1. Uvod u programski jezik C++

Pre nego što započnemo ovo poglavlje, potrebno je da razumemo da veliki broj programa validno napisanih u programskom jeziku C ujedno predstavljaju i validno napisane C++ programe. To znači da se razni elementi programskog jezika C mogu koristiti i u jeziku C++, kao što su osnovni tipovi podataka poput `int`, `float`, `double`, `char`, nizovi podataka, funkcije i mnogi drugi. Semantika ovakvih konstrukata ima gotovo identičnu namenu u oba jezika, te ih je dozvoljeno koristiti i u C++ programima. U ovom tekstu nećemo detaljno diskutovati o takvim razlikama, već ćemo uvoditi samo one konstrukte koji su nam neophodni za rešavanje primera, a koji nisu deo C standarda, već uvode novu semantiku u C++ programima.

Programski jezik C++ je nastao 1980-ih godina i kao takav predstavlja jedan od najstarijih jezika koji su i dan danas u širokoj upotrebi. Iako se osnovni elementi tog jezika i dalje koriste (upravo zato što predstavljaju osnovu jezika), godine 2011. internacionalna organizacija za standardizaciju (ISO) objavila je tadašnju verziju standarda za programski jezik C++ koja je značajno promenila način rada i ponovo vratila veliku popularnost ovom jeziku. Taj standard koji se često naziva *C++11* — zajedno za verzijama nakon njega: *C++14*, *C++17* i najnoviji predlog standarda *C++20* (u toku pisanja ovog teksta) — čine ono što se među današnjim razvijačima softvera smatra za *savremeni* C++ jezik. U ovoj skripti, veliki akcenat će biti postavljen na razne elemente savremenog C++ jezika, ali, naravno, diskutovaćemo i o osnovnim konceptima koji su tu od samog nastanka jezika. Pogledajmo zahtev našeg prvog primera.

Primer 1.1 Napisati program u programskom jeziku C++ koji sa standardnog ulaznog toka učitava niz kompleksnih brojeva do pojave karaktera EOF i izračunava njihovu aritmetičku sredinu.

Ovako jednostavan primer će nam do kraja njegovog rešavanja odmah ilustrovati "moć" savremenog programskog jezika C++ u odnosu na (slabo) ekvivalentan kod koji bismo napisali u programskom jeziku C.

1.1 Inicijalizacija vrednosti promenljivih

Deklarisanje promenljivih određenog tipa čija se lokacija nalazi na steku funkcije se izvršava isto kao i u programskom jeziku C — navođenjem tipa promenljive i praćenjem identifikatora koji se vezuje za taj memorijski prostor:

```
int a;
```

Ukoliko nismo inicijalizovali sadržaj promenljive `a`, njena vrednost će biti ono što je se u trenutku deklarisanja nalazilo u memoriji na toj lokaciji, te je često bolje inicijalizovati promenljivu pre njenog korišćenja:

```
int a;
int b = a + 1;
```

Na primer, prilikom kompiliranja datog fragmenta koda korišćenjem Microsoft C/C++ kompilatora verzije 19.16.27027.1 dobija se naredno upozorenje za drugu liniju:

```
warning C4700: uninitialized local variable 'a' used
```

U programskom jeziku C++ inicijalizacija ove promenljive se može izvršiti na više načina. Jedan način je korišćenje *inicijalizatora* (engl. *braced initializer*), koji služi da definiše inicijalnu vrednost promenljive koja se deklariše prilikom njene konstrukcije u memoriji. Na primer, inicijalizacija promenljive tipa `int` može se izvršiti na sledeći način:

```
int a{1};
```

Broj parametara za inicijalizaciju promenljive zavisi isključivo od njenog tipa, te inicijalizator može sadržati i više od jedne vrednosti. Očigledno, inicijalizator za tip `int` može imati jednu vrednost, koja predstavlja upravo vrednost promenljive koja se inicijalizuje, ali imati i nula vrednosti, čime se promenljiva inicijalizuje na vrednost 0:

```
int a;   // nije definisana vrednost
int a{}; // 0
```

Inicijalizator koji prihvata nula argumenata se naziva *nulski inicijalizator* (engl. *zero initializer*) i radi za sve osnovne tipove podataka.

Postoje i drugi načini za inicijalizaciju vrednosti prilikom deklarisanja. Na primer, *funkcionalna notacija* ili *konstrukcija* izgleda ovako:

```
int a(1);
```

Sa druge strane, moguće je inicijalizovati vrednost *dodeljivanjem* vrednosti, što je bio jedini način za inicijalizaciju vrednosti u programskom jeziku C:

```
int a = 1;
```

Inicijalizator je uveden standardom C++11 sa ciljem da se uniformiše inicijalizacija raznih tipova podataka. Postoji prednost u korišćenju inicijalizatora, na primer, prilikom inicijalizovanja u kojem dolazi do implicitne konverzije, te ćemo zbog toga njega i najčešće koristiti:

```
int banana_count(7.5);   // Moze proci kompiliranje bez upozorenja
int coconut_count = 5.3; // Moze proci kompiliranje bez upozorenja
int papaya_count{0.3};   // Dobija se makar upozorenje, cesto i greska
```

Na primer, prilikom kompiliranja datog fragmenta koda korišćenjem Microsoft C/C++ kompilatora verzije 19.16.27027.1 dobija se naredna greška samo za treću liniju koda, dok prva i druga linija prolaze bez upozorenja:

```
error C2397: conversion from 'double' to 'int' requires a narrowing conversion
```

1.2 `vector` kao deo standardne biblioteke

Programski jezik C++ je sam po sebi, poprilično limitiran, slično kao i jezik C. Ukoliko bismo koristili isključivo elemente koji se nalaze u njegovoj srži, ne bismo mogli da kreiramo preterano korisne programe. Zbog toga, da bi se povećala njegova upotrebljivost, na raspolaganju nam je *standardna biblioteka* (engl. *standard library*, skr. *STL*) koja sadrži pregršt *klasa*[1] koje nam omogućavaju razne pogodnosti. Pored osnovnih koncepata koje čine jezik C++, poznavanje standardne biblioteke je od izuzetne važnosti za svakog C++ programera. Jedan od značajnih elemenata standardne biblioteke čine *kolekcije* (engl. *containers*).

Iako je u programskom jeziku C++ moguće koristiti nizove kao što smo to radili u jeziku C — bilo statički alocirane na steku funkcije, bilo dinamički alocirane na hipu — u 99% slučajeva, kolekcija `vector` nam završava posao kada je potrebno čuvati veći broj podataka istog tipa, sekvencijalno zadato u memoriji poređanih prema svom indeksu — drugim rečima, ono što obično smatramo za *niz podataka*. Ova kolekcija je definisana u zaglavlju `<vector>`.

`vector` se obično koristi tako što se inicijalizuje prazan vektor, a zatim se dinamički popunjava elementima po potrebi. Ovo je moguće zbog toga što `vector` automatski raste po potrebi, čime se osiguravamo da uvek imamo prostor za elemente koji se skladište (naravno, sve dok nam memorija računara to dozvoljava) bez da znamo unapred njihov broj.

`vector` čuva elemente istog tipa, koji se navodi između karaktera `<` i `>` nakon navođenja tipa `vector`. Na primer, kreiranje praznog vektora koji čuva vrednosti zapisane u pokretnom zarezu sa dvostrukom preciznošću se vrši na sledeći način[2]:

```
std::vector<double> vektor;
```

Dodavanje novog elementa na kraj `vector` objekta se vrši pozivanjem metoda `push_back`[3] koji kao argument ima element koji se dodaje `vector` objektu:

```
vektor.push_back(3.14);
```

Ukoliko želimo da saznamo koliko `vector` objekat ima trenutno elemenata, na raspolaganju nam je metod `size`[4]:

```
std::vector<double>::size_type velicina = vektor.size(); // 1
```

[1] Ali i drugih konstrukata jezika, poput funkcija.

[2] Za sada ignorišemo fragment koda `std::` koji se navodi ispred `vector`; na to ćemo se osvrnuti u sekciji 1.6, a videćemo ga još nekoliko puta do tada.

[3] Vremenska složenost dodavanja novog elementa na kraj `vector` objekta je *amortizovano konstantno*, za šta koristimo notaciju $O(1)+$. Amortizovana analiza vremena podrazumeva određivanje klase složenosti za algoritam na nivou izračunavanja celog algoritma, umesto na nivou izračunavanja koraka. Drugim rečima, amortizovana cena n operacija predstavlja količnik ukupne cene za izračunavanje tih n operacija podeljena brojem n.

[4] Vremenska složenost izračunavanja broja elemenata u `vector` objektu je konstantna, tj. $O(1)$.

Kao što vidimo, tip povratne vrednosti metoda `size` jeste statički `typedef` za tip `size_type` definisan u klasi `vector<double>`, kojim se pristupa korišćenjem notacije `::` između `std::vector<double>` i `size_type`, kao što je i ilustrovano. Iako možemo da koristimo i `size_t` kao tip promenljive u koju će biti smeštena povratna vrednost metoda `size`, uvek je dobra praksa pratiti C++ standard, čak i po cenu nešto većeg broja otkucanih karaktera. Korišćenjem modernog C++ jezika, moći ćemo da smanjimo broj karaktera, ali i da sačuvamo ekvivalentnu semantiku, ali o tome ćemo pričati detaljnije kasnije[5].

Pristup elementima vektora na osnovu njihovog indeksa se vrši kao i za obične nizove — korišćenjem operatora `[]`[6]:

```
double prviElement{vektor[0]};
```

Iz ovog koda takođe primećujemo da indeksiranje vektora počinje od indeksa 0. Dodatno, operator `[]` ne proverava granice vektora. Za indeksiranje elementa sa proverom granica vektora, može se koristiti metod `at`, čiji je argument indeks elementa koji se dohvata. Ukoliko je indeks van granica vektora, biće ispaljen izuzetak tipa `out_of_range`. O izuzecima ćemo govoriti naknadno.

Metodi `first` i `last` služe za dohvatanje prvog, odnosno, poslednjeg elementa, redom. Metod `empty` proverava da li je kolekcija prazna ili ne (povratna vrednost je tipa `bool`)[7].

Za sada nam je ovo dovoljno. Kroz neke od ostalih primera ćemo ilustrovati još neke mogućnosti rada sa `vector` objektom.

1.3 Kompleksni brojevi

Ukoliko bi neko od Vas zatražio da implementirate rad sa kompleksnim brojevima u programskom jeziku C, sigurno bi Vam prvo palo na pamet da implementirate strukturu koja sadrži dva polja tipa `double` (realni i imaginarni deo kompleksnog broja) i veliki broj funkcija za njihovo sabiranje, oduzimanje, itd. Međutim, ukoliko bi neko odlučio da mu nije potrebno da čuva `double` kao tip realnog i imaginalnog dela, već želi da koristi `float`, onda bismo morali da promenimo ne samo definiciju strukture već i potencijalno neke od funkcija za rad sa tom strukturom (makar kreiranje novog kompleksnog broja i njegov ispis).

U standardnoj biblioteci jezika C++ na raspolaganju nam je klasa `complex` koja već ima sve implementirano za nas, definisana u zaglavlju `<complex>`. Jedino što je potrebno da uradimo jeste da navedemo koji se tip podataka koristi za instancu kompleksnog broja pri njegovom kreiranju, ponovo navođenjem tipa `float`, `double` ili `long double` između zagrada `<` i `>`. Prednost ovoga jeste što se izmena tipa vrši samo na jednom mestu, upravo prilikom deklaracije promenljive.

Generalno, ovakve klase se nazivaju *šablonske klase* (engl. *template class*) i o njima će biti reči nešto kasnije. Tip koji se navodi između zagrada `<` i `>` je *tipski parametar* (engl. *type parameter*) te šablonske klase. Videćemo da ne postoje samo šablonske klase, već i šablonske funkcije, ali sve u svoje vreme.

Kreiranje kompleksnih brojeva, na primer, $z_1 = 2 - 3.96i$, $z_2 = 5 + 0i$ i $z_3 = 0 + 0i$, mogli bismo izvršiti na sledeći način:

[5] O čemu je reč? Saznaćete u sekciji 1.10.

[6] Vremenska složenost indeksiranja elementa iz `vector` objekta je konstantna, tj. $O(1)$.

[7] Vremenska složenost svih nabrojanih metoda: $O(1)$.

```
std::complex<double> z1{2, −3.96};
std::complex<double> z2{5};
std::complex<double> z3{};          // ili samo std::complex<double> z3;
```

Ono što primećujemo jeste da se nakon identifikatora kompleksnog broja specifikuje nula, jedna ili više vrednosti. Kako je moguće da smo koristili različit broj parametara prilikom inicijalizacije kompleksnih brojeva `z1`, `z2` i `z3`? Odgovor na to ćemo videti uskoro, kada budemo govorili o klasama i specijalnoj metodi koja se naziva *konstruktor*, kao i o mogućnosti jezika C++ koja se naziva *podrazumevani parametri*.

Za sada, samo shvatimo da inicijalizator za `complex` objekte prima najviše dve vrednosti koje predstavljaju realni i imaginarni deo, redom, ali da može da primi i jednu, koja predstavlja realni deo (imaginarni deo je u tom slučaju 0) ili nijednu (čime i realni i imaginarni deo imaju vrednost 0).

Klasa definiše razne aritmetičke metode, kao što su sabiranje, oduzimanje, množenje i deljenje koje su predstavljene operatorima `+`, `–`, `*` i `/`, redom, kao i metode `real` i `imag` koje vraćaju realni, odnosno, imaginarni deo kompleksnog broja, redom.

1.4 Ulazni i izlazni tokovi

U programskom jeziku C++, svaki ulazni ili izlazni resurs koji program može da koristi, apstrahuje se kroz koncept poznat pod nazivom *tok* (engl. *stream*). Ulazni tokovi predstavljaju izvore dohvatanja podataka, dok izlazni tokovi služe za čuvanje podataka. Zbog ovakve apstrakcije, čitanje podataka sa tastature i sa diska se fundamentalno ne razlikuje, što predstavlja jednu od dobrih strana ovog jezika.

U jeziku C++, standardni ulaz je reprezentovan kao `cin`, dok je standardni izlaz reprezentovan kao `cout`. Da bismo mogli da ih koristimo, potrebno je da uključimo zaglavlje `<iostream>`.

Ispisivanje na izlazni tok se vrši operatorom `<<`, čiji je levi operand neki izlazni tok (na primer, `cout`), dok je desni operand vrednost koja se ispisuje (na primer, `int`):

```
int a{7}, b{5};

std::cout << a; // bice ispisan broj 7, ...
std::cout << b; // ... a odmah za njim broj 5
```

U standardnoj biblioteci postoje implementacije operatora `<<` za sve osnovne tipove podataka, kao i za niske, tako da ne moramo eksplicitno da naznačimo koji je tip operanda koji se ispisuje, kao što smo to morali da uradimo pri korišćenju C funkcije `printf`. Ukoliko za neki tip ne postoji implementacije, potrebno ju je napisati, na šta ćemo se kasnije vratiti.

Ono što je zanimljivo napomenuti jeste da operator `<<` kao povratnu vrednost ima isti izlazni tok nad kojim se primenjuje. Drugim rečima, moguće je *ulančavati* pozive ovog operatora jedan za drugim:

```
int a{7}, b{5};

// Bice ispisan broj 7, a odmah za njim broj 5.
// Dakle, isto kao u prethodnom kodu.
std::cout << a << b;
```

Ukoliko želimo da ispišemo "novi red" u izlazni tok, preporučuje se korišćenje `endl`, kao u narednom kodu:

```
int a{7}, b{5};

// Bice ispisan broj 7, zatim novi red, i na kraju broj 5.
std::cout << a << std::endl << b;
```

Učitavanje sa ulaznog toka se vrši operatorom `>>`, čiji je levi operand neki ulazni tok (na primer, `cin`), dok je desni operand identifikator promenljive u koju se smešta pročitana vrednost (na primer, `int`):

```
int a;

// Ucitava ceo broj sa standardnog ulaza u promenljivu "a"
std::cin >> a;
```

Napomene koje su važile za operator `<<` važe i za operator `>>`, tako da možemo učitavati sve osnovne tipove i ulančavati čitanje:

```
int a;
double b;

// Sa standardnog ulaza
// prvo se ucitava ceo broj i smesta u "a",
// a zatim se ucitava broj u pokretnom zarezu i smesta u "b".
std::cin >> a >> b;
```

1.5 Sabiranje elemenata `vector` objekta

S obzirom da je potrebno da izračunamo aritmetičku sredinu naših kompleksnih brojeva, očigledno, potrebno je da prođemo kroz sve njegove elemente i da ih sumiramo, da bismo na kraju tu sumu podelili ukupnim brojem elemenata. Za početak, pogledajmo kako bismo rešili ovaj problem ukoliko bi nam elementi niza bili tipa `double`.

U klasičnom C pristupu rešavanja ovog problema, samo korišćenjem C++ jezika, deo koda koji bi ovo rešio bi mogao da izgleda ovako:

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
double suma{};

for (std::vector<double>::size_type i{}; i < niz.size(); ++i)
{
    suma += niz[i];
}

std::cout << suma / niz.size() << std::endl; // 3.5
```

Petlja koja prolazi kroz sve elemente niza je toliko česta u programskim jezicima, da je u savremenom C++ jeziku uvedena *kolekcijska petlja* (engl. *for-each loop*) koja ubrzava pisanje takvog koda i koja ne zahteva korišćenje brojača iteracije (promenljiva `i` u prethodnom kodu). Njena forma je:

```
for (<tip> <id elementa> : <id kolekcije>)
{
    // Telo petlje
}
```

U ovoj sintaksi, `<tip>` je tip elementa kolekcije čiji je identifikator `<id kolekcije>`. U telu petlje, dostupna je promenljiva `<id elementa>` koja predstavlja i-ti element te kolekcije, u iteraciji i, za svako $i \in \{0, 1, ..., n-1\}$, gde je n broj elemenata u kolekciji. Prethodni fragment koda se može napisati kolekcijskom petljom na sledeći način:

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
double suma{};

for (double element : niz)
{
    suma += element;
}

std::cout << suma / niz.size() << std::endl; // 3.5
```

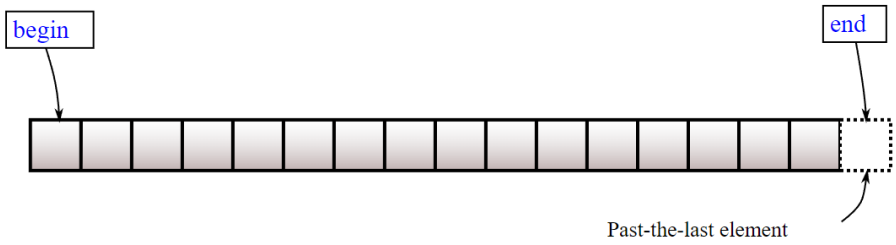
Mada je ovaj kod sasvim korektan i radi ispravno, i sumiranje elemenata niza predstavlja algoritam koji ima čestu upotrebu. Opštije, sumarne statistike nad nizom elemenata su veoma rasprostranjene u programima realne primene, i sve one predstavljaju *akumulaciju* elemenata na neki način, bilo njihovim sabiranjem, množenjem, ili nekom operacijom koja je definisana nad korisnički definisanim tipom.

U tu svrhu, standardna biblioteka jezika C++ nudi veliki broj algoritama, koji su definisani bilo u zaglavlju `<algorithm>` (kada su u pitanju neki opštiji algoritmi) ili u nekim specifičnim zaglavljima kao što je, na primer, zaglavlje `<numeric>` (za rad sa matematičkim objektima).

Tako, na primer, akumuliranje elemenata niza sa nekom početnom vrednošću, može se izvršiti šablonskom funkcijom `accumulate`, definisanom u zaglavlju `<numeric>`, koja podrazumevano akumulira vrednosti iz kolekcije pozivanjem operatora `+` nad tim elementima, pri čemu je potrebno specifikovati početnu vrednost nad kojom se poziva operator `+`. Njena povratna vrednost je suma svih elemenata kolekcije. Pogledajmo za početak njenu deklaraciju:

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );
```

Prva linija nam govori da je u pitanju šablonska funkcija koja ima dva tipa. O ovome ćemo detaljnije diskutovati kasnije, tako da za sada to zanemarujemo. U drugoj liniji vidimo da funkcija prima 3 argumenta i to su: (1) iterator na početak kolekcije, (2) iterator na kraj kolekcije i (3) inicijalna vrednost za akumulaciju.



Slika 1.1: Ilustracija iteratora na početak i kraj kolekcije. Primetimo da se za "kraj" kolekcije smatra iterator koji "pokazuje" na "element nakon poslednjeg".

O iteratorima će takođe biti detaljnije rečeno kasnije, ali za sada možemo da ih razumemo kao apstrakciju pokazivača (videti sliku 1.1). Drugim rečima, funkciji treba proslediti

”pokazivače” na početak i kraj kolekcije koju želimo da akumuliramo, kao i inicijalnu vrednost. Ove iteratore je moguće dobiti na dva načina:

1. Pozivanjem metoda `begin` i `end` (i njima sličnih) nad kolekcijom, koji vraćaju iteratore na početak, odnosno, kraj kolekcije, redom.
2. Pozivanjem šablonskih funkcija `begin` i `end` (i njima sličnih) čiji je argument kolekcija, i koji vraćaju iteratore na početak, odnosno, kraj kolekcije, redom. Ove funkcije su definisane u zaglavlju `<iterator>`.

Oba pristupa su validna i koriste se u praksi. Neki razvijači preferiraju jedan stil, neki drugi, ali bitno je odabrati jedan i držati se njega. Mi ćemo trenutno pokazati oba pristupa, ali držaćemo se drugog stila.

Sada kada smo opremljeni ovim znanjem, možemo da izračunamo airtmetičku elemenata vektora na sledeći način:

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
double suma{ std::accumulate(niz.begin(), niz.end(), double{}) };

std::cout << suma / niz.size() << std::endl;
```

odnosno

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
double suma{ std::accumulate(std::begin(niz), std::end(niz), double{}) };

std::cout << suma / niz.size() << std::endl;
```

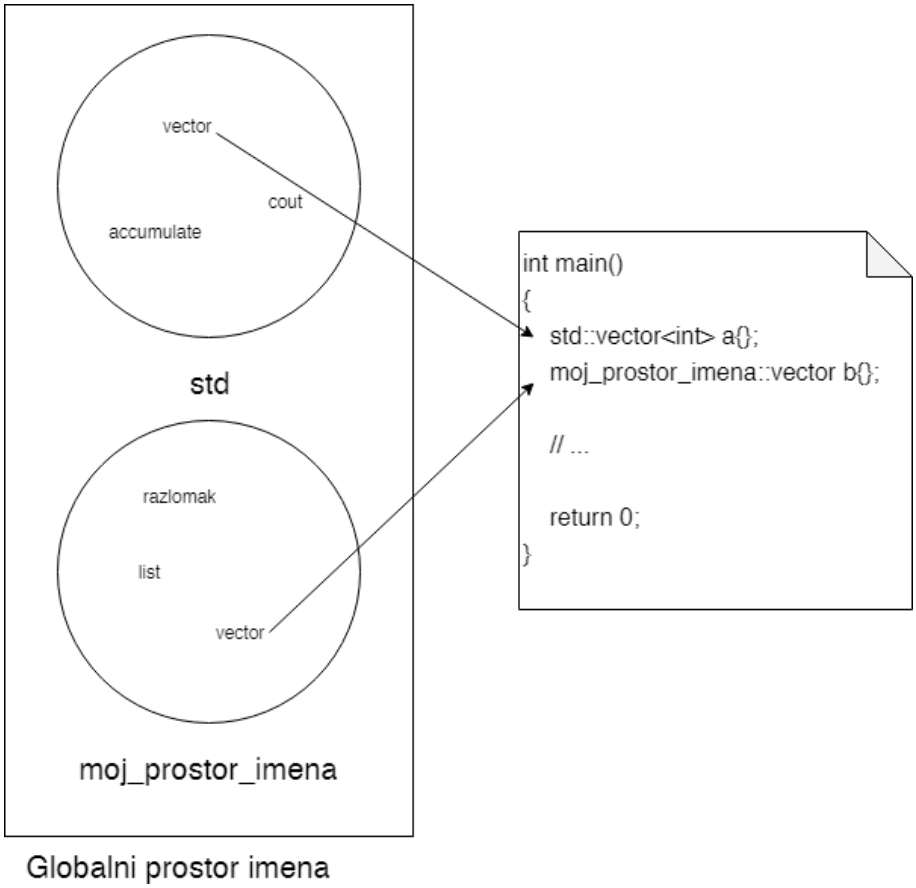
Zapravo, sada nam promenljiva `suma` uopšte nije ni potrebna:

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
std::cout
    << std::accumulate(niz.begin(), niz.end(), double{}) / niz.size()
    << std::endl;
```

1.6 Prostor imena i `std`

Iole veći projekti podrazumevaju rad većeg broja razvijača softvera koji u isto vreme rade na istim ili različitim delovima koda. Posledica ovoga jeste u tome da se može desiti da jedan programer koristi identifikator, bilo za neku promenljivu, funkciju ili klasu, koje na nekom drugom mestu koristi drugi programer. U trenutku kada se moduli na kojima oni rade spajaju, dolazi do problema jer kompilator ima dve definicije za dva konstrukta koji se nazivaju identično. Slično tome, standardna biblioteka uvodi ogroman broj imena koje jednostavno ne možemo sve zapamtiti, te može doći do kolizicije između tih imena i imena u bibliotekama koje mi kreiramo (ili koje su kreirane od strane *trećih lica* (engl. *third-party*).

Rešenje ovog problema se sastoji u korišćenju mehanizma *prostora imena* (engl. *namespace*). Na prostor imena možemo posmatrati kao prezime za sve konstrukte koje su definisane u okviru njega. Imena koja su uvedena u okviru nekog prostora imena su dostupna prvo referisanjem na taj prostor imena, a zatim, korišćenjem operatora `::` čiji je naziv *operator rezolucije dosega*, referisanjem imena unutar tog prostora. Slika 1.2 ilustruje ovaj koncept.



Slika 1.2: U globalnom prostoru imena (levo) nalaze se prostor imena `std` i `moj_prostor_imena` koji definišu različite klase `vector`. Kod koji koristi ove klase (desno) zna da odredi koja promenljiva predstavlja instancu koje klase jer referiše na njih preko prostora imena u kojem su one definisane.

U okviru standardne biblioteke, definisan je prostor imena `std` u kojem su definisane razne vrednosti, funkcije, klase i dr. Zbog toga smo uvek morali da koristimo fragment koda `std::` ispred naziva klasa kao što je `vector`, funkcija kao što je `accumulate` i dr.

Iako će većina naših programa biti dovoljno mala, te nam naš prostor imena neće biti neophodan, korisno je znati da se prostor imena uvodi ključnom reči `namespace`:

```
namespace moj_prostor_imena
{
    // Ovde idu deklaracije/definicije konstrukata
    // koje se nalaze u ovom prostoru imena.
    // Na primer, ako imamo definisanu klasu vector ovde,
    // onda bismo na nju referisali
    // pomocu operatora rezolucije dosega na sledeci nacin:
    // moj_prostor_imena::vector
}
```

Ipak, nekada je zamorno stalno kucati kvalifikator prostora imena ispred naziva konstrukta, te je moguće navesti narednu liniju u kodu, čime se u tekućem dosegu uvode sva imena iz `std` prostora imena:

```
using namespace std;
```

U dosegu u kojem je ova linija navedena, mogu se referisati konstrukti na sledeći način:

```
vector<int> a{1, 2, 3};
int suma{ accumulate(begin(a), end(a), int{}) };
cout << suma << endl;
```

Na slici 1.2 pomenuli smo nešto što se naziva *globalni prostor imena* (engl. *global namespace*). On predstavlja podrazumevani prostor imena koji se koristi za sve konstrukte koji se ne definišu u okviru nekog prostora imena. Na takva imena se referiše jednostavno navođenjem njihovih identifikatora:

```
#include <iostream>

int a{5}; // u globalnom prostoru imena

int main() {
    std::cout << a << std::endl; // ispisuje 5

    return 0;
}
```

Može se koristiti operator rezolucije dosega za referisanje imena u globalnom prostoru, tako što se ne navede levi operand, na primer: `::a`. Jedna korisna upotrebna vrednost jeste ukoliko postoji isto ime za konstrukt u lokalnom dosegu i u globalnom prostoru imena:

```
#include <iostream>

int a{5}; // u globalnom prostoru imena

int main() {
    int a{7}; // lokalna promenljiva

    std::cout <<   a << std::endl; // ispisuje 7
    std::cout << ::a << std::endl; // ispisuje 5

    return 0;
}
```

Vežbanje Kod za ovaj primer se nalazi na lokaciji `Primeri/1/1/main.cpp`.

Test primer 1

```
Ulaz:
(1, 5)
(2, 4)
(7, 1)
Izlaz:
(3.33333,3.33333)
```

Test primer 2

```
Ulaz:
5
-2
6
Izlaz:
(3,0)
```

Test primer 3

```
Ulaz:
Izlaz:
(0,0)
```

Primer 1.2 Napisati program u programskom jeziku C++ koji sa standardnog ulaznog toka učitava niz linija teksta do pojave karaktera EOF, a zatim ispisuje ceo niz (bez praznih linija) "uokviren" karakterima $*$.

1.7 Konstantne vrednosti

Ukoliko želimo da deklarišemo neku promenljivu za konstantnu, odnosno, da njena vrednost ne može biti promenjena, potrebno je koristiti ključnu reč `const`:

```
const int a{1};

// Kod ispod rezultuje narednom greskom prilikom kompiliranja:
// error C3892: 'a': you cannot assign to a variable that is const
a = 2;
```

U C++ jeziku je ”konstantnost” veoma važna i generalno pravilo je da uvek treba razmisliti da li je neka vrednost konstantna ili ne i definisati je kao konstantnu ukoliko je to moguće. Ipak, u fragmentima kodova koje prikazujemo često nećemo primenjivati ovo pravilo, da bismo insistirali na konstantnosti tamo gde je to zaista neophodno. Rešenja će ipak imati konstantne vrednosti svugda gde je to poželjno. Na ovu temu ćemo se vratiti, pre svega, kada budemo govorili o definisanju korisničkih tipova podataka, tj. klasa.

1.8 Osnovni rad sa niskama

Kao i u jeziku C, moguće je koristiti nizove karaktera, odnosno, pokazivače na nizove karaktera za predstavljanje tekstualnih podataka. U zaglavlju `<cstring>` su nam dostupne razne funkcije koje mogu da se koriste za njihovu obradu. Međutim, daleko upotrebljivija je klasa `string`, definisana u zaglavlju `<string>`, na koju ćemo se osvrnuti u rešavanju ovog primera.

Deklarisanje niske karaktera se vrši kao i za svaki drugi tip podataka:

```
std::string prazna; // Prazna niska
```

Naravno, niske možemo inicijalizovati prosleđivanjem literala niske kao argument inicijalizatora:

```
std::string recenica{ "Jun je najbolji mesec od svih." };
```

Objekat `recenica` sadrži kopiju literala niske koja se prosleđuje kao argument inicijalizatora. Dodatno, niz karaktera koji se nalazi kao deo klase `string` uvek se završava terminirajućom nulom, iz razloga kompatibilnosti između te klase i klasičnih C nizova karaktera. Zapravo, u klasi `string` postoji metod `c_str` koji vraća pokazivač na niz karaktera:

```
const char *recenica_c_str{ recenica.c_str() };
```

Svi metodi klase `string` su dizajnirani tako da ne moramo da razmišljamo o terminirajućoj nuli prilikom njihove upotrebe. Na primer, dužinu niske možemo dobiti metodom `length`[8]:

```
std::cout << recenica.length() << std::endl; // 30
```

Možemo inicijalizovati nisku tako što *konstruktoru* prosledimo `size_type count` i `char ch`, a on će za nas kreirati nisku koja sadrži `count` karaktera `ch`:

```
std::cout << std::string(5, 'n') << std::endl; // nnnnn
```

Primetimo da je ovde naglašena upotreba *konstruktora*, a ne inicijalizatora. Ukoliko koristimo inicijalizator u ovakve svrhe, kompilator će to protumačiti kao da su elementi inicijalizatora karakteri od kojih se sastoji niska, odnosno, `std::string{5, 'n'}` će kreirati nisku od dva karaktera, čiji je prvi karakter onaj sa indeksom 5 u ASCII tabeli, a drugi karakter je `'n'`.

[8] Vremenska složenost: $O(1)$ (počevši od C++11 standarda; ranije je bila nepropisana standardom).

Niska je takođe i kolekcija karaktera, tako da su nam na raspolaganju i metodi: `size`, `first`, `last` i `empty`, među onima koje smo pomenuli u sekciji 1.2.

Učitavanje jedne linije iz ulaznog toga se izvršava pozivom funkcije `getline`, definisanoj u zaglavlju `<string>`. Ova funkcija prihvata dva argumenta: (1) ulazni tok iz kojeg se čita jedna linija i (2) niska u koju će biti smešten rezultat čitanja. Zapravo, moguće je koristiti i treći argument, karakter, koji predstavlja *razdvajač* (engl. *delimiter*) i koji je podrazumevano novi red (otuda funkcija podrazumevano čita jednu liniju iz ulaznog toka):

```
std::string ime_prezime;
std::cout << "Unesite ime i prezime: ";

std::getline(std::cin, ime_prezime); // Cita karaktere do novog reda

std::cout << "Zdravo, " << ime_prezime << "!" << std::endl;
```

1.9 Reference

Kao što smo napomenuli u predgovoru, pretpostavljamo da svaki čitalac razume rad sa pokazivačima, što sa sobom nosi i razumevanje grešaka koje mogu nastati pri radu sa njima. Programski jezik C++ uvodi koncept *referenci* (engl. *reference*) koji olakšava rad sa podacima, za koji bi nam bili neophodni pokazivači u programskom jeziku C. Naravno, sa mnoštvom upotrebnih vrednosti dolazi i mnoštvo načina da se pogreši, tako da ćemo u ovoj sekciji pokušati da razumemo reference, zajedno sa sličnostima i razlikama između pokazivača i referenci.

Referenca predstavlja ime koje služi kao alijas za neku drugu promenljivu (odnosno, adresabilnu memorijsku lokaciju u opštijem slučaju). Očigledno, referenca liči na pokazivač u smislu da mora da referiše na nešto u memoriji, ali postoji nekoliko značajnih razlika.

Za početak, ne možemo deklarisati referencu bez njene definicije. S obzirom da je referenca zapravo alijas, to znači da se promenljiva za koju ona predstavlja alijas mora navesti prilikom deklarisanja te reference. Takođe, referenca ne može biti modifikovana da bude alijas na nešto drugo — jednom kada predstavlja alijas za promenljivu, taj odnos važi sve do kraja životnog veka je reference.

Pogledajmo kako se reference inicijalizuju i koriste. Pretpostavimo da nam je na raspolaganju naredni podatak:

```
double broj{2.5};
```

Definisanje reference ka ovoj promenljivoj se može izvršiti na sledeći način:

```
double& r_broj{broj};
```

Karakter `&` nakon tipa `double` indikuje da promenljiva `r_broj` predstavlja referencu ka promenljivoj koja je tipa `double`. Promenljiva na koju se ta referenca odnosi se specifikuje u inicijalizatoru. Dakle, promenljiva `r_broj` je tipa "referenca ka `double`".

Referenca se može koristiti kao alternativa originalne promenljive:

```
r_broj += 1.0;

std::cout << r_broj << std::endl; // 3.5
std::cout <<   broj << std::endl; // 3.5
```

Primetimo da smo u prethodnom kodu koristili promenljivu `r_broj` bez ikakve upotrebe operatora dereferenciranja, kao što bismo morali da koristimo u radu sa pokazivačima — referencu jednostavno koristimo na isti način kao i originalnu promenljivu.

Promena vrednosti reference, kao što smo rekli, nije moguća. Pogledajmo efekat narednog koda:

```
double novi_broj{7.0};

r_broj = novi_broj; // Menja se vrednost promenljive "broj"
                    // kroz referencu "r_broj".
                    // Prakticno, kao da smo napisali "broj = novi_broj;".

std::cout << broj << std::endl; // 7
```

Konstantne reference je takođe moguće definisati:

```
const double& cr_broj{ broj };
```

U ovom slučaju, referenca `cr_broj` se ne može koristiti za promenu vrednosti originalne promenljive:

```
// error C3892: 'cr_broj': you cannot assign to a variable that is const
cr_broj *= 0;
```

Prisetimo se našeg koda koji koristi kolekcijsku petlju za izračunavanje sume elemenata vektora:

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
double suma{};

for (double element : niz)
{
    suma += element;
}

std::cout << suma / niz.size() << std::endl; // 3.5
```

Ono što nismo rekli jeste da se u svakoj iteraciji kreira kopija elementa vektora koja se čuva kao lokalna promenljiva `element` u telu kolekcijske petlje. Kopiranje `double` vrednosti možda i nije previše skupo, ali zamislimo da naš vektor čuva veliki broj objekata koji sadrže veliki broj podataka. Tada dolazi do previše kopiranja, koje se može veoma jednostavno rešiti korišćenjem referenci:

```
for (double& element : niz)
{
    suma += element;
}
```

Jedina izmena u kodu je ta što `element` sada postaje referenca na jedan element u vektoru `niz`, koji se inicijalizuje u svakoj iteraciji. Kao što smo rekli, prednost ovoga je u tome što smo izbegli ukupno n operacija kopiranja vrednost elemenata u nizu u lokalnu promenljivu pri svakoj iteraciji.

Kako smo rekli da se preko reference može vršiti promena vrednosti promenljive na koju ta referenca referiše, onda je moguće koristiti kolekcijsku petlju za promenu vrednosti u vektoru. Na primer, naredni kod udvostručava elemente vektora:

```
for (double& element : niz)
{
    element *= 2;
}
```

Ipak, nekada ne želimo da menjamo vrednosti u vektoru, već samo želimo da pristupamo njegovim elementima, te je u tom slučaju poželjno koristiti konstantne reference:

```
for (const double& element : niz)
{
    suma += element;
}
```

Osim što je dobra praksa, na ovaj način možemo sprečiti nenamerne greške u kodu koje se tiču menjanja elemenata koje ne bi trebalo menjati:

```
// error C3892: 'element': you cannot assign to a variable that is const
for (const double& element : niz)
{
    element *= 2;
}
```

1.10 Ključna reč `auto`

Česte su prilike kada nam je mukotrpno da eksplicitno pišemo tip promenljive koja se inicijalizuje nekom vrednošću. Najočigledniji primer toga sa kojim smo se do sada susreli jeste definisanje promenljive koja sadrži veličinu vektora:

```
std::vector<std::string> reci{ "danas", "je", "lep", "dan" };
std::vector<std::string>::size_type broj_reci{ reci.size() };
std::cout << broj_reci << std::endl; // 4
```

Iako smo napomenuli da bismo mogli da koristimo neki drugi tip koji je pogodan u te svrhe, mi ne znamo za koji osnovni tip `std::vector<std::string>::size_type` prestavlja njegov `typedef` alijas.

Na sreću, od verzije C++11 na raspolaganju nam je ključna reč `auto` koja ima sposobnost da dedukuje vrednost na osnovu koje se inicijalizuje promenljiva. Drugim rečima, možemo pisati kod kao u narednom primeru i prepustili kompilatoru da dedukuje odgovarajući tip za nas:

```
std::vector<std::string> reci{ "danas", "je", "lep", "dan" };

auto broj_reci{ reci.size() };

std::cout << broj_reci << std::endl; // 4
```

Pogledajmo kako kompilator razume izraz:

```
auto broj_reci{ reci.size() };
```

Kompilator prvo izračunava vrednost izraza `reci.size()`, a zatim utvrđuje da je u pitanju vrednost tipa `std::vector<std::string>::size_type` na osnovu definicije metoda `size`, i koristi taj tip na mesto ključne reči `auto`.

Automatsko dedukovanje tipa se može koristiti i na osnovu literala:

```
auto m{10};       // "m" ima tip "int"
auto n{200UL};    // "n" ima tip "unsigned long"
auto pi{3.14159}; // "pi" ima tip "double"
```

Isto važi i u slučaju konstrukcije ili dodeljivanja vrednosti:

```
auto m = 10;       // "m" ima tip "int"
auto n = 200UL;    // "n" ima tip "unsigned long"
auto pi = 3.14159; // "pi" ima tip "double"
```

Treba biti oprezan prilikom korišćenja `auto` i inicijalizatora. Na primer, postoji razlika između naredne dve naredbe:

```
auto broj1{ 7 };    // int (od C++17, a pre toga std::initializer_list<int>
    !!!)
auto broj2 = { 7 }; // std::initializer_list<int> (!!!)

auto ovo_nije_broj_sigurno1{ 1, 2, 3 };    // std::initializer_list<int> (!!!)
auto ovo_nije_broj_sigurno2 = { 1, 2, 3 }; // std::initializer_list<int> (!!!)
```

Vidimo da u raznim varijantama korišćenja inicijalizatora možemo dobiti različite tipove, pa čak i u različitim verzijama standarda! Zbog toga treba biti posebno oprezan pri kombinaciji inicijalizacije i dodeljivanja.

Šta je `std::initializer_list`? U pitanju je šablonska klasa za kreiranje liste objekata koji se koriste za inicijalizaciju kolekcija, poput `vector` objekata, koje smo koristili do sada. Dakle, prilikom inicijalizacije vektora `niz` narednim kodom

```
std::vector niz{ 1, 2, 3 };
```

ono što se zapravo dešava jeste da se kreira objekat tipa `std::initializer_list<int>` koji se zatim koristi da popuni `vector` objekat `niz` početnim vrednostima.

Vratimo se nazad na `auto`. Zanimljivo je videti da se `auto` može koristiti za dedukciju tipa u kolekcijskoj petlji:

```
std::vector<double> niz{1., 2., 3., 4., 5., 6.};
double suma{};

for (auto element : niz)
{
    suma += element;
}

std::cout << suma / niz.size() << std::endl; // 3.5
```

Korišćenje ključne reči `auto` ima jednu, potencijalno neočiglednu karakteristiku. `auto` nikad ne dedukuje referencni tip, već uvek na vrednosni tip. Posledica ovoga je da čak i kada dodelimo referencu `auto`, vrednost se uvek kopira. Štaviše, ova kopija nikad neće biti tipa `const`, te ju je potrebno dodatno okarakterisati kao `const auto`. Zaključak: ukoliko želimo da kompilator dedukuje referentni tip, moramo koristiti `auto&` ili `const auto&`. Na primer:

```
std::string pitanje{ "Da li volite cokoladu?" };
const std::string& ref_pitanje { pitanje };
auto auto_test = ref_pitanje;
```

U ovom fragmentu koda, promenljiva `auto_type` ima tip `std::string` i kao takva sadrži kopiju teksta koji se nalazi u promenljivoj `pitanje`. Dodatno, za razliku od `ref_pitanje`, ova nova kopija nije čak ni konstantna.

Vežbanje Kod za ovaj primer se nalazi na lokaciji `Primeri/1/2/main.cpp`.

Test primer 1

```
Ulaz:
Ovo je jedna srednja recenica
Ovo je kratka recenica
Ovo je veoma dugacka recenica koja ima dosta nepotrebnog teksta

Prethodna recenica je bila prazna
Ovo je poslednja recenica
Izlaz:
**************************************************************
* Ovo je jedna srednja recenica                              *
* Ovo je kratka recenica                                     *
* Ovo je veoma dugacka recenica koja ima dosta nepotrebnog teksta *
* Prethodna recenica je bila prazna                          *
* Ovo je poslednja recenica                                  *
**************************************************************
```

Test primer 2

```
Ulaz:
Izlaz:
****
****
```

2. Osnove objektno-orijentisanog programiranja

C++ ne bi bilo veoma koristan u praktičnim aplikacijama da ne podržava verovatno najrasprostanjeniju paradigmu programiranja — objektno-orijentisano programiranje. Klase, objekti, atributi, metodi i učauravanje su samo neki od pojmova sa kojima ćemo se upoznati u ovom poglavlju. Čitaocu bi već trebalo da ovi pojmovi budu poznati, bez obzira na koji programski jezik se do sada susreo, a mi ćemo tek ilustrovati OOP mehanizme koje se koriste u jeziku C++.

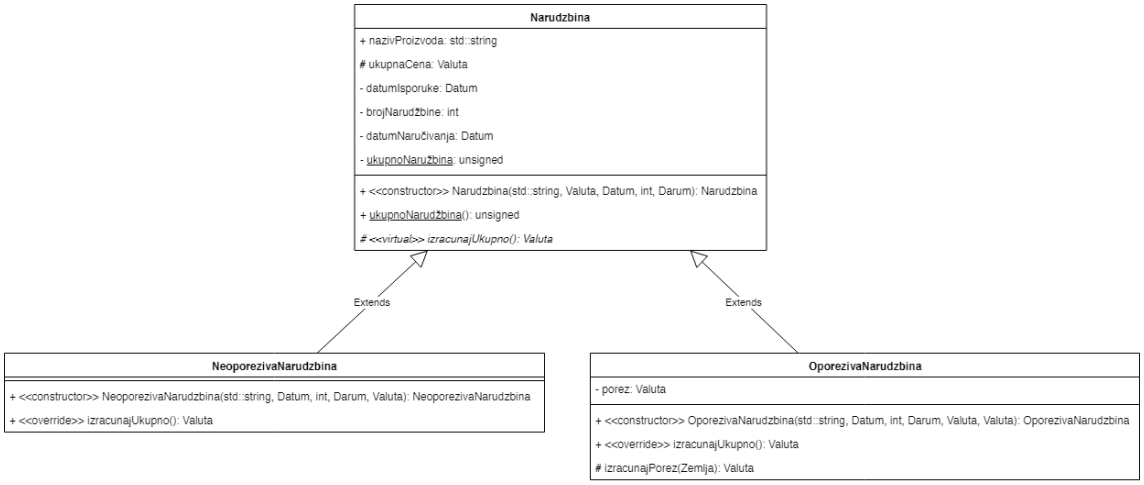
2.1 UML dijagram klasa

Jedan od problema razvoja softvera sa kojim se veći projekti susreću jeste komunikacija između *učesnika u projektu* (engl. *stakeholder*) i razvijača softvera. Razvijačima softvera je neophodan precizan, tehnički opis problema od strane učesnika, dok učesnici najčešće definišu problem u tekstualnom ili vizualnom formatu.

Da bi se napravio "kompromis" između ove dve strane, dizajniran je jezik koji se naziva *unifikovani jezik za modeliranje* (engl. *Unified Modeling Language*, skr. *UML*) i koji uvodi veliki broj dijagrama kojima se opisuju razni delovi projekta, kao što su: modeli, procesi, isporučivanje i dr.

Jedan od dijagrama koji UML propisuje jeste tzv. *dijagram klasa* (engl. *class diagram*). Kako mu samo ime kaže, dijagram klasa služi za prikazivanje klasa u sistemu, zajedno sa vezama između njih, kao i njihovih osobina (atributi) i ponašanja (metodi). Iako je celokupna diskusija o UML jeziku, pa i o svim mogućnostima dijagrama klasa van okvira ove skripte, prikazaćemo neke osnovne elemente dijagrama klasa kako bismo mogli da preciznije definišemo zahteve primera koji slede. Napomenimo da ćemo koristiti specifičnosti jezika C++ u diskutovanju o dijagramima klasa, tako da će se neki elementi razlikovati ukoliko se kreira dijagram klasa za, na primer, jezike C# ili Java.

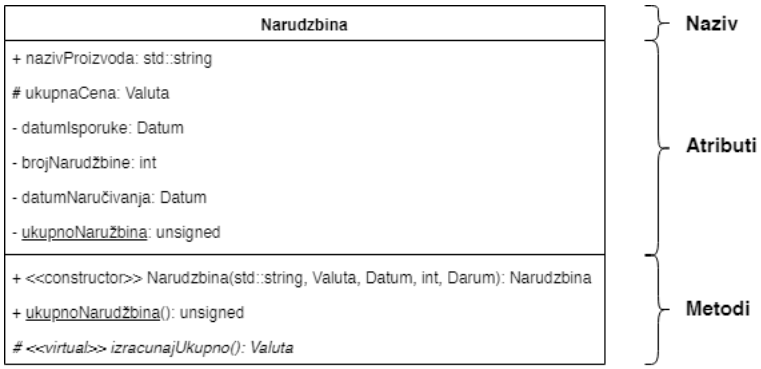
Postoje različite verzije ovog dijagrama: *dijagram klasa za analizu* i *dijagram klasa za*



Slika 2.1: Primer dijagrama klasa.

dizajn, ali mi ćemo koristiti drugi od njih, s obzirom da on služi da detaljno opiše implementaciju klase[1]. Primer jednog dijagrama klase dat je na slici 2.1.

Na prvi pogled može delovati da ovaj dijagram sadrži previše informacija koje se analiziraju odjednom, ali ako znamo da ga posmatramo na pravi način, onda nam neće biti problem da ga razumemo.



Slika 2.2: Primer modela jedne klase.

Započnimo od modela jedne klase, koji je prikazan na slici 2.2. Svaki model klase se sastoji od naziva klase, liste atributa (koji definišu osobine klase) i liste metoda (koji definišu ponašanje klase). Neke klase, kao što je, na primer, klasa `NeoporezivaNarudzbina` na slici 2.1, nemaju svoje atribute ili metode. U tom slučaju, odgovarajuća lista je prazna, ali se ostavlja prazan prostor na mestu gde bi stajala popunjena lista, da ta klasa ima sopstvene atribute/metode.

Svaki od atributa, odnosno, metoda zadaje je jednim *ajtemom* (engl. *item*) čija je forma prikazana na slici 2.3. Svaki ajtem se sastoji od: modifikatora pristupa, naziva i tipa. Bilo da li je u pitanju atribut ili metod, modifikator pristupa uvek ima isto značenje:

- Znak + označava da je modifikator pristupa `public`.

[1] Za razliku od njega, dijagram klasa za analizu služi samo da opiše klasu navođenjem podataka koji se sadrže u klasi i operacija koje klasa može da izvrši, bez ulaženja u detalje implementacije.

- Znak `#` označava da je modifikator pristupa `protected`.
- Znak `-` označava da je modifikator pristupa `private`.

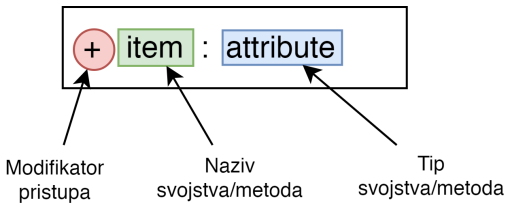
Primetimo da se ajtemi obično navode po restrikciji modifikatora pristupa opadajuće.

Ukoliko ajtem opisuje jedan atribut, onda naziv ajtema odgovara nazivu atributa koji se koristi u implementaciji te klase, na primer, `nazivProizvoda`. Sa druge strane, ako ajtem opisuje jedan metod, onda se pored naziva metoda (koji je ponovo isti u ajtemu i u implementaciji klase), specifikuju i tipovi argumenata tog metoda. Na primer, metod `void ispisi(int broj, char slovo)` bi imao ajtem čiji bi naziv bio oblika `ispisi(int, char)`.

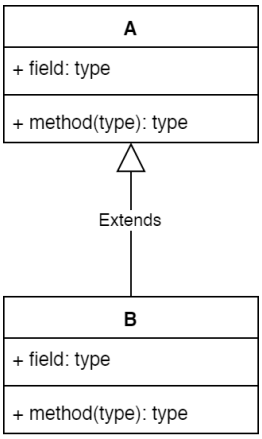
Ponovo, tip ajtema se razlikuje u zavisnosti od toga da li on opisuje atribut ili metod. U slučaju atributa, tip ajtema predstavlja upravo tip kojim je taj atribut definisan u klasi. U slučaju metoda, tip ajtema predstavlja tip povratne vrednosti tog metoda.

Postoje još neke specifičnosti jezika C++ koje se jednostavno prevode na dijagram klasa:

- Statički atributi i metodi se označavaju podvučenom linijom.
- Konstruktori se označavaju postavljanjem oznake `<<constructor>>` između modifikatora pristupa i naziva metoda. S obzirom da se nazivi metoda konstruktora uvek poklapaju sa nazivom klase, onda nije neophodno koristiti ovu oznaku.
- Apstraktni metodi se označavaju *iskošenim slovima*.
- Virtualni metodi se označavaju postavljanjem oznake `<<virtual>>` u natklasi, a u klasi koja implementira taj virtualni metod se koristi oznaka `<<override>>`.
- Konstantne članove je moguće označiti korišćenjem oznake `<<const>>`.



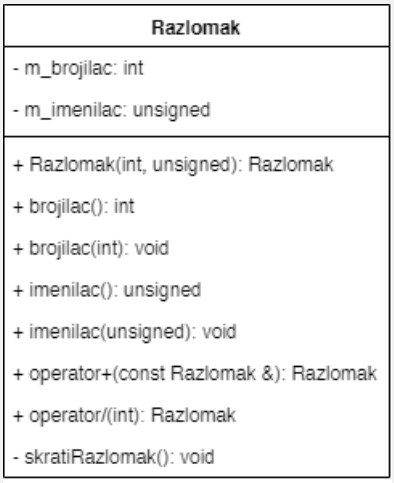
Slika 2.3: Elementi jednog ajtema u modelu klase.



Slika 2.4: Primer nasleđivanja u dijagramu klasa. Obratiti pažnju na smer strelice koja označava nasleđivanje.

Nasleđivanje klasa je ilustrovano na slici 2.4. Nasleđivanje se jednostavno ilustruje korišćenjem linije koja ima oznaku `Extends` sa nepopunjenom strelicom u smeru od potklase ka natklasi. Drugim rečima, iz dijagrama na pomenutoj slici možemo zaključiti da klasa `B` nasleđuje klasu `A`.

Primer 2.1 Implementirati klasu koja omogućava rad sa razlomcima i koja je data narednim dijagramom klasa:



Takođe, implementirati funkcije za ispis razlomka na izlazni tok i učitavanje iz ulaznog toka. Razmisliti koji članovi klase treba da budu definisani kao konstantni.

2.2 Rad sa klasama

Klasa predstavlja korisnički-definisan tip podataka. Definicija klase započinje ključnom rečju `class`, koja je praćena identifikatorom klase, vitičastim zagradama u okviru kojih se navodi definicija klase i tačkom-zapetom:

```
class MojaKlasa
{
    // Ovde ide definicija klase
};
```

Svi članovi klase su podrazumevano privatne vidljivosti. Drugim rečima, njima ne možemo pristupiti ni iz jednog dela koda koji nije deo klase u kojoj su definisani. Generalno, da bismo napomenuli kompilatoru koje članove naše klase kod koji je van te klase može da pristupi i na koji način, koristimo naredne *modifikatore pristupa* (engl. *access specifier*):

- `public` — definiše da se član klase može koristiti u bilo kod delu koda.
- `protected` — definiše da se član klase može koristiti u toj klasi ili u nekoj od nasleđenih klasa.
- `private` — definiše da se član klase može koristiti isključivo u toj klasi.

Naredni primer ilustruje korišćenje modifikatora pristupa:

```
class MojaKlasa
{
public:
    // Ovde idu deklaracije/definicije javnih clanova klase

private:
    // Ovde idu deklaracije/definicije privatnih clanova klase
};
```

Nakon navođenja modifikatora pristupa sledi niz deklaracija i definicija članova te klase. Ovaj modifikator pristupa važi sve do pojave narednog modifikatora pristupa ili do kraja definicije klase. Možemo imati više modifikatora pristupa različitog ili istog tipa, u proizvoljnom redosledu.

Svaka klasa ima dva osnovna tipa članova: atributi i metodi. Atributi predstavljaju promenljive koje određuju stanje, dok metodi predstavljaju funkcije koje opisuju ponašanje objekata te klase. Iako su metodi zapravo funkcije, namerno koristimo različitu terminologiju za funkcije koje su definisane van klase (termin *funkcija* (engl. *function*)) i za funkcije koje su definisane kao deo neke klase (termin *metod* (engl. *method*)).

Prikažimo jednu jednostavnu klasu:

```
class NapitakUSolji
{
private:
    const double m_najvecaKolicina{250.0};
    double m_kolicinaNapitka{};

private:
    bool mozeLiDaStane(const double _novaKolicina) const
    {
        if (m_najvecaKolicina - m_kolicinaNapitka < _novaKolicina)
        {
            return false;
        }
        return true;
    }

public:
    void sipajNapitakAkoMoze(const double _novaKolicina)
    {
        if (!mozeLiDaStane(_novaKolicina))
        {
            return;
        }
        m_kolicinaNapitka += _novaKolicina;
    }

    void ispisiKolicinu() const
    {
        std::cout
            << "Trenutna kolicina napitka je "
            << m_kolicinaNapitka
            << std::endl;
    }
};
```

Ono što primećujemo jeste da ova klasa sadrži dva privatna atributa tipa `double`: promenljive `m_najvecaKolicina` i `m_kolicinaNapitka`, koji opisuju najveću količinu napitka koja može da stane u šolju i tekuću količinu napitka u šolji, redom. Dodatno, obe vrednosti su inicijalizovane za svaki objekat koji se napravi — šolja ima zapreminu od 250*cl* (inicijalizacija `m_najvecaKolicina` na `250.0`) i inicijalno je prazna (inicijalizacija `m_kolicinaNapitka` na podrazumevanu vrednost `0.0`). Ovim atributima je opisano *stanje* naše klase.

Klasa takođe definiše ukupno tri metoda: jedan privatan i dva javna. Javni metod `sipajNapitakAkoMoze` služi da doda novu količinu napitka na tekuću količinu napitka u šolji. Da bi to uradio, potrebno je da proveri da li nova kolicina može da stane, pre promene

atributa `m_kolicinaNapitka`. Za to koristi privatni metod `mozeLiDaStane` koji vraća `true` ukoliko može i `false` u suprotnom. Imamo još jedan javni metod `ispisiKolicinu` koji na standardni izlaz ispisuje informaciju o tekućem stanju napitka. Ove funkcije definišu *ponašanje* naše klase.

Razmotrimo sada na koji način su rešena neka pitanja objektno-orijentisane paradigme u C++ implementaciji ove klase:

1. *Učauravanje* (engl. *encapsulation*) — Kao što smo rekli, stanje klase `NapitakUSolji` određeno je njegovom najvećom zapreminom i tekućom količinom napitka. Oba ova podatka su deklarisana pod `private` modifikatorom pristupa da bismo se osigurali da nijedan kod koji se nalazi van ove klase ne može da ih promeni. Dodatno, metod `mozeLiDaStane` služi samo kao pomoćni metod i nije neophodan da bude dostupan korisniku, zbog toga je i on deklarisan pod `private` modifikatorom pristupa. U prethodnom kodu vidimo da možemo da koristimo `private` dva puta, jedan za drugim, ali nam to nije neophodno — isti efekat bismo postigli i da smo napisali:

   ```
   class NapitakUSolji
   {
   private:
       const double m_najvecaKolicina{250.0};
       double m_kolicinaNapitka{};

       bool mozeLiDaStane(const double _novaKolicina) const
       {

       // Ostatak koda...
   ```

   Zapravo, s obzirom da su podrazumevano članovi klase privatni, onda ne moramo ni da pišemo `private` modifikator:

   ```
   class NapitakUSolji
   {
       const double m_najvecaKolicina{250.0};
       double m_kolicinaNapitka{};

       bool mozeLiDaStane(const double _novaKolicina) const
       {

       // Ostatak koda...

   public:
       void sipajNapitakAkoMoze(const double _novaKolicina)
       {

       // Ostatak koda...
   ```

   Naravno, ovo ne bismo mogli da uradimo da smo prvo napisali javne metode, pa nakon njih privatne — u tom slučaju je neophodno korišćenje `private` modifikatora pristupa:

   ```
   class NapitakUSolji
   {
   public:
       void sipajNapitakAkoMoze(const double _novaKolicina)
       {

       // Ostatak koda...

   private:
   ```

```
    const double m_najvecaKolicina{250.0};

    // Ostatak koda...
```

2. *Interfejs* (engl. *interface*) — Sa druge strane, korisniku su na raspolaganju dve operacije: dodavanje nove i ispisivanje tekuće količine napitka, te su zbog toga one deklarisane pod `public` modifikatorom pristupa. Metodi koji omogućavaju korisniku da dobiju ili menjaju informacije o objektu ulaze u sastav njegovog interfejsa.
3. *Konstantnost* (engl. `const`-*modification*) — U sekciji 1.7 govorili smo o konstantnim vrednostima i ključnoj reči `const`. Konstantnost podataka igra značajnu ulogu u definisanju klasa. Svi atributi čije vrednosti ne bi trebalo da se menjaju treba definisati kao konstantne — to već znamo. Zbog toga je atribut `m_najvecaKolicina` definisan kao konstantan, jer ne možemo da menjamo zapreminu šolje. Međutim, ono što je novo jeste da svi metodi koji ne menjaju stanje objekta, takođe treba da budu deklarisani kao konstantni. Ovo se izvodi navođenjem `const` kvalifikatora na kraju deklaracije metoda. Metodi `mozeLiDaStane` i `ispisiKolicinu` samo dohvataju trenutnu informaciju o objektu, ne menjajući njihove vrednosti, te su oni kandidati za konstantne metode. Sa druge strane, metod `sipajNapitakAkoMoze` *potencijalno* menja atribut `m_kolicinaNapitka`, te on ne može biti konstantan. Ključno je primetiti reč *potencijalno* u prethodnoj rečenici — ako postoji redosled izvršavanja operacija u metodu koji može da promeni stanje objekta, onda on ne sme biti konstantan. Napomenimo još i to da nekonstantni metodi mogu da vrše pozive ka konstantnim metodima (u primeru vidimo da metod `sipajNapitakAkoMoze` poziva metod `mozeLiDaStane`), dok konstantan metod može da zove isključivo konstantne metode.

Prikažimo i primer rada programa koji koristi ovu klasu:

```
int main()
{
    NapitakUSolji solja{};

    solja.sipajNapitakAkoMoze(127.5);
    solja.ispisiKolicinu(); // Trenutna kolicina napitka je 127.5

    solja.sipajNapitakAkoMoze(79.1245);
    solja.ispisiKolicinu(); // Trenutna kolicina napitka je 206.625

    solja.sipajNapitakAkoMoze(98.36);
    solja.ispisiKolicinu(); // Trenutna kolicina napitka je 206.625

    return 0;
}
```

Vidimo da se definisanje promenljive `solja` koja objekat klase `NapitakUSolji` kreira na isti način kao i definisanje promenljive osnovnog tipa. Možemo kreirati objekat i bez navođenja inicijalizatora:

```
NapitakUSolji solja;
```

S obzirom na podrazumevane vrednosti atributa klase, svi objekti kreirani na ovaj način imaju iste početne vrednosti. Naravno, njihove vrednosti se menjaju odvojeno jer oni ne dele podatke među sobom[2].

[2]Samo su statički članovi klase zajednički za sve instance objekata. O tome će biti reči u sekciji 2.12.

Što se tiče funkcionalne notacije, tj. konstrukcije, za sada, ona se jedino može koristiti zajedno sa dodeljivanjem, kao u narednom primeru:

```
NapitakUSolji solja = NapitakUSolji();
```

Sa dodeljivanjem se može koristiti i inicijalizaciona lista, odnosno, inicijalizator, kao u narednim primerima:

```
NapitakUSolji solja1 = {};
NapitakUSolji solja2 = NapitakUSolji{};
```

2.3 Konstruktor

Za sada naša klasa `NapitakUSolji` radi dobro, međutim, ona ipak ima jedan problem. Sve šolje koje se kreiraju imaju istu zapreminu od $250cl$. Šta ukoliko bismo želeli da kreiramo šolje koje imaju različitu zapreminu? Za početak, razmislimo kako bismo ovo mogli da uradimo.

Jedan način je da kreiramo javan metod koji će atribut `m_najvecaKolicina` postaviti na vrednost koja se prosleđuje kao argument:

```
// U klasi NapitakUSolji

void najvecaKolicina(double _najvecaKolicina)
{
    m_najvecaKolicina = _najvecaKolicina;
}

// Van klase NapitakUSolji

NapitakUSolji solja;
solja.najvecaKolicina(500.0);

solja.sipajNapitakAkoMoze(127.5);
solja.ispisiKolicinu();
solja.sipajNapitakAkoMoze(79.1245);
solja.ispisiKolicinu();
solja.sipajNapitakAkoMoze(98.36);
solja.ispisiKolicinu();
```

Međutim, sa ovom izmenom smo uveli grešku u narednoj liniji koda (komentar iznad prikazuje grešku do koje se dolazi):

```
// error C2166: l-value specifies const object
m_najvecaKolicina = _najvecaKolicina;
```

Drugim rečima, mi želimo da promenimo vrednost atributa `m_najvecaKolicina` koji je deklarisan kao konstantan. Dakle, da bismo ovo rešili, moramo da uklonimo konstantnost tog atributa:

```
double m_najvecaKolicina{};
```

Sada se kod kompilira i radi kao što je očekivano:

```
Trenutna kolicina napitka je 127.5
Trenutna kolicina napitka je 206.625
Trenutna kolicina napitka je 304.985
```

Međutim, ovim smo sad uveli dva nova problema, ali ovoga puta semantičke prirode. Prvo, atribut `m_najvecaKolicina` vise nije konstantan, tako da možemo da ga menjamo više puta. Ovo očigledno nije korektno jer smo rekli da jednom kada se kreira šolja, njena zapremina bi trebalo da bude nepromenjena, a nas ništa ne sprečava da napišemo naredni kod:

```
NapitakUSolji solja;
solja.najvecaKolicina(500.0);
solja.najvecaKolicina(1000.0);
solja.najvecaKolicina(1500.0);
```

Posledica ovog problema je drugi problem: mi možemo da dodajemo napitak do neke vrednosti, a nakon toga da promenimo vrednost zapremine šolje da vrednost koja je manja od trenutne količine:

```
NapitakUSolji solja;
solja.najvecaKolicina(500.0);

solja.sipajNapitakAkoMoze(127.5);
solja.ispisiStanje();
solja.sipajNapitakAkoMoze(79.1245);
solja.ispisiStanje();
solja.sipajNapitakAkoMoze(98.36);
solja.ispisiStanje();

solja.najvecaKolicina(100.0);
solja.ispisiStanje();
```

čime se dobija nevalidna situacija u četvrtoj liniji ispisa:

```
Trenutna kolicina napitka je 127.5 od najvece kolicine 500
Trenutna kolicina napitka je 206.625 od najvece kolicine 500
Trenutna kolicina napitka je 304.985 od najvece kolicine 500
Trenutna kolicina napitka je 304.985 od najvece kolicine 100
```

Možemo da zamislimo problematiku do koje dolazi prilikom korišćenja ovakve klase u implementaciji, na primer, aparata za kafu.

Napomenimo samo da smo umesto metoda `ispisiKolicinu` implementirali naredni metod:

```
void ispisiStanje() const
{
    std::cout
        << "Trenutna kolicina napitka je "
        << m_kolicinaNapitka
        << " od najvece kolicine "
        << m_najvecaKolicina
        << std::endl;
}
```

Rešenje oba ova problema je moguće izvršiti kreiranjem pomoćnog, privatnog atributa `m_vecJePostavljenaKolicina` koji čuva Bulovu vrednost da li je najveća zapremina šolje već postavljena. Njegovom proverom onemogućujemo da se metod `najvecaKolicina` poziva više od jednog puta:

```
private:
    bool m_vecJePostavljenaKolicina{false};
    // ...

public:
    void najvecaKolicina(double _najvecaKolicina)
```

```
    {
        if (m_vecJePostavljenaKolicina)
        {
            return;
        }
        m_najvecaKolicina = _najvecaKolicina;
        m_vecJePostavljenaKolicina = true;
    }
```

Kompiliranje prolazi i program radi kako treba:

```
Trenutna kolicina napitka je 127.5 od najvece kolicine 500
Trenutna kolicina napitka je 206.625 od najvece kolicine 500
Trenutna kolicina napitka je 304.985 od najvece kolicine 500
Trenutna kolicina napitka je 304.985 od najvece kolicine 500
```

Dakle, za inicijalizaciju jednog konstantnog atributa potrebno nam je da implementiramo metod od 6 linija koda i pomoćni atribut koji "kontroliše konstantnost" tog konstantnog atributa. Količina ovog (videćemo uskoro zašto bespotrebnog) koda relativno brzo raste povećanjem broja konstantnih atributa koje se koriste. Dodatno, implementacija koda na ovakav način odlaže proveru konstantnosti koda (statička greška) do faze izvršavanja, a kad smo definisali atribut `m_najvecaKolicina` kao konstantan, tad smo definisali njegovu semantiku još u fazi kompiliranja — bolje je ostaviti kompilatoru da detektuje statičke greške u fazi kompiliranja nego debageru u fazi izvršavanja.

Ako malo bolje razmislimo (što smo mogli odmah da uradimo na početku sekcije), postavljanje zapremine šolje se dešava samo jednom i to prilikom *konstrukcije* objekta koji predstavlja instancu te klase. Za ovaj posao možemo iskoristiti mehanizam programskog jezika C++ koji se naziva *konstruktorski metodi* ili samo *konstruktori* (engl. *constructor*). Konstruktor nije ništa drugo do metod koji se izvršava tokom konstrukcije objekta i njegove inicijalizacije. Pogledajmo jedan primer:

```
#include <iostream>

class Broj
{
private:
    unsigned m_broj;

public:
    // Konstruktor klase Broj:
    Broj(const unsigned _broj)
    {
        m_broj = _broj;
    }

    unsigned broj() const
    {
        return m_broj;
    }
};

int main()
{
    Broj a{3u};
    Broj b{4u};

    std::cout << a.broj() + b.broj() << std::endl; // 7
}
```

Klasa `Broj` ima jedan konstruktor čiji naziv mora da se poklopi sa nazivom klase. Taj konstruktor prihvata jedan argument kojim se inicijalizuje privatni atribut `m_broj`. Primetimo da nismo implementirali metod koji postavlja vrednost tog atributa. Nismo ni imali potrebe za time jer je konstruktor izvršio posao inicijalizacije za nas.

Prodiskutujmo narednu liniju koda:

```
Broj a{3u};
```

Šta se ovde zapravo desilo? Tokom izvršavanja ove linije koda, program je kreirao novi objekat klase `Broj` (na steku funkcije) i pozvao konstruktor klase koji prihvata jedan broj i prosledio mu je vrednost `3u`. U pozvanom konstruktoru, vrednost `3u` se iskoristila za inicijalizaciju atributa `m_broj`. Konačno, vezan je identifikator `a` za kreirani objekat klase `Broj`. Zbog toga, kada smo koristili metod `broj` za dohvatanje te vrednosti, dohvaćene su očekivane vrednosti koje smo prosledili prilikom inicijalizacije.

Pozivanje konstruktora se sada može izvršiti na više načina:

```
Broj a{3u};
Broj b(4u);
Broj c = {5u};
Broj d = Broj{6u};
Broj e = Broj(7u);

std::cout
    << a.broj() + b.broj() + c.broj() + d.broj() + e.broj()
    << std::endl; // 25
```

Kreiranje promenljivih `c`, `d` i `e` ima nešto drugačiju semantiku sa kojom ćemo se upoznati uskoro, te ćemo zbog toga najčešće koristiti princip kreiranja koji je korišćen za kreiranje promenljive `a` (ili, eventualno, `b`).

Pre nego što se vratimo na rešavanje našeg problema sa klasom `NapitakUSolji`, pogledajmo još nekoliko interesantnih karakteristika konstruktora.

Šta mislite da bi se desilo ako bismo kreirali objekat klase `Broj` bez da mu prosledimo vrednost za inicijalizaciju, kao u narednom kodu:

```
Broj a; // ili Broj a{};
```

Ako ste pretpostavili da će se pojaviti greška, bili ste u pravu:

```
error C2512: 'Broj': no appropriate default constructor available
```

Problem je u tome što se prilikom konstrukcije objekata korišćenjem naredbe:

```
Broj a; // ili Broj a{};
```

poziva specijalan konstruktor koji se naziva *podrazumevani konstruktor* (engl. *default constructor*). Zapravo, prilikom kreiranja objekata naše problematične klase `NapitakUSolji`, koristili smo podrazumevani konstruktor za kreiranje objekta:

```
NapitakUSolji solja; // Poziva se podrazumevani konstruktor: NapitakUSolji()
```

Sada se možete zapitati — kako je moguće da ta naredba poziva metod koji nismo definisali u našoj klasi? Razlog zašto ovo funkcioniše je u tome što, ukoliko klasa nema

definisan nijedan konstruktor, kompilator automatski definiše *podrazumevani podrazumevani konstruktor*[3] čija je definicija sledeća:

```
NapitakUSolji()
{}
```

Drugim rečima, podrazumevani podrazumevani konstruktor[4] predstavlja javni konstruktorski metod koji nema argumenata i ima prazno telo.

Međutim, šta ako želimo da imamo i podrazumevani konstruktor i neki drugi konstruktor? Rešenje je vrlo jednostavno — klasa može imati više od jednog konstruktora:

```
class Broj
{
private:
    unsigned m_broj;

public:
    // Konstruktor klase Broj:
    Broj(const unsigned _broj)
    {
        m_broj = _broj;
    }

    // Podrazumevani konstruktor klase Broj:
    Broj()
    {
        m_broj = 0u;
    }

    unsigned broj() const
    {
        return m_broj;
    }
};
```

Sada možemo kreirati objekte korišćenjem bilo kog konstruktora:

```
Broj a;
Broj b{4u};

std::cout << a.broj() + b.broj() << std::endl; // 4
```

Već smo napomenuli, ali s obzirom na novu terminologiju, napomenimo još jednom da se podrazumevani konstruktor može pozivati na jedan od narednih načina:

```
// Ispravno:
Broj a;
Broj b{};

// Poziva se podrazumevani konstruktor,
// ali ne radi tacno ono sto ocekujemo:
Broj c = {};
Broj d = Broj{};
Broj e = Broj();

// Neispravno:
// Broj f();
```

[3] Nije greška u kucanju — kompilator kreira podrazumevani konstruktor koji ima podrazumevanu definiciju — otuda "podrazumevani podrazumevani konstruktor"

[4] I dalje nije greška u kucanju!

2.4 Podrazumevane vrednosti parametara funkcije

Pogledajmo naredni fragment koda:

```
void prikazi_sistemsku_poruku(std::string poruka)
{
    std::cout << "Sistemska poruka: " << poruka << std::endl;
}

void prikazi_sistemsku_poruku()
{
    std::cout << "Sistemska poruka: " << "Sve je u redu!" << std::endl;
}

int main()
{
    prikazi_sistemsku_poruku("Proveravam da li postoji problem...");
    prikazi_sistemsku_poruku();

    return 0;
}
```

Na standardni izlaz biće ispisano:

```
Sistemska poruka: Proveravam da li postoji problem...
Sistemska poruka: Sve je u redu!
```

Da bismo implementirali mogućnost da pozivamo ”istu” funkciju koja izvršava neku podrazumevanu akciju, treba da implementiramo dve funkcije sa različitim potpisom, a sa gotovo identičnom definicijom. U slučaju kada je definicija funkcije jednostavna, to nam možda i nije problem, ali za iole kompleksnije funkcije dolazi do nepotrebnog dupliciranja gotovo identičnog koda, koji zatim treba održavati u obe funkcije.

U programskom jeziku C++ postoji jednostavniji način da se ovo implementira kroz mehanizam podrazumevanih vrednosti parametara funkcija. Pogledajmo kako bi izgledala ekvivalentna implementacija prethodnih funkcija:

```
void prikazi_sistemsku_poruku(std::string poruka = "Sve je u redu!")
{
    std::cout << "Sistemska poruka: " << poruka << std::endl;
}
```

Korišćenje funkcije ostaje isto kao u prethodnom slučaju, a pritom i rezultat koji se dobija je identičan, ali imamo samo jedno mesto koje treba izmeniti u slučaju dopunjavanja ili ispravljanja grešaka.

Napomenimo da ako je definicija funkcije odvojena od njene deklaracije, onda se podrazumevani parametri navode samo u deklaraciji:

```
void prikazi_sistemsku_poruku(std::string poruka = "Sve je u redu!");

void prikazi_sistemsku_poruku(std::string poruka)
{
    std::cout << "Sistemska poruka: " << poruka << std::endl;
}
```

Specifikovanje podrazumevanih vrednosti se može izvršiti i za vrednosti koje se prosleđuju po vrednosti, ali i po referenci:

```
void prikazi_sistemsku_poruku(const std::string & poruka = "Sve je u redu!");
```

Zbog toga što je literal niske tipa `const char[]`, funkcija `prikazi_sistemsku_poruku` mora da prihvati konstantnu referencu, inače se neće kompilirati:

```
// error C2440: 'default argument':
//     cannot convert from 'const char [15]' to 'std::string &'
void prikazi_sistemsku_poruku(std::string & poruka = "Sve je u redu!");
```

Naravno, funkcija može imati i više podrazumevanih vrednosti parametara:

```
void prikazi_podatke(const int podaci[],
                     size_t broj = 1,
                     const std::string & naslov = "Vrednosti podataka",
                     size_t sirina = 10,
                     size_t po_liniji = 5)
{
    std::cout << naslov << std::endl;

    for (size_t i{}; i < broj; ++i)
    {
        // std::setw podesava sirinu parametara u toku
        // https://en.cppreference.com/w/cpp/io/manip/setw
        std::cout << std::setw(sirina) << data[i];
        if ((i+1) % po_liniji == 0)
        {
            std::cout << std::endl;
        }
    }
    std::cout << std::endl;
}

int main()
{
    int jedanPodatak{-99};
    prikazi_podatke(&jedanPodatak);

    jedanPodatak = 13;
    prikazi_podatke(&jedanPodatak, 1, "Za neke, nesrecan podatak!");

    // std::size racuna duzinu kolekcije ili C niza
    // https://en.cppreference.com/w/cpp/iterator/size
    int uzorak[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    prikazi_podatke(uzorak, std::size(uzorak));
    prikazi_podatke(uzorak, std::size(uzorak), "Uzorak");
    prikazi_podatke(uzorak, std::size(uzorak), "Uzorak", 6);
    prikazi_podatke(uzorak, std::size(uzorak), "Uzorak", 8, 4);

    return 0;
}
```

Rezultat izvršavanja ovog programa je:

```
Vrednosti podataka
       -99
Za neke, nesrecan podatak!
        13
Vrednosti podataka
         1         2         3         4         5
         6         7         8         9        10
        11        12
Uzorak
         1         2         3         4         5
         6         7         8         9        10
```

```
        11        12
Uzorak
     1     2     3     4     5
     6     7     8     9    10
    11    12
Uzorak
       1       2       3       4
       5       6       7       8
       9      10      11      12
```

Jedno pravilo koje treba zapamtiti jeste naredno: ako i-ti parametar ima postavljenu podrazumevanu vrednost, onda svaki od $(i+1)$-og, $(i+2)$-og, ..., n-tog parametara mora imati postavljenu podrazumevanu vrednost:

```
// error C2548: 'ispisi_dva': missing default parameter for parameter 2
void ispisi_dva(int a = 0, int b)
{
    std::cout << a << ", " << b << std::endl;
}
```

Naravno, i metodi klasa su funkcije, pa i oni mogu imati podrazumevane vrednosti; čak i konstruktori:

```
class Broj
{
private:
    unsigned m_broj;

public:
    Broj(const unsigned _broj = 0u)
    {
        m_broj = _broj;
    }

    unsigned broj() const
    {
        return m_broj;
    }
};

int main()
{
    Broj a;
    Broj b{4u};

    std::cout << a.broj() + b.broj() << std::endl; // 4
}
```

2.5 Inicijalizatorska lista u konstruktoru

Sa dosadašnjim znanjem, možemo krenuti u izmenu naše klase `NapitakUSolji` tako da očuvamo konstantnost atributa `m_najvecaKolicina`. Prvo što je potrebno uraditi jeste vratiti `const` kvalifikator atributu:

```
class NapitakUSolji
{
private:
    const double m_najvecaKolicina{};
```

```
    // ...
```

Sada znamo da možemo da koristimo konstruktor za inicijalizaciju, te bismo pokušali da uradimo sledeće:

```
NapitakUSolji(double _najvecaKolicina)
{
    m_najvecaKolicina = _najvecaKolicina;
}
```

Međutim, ubrzo bismo se razočavari kada se pojavi greška prilikom kompiliranja naredne linije:

```
// error C2166: l-value specifies const object
m_najvecaKolicina = _najvecaKolicina;
```

Konstantnim objektima, kao što smo videli do sada, ne možemo dodavati vrednosti. Zašto smo onda toliko diskutovali o konstruktorima ako ne možemo da ih iskoristimo? Zapravo, ono što nismo napomenuli do sada jeste da konstruktori imaju još jedan interesantan mehanizam koji se naziva `inicijalizatorska lista` (engl. *member initializer list*). Pogledajmo naredni kod:

```
class UredjenPar
{
private:
    int m_prvi{};
    int m_drugi{};

public:
    UredjenPar(const int _prvi, const int _drugi)
        : m_prvi{_prvi}
        , m_drugi{_drugi}
    {}
};
```

Umesto da u telu konstruktora dodeljujemo vrednosti atributima klase, taj posao se obavlja u delu koda:

```
: m_prvi{_prvi}
, m_drugi{_drugi}
```

koji predstavlja inicijalizatorsku listu. Efekat ove liste je inicijalizacija atributa klase *prilikom njihove konstrukcije*, što znači da se vrednosti u zagrada (u prethodnom kodu `_prvi` i `_drugi`) koriste prilikom inicijalizacije tih atributa. Da je konstruktor imao definiciju:

```
UredjenPar(const int _prvi, const int _drugi)
{
    m_prvi = _prvi;
    m_drugi = _drugi;
}
```

onda bi se odvojeno vršila inicijalizacija atributa `m_prvi` i `m_drugi`, a zatim bi se vršilo dodeljivanje vrednosti `_prvi` i `_drugi` tim atributima, redom.

Postoji samo jedna napomena u korišćenju inicijalizatorske liste — redosled inicijalizacije atributa u inicijalizatorskoj listi odgovara redosledu navođenja atributa pri definiciji klase. To znači da ako vrednost jednog atributa zavisi od drugog, onda bez obzira na njihov

redosled u inicijalizatorskoj listi, njihov redosled mora biti ispravan prilikom definicije klase čiji su oni članovi. Pogledajmo primer gde ovo pravilo nije ispoštovano:

```
class DvostrukiBroj
{
private:
    int m_dvostrukiBroj{};
    int m_broj{};

public:
    DvostrukiBroj(const int _broj)
        : m_broj{_broj}
        , m_dvostrukiBroj{2 * m_broj}
    {}

    int dvostrukiBroj() const { return m_dvostrukiBroj; }
};

int main()
{
    DvostrukiBroj broj{5};
    std::cout << broj.dvostrukiBroj() << std::endl; // 31453432
                                                    // GRESKA!!!
}
```

Pošto se vrednosti u inicijalizatorskoj listi koriste prilikom konstrukcije atributa, onda možemo na ovaj način dodeliti inicijalne vrednosti čak i konstantnim atributima, što konačno rešava opisane probleme koje smo imali sa klasom `NapitakUSolji`:

```
class NapitakUSolji
{
private:
    const double m_najvecaKolicina{};
    double m_kolicinaNapitka{};

private:
    bool mozeLiDaStane(const double _novaKolicina) const
    {
        if (m_najvecaKolicina - m_kolicinaNapitka < _novaKolicina)
        {
            return false;
        }
        return true;
    }

public:
    NapitakUSolji(double _najvecaKolicina)
        : m_najvecaKolicina{_najvecaKolicina}
    {
    }

    void sipajNapitakAkoMoze(const double _novaKolicina)
    {
        if (!mozeLiDaStane(_novaKolicina))
        {
            return;
        }
        m_kolicinaNapitka += _novaKolicina;
    }

    void ispisiStanje() const
```

```
    {
        std::cout
            << "Trenutna kolicina napitka je "
            << m_kolicinaNapitka
            << " od najvece kolicine "
            << m_najvecaKolicina
            << std::endl;
    }
};

int main()
{
    NapitakUSolji solja{500};
    solja.sipajNapitakAkoMoze(127.5);
    solja.ispisiStanje();
    solja.sipajNapitakAkoMoze(79.1245);
    solja.ispisiStanje();
    solja.sipajNapitakAkoMoze(98.36);
    solja.ispisiStanje();

    return 0;
}
```

Radi kompletnosti dajemo i rezultat rada programa:

```
Trenutna kolicina napitka je 127.5 od najvece kolicine 500
Trenutna kolicina napitka je 206.625 od najvece kolicine 500
Trenutna kolicina napitka je 304.985 od najvece kolicine 500
```

2.6 Obrada izuzetaka

2.7 Privremeni objekti i reference kao povratne vrednosti funkcija

2.8 Implementiranje operatora

2.9 Konstruktor kopije

2.10 Operator dodeljivanja

2.11 Semantika pomeranja

2.12 Statički članovi