s;
}
```

Funkcija *sumaIzabranih* vraća sumu izabranih elemenata vektora. Ova funkcija ne zna kriterij po kojemu se izabiru elementi za sumiranje već se kriterij određuje nekom drugom funkcijom koja se predaje kao dodatni parametar (*function<bool(int)>kriterij*).

Obično u slučajevima kada funkciju predajemo drugoj funkciji koristimo pokazivače na funkcije, no izraz *function* (zaglavlje *functional*) omogućava nam predavanje obične funkcije, funkcijskog objekta te čak i lambda funkcije kao parametra drugoj funkciji. U prikazanom primjeru funkcija koja se predaje ima povratnu vrijednost *bool* te ulazni parametar tipa *int*, a u daljnjem kodu na nju ćemo se pozivati korištenjem imena (delegata) *kriterij*.

Da bismo pozvali funkciju *sumaIzabranih* prvo treba napisati funkciju koja definira kriterij po kojemu se vrši sumiranje elemenata u vektoru. Primjerice,

```
bool isParan(int n){
	return !(n%2);
}
…
vector<int> v = {1, 2, 3, 4, 5, 6};
cout << sumaIzabranih(v, isParan) << endl; // 12
```

Pri pozivu funkcije *sumaIzabranih* koristili smo globalnu funkciju *isParan* koja vraća informaciju je li neki broj paran ili nije. Drugim riječima, funkcija *sumaIzabranih* vratit će samo sumu parnih brojeva unutar vektora. Pokušajmo ovo isto izvesti upotrebom funkcijskog objekta.

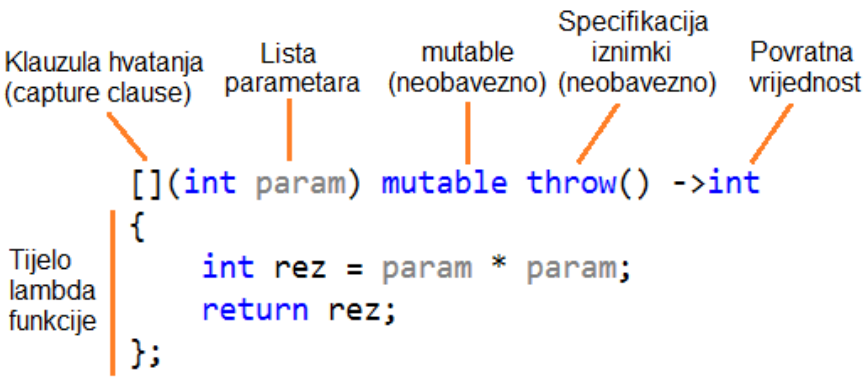
```
class CParan{
	public:
	bool operator()(int n){
		return !(n%2);
	}
};
…
vector<int> v = {1, 2, 3, 4, 5, 6};
cout << sumaIzabranih(v, CParan()) << endl; // 12
```

Korištenjem privremenog funkcijskog objekta tipa *CParan* funkcija *sumaIzabranih* prima kriterij za sumiranje elemenata vektora. Funkcijski objekt unutar svog operatora *( )* vraća informaciju je li neki broj paran ili nije, pa je krajnji rezultat isti kao i u prethodnom slučaju, tj. 12.

Korištenjem lambda funkcije gornje računanje možemo obaviti na sljedeći način.

```
cout << sumaIzabranih(v, [](int n){return !(n%2);}); // 12
```

U samo jednoj liniji koda napravili smo poziv funkcije *sumaIzabranih* te definirali kriterij za sumiranje elemenata vektora. Kriterij je definiran lambda funkcijom. Ovim primjerom vidimo jedan od najjednostavnijih oblika lambda funkcije pa pogledajmo njihov opći oblik.



Slika 10.2.1. Opći oblik lambda funkcije

Svaka lambda funkcija počinje uglatim zagradama *[]* tj. klauzulom hvatanja (eng. *capture clause*). Unutra se navodi popis varijabli s automatskim trajanjem (lokalne varijable) koje lambda funkcija kani koristiti. Još bitnije, u klauzuli hvatanja definiramo na koji način će te varijable biti korištene unutar lambda funkcije. Primjerice,

```
int main(){
int a = 1, b = 2, suma;
```

Ukoliko bismo htjeli napisati lambda funkciju koja računa sumu varijabli *a* i *b* to bismo mogli napraviti na više načina. Primjerice,

```
suma = [](int x, int y){return x + y;} (a, b); // klauzula hvatanja je prazna!
```

Općenito, lambda funkcija će nakon svog poziva generirati anonimni funkcijski objekt s operatorom *( )*. Da bi unutar tog funkcijskog objekta lokalne varijable pozivajuće funkcije (*main*) bile vidljive, obično ih treba prenijeti u sam lambda objekt preko klauzule hvatanja. U ovom slučaju to nije potrebno jer smo varijable *a* i *b* predali kao ulazne parametre lambda funkciji. Te varijable će se preslikati u *x* i *y* i *suma* će imati vrijednost 3.

```
suma = [a, b](){return a + b;}(); // a i b prijenos po vrijednosti, suma = 3
```

Sada lambda funkcija ne koristi ulazne parametre već izravno pristupa varijablama *a* i *b* koje se nalaze u pozivajućoj funkciji *main*. Stoga, te se varijable trebaju prenijeti u generirani lambda objekt preko klauzule hvatanja. To može biti prijenos po vrijednosti i prijenos po referenci. U ovom slučaju varijable *a* i *b* se prenose po vrijednosti te lambda objekt generira njihove kopije.

```
[&suma, a, b](){suma = a + b;}(); // prijenos po referenci, a i b po vrijednosti
```

Trećim načinom prikazano je kako napraviti prijenos po referenci (varijabla *suma*). Varijable *a* i *b* se opet prenose po vrijednosti pa vidimo da načine prijenosa možemo i kombinirati. Na kraju, i u ovom slučaju varijabla *suma* poprima vrijednost 3.

Lambda funkcije ne zahtijevaju da se u klauzuli hvatanja nužno navodi svaka varijabla koja će biti korištena u lambda funkciji. Primjerice,

```
suma = [a, b](){return a + b;}(); // a i b imaju prijenos po vrijednosti
suma = [=](){return a + b;}(); // podrazumijevani je prijenos po vrijednosti
```

Obje linije rade identičnu stvar. Umjesto da navodimo svaku varijablu koju želimo prenijeti po vrijednosti, možemo koristiti *[=]* klauzulu. Ovime definiramo da se za svaku lokalnu varijablu pozivajuće funkcije podrazumijeva upravo prijenos po vrijednosti. Slično se može definirati i za podrazumijevani prijenos po referenci korištenjem *[&]* klauzule.

Ukoliko koristite *[=]* ili *[&]* klauzulu treba napomenuti da se lambda funkciji (njegovom funkcijskom objektu) prenosi samo ono što se u lambda funkciji zaista i koristi. Primjerice,

```
int a = 1, b = 2, suma;
vector<int> veliki_vektor(1000000);
...
suma = [=](){return a + b;}(); // veliki_vektor se ne prenosi lambda funkciji!
```

Lambda funkcija će izvršiti prijenos po vrijednosti tek za varijable *a* i *b*. S obzirom da se objekt *veliki_vektor* ne koristi u lambda funkciji neće se niti izraditi njegova kopija. Prikažimo još neke primjere.

```
[&, a]   // 'a' ima prijenos po vrijednosti. Svi ostali: prijenos po referenci!
[&, &a]  // Greška! Podrazumijevani prijenos je već definiran (po referenci).
[=, &suma] // 'suma' ima prijenos po referenci. Ostali: prijenos po vrijednosti.
```

Globalne i statičke varijable nije potrebno navoditi u klauzuli hvatanja jer su one lambda funkciji automatski dostupne. Primjerice,

```
int globalna = 0;
int main(){
	static int staticna = 0;
	[](){globalna++; staticna++;}();
	cout << globalna << " " << staticna; // 1 1
	return 0;
}
```

Do sada u lambda funkcijama nismo definirali njihovu povratnu vrijednost. Naime, ona se automatski odredi ukoliko tijelo lambda funkcije ima samo jednu naredbu.

```
suma = [=](){return a + b;}(); // automatska dedukcija povratnog tipa
```

Povratna vrijednost ove lambda funkcije odredit će se automatski nakon računanja izraza *a+b*. Primjerice, ukoliko su obje varijable tipa *int* onda će i krajnji rezultat izraza biti tipa *int* pa time i povratna vrijednost lambda funkcije.

```
int a = 1, b = 2, suma;
suma = [=]()->int{
                    cout <<"Prva naredba!\n";
                    return a + b; // druga naredba
             }();
```

Ako tijelo lambda funkcije ima više naredbi stariji prevoditelji će zahtijevati da se eksplicitno navede njen povratni tip (`->int`). Ukoliko se koristi noviji C++ prevoditelj (C++14 ili noviji) ovo nije potrebno, te će povratni tip automatski biti određen bez obzira na broj naredbi u lambda funkciji.

Često je i pitanje čemu služi specifikacija *mutable* pri pisanju lambda funkcije. Naime, prilikom prijenosa varijable po vrijednosti (unutar klauzule hvatanja) ona se u lambda funkciji (njenom anonimnom funkcijskom objektu) ne može mijenjati jer je funkcijski operator () tog objekta deklariran kao tipa *const*. Zbog toga se sljedeći primjer koda neće uspješno prevesti.

```
int a = 1, b = 2, suma;
[=](){suma = a + b;}();
```

Varijabla *suma* prenesena je po vrijednosti, a budući da se ona u anonimnom funkcijskom objektu (lambda funkciji) nalazi u funkcijskom operatoru () tipa *const* nikakve modifikacije te varijable nisu dopuštene. Da bi se zaobišlo to ograničenje koristi se specifikacija *mutable*.

```
[=]()mutable{suma = a + b;}();
```

Sada se prevoditelj neće buniti, no treba biti svjestan da je za varijablu *suma* obavljen prijenos po vrijednosti. Stoga, varijabla *suma* unutar pozivajuće funkcije *main* će i dalje imati nedefiniranu vrijednost.

10.3. Rekurzivne lambda funkcije

Također treba napomenuti da kao i u slučaju običnih funkcija tako i lambda funkcije se mogu pozivati rekurzivno. Primjerice,

```
function<int(int, int)> suma = [&suma](int a, int b) {
	if (a > b)
		return 0;
	return a + suma(a + 1, b);
};
cout << suma(1, 3); // 6
```

S obzirom da lambda funkcije nemaju svoje ime osnovni problem je kako uopće napraviti rekurzivni poziv takve funkcije. Da bi to bilo moguće izrazom *function* se kreira delegat zvan *suma*. On predstavlja lambda funkciju kojoj se treba vratiti rezultat njenog sljedećeg rekurzivnog poziva. Upravo zato se njegova referenca stavlja u klauzulu hvatanja.

11. Zadaci za vježbu

11.01. [VJ-1] Kompleksni broj

Napišite program koji u dinamički alocirano polje učitava N kompleksnih brojeva te ih ispisuje sortirane po vrijednosti njihovih modula (od najmanjeg prema najvećem).

- Kompleksni broj treba predstaviti klasom *Kompleksni* koja sadrži realni i imaginarni dio (realni brojevi).
- Modul kompleksnog broja je udaljenost kompleksnog broja od ishodišta i treba se računati globalnom funkcijom *Modul*.
- Za sortiranje kompleksnih brojeva napišite funkciju *Sortiraj* koja će sortirati kompleksne brojeve upotrebom *BubbleSort* algoritma.

Primjer izvođenja

```
Unesi N: 3
niz[0].re = 3
niz[0].im = -1
niz[1].re = 1
niz[1].im = 1
niz[2].re = 2
niz[2].im = -3
Z(1+1i) Modul: 1.41421
Z(3-1i) Modul: 3.16228
Z(2-3i) Modul: 3.60555
```

Nakon što ste zadatak riješili upotrebom dinamički alociranog polja zamijenite ga vektorom kompleksnih brojeva te sortirajte upotrebom funkcije *std::sort* (zaglavlje *algorithm*).

11.02. [VJ-1] Prosjek ocjena

Napišite program koji u vektor učitava niz studenata, a za svakog studenta i popis njegovih upisanih kolegija te ocjenu iz svakog od tih kolegija. Nakon učitavanja svih podataka potrebno je napisati globalnu funkciju *ProsjekKolegija* koja prima taj vektor i naziv odabranog kolegija te vraća prosjek ocjena svih studenata iz tog kolegija. Studente i kolegije potrebno je predstaviti istoimenim klasama.

Primjer izvođenja

```
Unesite broj studenata: 3

Unesi ime i prezime 1.studenta: Ante Antic
Unesi broj kolegija studenta: 2
Unesite naziv i ocjenu za 1. kolegij: Matematika 3
Unesite naziv i ocjenu za 2. kolegij: Fizika 4

Unesi ime i prezime 2.studenta: Pero Peric
Unesi broj kolegija studenta: 3
Unesite naziv i ocjenu za 1. kolegij: Matematika 4
Unesite naziv i ocjenu za 2. kolegij: Fizika 3
Unesite naziv i ocjenu za 3. kolegij: Programiranje 4

Unesi ime i prezime 3.studenta: Ana Anic
Unesi broj kolegija studenta: 2
Unesite naziv i ocjenu za 1. kolegij: Matematika 4
Unesite naziv i ocjenu za 2. kolegij: Programiranje 5

Unesite naziv kolegija: Matematika
Prosjek ocjena iz kolegija Matematika iznosi 3.66667
```

11.03. [VJ-1] Evidencija garaža

Pretpostavimo da ste vlasnik niza garaža u kojima spremate razne predmete. Napišite program za evidenciju garaža tako da za svaku od njih znate koje predmete sadrži.

Klasa **Garaza** ima sljedeća svojstva:

- veličinu (širina x duljina)
- podatak o tome jesu li vrata s automatskim upravljanjem ili ne
- lokaciju
- **niz predmeta**

Klasa **Predmet** ima sljedeća svojstva:

- naziv
- vrijednost

U glavnom programu potrebno je od korisnika tražiti unos broja **N**, nakon kojeg se unose podaci o N garaža. Nakon što se unesu sva polja, potrebno je tražiti unos broja **M** koji označava broj predmeta koji se nalaze u garažama. Učitajte sve predmete i ispišite sve garaže i predmete koji se u njima nalaze.

Primjer izvođenja

```
Unesite broj garaža (N):2

Unesite velicinu, lokaciju i podatak o vratima za 1. garazu:
10 20
Konavoska 3
DA

Unesite velicinu, lokaciju i podatak o vratima za 2. garazu:
15 25
Konavoska 3b
NE

Unesite broj predmeta (M): 1
Unesite redni broj garaze u koju spada 1. predmet: 2
Unesite naziv i vrijednost predmeta: Lopta 400

Ispis garaza i predmeta:
1. Konavoska 3 10x20 - predmeti:
2. Konavoska 3b 15x25 - predmeti: Lopta (400kn)
```

11.04. [VJ-2] Računi i artikli

Potrebno je napisati program za evidenciju računa. Primjerice, u dućanu svaki kupac ima račun s jedinstvenim rednim brojem, a na tom računu se nalazi popis kupljenih stavki (naziv, količina, jedinična cijena). Slijedeći programski odsječak u nastavku napišite sve potrebne klase i metode kojima se dobije očekivano ponašanje programskog koda.

```
int main(){
	Kupac Ante(Racun(1)); // Ante ima račun broj 1
	Ante.racun.dodaj(Artikl("Jabuka", 1, 6)); // 1 kg, 6 kn/kg
	Ante.racun.dodaj(Artikl("Banana", 2, 7.5)); // 2 kg, 7.5 kn/kg
	Ante.racun.dodaj(Artikl("Coca cola 2l", 2, 10)); // 2 boce, 10 kn/boca
	cout << "Ukupno: " << Ante.racun.ukupnaCijena << " kn" << endl; // 41 kn

	/* U nastavku ispišite koji je najskuplji artikl kojeg je Ante platio
	   (naziv i ukupna cijena). Npr.

	   Najskuplje placeni artikl je Coca cola 2l (20kn)
	*/
	return 0;
}
```

11.05. [VJ-2] Studenti i bodovi

Potrebno je napisati program za praćenje bodovnog stanja studenata na određenom kolegiju. Slijedeći programski odsječak u nastavku napišite sve potrebne klase i metode kojima se dobije očekivano ponašanje programskog koda.

```
int main(){
	vector<Bodovi> OOPBodovi{
		Bodovi(Student("Ivana Ivic", "0246002475"), 25),
		Bodovi(Student("Ivica Ivanovic", "0246005654"), 20),
		Bodovi(Student("Marko Markic", "0246004234"), 32)
	};
	Kolegij OOP("Objektno orijentirano programiranje", OOPBodovi);

	/* Preko objekta OOP ispišite bodove svih studenata tog kolegija počevši od
	   onih s najmanjim brojem bodova prema većim. Npr.;

	   Objektno orijentirano programiranje bodovi:
	   Ivica Ivanovic  0246005654      20
	   Ivana Ivic      0246002475      25
	   Marko Markic    0246004234      32
	*/
	return 0;
}
```

11.06. [VJ-2] Kartaški špil

U kartaškom špilu se nalaze 52 karte a svaka od karti je označena brojem 1-52. Napišite program koji će za N igrača (N <= 13) podijeliti po 4 karte iz kartaškog špila. Prilikom kreiranja kartaškog špila sve karte mogu biti poredane po redu (1-52) ili izmiješane slučajnim redoslijedom (*random_shuffle*, zaglavlje *algorithm*).

Na početku programa potrebno je unijeti N igrača te kreirati novi (izmiješani) špil karata. Upotrebom metode *void Spil::Podijeli4Karte(Igrac *igrac)* svakom od igrača treba podijeliti po 4 karte sa vrha špila. Svaki igrač ima svoje ime te popis karti koje su mu podijeljene. Program treba ispisati karte koje su podijeljene svakom od igrača a zatim i karte koje su ostale u špilu.

Primjer izvođenja (izmiješani špil karata):

```
Unesi broj igraca: 2
Unesi ime 1.igraca: Ante
Unesi ime 2.igraca: Ivica

Karte u spilu (52):
10      7       5       6       4       41      26      8       31      42
2       33      47      37      14      16      51      35      24      38
18      25      21      36      11      49      52      29      45      13
19      44      43      32      12      3       30      17      34      9
40      22      1       20      27      39      15      46      50      23
48      28

Ante je dobio sljedece karte:   10      7       5       6
Ivica je dobio sljedece karte:  4       41      26      8


Preostale karte u spilu (44):
31      42      2       33    47    37    14    16    51      35      24
38      18      25      21    36    11    49    52    29      45      13
19      44      43      32    12    3     30    17    34      9       40
22      1       20      27    39    15    46    50    23      48      28
```

11.07. [VJ-3] Bankovni račun

Neka u klasi BankovniRacun postoje podatkovni članovi brojRacuna (string), tipKlijenta (string) i tipRacuna (string). Potrebno je demonstrirati svojstvo enkapsulacije na način da se nametnu sljedeća pravila:

- Broj računa mora imati točno 8 znamenki te mora početi znamenkom 0.
- Tip klijenta može biti isključivo fizička ili privatna osoba
- Tip računa može biti tekući, žiro-račun ili devizni

Ukoliko se prekrši bilo koje od pravila potrebno je odustati od postavljanja vrijednosti te korisniku ispisati odgovarajuću poruku.

11.08. [VJ-3] Najstarija osoba

Napišite klasu *Osoba* koja ima sljedeće podatkovne članove: *Ime* (string), *Prezime* (string), *GodinaRodjenja* (int). U navedenoj klasi potrebno je demonstrirati svojstvo enkapsulacije na način da ime i prezime osobe moraju početi velikim slovom, dok ostala slova su mala. Također, godina rođenja osobe ne smije biti manja od 1900.

Unutar projekta potrebno je deklaraciju klase *Osoba*, njenih metoda i podatkovnih članova smjestiti u Osoba.h datoteku, dok tijela metoda smjestiti u Osoba.cpp datoteku. U projekt dodajte novu cpp datoteku u kojoj se treba nalaziti funkcija *main*. Unutar nje kreirajte polje od 5 objekata tipa *Osoba* te inicijalizirajte vrijednosti svih objekata preko konzole (tipkovnice). Na kraju, program treba ispisati koja je najstarija od unesenih osoba.

11.09. [VJ-4] Student

Napišite klasu *Student* koja u privatnom dijelu sadrži pokazivač *JMBAG* koji pokazuje na znakovni niz duljine 11. U javnom dijelu klase smjestite metode

```
Student(){}
Student(char* jmbag); // konstruktor (alocira JMBAG duljine 11 znakova)
~Student();  // destruktor (dealocira JMBAG)
char* GetJMBAG() const;
void SetJMBAG(char* noviJMBAG);
```

Nakon što napišete tijela metoda klase *Student* napišite funkciju *main* sa sljedećim sadržajem.

```
Student Ante("1122334455");
Student Ivica = Ante;

Ante.SetJMBAG("6677889900");
cout << Ante.GetJMBAG() << endl;  // 6677889900
cout << Ivica.GetJMBAG() << endl; // 6677889900 ?!?!

Student Marko;
Marko = Ante;
Marko.SetJMBAG("1234567890");
cout << Marko.GetJMBAG() << endl; // 1234567890
cout << Ante.GetJMBAG() << endl;  // 1234567890 ?!?!
```

Nakon što pokrenete program potrebno je primijetiti problematične ispise koji su naznačeni u komentarima. Oni su se dogodili kao rezultat plitkog kopiranja (eng. *shallow copy*) prilikom pozivanja podrazumijevanog kopirnog konstruktora i operatora pridruživanja. Za oba ova slučaja potrebno je realizirati duboko kopiranje tako da konačni ispis bude:

```
6677889900
1122334455
1234567890
6677889900
```

Dodatak: Napišite prijenosni konstruktor i operator pridruživanja sa semantikom prijenosa za klasu *Student*.

11.10. [VJ-4] Učenik i razred

Neka u programu postoje sljedeće klase:

```
struct Ucenik {
	string ime, prezime;
};
class Razred {
public:
	vector<Ucenik*> ucenik;
	float prosjecnaOcjena;
};
```

Napišite sve potrebne metode unutar klase Razred kojima ćete demonstrirati semantiku kopiranja u funkciji main. Također, u funkciji main demonstrirajte primjerima kako bi izgledala semantika prijenosa (nije potrebno pisati metode unutar klase Razred).

11.11. [VJ-5] Objekt ID

Napišite klasu *Objekt* koja interno prati koliko njenih instanci postoji te koji je jedinstveni identifikator (ID) svake instance. Identifikator je cijeli broj koji počinje od 0 te se za svaku instancu klase *Objekt* uvećava za 1. Jednom dodijeljen, ID više ne može biti prenesen na drugi objekt (instancu). Primjerice,

```
#include <iostream>
#include "Objekt.h"
using namespace std;

int main(){
	Objekt obj1; // ID = 1, brojInstanci  = 1
	Objekt niz[10]; // ID = 2..11, brojInstanci  = 11
	Objekt* p = new Objekt[10];	// ID = 12..21, brojInstanci  = 21
	delete[] p; // brojInstanci  = 11!

	Objekt zadnji;
	cout << zadnji.ID << " " << Objekt::BrojInstanci() << endl; // 22 12
	return 0;
}
```

Napisana klasa *Objekt* treba proizvesti efekte koji su opisani u komentarima gornjeg primjera programskog koda.

11.12. [VJ-5] Slika

Sva slika ima svoju cijenu izraženu u eurima. Slijedeći programski kod u nastavku napišite klasu Slika i sve potrebne podatkovne članove i metode kojima se dobije očekivano ponašanje programskog koda.

```
Slika Slika1(2000); // 2000 eura
Slika Slika2(3000); // 3000 eura
Slika Slika3(1800); // 1800 eura
cout << Slika::ProsjecnaCijena() << " eura"; // 2266.67 eura
```

11.13. [VJ-5] Matematička operacija

Neka u C++ projektu postoji sljedeći programski odsječak:

```
int main(){
	short a, b;
	cin >> a >> b;
	try{
		cout << Matematika::Operacija(a, b, '+').rezultat() << endl;
		cout << Matematika::Operacija(a, b, '-').rezultat() << endl;
		cout << Matematika::Operacija(a, b, '*').rezultat() << endl;
		cout << Matematika::Operacija(a, b, '/').rezultat() << endl;
	}
	catch (short n){
		cout << "Broj " << n << " se ne moze dijeliti s 0!\n";
	}
	catch (const char* s){
		cout << s;
	}
	return 0;
}
```

Primjeri izvršavanja:

100 100 200 0 10000 1	32000 2000 Integer overflow!	6000 6 6006 5994 Integer overflow!	5000 0 5000 5000 0 Broj 5000 se ne moze dijeliti s 0!

Napišite imenik *Matematika* i klasu *Operacija* koji uzrokuju ovakvo ponašanje programskog koda.

11.14. [VJ-6] Kompleksni broj

Neka je zadana sljedeća funkcija *main*. Napišite klasu *Kompleksni* koja u sebi ima sve potrebne podatkovne članove i metode kako bi glavni program (funkcija *main*) mogla raditi kao što je to u kodu i predviđeno.

```
#include <iostream>
#include "kompleksni.h"
using namespace std;

ostream& operator << (ostream& izlaz, Kompleksni Z){
	izlaz << Z.re << ((Z.im >= 0) ? "+" : "") << Z.im << "i";
	return izlaz;
}
int main() {
	// Z1 = 1, Z2 = 2 - i
	Kompleksni Z1(1, 0), Z2(2, -1);

	// članska operatorska funkcija + (Kompleksni)
	Kompleksni Suma = Z1 + Z2;
	cout << Suma << endl; // 3 - 1i

	// članska operatorska funkcija * (Kompleksni)
	Kompleksni Umnozak = Z1 * Z2;
	cout << Umnozak << endl; // 2 - 1i

	// ne-članska operatorska funkcija - (Kompleksni, double)
	Kompleksni Razlika = Suma - 3;
	cout << Razlika << endl; // 0 - 1i

	// ne-članska operatorska funkcija - (double, Kompleksni)
	Kompleksni Razlika2 = 3 - Suma;
	cout << Razlika2 << endl; // 0 - 1i
	return 0;
}
```

Upute: Deklarirajte klasu *Kompleksni* koja kao privatne članove sadrži *re* i *im* tj. realni i imaginarni dio kompleksnog broja. U javnom dijelu klase napišite konstruktor s dva parametra kojim se inicijaliziraju ti podatkovni članovi, a zatim i deklaracije potrebnih operatora (operatorskih funkcija). Deklaraciju klase, njenih metoda i friend funkcija napišite u Kompleksni.h a tijela metoda i funkcija u Kompleksni.cpp.

Dodatni zadatak: Deklarirajte i napišite tijela postfiks i prefiks operatora ++ za klasu *Kompleksni* te u funkciji main na primjeru prikažite njihovu upotrebu. Oba operatora povećavaju imaginarni dio i realni dio kompleksnog broja za 1.

11.15. [VJ-6] Bubble

Potrebno je napisati klasu Bubble sa svojstvima boja i radijus. Implementacijom operatora zbrajanja (+) potrebno je omogućiti "spajanje" dva bubble-a u jedan. Novi bubble ima obujam jednak zbroju obujama bubble-a od kojih je nastao, a preuzima boju većeg bubble-a. Pri izdvajanju novog bubble-a iz postojećeg (operator -) nastaje novi sa smanjenim obujmom. Boja ostaje od originalnog bubble-a.

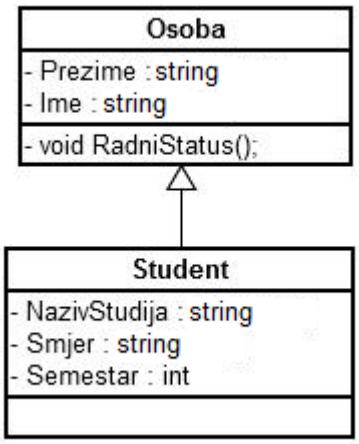
Obujam se računa prema obujmu kugle - V = 4/3 * r * r * r * PI. Također, treći korjen se računa kao pow(x, 1.0/3) (potrebno pri izračunavanju radijusa iz obujma).

Potrebno je dodati i operator == koji uspoređuje dva Bubble objekta. Dva bubble-a su isti ako imaju istu boju, i ako im se radijus razlikuje za ne više od 0.0001.

```
int main(){
	Bubble a("blue", 10.4);
	Bubble b("red", 7.2);
	Bubble c("green", 18.8);

	Bubble x = a + b; //nastat će novi bubble obujma 6275.27
	cout << x; //blue: 11.44
	Bubble y = x + c;
	cout << y; //green: 20.12

	Bubble z = y - x;
	cout << z; //green: 18.8

	if (z == c)
		cout << "OK" << endl; //OK
	return 0;
}
```

11.16. [VJ-7] Osoba i student

Napišite klase kao na sljedećem dijagramu (deklaracije u .h, implementacije u .cpp);



Klasa *Student* koristi javno (*public*) nasljeđivanje klase *Osoba*, dok podatkovni članovi u obje klase imaju javno pravo pristupa. Također, metoda *RadniStatus* ispisuje "Osoba nema radni status!". Obje klase je potrebno spremiti u odgovarajuće datoteke na način da se prototipi i deklaracije spreme u <ime_klase>.h datoteke a tijela metoda u <ime_klase>.cpp datoteke.

a) U klasi *Osoba* dodajte podatkovni član *OIB* (string), te konstruktor sljedećeg oblika: **Osoba(string oib);**

   Navedeni konstruktor postavlja člansku varijablu *OIB* na vrijednost predanu parametrom. Sukladno tome, u klasi *Student* također dodajte odgovarajući konstruktor kojim se inicijalizira pod-objekt *Osoba*.

b) U klasi *Student* napravite prijepis (eng. *overriding*) metode *RadniStatus* tako da ispisuje "Redovni student".

c) U funkciji main (main.cpp) deklarirajte statičko polje *GrupaA* tipa *Student* koje sadrži 5 elemenata. Pri deklaraciji polja svaki element treba inicijalizirati proizvoljnim OIB-om.

d) Napišite funkciju

   ```
   int BrojStudenata(Student* p, int elem, int semestar);
   ```

   koja za predano polje *p* vraća broj studenata koji se nalazi u traženom semestru.

11.17. [VJ-7] Računi

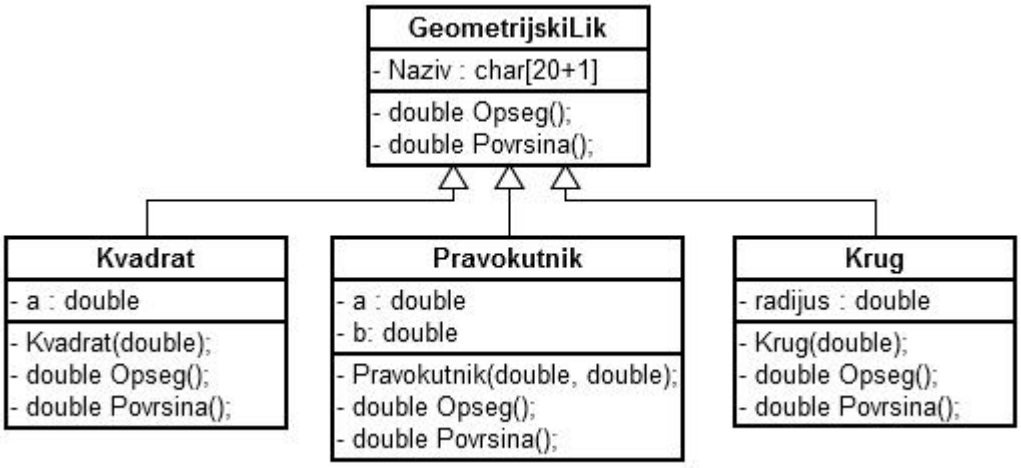
Sukladno priči u nastavku napišite sve potrebne klase sa svim podatkovnim članovima i metodama te adekvatnom hijerarhijom nasljeđivanja.

Nadograđujete sustav koji se bavi informatizacijom poduzeća te ste zaduženi za informatizaciju dokumenata. Trenutno je poznato da postoji nekoliko vrsta dokumenata. Za svaki dokument se zna da ima podatke o tome tko ga je napravio, datum i vrijeme nastanka dokumenta, te njegov jedinstveni broj (svaki dokument ima jedinstveni broj - string koji može sadržavati brojke, crtice "-" i slasheve "/").

Vi ste zaduženi da napravite dokument koji predstavlja račun. Račun, uz osnovne podatke koje ima svaki dokument, ima također JIR i tip računa (R1 ili obični). Račun ima niz stavki. Svaka stavka sadrži podatak o nazivu usluge koja je napravljena, količinu i jediničnu cijenu. Za račun se također mora znati kolika je ukupna suma njegovih stavki. Račun mora imati mogućnost obračunavanja popusta, na način da se postotak popusta primijeni na svaku stavku. Također, račun se mora moći ispisati (zajedno sa svim podacima i stavkama).

11.18. [VJ-8] Geometrijski lik

Napišite deklaracije i implementacije klase kao na sljedećem dijagramu;



Klase *Kvadrat, Pravokutnik* i *Krug* koriste javno (*public*) nasljeđivanje klase *GeometrijskiLik*, dok podatkovni članovi u svim klasama imaju javno pravo pristupa. Klasa *GeometrijskiLik* je apstraktna tj. njene metode *Opseg* i *Povrsina* su *čiste virtualne* metode. Zatim, za funkciju main kreirajte datoteku main.cpp te u nju napišite sljedeće;

```
#include <iostream>
#include "GeometrijskiLik.h"
#include "Kvadrat.h"
#include "Pravokutnik.h"
#include "Krug.h"
using namespace std;

int main(){
	GeometrijskiLik* Lik[3];
	Kvadrat Kvadrat1(1); // kvadrat sa stranicama duljine 1
	Pravokutnik Pravokutnik1(1, 2);  // pravokutnik sa stranicama duljine 1 i 2
	Krug Krug1(1);  // krug sa radijusom 1

	Lik[0] = &Kvadrat1;
	Lik[1] = &Pravokutnik1;
	Lik[2] = &Krug1;
	for (int i = 0; i < 3; i++)
		cout << Lik[i]->Naziv << " O=" << Lik[i]->Opseg()
		<< " P=" << Lik[i]->Povrsina() << endl;
	return 0;
}
```

Ispisi redom trebaju biti:

```
Kvadrat O=4 P=1
Pravokutnik O=6 P=2
Krug O=6.28 P=3.14
```

11.19. [VJ-8] Računalo i OS

U učionici se nalazi nekoliko prenosivih računala. Riječ je tablet i laptop računalima različitih proizvođača na kojima se nalaze različiti operacijski sustavi. Slijedeći programski kod u nastavku napišite sve potrebne klase, metode i funkcije koje su potrebne da bi funkcija main radila kao što je to i predviđeno.

```
int main() {
	int brTableta, brLaptopa;
	int brAndroida, brWindowsa, brLinuxa;

	Tablet Acer("Android"), Prestigio("Windows");
	Laptop Dell("Linux"), IBM("Windows"), Toshiba("Windows");

	vector<Racunalo*> Ucionica1 = { &Acer, &Prestigio, &Dell, &IBM, &Toshiba };
	AnalizaUcionice(Ucionica1, &brTableta, &brLaptopa,
			&brAndroida, &brWindowsa, &brLinuxa);


	/* u nastavku ispisite dobivenu analizu ucionice u sljedecem obliku:

	U ucionici se nalazi 5 racunala
	Broj tablet racunala : 2
	Broj laptop racunala : 3
	Android OS : 1
	Windows OS : 3
	Linux OS : 1
	*/
	return 0;
}
```

Napomena: Klasa Racunalo je apstraktna - demonstrirati polimorfizam unutar funkcije AnalizaUcionice prilikom provjere tipa računala (tablet ili laptop).

11.20. [VJ-8] Sortiranje

Dopišite potrebne dijelove koda da bi funkcija main radila kako je opisano.

```
class Sortiranje {
public:
	virtual void sortiraj(vector<int>* vec) final {
		for (int i = 0; i < vec->size() - 1; ++i)
			for (int j = i + 1; j < vec->size(); ++j)
				if (!usporedba((*vec)[i], (*vec)[j])) {
					swap((*vec)[i], (*vec)[j]);
				}
	}
	virtual bool usporedba(int e1, int e2) = 0;
};

int main() {
	vector<int> v = { 8, 17, 1, 14, 5, 2, 19, 3, 15, 11 };

	Sortiranje* sort1 = new SortVeciPremaManjem();
	sort1->sortiraj(&v);
	cout << v;		// ispisuje 19 17 15 14 11 8 5 3 2 1
	delete sort1;

	Sortiranje* sort2 = new SortManjiPremaVecem();
	sort2->sortiraj(&v);
	cout << v;		// ispisuje 1 2 3 5 8 11 14 15 17 19
	delete sort2;
	return 0;
}
```

11.21. [VJ-9] Generički kontejner

Napišite vlastiti generički kontejner (predložak klase) koji predstavlja polje podataka generičkog tipa. Broj elemenata tog polja se određuje konstruktorom klase, a klasa za svaku instancu (polje) mora omogućiti informaciju o broju elemenata, direktan pristup svakom elementu preko operatora [], te pretragu pojedine vrijednosti u polju.

Primjer korištenja

```
Polje<int> A(100);

// inicijalizacija elemenata slučajnim vrijednostima 1-100
srand((unsigned)time(NULL));
for(int i = 0; i < 100; i++)
       A[i] = rand() % 100 + 1; // A.polje[i] ...

// ispiši broj elemenata
cout << "Polje ima " << A.BrElem << " elemenata " << endl;

// da li se u polju nalazi broj x?
int x = 58;
int indeks = A.Sadrzi(x);
if(indeks != -1)
    cout << "Broj " << x << " se nalazi u elementu s indeksom " << indeks << "!\n";
else
    cout << "Broj " << x << " se ne nalazi u polju!\n";
```

11.22. [VJ-9] Najčešće ime

Napiši program koji unosi cijeli broj N a zatim isto toliko imena osoba. Program treba ispisati koje ime je najčešće i koliko puta se pojavilo.

Primjer izvođenja

Unesite N: 5
Unesite ime: Ante
Unesite ime: Ivica
Unesite ime: Davor
Unesite ime: Ivica
Unesite ime: Ivan
Najcesce ime: Ivica (2 puta)

Napomena: Potrebno je razlikovati mala i velika slova u imenu osobe.

11.23. [VJ-9] Matematika

Neka postoji sljedeći programski odsječak:

```
int main(){
	Matematika Racun;
	cout << Racun.suma(2, 4.3) << endl; // 6.3
	cout << Racun.suma(2.3, 4) << endl; // 6.3
	cout << Racun.suma(Kompleksni(2.5, 1), 4) << endl; // 6.5 1i
	return 0;
}
```

Napišite sve potrebne klase, metode i funkcije kako bi prikazani programski odsječak radio kao što je to u komentarima i predviđeno. Napomena: metoda **suma** vraća sumu bilo koja dva podatka koje je moguće zbrojiti operatorom +. Za tipove podatka koje nije moguće direktno zbrojiti operatorom + potrebno je preopteretiti taj operator za navedeni tip.

11.24. [VJ-10] Lambda izrazi 1

Slijedeći komentare u funkciji main nadopunite sve potrebne programske odsječke.

```
int main(){
	// 1. zadatak
	// umjesto ? napisati funkcijski objekt a zatim i lambda izraz kojim se vektor
	// x sortira od najmanje vrijednosti prema većoj
	// vector<int> x = { 3, -1, 0, 4, 1 };
	// sort(x.begin(), x.end(), ?);

	// 2. zadatak
	// neka postoje cjelobrojne varijable x, y i umnozak. pomoću barem 3 različite
	// lambda funkcije prikažite da je umnožak = x * y
	int x = 2, y = 3, umnozak;

	// 3. zadatak
	// napišite rekurzivnu lambda funkciju koja vraća n-ti član fibonaccijevog niza
	// upotrebom delegata proizvoljnog imena
	return 0;
}
```

11.25. [VJ-10] Lambda izrazi 2

Potrebno je nadopuniti donji programski odsječak na način da napišete funkciju "izdvoji", za koju vrijedi sljedeće:

- vraća vector<int> (novi vektor)
- prvi argument: vector<int>&
- drugi argument: funkcija

Funkcija izdvoji mora iz vektora "brojevi" izdvojiti samo one koji su djeljivi s 3. Drugi argument funkcije potrebno je poslati kao lambda izraz (nadopuniti).

```
int main(){
	vector<int> brojevi = { 1, 4, 5, 7, 3, 6, 12, 65, 32, 8, 87, 55, 23, 22,
	                        1, 1, 433, 66, 7, 433, 3, 32, 76, 8, 72, 256, 42 };
	vector<int> rez = izdvoji(brojevi, /*lambda izraz*/);
	for (int i = 0; i < rez.size(); i++)
		cout << rez[i] << " ";
	//ispis: 3 6 12 87 66 3 72 42
}
```

11.26. Primjeri prijašnjih zadataka sa prvog parcijalnog ispita

Potrebno je napraviti klase Klub i Osoba. Klub ima obilježja naziv, adresa i trener (pokazivač na objekt tipa Osoba). Osoba ima ime i prezime.

- Razdvojiti implementaciju klase Klub na .h i .cpp datoteku.
- Napisati get i set metode u klasi Osoba za atribute ime i prezime.
- Definirati potrebna const ograničenja.
- Napisati funkciju za dohvat trenera imajući na umu da se pri dohvatu ne smije stvoriti novi objekt tipa Osoba.
- Napisati konstruktor klase Osoba s prototipom Osoba(string ime, string prezime). Nije dopušteno mijenjati imena argumenata.
- Napisati destruktor klase Klub.
- Napisati kopirni konstruktor i operator pridruživanja za klasu Klub.
- Napisati friend funkciju za ispis podataka o osobi oblika void ispisOsobe(const Osoba& o).
- U main programu stvoriti dinamički objekt tipa Klub k1 i dinamički objekt tipa Osoba o1. Definirati da je o1 trener kluba k1.
- Stvoriti objekt k2 iz objekta k1 koristeći kopirni konstruktor.
- Promjeniti podatke o treneru kluba k2 i ispisati podatke o oba kluba.

11.27. Primjeri prijašnjih zadataka sa drugog parcijalnog ispita

Potrebno je napraviti objektni model za aplikaciju koja služi potpori poslovanju hotela. U aplikaciji se vodi evidencija o rezervacijama prostora u hotelu. Prostori mogu biti javni prostor (primjerice, kongresna dvorana, restoran, klub, bazeni) ili sobe. Za svaki prostor se pamti niz rezervacija (datum i vrijeme, te trajanje rezervacije). Za pojedini javni prostor se također vodi evidencija o kategoriji (bazen, restoran,...) te kapacitet. Za pojedinu sobu se vodi evidencija je li jednokrevetna, dvokrevetna ili trokrevetna.

- Ispravno modelirati nasljeđivanje i osmisliti cjelokupni objektni model zajedno sa svim potrbenim atributima. Očekuje se da model ima 5 klasa + main program. Dopušteno je dodavanje novih klasa u svrhu lakše implementacije.
- Odabrati jednu izvedenu klasu, i u njoj i baznoj klasi implementirati ispravne modifikatore pristupa za sve atribute (postaviti private, protected, public gdje treba).
- Definirati operator ispisa za neku klasu po izboru i demonstrirati upotrebu.
- Ostale klase trebaju imati ili operator ispisa ili funkciju "ispisi".
- Demonstrirati mehanizme polimorfizma ili preko te funkcije ili neke druge funkcije prema želji.
- U main programu statički napuniti objekte sa inicijalnim podacima i ispisati ih koristeći operator << ili funkciju "ispis".
- Baciti iznimku ako prostor nije moguće rezervirati u tom terminu i uhvatiti ju, te ispisati poruku korisniku.

11.28. Primjeri prijašnjih zadataka sa pismenih ispita

1. Napišite program koji u mapi pronalazi najveći datum kako je pokazano u donjem primjeru. Funkcija "najveci_dan" mora vraćati vrijednost tipa Datum.

```
void test2() {
	map<string, vector<Datum>> mapa;
	mapa["a"] = { Datum(4, 7, 2013), Datum(18, 6, 2015), Datum(20, 7, 2016),
		      Datum(20, 7, 2014), Datum(1, 7, 2015) };
	mapa["b"] = { Datum(11, 9, 2016), Datum(3, 10, 2016), Datum(4, 10, 2016),
		      Datum(30, 10, 2016) };
	mapa["c"] = { Datum(1, 2, 2012), Datum(7, 2, 2013), Datum(12, 1, 2014),
		      Datum(11, 1, 2015) };
	// mogu postojati i drugi kljucevi ...

	cout << "Najveci: " << najveci_dan(mapa); // ispisuje 30.10.2016.
}
```

2. Napišite klase Par i Kolekcija, gdje objekt klase Kolekcija sadrži skupinu objekata klase Par i omogućava pretraživanje parova kako je ispod pokazano.

```
void test3() {
	Par<string, string> dok1("a", "test1");
	Par<string, string> dok2("b", "test2");

	Kolekcija<Par<string, string>> kol_str;
	kol_str.dodaj(dok1);
	kol_str.dodaj(dok2);

	// ispisuje b:test2
	try {
		Par<string, string>* x = kol_str.nadji(Par<string, string>("b", "test2"));
		cout << x->kljuc() << ":" << x->vrijednost() << endl;
	}
	catch (KolekcijaException& e) {
		cout << e.what();
	}

	// ispisuje "trazeni element ne postoji"
	try {
		Par<string, string>* y = kol_str.nadji(Par<string, string>("X", "TEST"));
		cout << y->kljuc() << ":" << y->vrijednost() << endl;
	}
	catch (KolekcijaException& e) {
		cout << e.what();
	}
}
```

3. Definirajte klasu Recenica koja podržava sljedeće operacije, kako je pokazano u kodu ispod:

* Usporedba dvije rečenice operatorom ==;

* Nečlanska funkcija "iste_rijeci" koja daje True samo ako dvije rečenice

sadrže iste riječi, bez obzira na njihov poredak ili razmake medju riječima;

* Konverzija iz tipa Recenica u string;

* Nečlanska funkcija "razlika" koja daje broj riječi u kojima se dvije rečenice razlikuju.

```
void test8() {
	Recenica r1("ne znam koliko je sati");
	Recenica r2("sutra je praznik");
	Recenica r3("koliko je sati ne znam");
	Recenica r4("ne      znam          koliko   je      sati");
	Recenica r5("sutra je slavlje");

	// ispisuje "sutra je praznik"
	string recenica = r2;
	cout << recenica << endl;

	// ispisuje "recenice imaju iste rijeci"
	if (iste_rijeci(r4, r3)) {
		cout << "recenice imaju iste rijeci\n";
	}
	else {
		cout << "recenice nemaju iste rijeci\n";
	}

	// ispisuje "nisu iste recenice"
	if (r1 == r2) {
		cout << "iste recenice\n";
	}
	else {
		cout << "nisu iste recenice\n";
	}

	// ispisuje 0
	double razlika1 = razlika(r1, r4);
	cout << razlika1 << endl;

	// ispisuje 6
	double razlika2 = razlika(r1, r2);
	cout << razlika2 << endl;

	// ispisuje 2
	double razlika3 = razlika(r2, r5);
	cout << razlika3 << endl;
}
```

12. Literatura

1. M. Slamić, Slajdovi s predavanja, na WEB stranici moj.tvz.hr
2. M. Slamić, Lekcije u Wordu, WEB stranica
3. I.J. Štribar, B. Motik, “Demistificirani C++”, Element, 1977.
4. D. Radošević, Programiranje 2, TIVA Tiskara Varaždin, 2007.
5. Eckel–„*Thinkingin C++ Vol 1 i Vol 2*“, PrenticeHall, dostupno na WEB-u.
6. Stroustrup–„*TheC++ ProgrammingLanguage*“, Addison-Wesley