Sadržaj

Predgovor **9**

1 Programski jezik C **11**

1.1 Odnos jezika C i C++ . . . 11

Tipovi podataka i deklaracije . . . 11

Ulazno–izlazne operacije . . . 12

Rad sa memorijom . . . 12

Funkcije i pokazivači na funkcije . . . 12

Standardna biblioteka . . . 12

Prevođenje i organizacija programa . . . 13

Bezbednost i tipizacija . . . 13

Zaključak . . . 13

1.2 Reprezentacija osnovnih tipova . . . 13

1.2.1 Celobrojni tipovi . . . 13

1.2.2 Realni tipovi (pokretni zarez) . . . 14

1.2.3 Logički tip . . . 14

1.2.4 Označenost tipa `char` . . . 15

1.2.5 Karakter-konstante . . . 15

1.2.6 Formati za ulaz/izlaz . . . 15

1.2.7 Tip `void` . . . 16

1.2.8 Konverzije tipova . . . 16

1.2.9 Bitovske operacije . . . 18

Algoritmi zasnovani na bitovskim operatorima . . . 20

1.2.10 Niske . . . 27

Kratak rezime: razlika između C i C++ niski . . . 29

1.3 Organizacija ulaznog i izvršivog programa . . . 29

1.3.1 Pretprocesiranje . . . 30

1.3.2 Kompilacija . . . 30

1.3.3 Povezivanje . . . 31

1.3.4 Odvojena kompilacija i povezivanje . . . 31
1.3.5 Automatizacija: program *make* . . . 31
1.3.6 Izvršavanje programa . . . 32
1.3.7 Organizacija izvornog programa . . . 32
Pretprocesor . . . 33
Doseg identifikatora . . . 36
Životni vek objekata i kvalifikatori `static` i `auto` . . . 37
Povezanost identifikatora i kvalifikatori `static` i `extern` . . . 39
1.3.8 Greške u fazi prevođenja i povezivanja . . . 43
Primer organizacije programa u više datoteka . . . 45
1.3.9 Organizacija izvršivog programa . . . 48
Izvršno okruženje — operativni sistem i rantajm biblioteka . . . 48
Vrste i organizacija podataka u izvršnom programu . . . 49
Organizacija memorije . . . 50
Segment koda . . . 50
Segment podataka . . . 51
Stek segment . . . 51
Prenos argumenata u funkciju preko steka . . . 52
Implementacija rekurzije . . . 55
Primer organizacije objektnih modula i izvršivog programa . . . 56
1.3.10 Greške u fazi izvršavanja . . . 60
1.4 Pokazivači . . . 61
1.4.1 Adrese i pokazivači . . . 61
Deklaracija pokazivača . . . 61
Operator `&` – operator referenciranja . . . 62
Operator `*` – dereferenciranje . . . 62
Pokazivači i celobrojne vrednosti . . . 62
Konverzije pokazivača . . . 63
Tip `void*` . . . 63
`const` i pokazivači . . . 63
Plitko i duboko kopiranje . . . 64
Poravnanje u memoriji . . . 64
1.4.2 Pokazivači i argumenti funkcija . . . 66
1.4.3 Pokazivačka aritmetika . . . 68
1.4.4 Pokazivači na funkcije . . . 69
Deklaracije pokazivača na funkcije . . . 71
Operacije sa pokazivačima na funkcije . . . 71

1.5 Pokazivači i nizovi, dinamička alokacija memorije . . . 71
1.5.1 Pokazivači i nizovi . . . 71
Deklaracija i veličina niza . . . 71
Imena nizova vs. pokazivači . . . 72
Indeksiranje pokazivača . . . 72
Razlika između a i &a . . . 73
Nizovi kao argumenti funkcija . . . 74
1.5.2 Pokazivači i niske . . . 75
1.5.3 Nizovi pokazivača i višedimenzioni nizovi . . . 77
Dvodimenzioni niz naspram niza pokazivača . . . 78
1.5.4 Pokazivači i strukture . . . 80
Plitko i duboko kopiranje struktura . . . 80
Strukture kao argumenti funkcija . . . 80
1.5.5 Dinamička alokacija memorije . . . 81
Funkcije standardne biblioteke za rad sa dinamičkom memorijom . . . 81
Amortizovana analiza složenosti i realokacija dinamičkih nizova . . . 87
Greške u radu sa dinamičkom memorijom . . . 89
Fragmentisanje memorije . . . 90
Hip i dinamički životni vek . . . 91
1.6 Pregled standardne biblioteke . . . 91
1.6.1 Zaglavlje `string.h` . . . 93
1.6.2 Zaglavlje `stdlib.h` . . . 94
1.6.3 Zaglavlje `ctype.h` . . . 96
1.6.4 Zaglavlje `math.h` . . . 97
1.6.5 Zaglavlje `assert.h` . . . 98
1.7 Ulaz i izlaz programa . . . 99
1.7.1 Standardni tokovi . . . 99
1.7.2 Ulaz i izlaz pojedinačnih karaktera . . . 99
Linijski izlaz . . . 100
1.7.3 Formatirani izlaz — `printf` . . . 101
Konverzioni karakteri . . . 101
1.7.4 Formatirani ulaz — `scanf` . . . 103
1.7.5 Ulaz iz niske i izlaz u nisku . . . 106
1.7.6 Ulaz iz datoteka i izlaz u datoteke . . . 107
Tekstualne i binarne datoteke . . . 107
Pristupanje datoteci . . . 107
Zatvaranje datoteka . . . 108

Ulaz i izlaz pojedinačnih karaktera . . . 108
Provera grešaka i kraja datoteke . . . 110
Linijski ulaz i izlaz . . . 110
Formatirani ulaz i izlaz . . . 111
Rad sa binarnim datotekama . . . 112
1.7.7 Argumenti komandne linije programa . . . 113

2 Neki napredni koncepti jezika C++ 117

2.1 Tipovi, zaključivanje tipova i životni vek objekata . . . 117
2.1.1 Zaključivanje tipova pomoću `auto` . . . 117
Zaključivanje tipova u funkcijama . . . 118
2.1.2 Zagrade `{}` za inicijalizaciju . . . 118
2.1.3 Liste inicijalizacije i `std::initializer_list<T>` . . . 120
2.1.4 Statički članovi klase . . . 120
2.1.5 Doseg i životni vek objekata . . . 121
2.1.6 Izuzeci i rukovanje greškama (`try–catch`) . . . 121
2.1.7 Poređenje nizova i vektora . . . 122
2.1.8 Poređenje niske i stringa . . . 123
2.1.9 Const i reference . . . 123
2.1.10 Preopterećenje operatora . . . 124
2.1.11 Funkcijski objekti (funktori) . . . 124
2.2 Dinamička alokacija memorije u C++ . . . 125
Prednost u odnosu na `malloc` i `free` . . . 126
Dinamička alokacija nizova . . . 127
Šta se dešava kada nema dovoljno memorije? . . . 128
RAII . . . 128
2.3 Pametni pokazivači . . . 129
2.3.1 Otkrivanje curenja memorije . . . 129
2.3.2 `std::unique_ptr` — pametni pokazivač sa ekskluzivnim vlasništvom . . . 131
2.3.3 `std::shared_ptr`: zajedničko vlasništvo i upravljanje vekom trajanja . . . 136
2.3.4 `std::weak_ptr`: posmatrač bez vlasništva . . . 138
2.4 Šabloni . . . 139
2.4.1 Klasa parametarskog tipa . . . 140
2.4.2 Vrednosni parametar u šablonu . . . 141
2.4.3 Dedukcija argumenata šablona (C++17) . . . 141
2.4.4 Funkcijski šabloni . . . 142
2.4.5 Alijasi tipova . . . 143

2.5 Sementika premeštanja . . . 145
2.5.1 U čemu je razlika između kopiranja i pomeranja . . . 145
2.6 Rad sa datotekama u jeziku C++ . . . 147
2.6.1 Vrste file streamova . . . 147
2.6.2 Pisanje u datoteku . . . 147
2.6.3 Dodavanje podataka (režim *append*) . . . 148
2.6.4 Čitanje iz datoteke . . . 148
2.6.5 Prenos streamova funkcijama . . . 149
2.6.6 String streamovi . . . 150
2.7 Ranges (C++20) . . . 151

3 Sekvencijalne strukture podataka 155
3.1 Jednostruko povezane liste . . . 155
Odnos povezanih lista i nizova u teoriji i praksi . . . 155
Jednostruko povezana lista u standardnoj biblioteci C++: `std::forward_list` 156
Elementi povezane liste . . . 157
Sentineli . . . 160
Osnovne operacije za rad sa povezanim listama . . . 161
3.2 Dvostruko povezane liste . . . 172
3.2.1 Dvostruko povezana lista u standardnoj biblioteci C++: `std::list` . . . 175
3.2.2 Primer primene dvostruko povezane liste: linijski editor . . . 176
3.2.3 Dvostruko povezana lista u jeziku C++ . . . 178
Struktura čvora . . . 179
Struktura liste . . . 180
Konstrukcija, destrukcija i pravila kopiranja . . . 180
Iterator dvostruko povezane liste . . . 182
Operacije u radu sa dvostruko povezanom listom . . . 186
Primer korišćenja opisane klase . . . 189
3.3 Kružne liste . . . 191
3.3.1 Primer primene strukture kružna lista: plesači . . . 192
3.3.2 Detekcija ciklusa u jednostruko povezanoj listi (Flojdov algoritam) . . . 195
3.4 Stek . . . 198
3.4.1 Implementacija steka korišćenjem niza . . . 199
3.4.2 Implementacija steka korišćenjem povezane liste . . . 201
3.4.3 Generički stek u C++: šablon zasnovan na `std::vector` . . . 204
3.4.4 Kako je organizovan `std::stack` u STL-u . . . 206
3.4.5 Stek i aritmetički izrazi . . . 207

Vrednost postfiksnog izraza . . . 207
Prevođenje potpuno zagrađenog izraza u postfiksni oblik . . . 210
Računanje vrednosti opštih aritmetičkih izraza . . . 212
3.4.6 Primer primene strukture stek: pretraga u dubinu i ispitivanje da li postoji put na mapi . . . 216
3.5 Red . . . 219
3.5.1 Implementacija reda korišćenjem niza . . . 220
3.5.2 Implementacija reda korišćenjem povezane liste . . . 223
3.5.3 Implementacija reda korišćenjem dva steka . . . 227
3.5.4 Primer primene strukture red: poslednjih k učitanih brojeva . . . 229
3.5.5 Primer primene strukture red: pretraga u širinu i pronalaženje puta na mapi . 231
3.6 Dek (red sa dva kraja) . . . 231
3.6.1 Osnovna ideja: niz segmenata . . . 231
3.6.2 Primer implementacije deka . . . 232
Dek kao "mapa" segmenata . . . 233
Inicijalizacija deka . . . 234
Dodavanje elementa . . . 236
Uklanjanje elementa sa početka i sa kraja deka . . . 241
Pristup elementu na datoj poziciji . . . 243
Oslobađanje memorije . . . 245
Primer upotrebe deka . . . 246
3.7 Hip i red sa prioritetom . . . 247
3.7.1 Osnovna svojstva hipa . . . 248
3.7.2 Primer implementacije maks-hipa . . . 249
Struktura podataka za maks-hip . . . 250
Indeksne relacije u implicitnoj reprezentaciji . . . 250
Inicijalizacija hipa . . . 250
Čišćenje i uništavanje hipa . . . 251
Osnovne informativne operacije . . . 251
Umetanje elementa u maks-hip . . . 251
Brisanje elementa iz maks-hipa . . . 254
3.7.3 Hipsort . . . 256
Formiranje hipa naniže (Vilijamsov metod) . . . 256
Formiranje hipa naviše (Flojdov metod) . . . 257
Završna faza: sortiranje pomoću hipa . . . 260

4 Asocijativne strukture podataka 263

4.1 Binarna stabla . . . 263

4.1.1 Uređeno binarno stablo . . . 264

Dodavanje elementa u uređeno binarno stablo . . . 265

Pronalaženje elementa u uređenom binarnom stablu . . . 268

Obilazak binarnog stabla . . . 268

Brisanje stabla . . . 270

Obilazak binarnog stabla u širinu . . . 270

Generičko uređeno binarno stablo . . . 271

4.2 Balansirana binarna stabla . . . 275

Umetanje elementa u AVL stablo . . . 280

C implementacija: rotacije + umetanje (AVL) . . . 284

Brisanje elementa iz AVL drveta . . . 287

Primene AVL drveta . . . 290

4.3 Heširanje i heš-tabela . . . 291

4.3.1 Heš-funkcije . . . 293

Uloga heš-funkcije i osnovni pojmovi . . . 293

Kolizije kao neizbežna pojava . . . 293

Izbor heš-funkcije . . . 295

Množenje i šiftovanje (engl. *multiply–shift*) . . . 297

Heširanje složenih ključeva (stringovi, nizovi) . . . 298

Zaključak: svojstva dobre heš-funkcije . . . 300

4.3.2 Razrešavanje kolizija . . . 301

Odvojeno ulančavanje . . . 301

Otvoreno adresiranje . . . 307

4.3.3 Veza sa C++ standardnom bibliotekom: `unordered_set` i `unordered_map` 313

Podržane operacije i složenosti . . . 313

Faktor opterećenja i proširivanje u C++ . . . 314

5 Napredne strukture podataka 315

5.1 Skip-lista . . . 315

5.2 B-stablo . . . 315

Predgovor

Glava 1

Programski jezik C

1.1 Odnos jezika C i C++

Jezici **C** i **C++** usko su povezani i dele zajedničke korene. C++ je nastao kao proširenje jezika C, sa ciljem da obezbedi podršku za objektno orijentisano programiranje i dodatne mehanizme apstrakcije. Ipak, iako su C i C++ sintaksički veoma slični, oni danas predstavljaju **dva različita jezika** — kako po filozofiji, tako i po praktičnim aspektima upotrebe.

U ovom kursu polazimo od pretpostavke da su studenti već upoznati sa osnovama programiranja u jeziku C++. U narednim poglavljima biće predstavljene karakteristike jezika C, sa posebnim naglaskom na njegove koncepte koji čine osnovu efikasne implementacije struktura podataka. Cilj ovog poglavlja je da ukaže na najvažnije razlike između C i C++ jezika, koje će se kasnije detaljno razmatrati.

C je minimalistički proceduralni jezik, projektovan da pruži **direktnu kontrolu nad memorijom i resursima sistema**. Njegov osnovni cilj je efikasnost — kako vremenska, tako i prostorna.

C++ zadržava sve osobine jezika C, ali uvodi složeniji model programiranja koji obuhvata **objektno orijentisane** i **generičke** koncepte. C++ programer često razmišlja u terminima apstrakcija (klasa, šablona, operatora), dok C programer razmišlja u terminima **podataka, memorije i funkcija**.

U programskom jeziku C programer eksplicitno upravlja svakim aspektom toka podataka i alokacije memorije. Takav pristup zahteva veću pažnju i disciplinu, ali pruža dublje razumevanje načina na koji računar izvršava program.

Tipovi podataka i deklaracije

C koristi osnovne i izvedene tipove podataka: cele brojeve (`int`), realne brojeve (`float`, `double`), karaktere (`char`), pokazivače, nizove, strukture i unije. C++ zadržava sve ove tipove, ali uvodi i dodatne mehanizme apstrakcije — **reference**, **klase**, **šablone**, kao i **nasleđivanje** i **enkapsulaciju**.

Strukture u programskom jeziku C su jednostavni skupovi podataka bez funkcija i konstrukora. U jeziku C++ strukture i klase dele isti model, pa strukture mogu imati metode, pristupne modifikatore i konstruktore. U jeziku C se strukturama može pristupati direktno, a često se koriste u kombinaciji sa pokazivačima. Unije u jeziku C imaju sličnu sintaksu kao strukture, ali omogućavaju da više polja dele istu memorijsku lokaciju, čime se štedi prostor — koncept koji se u jeziku C++ koristi ređe.

U jeziku C **ne postoje reference**. Takav efekat mora biti eksplicitno postignut prosleđivanjem adrese, odnosno pokazivača. Ova razlika je suštinska — C ne uvodi novi jezički nivo apstrakcije

iznad pokazivača, već programeru prepušta punu kontrolu nad adresama i memorijom. Razumevanje pokazivača, njihovog odnosa sa nizovima i pristupa elementima predstavlja osnovu daljeg rada u C-u i biće detaljno razmotreno u posebnom poglavlju.

Vredi napomenuti i da su tipovi u C-u manje strogi nego u C++. C dozvoljava implicitne konverzije između različitih tipova, kao i određene aritmetičke operacije nad pokazivačima i celobrojnim vrednostima, što u jeziku C++- često izaziva grešku pri prevođenju.

Ulazno–izlazne operacije

U programskom jeziku C se ulazno–izlazne operacije obavljaju pomoću **biblioteke** `<stdio.h>`, kroz funkcije kao što su `printf`, `scanf`, `fopen`, `fread` i druge. C++ koristi **tokove (streams)** i zaglavlja poput `<iostream>`, sa operatorima << i >>.

U narednim poglavljima biće prikazano kako funkcije standardne biblioteke jezika C omogućavaju preciznu kontrolu nad formatom, tokovima i datotekama.

Rad sa memorijom

C ne poseduje automatsko upravljanje memorijom niti klase koje enkapsuliraju dinamičke objekte. Umesto toga, koristi funkcije `malloc`, `calloc`, `realloc` i `free` iz biblioteke `<stdlib.h>`. U jeziku C++ se koriste operatori `new` i `delete`, koji su bezbedniji i integrisani sa konstruktorima i destruktorima.

U programskom jeziku C je programer odgovoran za svaku alokaciju i oslobađanje memorije, što često predstavlja izvor grešaka, ali i daje potpunu kontrolu nad performansama.

Funkcije i pokazivači na funkcije

C je stariji jezik i ne poznaje koncepte poput **preopterećenja funkcija (eng. function overloading)** niti ključnu reč inline. U jeziku C ne mogu postojati dve funkcije sa istim imenom, čak ni ako se razlikuju po broju ili tipu parametara — svaka funkcija mora imati jedinstveno ime. Kasniji razvoj jezika C++ uveo je preopterećenje funkcija kao deo sistema tipova, čime kompajler može izabrati odgovarajuću funkciju na osnovu liste argumenata.

U jeziku C, međutim, programer sličnu fleksibilnost može postići drugačijim pristupom — pomoću **pokazivača na funkcije**. Ovaj mehanizam omogućava da se funkcije prosleđuju kao argumenti drugim funkcijama, čuvaju u strukturama podataka ili dinamički biraju tokom izvršavanja programa. Takav pristup, iako jednostavniji od polimorfizma jezika C++, daje veliku izražajnost i predstavlja osnovu mnogih naprednih konstrukcija koje se u jeziku C zasnivaju na eksplicitnom upravljanju funkcijama i podacima.

Standardna biblioteka

C raspolaže relativno malim, ali efikasnim skupom biblioteka: `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<math.h>`, `<assert.h>`, i dr. C++ standardna biblioteka obuhvata iste funkcionalnosti, ali i dodatne module poput **STL** (Standard Template Library), sa klasama kao što su `vector`, `string`, `map` i dr.

U C-u se svi ovi koncepti mogu izgraditi ručno — i upravo to omogućava bolje razumevanje njihovog funkcionisanja.

Prevođenje i organizacija programa

U jeziku C program se najčešće sastoji od više izvornih datoteka koje se zasebno prevode, a zatim povezuju (*linkuju*) u izvršivu datoteku. Iako i C++ koristi sličan proces, u jeziku C je taj postupak vidljiviji i eksplicitniji: programer mora sam da uključuje zaglavlja (`.h` datoteke), vodi računa o prototipima funkcija i koristi preprocesorske direktive (`#define`, `#include`, `#ifdef`, `#ifndef`) radi upravljanja zavisnostima i ponovnog korišćenja koda.

Za razliku od jezika C, C++ se u većoj meri oslanja se na **prostore imena** (*namespaces*) za organizaciju koda i sprečavanje kolizija među imenima. U jeziku C takav mehanizam ne postoji: svi globalni simboli (funkcije i promenljive) dele isti imenski prostor, pa se njihova razdvojenost postiže konvencijama u imenovanju i korišćenjem modifikatora `static` i `extern`, koji određuju vidljivost simbola tokom povezivanja.

Bezbednost i tipizacija

C poseduje **slabu statičku tipizaciju**, što znači da kompajler često dozvoljava konverzije između nepovezanih tipova (npr. pokazivača i celih brojeva). C++ uvodi striktniju kontrolu tipova i više mehanizama zaštite (kao što su reference i `const` ispravnost). Razumevanje kako i zašto C dopušta određene konverzije biće važno za razumevanje ponašanja programa na niskom nivou.

Zaključak

Iako C++ obuhvata C, **učenje C-a nije korak unazad**, već povratak ka osnovama. Programer koji poznaje C++ zna kako da koristi apstrakcije; programer koji poznaje C razume **kako su te apstrakcije izgrađene**. U tom smislu, C predstavlja nezaobilazan temelj svakog ozbiljnog razumevanja računarskih sistema i struktura podataka.

1.2 Reprezentacija osnovnih tipova

U ovoj glavi dajemo sažet pregled *osnovnih tipova podataka u jeziku C*, sa naglaskom na one aspekte koji se razlikuju od C++ i koji su važni za razumevanje daljeg teksta. Podrazumevamo da je čitalac upoznat sa osnovnim tipovima podataka u jeziku C++ i njihovom reprezentacijom.

> **Napomena (C naspram C++):** C i C++ dele veliki deo leksičke i semantičke baze, ali **standard C ne fiksira veličine tipova** (već samo minimume i međusobne odnose), **formati u `printf/scanf` moraju tačno odgovarati tipovima**, a pojedine naizgled sitne razlike (npr. tip karakter-konstante ili označenost/neoznačenost `char`) imaju semantičke posledice. Više o ovome u nastavku.

1.2.1 Celobrojni tipovi

Osnovni celobrojni tip u C je `int`. Standard ne propisuje tačan broj bitova, već samo odnose i minimalne širine (npr. `sizeof(short)` $\le$ `sizeof(int)` $\le$ `sizeof(long)`, pri čemu je `short` najmanje 16 bita, a `long` najmanje 32 bita). Praktično, veličine zavise od platforme i kompajlera.

Tipovi se proširuju kvalifikatorima:

- **Širina:** `short`, `long`, `long long`
- **Znak:** `signed`, `unsigned`

Minimalne garancije:

- `short` 16 bita, `long` 32 bita, `long long` 64 bita;
- `signed` označava reprezentaciju sa znakom (u praksi potpuni komplement), `unsigned` bez znaka.

Za prenosivost pri radu sa binarnim formatima koriste se **tipovi fiksnih širina** iz `<stdint.h>`: `int8_t`, `uint16_t`, `uint32_t`, `uint64_t`, itd. Za granice opsega koriste se konstante iz `<limits.h>` (npr. `INT_MAX`, `LLONG_MIN`).

```c
#include <stdio.h>
#include <limits.h>
#include <stdint.h>

int main(void) {
    printf("sizeof(int) = %zu, INT_MIN = %d, INT_MAX = %d\n",
           sizeof(int), INT_MIN, INT_MAX);
    printf("sizeof(uint32_t) = %zu\n", sizeof(uint32_t));
    return 0;
}
```

1.2.2 Realni tipovi (pokretni zarez)

Tipovi `float`, `double`, `long double` opisuju brojeve u pokretnom zarezu osnovne, dvostruke i proširene tačnosti. Standard C ne fiksira broj bitova (u praksi se sledi IEEE-754). Granice i parametri tačnosti dostupni su u `<float.h>` (npr. `DBL_MAX`, `FLT_EPSILON`).

```c
#include <stdio.h>
#include <float.h>

int main(void) {
    printf("DBL_MAX = %.17e, DBL_EPSILON = %.17e\n", DBL_MAX, DBL_EPSILON);
    return 0;
}
```

1.2.3 Logički tip

U C se `bool` uvodi zaglavljem `<stdbool.h>` (`true`/`false`). U C++ je ugrađen bez posebnog zaglavlja.

1.2.4 Označenost tipa `char`

Pri korišćenju `char` u C postoje tri relevantne varijante:

- `signed char` — uobičajen opseg: -128 … +127
- `unsigned char` — opseg: 0 … 255
- `char` — može biti ili `signed` ili `unsigned`, zavisi od kompajlera/arhitekture (standard to prepušta implementaciji)

Dakle, isti izvorni kod može imati drugačiji rezultat na različitim platformama ako se oslanja na to da je `char` negativan ili nenegativan.

```c
#include <stdio.h>
#include <limits.h>

int main(void) {
    printf("CHAR_MIN=%d, CHAR_MAX=%d\n", CHAR_MIN, CHAR_MAX);
    return 0;
}
```

Ako je `CHAR_MIN < 0`, tada je `char` označen (`signed`); u suprotnom je neoznačen (`unsigned`). Radi prenosivosti, preporuka je **eksplicitno** koristiti `signed char` ili `unsigned char` kada `char` ne sadrži „obične“ štampajuće znakove.

1.2.5 Karakter-konstante

U C `'A'` **ima tip** `int` (a ne `char`, kao u C++). To je važno za `sizeof` i u konverzijama.

```c
#include <stdio.h>
#include <stdbool.h>

int main(void) {
    char c = 'A';
    bool upper = (c >= 'A' && c <= 'Z');
    printf("'A' kao int = %d; sizeof('A') u C = %zu; upper = %d\n",
           'A', sizeof('A'), upper);
    return 0;
}
```

1.2.6 Formati za ulaz/izlaz

Zbog odsustva provere tipova pri pozivu funkcija sa promenljivim brojem argumenata, **specifikator formata mora odgovarati tačnom tipu argumenta**:

- `int` → `%d`, `unsigned` → `%u`, `long` → `%ld`, `long long` → `%lld`;
- „Uži“ celi tipovi (`char`, `short`) se u `printf` promovišu u `int` → koristi se `%d`

- Prilikom poziva `printf` funkcije: `float` se promoviše u `double` → za oba se koristi `%f`;
- Prilikom poziva `scanf` funkcije: `double` → `%lf`, `float` → `%f`
- `void*` → `%p` (standardni način za ispis adrese)

```c
#include <stdio.h>

int main(void){
    short s = -12; unsigned u = 4000000000u; double z = 0.1;
    printf("s=%d, u=%u, z=%.17g\n", s, u, z);
    return 0;
}
```

1.2.7 Tip `void`

Tip `void` označava prazan skup vrednosti i koristi se:

- kao povratni tip funkcija bez rezultata (`void f(void)`),
- kao „generički“ pokazivač `void*` - može da adresira bilo koji objekat, ali se mora eksplicitno kastovati pre dereferenciranja (o ovome će biti reči kasnije),
- kao signalizacija „ništa“ u kontekstima gde je potreban tip (npr. stari stil deklarisanja bez parametara).

1.2.8 Konverzije tipova

Konverzija tipova predstavlja pretvaranje vrednosti jednog tipa u vrednost drugog tipa. C je fleksibilan: mnoge konverzije se obavljaju implicitno. Ipak, konverzije mogu dovesti do gubitka informacija ili promenjene interpretacije podataka. Jezik je statički tipiziran: tip promenljive zadržava se bez obzira na konverzije vrednosti izraza.

Uopšteno, postoje dve vrste konverzija:

- **Eksplicitne konverzije (kastovanja):** programer ih zahteva pomoću operatora kastovanja.
- **Implicitne konverzije:** prevodilac ih obavlja automatski (npr. pri dodeli, aritmetičkim operacijama, promocijama).

Neke konverzije su bez gubitka (npr. iz užeg tipa u širi), a neke nisu (npr. `double` → `int`).

Primer „promocije“ (u širi tip):

```c
float a = 4;  /* 4 je int i implicitno se konvertuje u float */
```

Ipak, može doći do gubitka informacija zbog ograničene preciznosti:

```
float f = 16777217;
printf("%f\n", f);
```

Na tipičnim sistemima (IEEE-754, 32-bitni `float`) ispis je `16777216.000000`, jer `16777217 = 2^24 + 1` ne može biti tačno predstavljen u 24 bita mantise.

Primer „democije“ (u uži tip):

```
int b = 7.0f;   /* 7.0f → 7 */
int c = 7.7f;   /* 7.7f → 7 (odseca se, ne zaokružuje na 8) */
unsigned char d = 256; /* d dobija vrednost 0 (mod 256) */
```

Pri prelazu između celih i pokretnog zareza menja se čitava interna reprezentacija (npr. IEEE-754 potpuni komplement). Dok, kod proširivanja celih tipova:

- iz neoznačenih u širi → dopisivanje vodećih nula (eng. *zero extension*),
- iz označenih u širi → proširivanje znakovnog bita (eng. *sign extension*).

Eksplicitne konverzije (kastovanja). Sintaksa:

```
(tip) izraz
```

Operator kastovanja je prefiksni, unarni i **više prioritetan** od svih binarnih operatora. Kastovanje menja vrednost izraza, ali **ne menja tip promenljive**.

Primer 1.2.1. *Želimo deljenje u pokretnom zarezu, iako su operandi celi:*

```
int a = 13, b = 4;
printf("%d\t", a/b);
printf("%f\n", (double)a/(double)b);  /* 3       3.250000 */
```

*Dovoljno je kastovati **jedan** operand u `double`; drugi će se implicitno konvertovati.*

Implicitne konverzije u izrazima. Pri dodeli, vrednost desne strane se (po potrebi) implicitno konvertuje u tip leve strane:

```
int a;
double b = (a = 3.5);  /* a = 3; b = 3.0 */
```

Aritmetički operatori se ne primenjuju nad „malim“ celim tipovima (`char`, `short`) direktno — pre primene operacije vrednosti se promovišu u `int`.

```c
unsigned char cresult, c1, c2, c3;
c1 = 100; c2 = 3; c3 = 4;
cresult = c1 * c2 / c3;  /* c1,c2,c3 → int; rezultat potom u char */
```

Bez promocije, `c1 * c2` (= 300) bi prekoracio `char`.

Pravila konverzije (sažeto):

1. Ako je bar jedan operand `long double` → drugi se konvertuje u `long double`.
2. Inače, ako je jedan operand `double` → drugi operand se konvertuje u `double`.
3. Inače, ako je jedan operand `float` → drugi operand se konvertuje u `float`.
4. Inače, `char`/`short` se promovišu u `int`.
5. Ako je jedan operand `long long` → drugi operand se konvertuje u `long long`.
6. Inače, ako je jedan operand `long` → drugi operand se konvertuje u `long`.

Kod mešanja označenih i neoznačenih tipova pravila zavise od mogućnosti predstavljanja:

- Ako je **neoznačeni operand širi**, označeni se konvertuje u taj neoznačeni širi tip.
- Inače, ako **označeni može predstaviti sve vrednosti neoznačenog**, neoznačeni se prevodi u označeni širi tip.
- Inače, oba se konvertuju u **neoznačeni tip** koji odgovara označenom operandu.

Tipični primeri poređenja:

- Ako `int` ima 16 bita, a `long` 32 bita, tada je `-1l < 1u` (oba u `signed long`).
- Ali `-1l > 1ul` (oba u `unsigned long`; `-1l` postaje `ULONG_MAX`).

Prenosivo i bezbedno poređenje označenog i neoznačenog:

```c
signed   si = /* neka vrednost */;
unsigned ui = /* neka vrednost */;

/* if (si < ui) - nije uvek korektno */
if (si < 0 || (unsigned)si < ui) {
    /* ... */
}
```

1.2.9 Bitovske operacije

C podržava naredne operatore za rad nad pojedinačnim bitovima, koji se mogu primenjivati samo na celobrojne argumente.

Operator	Značenje
~	bitovska negacija
&	bitovsko i

Operator	Značenje
|	bitovsko ili
^	bitovsko ekskluzivno ili
<<	pomeranje ulevo
>>	pomeranje udesno

- `&` – **bitovsko *i*** — primenom ovog operatora vrši se konjunkcija pojedinačnih bitova dva navedena argumenta (*i*-ti bit rezultata predstavlja konjunkciju *i*-tih bitova argumenata). Na primer, neka su promenljive `x1` i `x2` tipa `unsigned char`.

Vrednost	Dekadno	Binarno	Heks
`x1`	74	01001010	0x4A
`x2`	87	01010111	0x57
`x1 & x2`	66	01000010	0x42
`x1 | x2`	95	01011111	0x5F
`x1 ^ x2`	29	00011101	0x1D

  S obzirom na to da se prevođenje u binarni sistem efikasnije sprovodi iz heksadekadnog sistema nego iz dekadnog, prilikom korišćenja bitovskih operatora konstante se obično zapisuju heksadekadno.

- `|` – **bitovsko *ili*** — primenom ovog operatora vrši se (obična) disjunkcija pojedinačnih bitova dva navedena argumenta.

- `^` – **bitovsko *ekskluzivno ili*** — primenom ovog operatora vrši se ekskluzivna disjunkcija pojedinačnih bitova dva navedena argumenta
  (vrednost ekskluzivne disjunkcije dva bita je `1` ako i samo ako ta dva bita imaju različite vrednosti).

- `~` – **jedinični komplement** — primenom ovog operatora dobija se vrednost u kojoj su svi bitovi argumenta komplementirani (invertovani), a sam argument ostaje nepromenjen. Na primer, vrednost izraza `~x1` u tekućem primeru je `10110101`, tj. `0xB5`.

- `<<` – **pomeranje ulevo (šiftovanje)** — primenom ovog operatora dobija se vrednost u kojoj su bitovi prvog argumenta pomereni ulevo za broj pozicija naveden kao drugi argument (a argumenti operatora se ne menjaju).
  Početni bitovi (sleva) prvog argumenta se zanemaruju, dok su na završnim mestima rezultata nule. Pomeranje ulevo broja za jednu poziciju odgovara množenju sa dva (osim kada dođe do prekoračenja). Na primer, ukoliko promenljiva `x` ima tip `unsigned char` i vrednost `01010101`, tj. `0x55`, tj. 85, vrednost izraza `x << 1` jednaka je `10101010`, tj. `0xAA`, tj. 170.

- `>>` – **pomeranje udesno (šiftovanje)** — primenom ovog operatora dobija se vrednost u kojoj su bitovi prvog argumenta pomereni udesno za broj pozicija naveden kao drugi argument (a argumenti operatora se ne menjaju).
  U izrazu `x >> n`, poslednjih `n` bitova vrednosti `x` (bitovi najmanje težine) se zanemaruje. Početni bitovi rezultata `x >> n` zavise od tipa i vrednosti promenljive `x`. Ako `x` ima neoznačen tip ili ima označen tip a nenegativnu vrednost, onda izraz `x >> n` ima vrednost dobijenu pomeranjem bitova `x` udesno za `n` pozicija, pri čemu je početnih `n` bitova rezultata jednako

nuli (to je tzv. *logičko pomeranje*). U ovom slučaju, vrednost `x >> n` jednaka je vrednosti celobrojnog deljenja `x` sa `2^n`. Na primer, ukoliko je tip promenljive `x` `unsigned char` a vrednost 10010100, tj. 0x94, tj. 148, tada je vrednost izraza `x >> 1` broj 01001010, tj. 0x4A, tj. 74. Standard ne propisuje vrednost izraza `x >> n` za slučaj da `x` ima označen tip i negativnu vrednost, a većina kompajlera u ovakvim situacijama vrši tzv. *aritmetičko pomeranje* u kojem je početnih `n` bitova rezultata jednako početnom bitu vrednosti `x`.

- `&=`, `|=`, `^=`, `<<=`, `>>=` – **bitovske dodele** — ovi operatori kombinuju bitovske operatore sa operatorom dodele. Na primer, `a &= 1` ima isto značenje kao `a = a & 1`.

Kao unarni operator, operator ~ ima najviši prioritet i desno je asocijativan.
Prioritet operatora pomeranja najviši je od svih binarnih bitovskih operatora — nalazi se između prioriteta aritmetičkih i relacijskih operatora.

Ostali bitovski operatori imaju prioritet između relacijskih i logičkih operatora.
Operator & ima viši prioritet od ^ koji ima viši prioritet od |.
Ovi operatori imaju levu asocijativnost.

Bitovski operatori dodele imaju niži prioritet (jednak ostalim operatorima dodele) i desnu asocijativnost.

Pregled prioriteta ooperatora: | Prioritet (viši → niži) | Operatori | Asocijativnost | |———————————|————————————|————————-| | Najviši (ovde bitovski) | ~ (bitovski NOT, unarni) | desna | | Aritmetički | *, /, %, +, - | leva | | Pomeranje | <<, >> | leva | | Relacioni | <, <=, >, >= | leva | | Relacioni | ==, != | leva | | Bitovski AND | & | leva | | Bitovski XOR | ^ | leva | | Bitovski OR | \| | leva | | Logički AND | && | leva | | Logički OR | \|\| | leva | | Dodele | =, &=, \|=, ^=, <<=, >>= | desna |

Ponašanje i uloga bitovskih operatora ne smeju se mešati sa ponašanjem i ulogom logičkih operatora. Na primer, vrednost izraza `1 && 2` jednaka je 1 (tačno i tačno je tačno), dok je vrednost izraza `1 & 2` jednaka 0 (`000...001 & 000..010 = 000...000`).

Algoritmi zasnovani na bitovskim operatorima

U nekim slučajevima, efikasnija rešenja mogu se dobiti korišćenjem pojedinačnih bitova u zapisu celobrojnih izraza.

Maskiranje.
Kako bi se postigao željeni efekat nad bitovima datog broja, često se vrši njegovo kombinovanje bitovskim operatorima sa specijalno pripremljenim konstantama koje se obično nazivaju *maske*. U nastavku će biti prikazana upotreba nekih od najčešće korišćenih operacija ovog oblika.

```
x:           00010010 00110100 10101011 11001101
maska:       00000000 00000000 00000000 11111111
-----------------------------------------------
x & maska:   00000000 00000000 00000000 11001101
```

Bitovska konjunkcija ima osobinu da je za svaki bit `b`, vrednost `b & 0` jednaka 0, dok je vrednost `b & 1` jednaka `b`. Dakle, konjunkcijom broja `x` sa nekom maskom dobija se broj u kojem su očuvane sve cifre broja `x` na pozicijama na kojima maska ima bit 1, a na svim drugim pozicijama su cifre 0. Na primer, konjunkcijom sa maskom 0xFF dobija se poslednji bajt broja. Pretposlednji bajt može se izdvojiti izračunavanjem `(x >> 8) & 0xFF`. Konjunkcijom sa maskom 0x01 dobija se poslednji bit

broja. Da li je bit na poziciji `i` broja `x` jednak 1 može se ispitati naredbom `if (x & maska)` gde je vrednost `maska` kreirana tako da ima jedan jedini bit postavljen na 1 i to na poziciji `i` (na primer, `if (x & (1 << i))`). Često je potrebno postaviti bit na nekoj poziciji na 0, što se takođe može realizivati sa `x & ~(1 << i)`, na primer želimo da postavimo na 0 bit na poziciji 2:

```
   10110110   (x = 0xB6)
&  11111011   (~(1<<2))
------------
   10110010   (rezultat = 0xB2)
```

Osobina disjunkcije je da je za svaki bit `b`, vrednost `b | 0` jednaka `b`, dok je vrednost `b | 1` jednaka 1. Dakle, disjunkcijom broja `x` sa nekom maskom dobija se rezultat u kojem su jedinice upisane na sve one pozicije na kojima maska ima bit 1 (a ostali bitovi ostaju neizmenjeni). Na primer, disjunkcijom sa maskom `0xFF` upisuju se jedinice u poslednji bajt broja.

Kombinacijom prethodnih operacija maskiranja, moguće je umetati određene nizove bitova u dati broj. Tako se na primer, izrazom `(x & ~0xFF) | 0xAB` upisuje 8 bitova `10101011` u poslednji bajt broja `x` (konjunkcijom se čiste postojeći bitovi, da bi se disjunkcijom upisao novi sadržaj).

Osobina ekskluzivne disjunkcije je da za svaki bit `b`, vrednost `b ^ 0` jednaka `b`, dok je vrednost `b ^ 1` jednaka `~b`. Ovo znači da se ekskluzivnom disjunkcijom sa nekom maskom dobija rezultat koji je jednak broju koji se dobije kada se invertuju bitovi na onim pozicijama na kojima se u maski nalazi jedinica. Na primer, ekskluzivna konjunkcija sa maskom `0xFF` invertuje poslednji bajt broja. Zanimljivo svojstvo ekskluzivne disjunkcije je da primenjena na vrednosti `x ^ maska` i `maska` daje vrednost `x`. To svojstvo omogućava, između ostalog, šifrovanje podataka: ako (samo) pošiljalac i primalac znaju šifru `maska`, pošiljalac može da šifruje svoju poruku `x` vrednošću `x ^ maska` i pošalje je primaocu; primalac, s druge strane, od vrednosti `x ^ maska` računa `(x ^ maska) ^ maska`, a to je upravo polazna poruka `x`.

Ispisivanje bitova broja.
Naredna funkcija ispisuje bitove datog celog broja `x`. Vrednost bita na poziciji `i` je 0 ako i samo ako se pri konjunkciji broja `x` sa maskom `000..010....000` (sve nule osim jedinice na poziciji `i`) dobija 0. Funkcija kreće od pozicije najveće težine kreirajući masku koja ima jednu jedinicu i to na mestu najveće težine i zatim pomera ovu masku za jedno mesto udesno u svakoj sledećoj iteraciji sve dok maska ne postane 0. Primetimo da je ovde bitno da maska ima tip `unsigned` kako bi se vršilo logičko pomeranje.

```c
void print_bits(int x)
{
  unsigned mask = 1 << (sizeof(x) * 8 - 1);
  while (mask) {
    putchar(x & mask ? '1' : '0');
    mask >>= 1;
  }
}
```

Imati na umu da će izlaz zavisiti od širine tipa `int` koji na različitim sistemima može zauzimati različit broj bajtova.

Brojanje bitova.
Prikažimo nekoliko različitih implementacija funkcije koja izračunava koliko ima bitova sa vrednošću

1 u zadatom broju n. Početne implementacije će biti slične i jednostavne, a kasnije nešto naprednije i sa boljom složenošću.

Osnovna ideja je da se razmatra jedan po jedan bit i da se brojač uveća kada god tekući bit ima vrednost 1. Bit na poziciji i može se izdvojiti tako što se izvrši bitovska konjunkcija sa maskom koja ima sve nule osim jedinice na poziciji i (ona se može dobiti tako što broj 1 šiftuje za odgovarajući broj mesta ulevo).

```
int bit_count(int n)
{
  int i;
  int count = 0;
  for (i = 0; i < 8 * sizeof(n); i++)
    if (n & (1 << i))
      count++;
  return count;
}
```

Druga mogućnost je da se uvek ispituje poslednji (krajnji desni) bit broja, a da se u svakoj iteraciji broj pomera za jedno mesto udesno.

```
int bit_count(int n)
{
  int i;
  int count = 0;
  for (i = 0; i < 8 * sizeof(n); i++, n >>= 1)
    if (n & 1)
      count++;
}
```

Ukoliko je broj neoznačen, pošto se vrši logičko šiftovanje i na početne pozicije broja sleva upisuju se nule, nije potrebno vršiti sve iteracije već je moguće zaustaviti se u trenutku kada broj postane nula.

```
int bit_count(unsigned n)
{
  int count = 0;
  while (n) {
    if (n & 1) count++;
    n >>= 1;
  }
  return count;
}
```

Naglasimo da je moguće i malo drugačije rešenje, u kojem bi se umesto šiftovanja broja udesno u svakom koraku maska šiftovala ulevo (u takvom rešenju bilo bi potrebno ispitati sve bitove).

Ukoliko broj n nije nula, izraz `n & (n-1)` invertuje poslednju jedinicu u njegovom binarnom zapisu (ostale cifre ostaju nepromenjene). Zaista, ukoliko je broj n oblika ...10000, u slučaju potpunog

komplementa broj n-1 je oblika ...01111 pa je njihova konjunkcija oblika ...00000, pri čemu je prefiks (označen sa ...) nepromenjen. Imajući ovo u vidu, moguće je implementirati prethodnu funkciju tako da broj iteracija bude jednak broju jedinica u broju n.

```
int bit_count(unsigned n)
{
  int count = 0;
  while (n) {
    count++;
    n = n & (n-1);
  }
  return count;
}
```

Ovo je moguće iskoristiti kako bi se ispitalo da li je broj stepen broja dva. Naime, broj je stepen broja dva ako i samo ako ima tačno jednu jedinicu u binarnom zapisu (specijalan slučaj nule se mora posebno obraditi).

```
int is_pow2(unsigned n)
{
  return n != 0 && !(n & (n - 1));
}
```

Asimptotski brže rešenje za problem brojanja bitova moguće je dobiti ako se primeni takozvano paralelno brojanje bitova. Taj algoritam zasnovan je na algoritamskoj tehnici *podeli i vladaj (engl.~divide and conquer)*. Broj jedinica u binarnom zapisu 32-bitnog broja može se svesti na sabiranje broja jedinica u binarnom zapisu dve njegove 16-bitne polovine, broj jedinica u svakoj 16-bitnoj polovini na sabiranje broja jedinica u svakoj od njene dve 8-bitne polovine i tako dalje. „Trik“ koji se koristi je da se broj jedinica u svakoj od dve polovine broja može zapisati umesto samih bitova te polovine broja. Naime, pomoću k bitova neoznačenog broja mogu se zapisati brojevi od 0 do $2^k - 1$, a za svako $k \geq 1$ važi da je $2^k - 1 \geq k$. Nakon što prebrojimo jedinice u svakoj od dve 16-bitne polovine broja i zapišemo te brojeve u svaku od te dve 16-bitne polovine, preostaje da saberemo neoznačene brojeve koji su zapisani u te dve 16-bitne polovine. To možemo uraditi tako što levu polovinu šiftujemo za 16 mesta udesno i saberemo sa desnom polovinom. Postupak se nastavlja rekurzivno. Izlaz iz rekurzije predstavlja zadatak u kom je potrebno izračunati broj jedinica u svakom od tridest i dva 1-bitna bloka i zapisati rezultat u taj blok. Međutim, u tom slučaju ne treba raditi zapravo ništa jer bitovi koji imaju vrednost 0 imaju zapravo nula jedinica, a blokovi koji imaju vrednost 1 imaju zapravo jednu jedinicu, pa je na svakoj bitovskoj poziciji upravo zapisan broj jedinica na toj bitovskoj poziciji.

Algoritam se može realizovati i iterativno, eliminisanjem rekurzije. Ideja je razmatrati tridesetdvobitni broj kao niz od 16 uzastopnih parova jednobitnih brojeva i zatim sabiranjem svakog para dobiti 16 uzastopnih parova dvobitnih brojeva koji sadrže njihove sume. Ovo je moguće uraditi izdvajanjem bitova na neparnim pozicijama (konjunkcijom sa maskom 0x55555555), zatim pomeranjem broja za jedno mesto udesno i ponovnim izdvajanjem bitova na neparnim pozicijama (ono što se dobije su bitovi na parnim pozicijama polaznog broja pomereni za jednu poziciju desno) i zatim primenom operatora sabiranja. Na primer, ukoliko je polazni broj

01011011001010100110110101011011,

sabiraju se brojevi

01 01 00 01 00 00 00 00 01 00 01 01 01 01 00 01 i

00 00 01 01 00 01 01 01 00 01 01 00 00 00 01 01.

Dobija se rezultat

01 01 01 10 00 01 01 01 01 01 10 01 01 01 01 10

Rezultat ukazuje da se u svakom od prva tri para bitova polaznog broja nalazi po jedna jedinica, da se u sledećem paru nalaze dve jedinice, zatim ide par sa nula jedinica itd.

Prilikom ovog sabiranja „paralelno“ se sabiraju svi parovi pojedinačnih bitova (kako je svaki bit na parnim pozicijama argumenata jednak 0, sabiranja pojedinih parova su nezavisna u smislu da ne može da nastupi prekoračenje koje bi se sa jednog para proširilo na sledeći).

Ovaj postupak dalje se nastavlja tako što se sabira 8 parova dvobitnih brojeva i dobija 8 četvorobitnih brojeva koji čuvaju brojeve jedinica u svakoj četvorki bitova polaznog broja. U prethodnom primeru se dobija

0010 0011 0001 0010 0010 0011 0010 0011

što ukazuje na to da prva četvorka bitova polaznog broja ima dve jedinice, druga tri jedinice, treća jednu, itd.

Zatim se ova 4 para četvorobitnih brojeva sabiraju i dobijaju se četiri osmobitna broja koja sadrže brojeve jedinica u pojedinačnim bajtovima polaznog broja. U prethodnom primeru se dobija

00000101 00000011 00000101 00000101

što govori da prvi bajt ima 5 jedinica, drugi 3, treći 5 i četvrti 5.

Zatim se ova dva para osmobitnih brojeva sabiraju i dobijaju se dva šesnaestobitna broja koji sadrže brojeve jedinica u dvobajtima polaznog broja. U prethodnom primeru dobija se vrednost

0000000000001000 0000000000001010

što govori da prvi dvobajt ima 8 jedinica, a drugi 10.

Napokon, sabira se i ovaj par šesnaestobitnih brojeva i dobija se tridesetdvobitni broj koji sadrži konačan rezultat.

00000000000000000000000000010010

Prethodni postupak implementira funkcija:

```
int bit_count(unsigned n)
{
   n  = (n & 0x55555555) + ((n >> 1) & 0x55555555);
   n  = (n & 0x33333333) + ((n >> 2) & 0x33333333);
   n  = (n & 0x0F0F0F0F) + ((n >> 4) & 0x0F0F0F0F);
   n  = (n & 0x00FF00FF) + ((n >> 8) & 0x00FF00FF);
   n  = (n & 0x0000FFFF) + ((n >> 16) & 0x0000FFFF);
   return n;
}
```

Još jedna od mogućih ideja je napraviti statički niz u kojem je preračunat broj bitova za svaku od 256 vrednosti bajta, a zatim čitati podatke iz tog niza.

```
int bit_count(unsigned n)
{
  static int lookup[256] = {0, 1, 1, 2, 1, 2, 2, 3, ...};
  return lookup[n & 0xFF] + lookup[n >> 8 & 0xFF] +
         lookup[n >> 16 & 0xFF] + lookup[n >> 24 & 0xFF];
}
```

Izdvajanje bitova.
Naredna funkcija vraća n bitova broja x koji počinju na poziciji p (pozicije se broje od nulte, zdesna nalevo) i prostiru se dalje ulevo (to su bitovi na pozicijama p, p+1, ..., p+n−1).

```
unsigned get_bits(unsigned x, int p, int n)
{
  /* Kreiramo masku koja ima poslednjih n jedinica:
     0000000...00011111 */
  unsigned last_n_1 = ~(~0 << n);
  /* x pomeramo udesno za odgovarajuci broj mesta, a zatim
     konjunkcijom sa  maskom brisemo suvisne cifre */
  return (x >> p) & last_n_1;
}
```

Izmena bitova.
Naredna funkcija vraća broj koji se dobija od broja x izmenom n bitova koji se prostiru od pozicije p (pozicije se broje od nulte, zdesna nalevo) pa dalje ulevo (to su bitovi na pozicijama p, p+1, ..., p+n−1), tako što se na ta mesta upisuje n krajnjih desnih bitova broja y. Na primer, ako je x jednako 10110110, y jednako 10111001 i razmatraju se n=2 bita počevši od pozicije p=3, u broju x menjaju se bitovi 10 (to su podvučeni bitovi 10110110) bitovima 01 broja y (to su podvučeni bitovi 10111001**) i dobija se broj** 10101__110.

```
unsigned set_bits(unsigned x, int p, int n, unsigned y)
{
  /* Maska 000000...000111111  - poslednjih n jedinica */
  unsigned last_n_1 = ~(~0 << n);
  /* Maska 1111100..000111111 - n nula pocevsi od pozicije p */
  unsigned middle_n_0 = ~(last_n_1 << p);
  /* Brisemo n bitova pocevsi od pozicije p */
  x = x & middle_n_0;
  /* Izdvajamo poslednjih n bitova broja y i
     pomeramo ih na poziciju p */
  y = (y & last_n_1) << p;
  /* Upisujemo bitove broja y u broj x i vracamo rezultat */
  return x | y;
}
```

Rotacija bitova.
Razmotrimo funkciju koja vrši rotaciju neoznačenog broja x za n pozicija udesno. Naivan način da se ona realizuje je da se n puta izvrši rotacija za jedno mesto udesno.

```c
unsigned right_rotate(unsigned x, int n)
{
  int i, width = sizeof(unsigned) * 8;
  /* Postupak se ponavlja n puta */
  for (i = 0; i < n; i++) {
    /* Poslednji bit (zdesna) broja x */
    unsigned last_bit = x & 1;
    /* x pomeramo za jedno mesto udesno */
    x >>= 1;
    /* Zapamceni poslednji bit stavljamo na pocetak broja x*/
    x |= last_bit << width - 1;
  }
  return x;
}
```

Bolje rešenje je da se rotacija izvrši u jednom koraku.

```c
unsigned right_rotate(unsigned x, int n)
{
  int i, width = sizeof(unsigned) * 8;
  /* Poslednjih n-bitova broja x */
  unsigned last_n_bits = x & ~(~0 << n);
  /* Pomeramo x za n mesta udesno */
  x >>= n;
  /* Na pocetak mu upisujemo upamcenih n bitova */
  x |= last_n_bits << width - n;
  return x;
}
```

Refleksija bitova.
Naredna funkcija obrće binarni zapis neoznačenog broja x tako što bitove „čita unatrag“.

```c
unsigned mirror(unsigned x)
{
  int i, width = sizeof(unsigned) * 8;
  /* Rezultat inicijalizujemo na poslednji bit broja x */
  unsigned y = x & 1;
  /* Postupak se ponavlja width - 1 puta */
  for (i = 1; i < width; i++) {
    /* x se pomera udesno za jedno mesto */
    x >>= 1;
    /* rezultat se pomera ulevo za jedno mesto */
    y <<= 1;
    /* Poslednji bit broja x upisujemo na kraj rezultata */
    y |= x & 1;
  }
```

```
  return y;
}
```

Hafmanovo kodiranje

...

1.2.10 Niske

Posebno mesto u programskom jeziku C zauzimaju nizovi koji sadrže karaktere — **niske karaktera** ili kraće **niske** (engl. *strings*). Za razliku od C++ jezika, gde se za rad sa tekstom koristi tip `std::string`, u C-u niska **nije** poseban tip podataka, već **niz karaktera** koji u memoriji sadrži dodatni znak `'\0'` na kraju. Ovaj terminator označava kraj niske. Upravo zbog te završne nule, niske u C-u se nazivaju *niske završene nulom* (engl. *null-terminated strings*). Posledica ovoga je da ne postoji ograničenje za dužinu niske, ali je neophodno proći kroz celu nisku da bi se odredila njena dužina.

Konstantne niske navode se između dvostrukih navodnika, na primer:

```
"ja sam niska"
```

U okviru niski, specijalni karakteri se navode pomoću specijalnih sekvenci, na primer:

```
"prvi red\ndrugi red"
```

Niske se mogu koristiti i prilikom inicijalizacije nizova karaktera:

```
char s1[] = {'Z', 'd', 'r', 'a', 'v', 'o'};
char s2[] = {'Z', 'd', 'r', 'a', 'v', 'o', '\0'};
char s3[] = "Zdravo";
```

Nakon ovih deklaracija, sadržaj niza `s1` je:

Indeks	0	1	2	3	4	5
Sadržaj	`'Z'`	`'d'`	`'r'`	`'a'`	`'v'`	`'o'`

a sadržaj nizova `s2` i `s3` je:

Indeks	0	1	2	3	4	5	6
Sadržaj	`'Z'`	`'d'`	`'r'`	`'a'`	`'v'`	`'o'`	`'\0'`

Niz `s1` sadrži 6 karaktera (6 bajtova), dok `s2` i `s3` sadrže po 7 karaktera (7 bajtova). Prvi niz **nije ispravna niska**, jer nema završnu nulu.

Važno je razlikovati:

- **karaktersku konstantu** `'x'` — jedan karakter
- **nisku** `"x"` — dva karaktera: `'x'` i `'\0'`

Spajanje niski u kodu

Ako se dve niske u programu nađu neposredno jedna uz drugu, one se automatski spajaju:

```
printf("Zdravo, " "svima");
```

ekvivalentno je sa:

```
printf("Zdravo, svima");
```

Funkcije za rad sa niskama

Standardna biblioteka C jezika sadrži veliki broj funkcija za rad sa niskama, čiji se prototipovi nalaze u zaglavlju:

```
#include <string.h>
```

Na primer, funkcija `strlen()` izračunava dužinu niske, **bez računanja završne nule**.

```
int duzina = strlen(s);
```

Dužina niske može se izračunati i ručno, prolaskom kroz sve karaktere:

```
int i = 0;
while (s[i] != '\0')
    i++;
```

Tipični oblici petlji za obradu niski

Kôd koji obrađuje niske obično prolazi karakter po karakter, najčešće pomoću petlji oblika:

```
while (s[i] != '\0')
    ...
```

ili:

```
for (i = 0; s[i] != '\0'; i++)
    ...
```

Pošto završna nula ima numeričku vrednost 0 (što u logičkom smislu znači *netačno*), uslov u petlji može se i skratiti:

```
for (i = 0; s[i]; i++)
    ...
```

Neefikasan pristup

Vrlo je neefikasno pozivati funkciju `strlen()` u svakom koraku iteracije:

```
for (i = 0; i < strlen(s); i++)
    ...
```

U ovom slučaju, funkcija `strlen()` se ponovo izvršava pri svakoj iteraciji, što značajno usporava program. Složenost ovog programa je $O(n^2)$.

Kopiranje niski

Funkcija `strcpy(dest, src)` kopira sadržaj niske `src` u nisku `dest`. Potrebno je da `dest` ima dovoljno prostora da primi kopiranu nisku, uključujući i završnu nulu. Ako nije tako, može doći do upisa izvan granica niza, što izaziva greške pri izvršavanju programa.

Jedan od načina da se kopiranje izvede ručno je:

```
int i = 0;
while ((dest[i] = src[i]) != '\0')
    i++;
```

ili, u kraćem obliku sa praznim telom petlje:

```
for (i = 0; dest[i] = src[i]; i++)
    ;  /* prazno telo petlje */
```

Kratak rezime: razlika između C i C++ niski

Osobina	**C niska**	**C++** `std::string`
Tip	Niz karaktera (`char[]`)	Klasa (`std::string`)
Terminator	Znak `'\0'`	Čuva dužinu interno
Promenljiva dužina	Ne (fiksna veličina niza)	Da
Dinamička alokacija	Ručno (`malloc`, `free`)	Automatski
Upoređivanje i dodela	Ručno, funkcijama (`strcmp`, `strcpy`)	Operatorima ==, =
Bezbednost	Niska, rizik od prepisivanja memorije	Visoka, proverava granice

> **Napomena:** U C-u postoje **polja bitova** (eng. bit fields) i **unije** (eng. unions) — posebni načini organizacije podataka koji omogućavaju uštedu memorije.

1.3 Organizacija ulaznog i izvršivog programa

Kada programer napiše **izvorni program**, to je samo tekstualna datoteka koja opisuje algoritme i strukture u nekom višem jeziku, poput C-a. Međutim, računar može izvršavati isključivo **mašinski kod** – niz instrukcija podržanih na nivou procesorske arhitekture. Da bi se iz izvornog programa dobio izvršivi, prevodilac prolazi kroz više faza.

Početnicima ceo proces često izgleda kao jedan jedini korak (na primer, poziv `gcc program.c`). Ipak, to je zapravo niz uzastopnih transformacija, od kojih svaka ima sopstvenu ulogu. U jeziku C uobičajeno razlikujemo tri osnovne faze: **pretprocesiranje**, **kompilaciju** i **povezivanje**. Tokom svake od tih faza mogu se javiti greške, a prijavljene poruke često ukazuju ne samo na sintaktičke propuste, već i na dublje probleme u organizaciji programa.

1.3.1 Pretprocesiranje

Prva faza je **pretprocesiranje** (eng. *preprocessing*). U njoj se vrše jednostavne transformacije teksta izvornog programa, pre nego što ga dobije sam kompilator. Najvažniji zadatak pretprocesora jeste objedinjavanje koda iz više datoteka. Na primer, u datoteke izvornog koda (`.c`) uključuju se **zaglavlja** (`.h`) koja sadrže deklaracije tipova, funkcija i promenljivih koje se koriste u različitim delovima programa.

Tako se ceo program organizuje u tzv. **jedinice prevođenja**, koje pretprocesor priprema i predaje kompilatoru. Drugim rečima, kompilator ne vidi baš onaj tekst koji je programer napisao, već "proširenu" verziju izvora nakon svih `#include` i `#define` direktiva.

Rezultat faze pretprocesiranja može se eksplicitno dobiti navođenjem opcije `-E` pri korišćenju GCC prevodioca:

```
gcc -E program.c
```

Ova komanda ispisuje prošireni tekst, koji kompilator zapravo prevodi.

1.3.2 Kompilacija

Nakon pretprocesora, sledeća faza je **kompilacija u užem smislu** (eng. *compilation*). Cilj je da se svaka jedinica prevođenja prevede u **objektni modul** – deo mašinskog koda koji još uvek nije spreman za izvršavanje.

Sam proces kompilacije prolazi kroz nekoliko slojeva analize i transformacije:

- **Leksička analiza** – prepoznavanje osnovnih jezičkih elemenata (leksema).
- **Sintaksička analiza** – izgradnja sintaksičkog stabla koje opisuje strukturu programa.
- **Semantička analiza** – provera značenja i ispravnosti programa na višem nivou (tipovi, opseg promenljivih, itd.).
- **Generisanje međukoda i optimizacija** – kod se prevodi u internu reprezentaciju i unapređuje radi efikasnosti.
- **Generisanje mašinskog koda** – stvara se objektni modul (`.o` ili `.obj` datoteka).

Objektni moduli sadrže funkcije prevedene u mašinski kod i podatke, ali se **ne mogu neposredno izvršavati**, jer im nedostaje informacija o tačnim adresama memorije.

Na primer, naredba:

```
gcc -c program.c
```

stvara datoteku `program.o`. Ovaj objektni modul je relokatibilan: tek prilikom kasnijeg povezivanja razrešavaju se stvarne memorijske adrese.

Opcije kompilatora dodatno utiču na proces. `-O3` omogućava opsežnu optimizaciju, dok `-Wall` uključuje većinu uobičajenih upozorenja, pomažući programeru da izbegne greške.

1.3.3 Povezivanje

Treća faza je **povezivanje** (eng. *linking*). Svrha ove faze je da objedini sve objektne module i potrebne biblioteke u jednu izvršivu datoteku. Povezivač razrešava sve nedefinisane adrese: gde se zapravo nalazi pozvana funkcija, gde se nalazi globalna promenljiva i slično.

Povezivanje može biti:

- **statičko** – kompletan kod biblioteka se kopira u izvršnu datoteku;
- **dinamičko** – kod ostaje u posebnoj datoteci (`.so` na Linuxu, `.dll` na Windowsu), a adrese se razrešavaju tek pri pokretanju programa.

Uobičajeno, kompajler automatski poziva i povezivač, pa naredba:

```
gcc -o program program.c
```

istovremeno prevodi i povezuje kod, dajući izvršni fajl `program`. Dodavanjem opcije `-l` moguće je uključiti dodatne biblioteke, npr. matematičku biblioteku:

```
gcc -o program program.c -lm
```

1.3.4 Odvojena kompilacija i povezivanje

Veći programi obično se sastoje od više datoteka izvornog koda. Tada je korisno razdvojiti proces na dva koraka:

1. Prevesti svaku `.c` datoteku u odgovarajući `.o` objektni modul.
2. Povezati sve `.o` fajlove u jedinstveni izvršni program.

Na primer:

```
gcc -c program1.c
gcc -c program2.c
gcc -o program program1.o program2.o
```

Prednost ovog pristupa je što se prilikom izmene jednog modula ne prevode svi ostali – smanjuje se vreme gradnje programa.

1.3.5 Automatizacija: program *make*

Ručna kompilacija i povezivanje mogu postati zamorni u većim projektima. Zbog toga se koristi program **make**, koji upravlja zavisnostima između datoteka. U datoteci *Makefile* definiše se kako se dobija jedna datoteka iz drugih, a `make` automatski pokreće potrebne komande samo kada je to neophodno.

Na primer:

```
program: program1.o program2.o biblio.o
    gcc -o program program1.o program2.o biblio.o

program1.o: program1.c biblio.h
    gcc -c program1.c

program2.o: program2.c biblio.h
    gcc -c program2.c
```

Ovaj opis omogućava da se nakon svake izmene prevedu samo datoteke koje zavise od izmenjenih, a ostalo se ne dira. Upravo to čini rad na većim projektima efikasnijim.

1.3.6 Izvršavanje programa

Kada je izvršna datoteka dobijena, njenim pokretanjem zadužuje se tzv. **punilac** (eng. *loader*). Punilac je deo operativnog sistema koji učitava program u memoriju, inicijalizuje potrebne registre i na kraju poziva ulaznu tačku programa.

Iako programeri očekuju da je to funkcija `main`, stvarna ulazna tačka je funkcija `_start`, koja se nalazi u *runtime* biblioteci. Ona poziva `main`, ali takođe obavlja i dodatne poslove: prosleđivanje argumenata komandne linije, inicijalizaciju standardnih biblioteka i čišćenje nakon završetka programa.

Za analizu rada programa koristi se **debager** (npr. `gdb`), ali je za to neophodno da izvršni fajl bude preveden sa opcijom `-g`, koja čuva informacije o izvornom kodu. Tako nastaje *debug verzija* programa, dok se korisnicima obično isporučuje *release verzija* bez tih dodatnih informacija.

1.3.7 Organizacija izvornog programa

Viši programski jezici namenjeni su prvenstveno čoveku (a ne računaru). Algoritam je po pravilu lakše izraziti na jeziku C nego na mašinskom jeziku, a važi i obrnuto: programe drugih autora lakše ćemo razumeti kada su napisani u C-u nego u asembleru. Ipak, pisanje, razumevanje i održavanje **velikih** programa ostaje ozbiljan izazov čak i na višem jeziku. Zato je u C-u (kao i u drugim jezicima) razvijen čitav skup mehanizama i konvencija koji vode ka *kraćem, preglednijem i efikasnijem* kodu – kako u fazi kompilacije, tako i u fazi izvršavanja.

U praksi, organizacija složenijih programa polazi od **funkcionalne dekompozicije**: složen problem deli se na manje potprobleme koji se izdvajaju u zasebne funkcije i module. Time se postiže manja međuzavisnost delova sistema (ideal kome se teži je da funkcije u programu budu što nezavisnije i što slabije međusobno uslovljene). Kod većih sistema uobičajeno je grupisati blisko povezane tipove i operacije nad njima u **module** (izvorne datoteke) – npr. definiciju tipa za kompleksne brojeve i prateće operacije smestiti u jedan modul. Takvi moduli često se izdvajaju i u **biblioteke**, radi ponovne upotrebe u različitim projektima.

Ovakav stil programiranja podržan je mehanizmima jezika C i njegovog alata za prevođenje. Pretprocesor i linker omogućavaju da se program prirodno podeli na više datoteka koje se, prema zavisnostima, spajaju u jedinstveni izvršivi program. Pored toga, C precizno razlikuje i na nivou jezika reguliše:

- **Doseg identifikatora** (eng. *scope*): gde je neki identifikator vidljiv i upotrebljiv (fajl/jedinica prevođenja, funkcija, blok). Kontrolisan doseg olakšava razumevanje i smanjuje mogućnost grešaka.
- **Životni vek promenljivih** (eng. *lifetime*): koliko dugo objekat postoji tokom izvršavanja (npr. automatske promenljive žive tokom poziva funkcije; statičke tokom čitavog izvršavanja).
- **Povezanost identifikatora** (eng. *linkage*): da li je simbol dostupan iz drugih jedinica prevođenja (spoljna povezanost), ili je ograničen na tekuću jedinicu (interna povezanost), ili nema povezanost (lokalni simbol).

Ovi aspekti određuju se **mestima deklaracije/definicije** i upotrebom kvalifikatora `auto`, `register`, `static` i `extern` (detaljno u nastavku poglavlja). Pravilna upotreba donosi jasniju strukturu i učinkovitiji kod; nepravilna upotreba, naprotiv, uvodi suptilne greške i nepoželjne zavisnosti.

Pretprocesor

Pri izgradnji izvršnog programa iz C izvora, pre samog prevodioca (kompilatora) pokreće se C pretprocesor. Pretprocesor izvodi **tekstualne** transformacije nad izvornim kodom na osnovu *pretprocesorskih direktiva* (one nisu deo „jezika C“ u užem smislu). On ne razume semantiku C naredbi, već uređuje i priprema tekst koji će kompilator analizirati.

Tipične uloge pretprocesora su:

- uključivanje drugih datoteka (`#include`);
- zamena tekstualnih obrazaca, uz mogućnost parametara (makroi preko `#define`);
- uslovna kompilacija (npr. različite grane koda za različite platforme).

Pošto zbog `#include` do kompilatora stiže *proširen* tekst iz više fizičkih datoteka, govorimo o **jedinicama prevođenja**: to su logičke celine koje kompilator vidi i od kojih gradi **objektne module**. Jedinice prevođenja imaju neposrednu vezu sa procesima kompilacije i povezivanja, i važnije su za prevodilac od same fizičke raspodele koda po fizičkim datotekama koje programer kreira.

> **Napomena.** Iako pretprocesor „samo“ menja tekst, on obavlja i elementarnu tokenizaciju, jer npr. zamene ne smeju da se vrše unutar literala niske ili identifikatora.

Uključivanje datoteka zaglavlja

Veliki programi dele deklaracije (tipova, funkcija, globalnih promenljivih) među više izvora. Umesto da ih ručno dupliramo po svakoj `.c` datoteci, izdvajamo ih u **datoteke zaglavlja** (`.h`) i uključujemo gde su potrebne:

```
#include "ime_datoteke"
#include <ime_datoteke>
```

Oba oblika zamenjuju liniju sadržajem zadate datoteke. Razlika je u tome **gde** se datoteka traži:

- `"..."` pretražuje projektne (korisničke) putanje — često uključujući i direktorijum tekuće datoteke; dodatne putanje zadaju se kompilatoru opcijama tipa `-I`.
- `<...>` traži u **sistemskim** putanjama predviđenim za standardna zaglavlja.

Ako se promeni uključena `.h` datoteka, **sve** jedinice koje od nje zavise moraju se **ponovo prevesti**.

> **Deklaracije vs. definicije.** Deklaracije obezbeđuju kompilatoru informacije dovoljne da *prevede* pozive i upotrebe (npr. prototip funkcije), dok su definicije potrebne *linkeru* da razreši simbole (npr. telo funkcije). Kompilator može da generiše poziv funkcije na osnovu **deklaracije**, ali da bi izvršni program nastao, linker mora da pronađe njenu **definiciju**.

Makro zamene

Direktiva `#define` uvodi **makro** – pravilo za zamenu jednog teksta drugim, pri čemu zamena nastupa *pre* kompilacije. Njen opšti oblik je:

```
#define originalni_tekst novi_tekst
```

Makro može biti bilo koji identifikator, pa čak i ključna reč jezika C. Tekst zamene (`novi_tekst` u navedenom opštem obliku) je tekst do kraja reda, a moguće je da se prostire na više redova ako se navede simbol `\` na kraju svakog reda koji se nastavlja.

(1) Proste simboličke konstante

```
#define MAX_LEN 80
```

`MAX_LEN` nije promenljiva (čak ni `const`), već čista tekstualna oznaka: pretprocesor *svako pojavljivanje* zamenjuje 80. Ne zauzima memoriju, ne vidi je ni kompilator ni linker.

(2) Makroi sa argumentima

```
#define max(A, B)  ((A) > (B) ? (A) : (B))
```

Poziv `max(2, 3)` postaje `((2) > (3) ? (2) : (3))`. Važna su dva upozorenja:

- **Nema provere tipova** ni implicitnih konverzija – za razliku od funkcija.
- **Višestruka evaluacija argumenata**: `max(a++, b++)` proširiće se na osnovu date definicije na `((a++) > (b++) ? (a++) : (b++))` i rezultat će biti inkrementiran **dva puta**, što najčešće nije željeno.

(3) Zagrade i prioriteti

Loša definicija:

```
#define kvadrat(x)  x*x
```

ukoliko se primeni na `kvadrat(a+2)`, tekst zamene će biti `a+2*a+2` i zbog toga bi trebalo koristiti:

```
#define kvadrat(x)  (x)*(x)
```

Navedena definicija, međutim, i dalje ne daje ponašanje koje je verovatno željeno. Naime, kada se makro primeni na izraz `a/kvadrat(b)`, on će biti zamenjen izrazom `a/(b)*b`, što je ekvivalentno sa `(a/b)*b`. Zbog toga je bolja definicija:

```
#define kvadrat(x)  ((x)*(x))
```

(4) Blokovi i „generičke“ operacije

```
#define swap(t, x, y) { t z; z = x; x = y; y = z; }
```

Tada `swap(int, a, b)` razmenjuje dve `int` promenljive.

(5) Prestanak važenja

Makro važi od mesta definicije do kraja datoteke, ili do eksplicitnog „brisanja“:

```
#undef MAX_LEN
```

(6) Makro naspram funkcija

Kao što je rečeno, makro zamene i funkcije se, iako mogu izgledati slično, suštinski razlikuju. Kod makroa nema provere tipova argumenata niti implicitnih konverzija što može da dovode do grešaka u kompilaciji ili do neočekivanog rada programa. Argument makroa u zamenjenoj verziji može biti naveden više puta što može da utiče na izvršavanje programa, a izostavljanje zagrada u tekstu zamene može da utiče na redosled izračunavanja operacija. S druge strane, kod poziva makroa nema prenosa argumenata te se oni izvršavaju brže nego odgovarajuće funkcije. Ipak, moderni kompilatori obično umeću (engl. *inline*) čitavo telo kratke funkcije na mesto poziva, starajući se o argumentima i tada nema gubitaka u vremenskoj ili prostornoj efikasnosti u odnosu na korišćenje makroa. Sve u svemu, makro zamene sa argumentima imaju i dobre i loše strane i treba ih koristiti oprezno.

> **Napomena.** Pojedine funkcije standardne biblioteke mogu biti implementirane i kao makroi (npr. `getchar` koji koristi funkciju `getc`), ali standard obezbeđuje semantiku poziva nezavisno od te tehničke realizacije.

Uslovno prevođenje

Pretprocesorske direktive omogućavaju da se **delovi koda uključe ili isključe iz prevođenja** u zavisnosti od uslova koji se računaju u fazi pretprocesiranja.

Osnovna direktiva je `#if`, koja ocenjuje **konstantni celobrojni izraz** (koji može, na primer, da sadrži konstante i simbolička imena definisana direktivom `#define`). Ako je vrednost izraza **0**, pretprocesor preskače linije do narednog `#elif`, `#else` ili `#endif`. U izrazima je korisna funkcija `defined(ime)` koja vraća 1 ako je dato simboličko ime definisano, u suprotnom 0.

Skraćeni oblici:

- `#ifdef ime` ≡ `#if defined(ime)`
- `#ifndef ime` ≡ `#if !defined(ime)`

Primer:

```
#define SRPSKI
#include <stdio.h>

int main(void) {
#ifdef SRPSKI
    printf("Zdravo, svete");
```

```
#else
    printf("Hello, world");
#endif
    return 0;
}
```

Ako je SRPSKI definisan, ispis je na srpskom; u suprotnom, na engleskom.

> **Važno:** Ovo nije isto što i C-naredba `if`. Kod „odsečen“ granom pretprocesora uopšte ne ulazi u izvršni fajl: prevodi se samo *jedna* varijanta. Kod grananja `if (...) { ... } else { ... }` u C-u, **obe grane postoje** u objektu, a izbor se dešava u toku izvršavanja.

Doseg identifikatora

Jedno od načela dobrog stila u C-u jeste da se promenljive *deklarišu što uže* (u funkciji ili čak u manjem bloku). Time se smanjuje zavisnost između funkcija i olakšava verifikacija ispravnosti funkcija na osnovu ulaza/izlaza, umesto globalnog stanja.

Doseg (eng. *scope*) određuje gde je identifikator vidljiv i može se koristiti. C ima *statička pravila dosega* – doseg se utvrđuje iz izvornog koda (ne zavisi od toka izvršavanja).

Vrste dosega:

- **Doseg datoteke** (eng. *file scope*): ime važi od tačke uvođenja do kraja datoteke.
- **Doseg bloka** (eng. *block scope*): ime važi od uvođenja do kraja bloka `{ ... }`.
- **Doseg funkcije** (eng. *function scope*): važi u celoj funkciji; samo labele (za `goto`) imaju ovaj doseg.
- **Doseg prototipa funkcije** (eng. *function prototype scope*): važi unutar prototipa; odnosi se na imena parametara u deklaraciji funkcije.

Najčešće razlikujemo **globalne** (doseg datoteke) i **lokalne** (doseg bloka) objekte. Preporuka je: promenljive – lokalne kad god je moguće; funkcije – tipično globalne. Od C99 standarda, moguće su i *lokalne funkcije* (funkcije sa dosegom bloka).

```
int a;
/* a: globalna promenljiva - doseg datoteke */

void f(int c) {
    /* c: lokalna promenljiva - doseg bloka funkcije f */
    int d;
    /* d: lokalna promenljiva - doseg bloka */

    void g(void) { printf("zdravo"); }
    /* g: lokalna funkcija - doseg bloka funkcije f */

    for (d = 0; d < 3; d++) {
        int e;
        /* e: lokalna promenljiva - doseg tela petlje */
        /* ... */
    }
```

```
kraj:
    /* labela 'kraj' - doseg funkcije */
}

void h(int b); /* b: doseg prototipa funkcije */
```

Konflikt identifikatora. tj. moguće je da postoji više identifikatora istog imena. Ako su njihovi dosezi jedan u okviru drugog, tada identifikator u užoj oblasti dosega sakriva identifikator u široj oblasti dosega.

```
void f(void) {
    int a = 3, i;
    for (i = 0; i < 4; i++) {
        int a = 5;           /* sakriva spoljnji 'a' */
        printf("%d ", a);
    }
}
```

```
Izlaz:
5 5 5 5
```

Ovim je omogućeno da prilikom uvođenja novih imena programer ne mora da brine da li je takvo ime već upotrebljeno u širem kontekstu.

Životni vek objekata i kvalifikatori `static` i `auto`

Životni vek (eng. *lifetime*) opisuje *koliko dugo* objekat postoji u memoriji tokom izvršavanja:

- **Statički** (`static`): objekat postoji tokom celog izvršavanja programa.
- **Automatski** (`automatic`): objekat postoji tokom u okviru bloka/funkcije (stvara se pri ulasku, uništava pri izlasku).
- **Dinamički** (`dynamic`): objekat postoji dok ga eksplicitno ne alociramo/dealociramo (vidi pogl. o dinamičkoj alokaciji).

Životni vek je određen mestima uvođenja i kvalifikatorima:

- `auto` – automatski životni vek (podrazumevan za lokalne promenljive u bloku; zato se retko piše).
- `static` – statički životni vek (postojanje tokom celog izvršavanja).

> **Napomena:** U C++ `auto` znači *zaključivanje tipa*, što je potpuno drugo značenje nego u C-u.

Lokalne automatske promenljive

Najčešće su to promenljive deklarisane unutar bloka (funkcije, petlje). One:

- imaju **doseg bloka**;
- imaju **automatski životni vek** (sem ako im se doda `static` ili `extern`);
- **nemaju definisanu početnu vrednost** (sadržaj memorije je neodređen dok se eksplicitno ne inicijalizuju).

Automatske promenljive tipično se čuvaju u **stek okviru** aktivne funkcije. Pri rekurziji svaka aktivna instanca dobija svoj nezavisan primerak promenljive. Parametri funkcija imaju isti status kao i lokalne automatske promenljive.

Globalne statičke promenljive

Globalne promenljive (deklarisane izvan svih funkcija) imaju **doseg datoteke** i **statički životni vek**: memorija se rezerviše pri pokretanju i oslobađa po završetku programa (tipično u **segmentu podataka**; vidi pogl. o memoriji). Ako nisu eksplicitno inicijalizovane, **podrazumevana vrednost je 0**.

Zbog stalnog postojanja i vidljivosti iz više funkcija, globalne promenljive mogu poslužiti za komunikaciju između funkcija — ali to čini kod *težim za razumevanje i održavanje*. Preporuka je koristiti umereno i sa jasnim razlogom.

Kada se za globalnu promenljivu napiše `static`, to ne menja njen životni vek (ionako je statički), već menja **povezanost** (linkage) u **internu** — ime nije vidljivo iz drugih jedinica prevođenja (vidi pogl. o povezivanju/povezanosti).

Lokalne statičke promenljive

Ponekad je potrebno **zapamtiti stanje između poziva funkcije** (npr. brojanje poziva). Globalna promenljiva bi to omogućila, ali bi bila vidljiva svuda, bilo bi moguće koristiti (i promeniti) i iz drugih funkcija, što je nepoželjno. Rešenje je **lokalna** promenljiva sa **statičkim životnim vekom**:

- deklariše se u telu funkcije sa `static`;
- smešta se u **segment podataka**, ne u stek;
- inicijalizuje se **jednom** na početku izvršavanja (konstantnim izrazom; podrazumevano 0);
- zadržava vrednost između poziva;
- i dalje ima **doseg bloka** (vidljiva je samo u toj funkciji).

Primer:

```c
#include <stdio.h>

void f(void) {
    int a = 0;
    printf("f: %d ", a);
    a = a + 1;
}

void g(void) {
    static int a = 0;
```

```
    printf("g: %d ", a);
    a = a + 1;
}

int main(void) {
    f(); f();
    g(); g();
    return 0;
}
```

```
Izlaz:
f: 0 f: 0 g: 0 g: 1
```

U `f` se lokalna automatska promenljiva `a` kreira tokom svakog poziva (uvek 0). U `g` lokalna statička promenljiva `a` čuva stanje između poziva (0, zatim 1).

Povezanost identifikatora i kvalifikatori `static` i `extern`

Pravila dosega određuju gde je ime vidljivo u okviru jedne jedinice prevođenja. Međutim, u programima koji se sastoje od više jedinica prevođenja, to nije dovoljno: potrebna je finija kontrola o tome *da li više pojavljivanja istog imena upućuju na isti ili različit objekat* preko granica datoteka. Tu nastupa pojam **povezanosti identifikatora** (eng. *linkage*).

Povezanost je direktno vezana za fazu povezivanja (eng. linking) i odgovara na pitanja tipa: *„Ako se ime `a` pojavi u više `.c` fajlova, da li se misli na jednu istu globalnu promenljivu ili na različite?“* Slično i za funkcije.

Vrste povezanosti. Jezik C razlikuje:

- **Bez povezanosti** (eng. *no linkage*): svako pojavljivanje imena uvodi nezavisan objekat (npr. lokalne promenljive u blokovima, formalni parametri, imena tipova, labele).
- **Spoljašnju povezanost** (eng. *external linkage*): pojavljivanja imena u različitim jedinicama prevođenja odnose se na isti objekat.
- **Unutrašnju povezanost** (eng. *internal linkage*): ime je vidljivo samo unutar jedne jedinice prevođenja (npr. globalni objekat označen `static` u `.c` fajlu).

  Ne mešati sa dosegom. Povezanost se tiče globalnih objekata (spoljašnjeg dosega) i ponašanja na nivou linkovanja; doseg je pojam vidljivosti u okviru izvornog teksta. Lokalni objekti su tipično bez povezanosti (osim posebnog slučaja lokalne `extern` deklaracije, videti dole).

  Životni vek. Povezanost „ima smisla” uglavnom za objekte statičkog životnog veka (globalne i lokalne statičke). Objekti automatskog životnog veka (lokalne automatske promenljive) su po pravilu bez povezanosti.

Deklaracija naspram definicije. Precizno razlikovanje je ključno:

- **Deklaracija**: uvodi ime i njegov **tip**; kompilatoru je to dovoljno da koristi objekat (npr. generiše poziv funkcije). Intuitivno: „negde u programu postoji objekat ovog tipa”.

- **Definicija**: stvarno **instancira** objekat (rezerviše memoriju za promenljivu, daje telo funkcije). Linkeru je neophodna **tačno jedna** definicija za svaki objekat sa spoljašnjom povezanošću.

Za funkcije je razlika očigledna: prototip je deklaracija; prototip + telo je definicija. Za globalne promenljive postoji nijansa:

- `int a = 3;` je **definicija** (i deklaracija).
- `int a;` (bez inicijalizacije) je **načelna definicija** (*tentative definition*):
  - ako negde postoji stvarna definicija a, ova linija se tretira kao deklaracija;
  - ako ne postoji stvarna definicija, načelna definicija postaje definicija sa inicijalnom vrednošću 0.

Identifikatori bez povezanosti

Deklaracije uvode **nezavisne objekte**; mogu se ponavljati u različitim funkcijama i datotekama bez međusobnog odnosa. Tipični slučajevi: lokalne promenljive (automatske i statičke), parametri funkcija, korisnički tipovi, labele.

Primer (iz različitih jedinica prevođenja):

```
int f() {
    int i;       /* lokalna bez povezanosti */
    /* ... */
}

int g1(int i) {  /* parametar bez povezanosti */
    static int j;/* lokalna statička bez povezanosti */
    /* ... */
}

int g2() {
    static int i, j; /* obe lokalne statičke, ali bez povezanosti */
    /* ... */
}

/* struct i je ime tipa (takođe bez povezanosti) */
struct i { int a; };
```

Svaka `i` i `j` ovde su različiti objekti.

Spoljašnja povezanost i kvalifikator `extern`

Spoljašnja povezanost omogućava da **jedan isti objekat** bude vidljiv u **više** jedinica prevođenja. Potrebna je **jedna definicija** u celom programu, a **više deklaracija** (po potrebi) u drugim datotekama.

- `extern` se koristi uz deklaracije da se naglasi da se ime odnosi na spoljašnji objekat definisan drugde.

- `extern` ne ide uz definiciju promenljive (nema inicijalizatora) niti uz telo funkcije.

Podrazumevana spoljašnja povezanost:

- Funkcije su podrazumevano sa spoljašnjom povezanošću (osim ako su globalne i označene sa `static`, tada dobijaju unutrašnju povezanost).
- Globalne promenljive su po podrazumevano spoljašnje (osim ako su globalne i označene sa `static`, tada imaju unutrašnju).

Iako su globalne „načelne definicije" (`int x;` bez inicijalizacije) dovoljne da kompilacija prođe, u zaglavljima se često preporučuje eksplicitno pisati `extern int x;` – jasno naglašava deklaraciju (a ne definiciju) i izbegava zavisnost od toga da li u nekoj drugoj datoteci postoji „stvarna" definicija.

Lokalna `extern` deklaracija. Ređi, ali važan obrazac: u lokalnom dosegu (npr. unutar funkcije) možemo napisati `extern int b;`. Time u tekućoj jedinici prevođenja dobijamo lokalnu deklaraciju imena koje se odnosi na spoljašnju globalnu promenljivu definisanu u nekoj drugoj datoteci. Dakle, deklaracija ima doseg bloka, ali ime i dalje ima spoljašnju povezanost. Razmotrimo naredni primer koji sadrži dve jedinice prevođenja.

```c
// mod1.c
#include <stdio.h>

int a;        /* načelna definicija → postaje definicija sa inic. 0,
                 jer nema „stvarne" definicije drugde */
int b = 3;    /* stvarna definicija (uz inicijalizaciju) */

void f(void) {
    printf("a=%d\n", a);
}
```

```c
// mod2.c
#include <stdio.h>

extern int a;   /* deklaracija 'a' - spoljašnja povezanost,
                   definisana u mod1.c */
void f(void);   /* deklaracija funkcije f definisane u mod1.c */

int g(void) {
    extern int b;       /* lokalna deklaracija:
                           odnosi se na globalnu 'b' iz mod1.c */
    printf("b=%d\n", b);
    f();                /* poziv globalne funkcije iz mod1.c */
    return 0;
}

int main(void) {
    a++;                /* menja globalnu 'a' (sada 1) */
    /* b = 2;           // greška: 'b' ovde nije u dosegu
                           (nije deklarisana u main) */
```

```
    g();
    return 0;
}
```

```
Izlaz:
b=3
a=1
```

U prvoj jedinici prevođenja postoje globalne promenljive `a` i `b`, kao i globalna funkcija `f`. Sve one podrazumevano imaju statički životni vek i spoljašnju povezanost. Linija `int b = 3;` sadrži inicijalizaciju tako da je jasno da je u pitanju definicija (i to stvarna). U prvom trenutku nije jasno da li je `int a;` deklaracija ili definicija i u pitanju je samo načelna definicija, međutim, pošto ne postoji druga definicija promenljive `a`, ova linija se ipak tumači kao definicija, uz podrazumevanu inicijalnu vrednost 0. Druga jedinica prevođenja sadrži dve deklaracije: deklaraciju promenljive `a` i funkcije `f`. U slučaju promenljive eksplicitno je upotrebljen kvalifikator `extern` čime je naglašeno da je u pitanju promenljiva sa spoljašnjom povezanošću i da se želi pristup promenljivoj `a` iz prve jedinice prevođenja. Čak i da kvalifikator `extern` nije naveden, program bi radio identično (obe definicije bile bi načelne i tek tokom povezivanja bio bi napravljen jedinstven objekat statičkog životnog veka inicijalizovan na nulu na koji bi se obe ove promenljive odnosile). U slučaju funkcije `f` nije bilo neophodno navoditi `extern` jer je jasno da je u pitanju deklaracija. Identifikator `b` je uveden u lokalni doseg funkcije `g` uz kvalifikator `extern` čime je naglašeno da se misli na promenljivu `b` definisanu u prvoj jedinici prevođenja. U funkciji `main` uvećava se vrednost promenljive `a` na 1. Pokušaj postavljanja vrednosti promenljive `b` (pod komentarom) ne bi uspeo jer identifikator `b` nije u dosegu funkcije `main`.

Unutrašnja povezanost i kvalifikator `static`

Globalni objekat (promenljiva ili funkcija) ima **unutrašnju povezanost** (eng. *internal linkage*) ako je uz nju ključna reč `static`. Tada se sva pojavljivanja istog identifikatora u okviru *jedne* jedinice prevođenja odnose na **isti objekat**, pri čemu u toj jedinici mora postojati tačno jedna definicija. S druge strane, to ime nije vidljivo iz drugih jedinica prevođenja: pokušaj pristupa tom objektu iz druge datoteke dovešće do greške u povezivanju.

Napomena o lokalnim promenljivim. `static` u lokalnom kontekstu *ne menja povezanost* (lokalni objekti su bez povezanosti), već dodeljuje statički životni vek (promenljiva čuva vrednost između poziva). O tome je već bilo reči u odeljku o životnom veku promenljivih.

Osnovna uloga unutrašnje povezanosti je **enkapsulacija**: deo koda postaje „zatvoren" za direktan spoljašnji pristup. Iz drugih jedinica prevođenja tom modulu se pristupa isključivo preko javnog interfejsa (globalnih objekata sa spoljašnjom povezanošću). Ako u jednoj jedinici prevođenja postoji `static int a;`, druga jedinica može takođe imati sopstveni globalni `int a;` — to su dve različite promenljive. Kao primer, razmotrimo naredne dve jedinice prevođenja:

```
// mod1.c
#include <stdio.h>

static int a = 3;        /* globalna, statička → unutrašnja povezanost */
int b = 5;               /* globalna, spoljašnja povezanost */

static int f(void) {     /* funkcija vidljiva samo u mod1.c */
    printf("f: a=%d, b=%d\n", a, b);
    return 0;
```

```
}

int g(void) {              /* globalna funkcija (spoljašnja povezanost) */
    f();
    return 0;
}
```

```
// mod2.c
#include <stdio.h>

int g(void);               /* deklaracija globalne funkcije iz mod1.c */
int a, b;                  /* globalne, spoljašnja povezanost u mod2.c */

int main(void) {
    printf("main: a=%d, b=%d\n", a, b);
    g();                   /* OK: g je „javno" iz mod1.c */
    /* f();                // GREŠKA pri povezivanju: f je „skrivena"
                              (static u mod1.c) */
    return 0;
}
```

```
Izlaz:
main: a=0, b=5
f: a=3, b=5
```

U prvoj jedinici prevođenja promenljiva `a` i funkcija `f` imaju unutrašnju povezanost (dok promenljiva `b` i funkcija `g` imaju spoljašnju povezanost (o kojoj će biti reči u nastavku poglavlja). U drugoj jedinici prevođenja promenljive `a`, `b` i funkcije `g` i `main` imaju spoljašnju povezanost. Ime `a` u dve različite jedinice prevođenja odnosi se na dve različite promenljive. Promenljiva `a`, definisana u prvoj jedinici prevođenja, ključnom rečju `static` sakrivena je i nije joj moguće pristupiti iz ostalih jedinica prevođenja. Slično je i sa funkcijom `f` (uklanjanje komentara u drugoj jedinici prevođenja dovelo bi do greške prilikom povezivanja jer je definicija funkcije `f` skrivena u prvoj jedinici prevođenja). Sa druge strane, sva pojavljivanja imena `b` odnose se na istu promenljivu i sva pojavljivanja imena `g` odnose se na istu funkciju.

Pošto u prvoj jedinici prevođenja postoji definicija promenljive `b`, u drugoj jedinici prevođenja `int b;` je samo njena deklaracija. Pošto je definicija promenljive `a` iz prve jedinice prevođenja skrivena, prilikom povezivanja druge jedinice prevođenja nije dostupna definicija promenljive `a`, onda se `int a;` u drugoj jedinici prevođenja smatra njenom definicijom (uz podrazumevanu vrednost 0).

1.3.8 Greške u fazi prevođenja i povezivanja

Tokom generisanja izvršnog programa, greške (eng. *errors*) mogu biti otkrivene u različitim fazama. Poruke se uobičajeno šalju na **standardni izlaz za greške** `stderr`. Čak i kratak program može ilustrovati greške na svim nivoima:

```
#Include <stdio.h>
int main(void) {
```

```
    int a = 9;
    if (a == 9)
        printf("9");
    return 0;
}
```

Greške u pretprocesiranju. Ako je direktiva pogrešno napisana (npr. `#Include` umesto `#include`):

```
primer.c:1:2: error: invalid preprocessing directive #Include
```

Pretprocesor radi *pre* kompilatora; greške su sintaktičko–leksičke, tj. u domenu direktiva (`#include`, `#define`, …).

Greške u kompilaciji. Kompilacija obuhvata leksičku, sintaksičku i semantičku analizu.

- **Leksička analiza (tokene/lekseme).** `int a = 09;` – neispravna oktalna konstanta:

  ```
  primer.c:4:9: error: invalid digit "9" in octal constant
  ```

- **Sintaksička analiza (gramatika).** `if a == 9 ...` – (bez zagrada):

  ```
  primer.c:5:6: error: expected ‘(’ before ‘a’
  ```

- **Semantička analiza (tipovi/pravila jezika).** `if ("a" * 9) ...` – operator `*` nije primenljiv na niske:

  ```
  primer.c:5:8: error: invalid operands to binary *
                       (have ‘char *’ and ‘int’)
  ```

Greške u povezivanju (linkovanju). Ako se poziva nedefinisana funkcija (npr. pogrešno ime `print` umesto `printf`):

```
primer.c:(.text+0x20): undefined reference to `print'
collect2: ld returned 1 exit status
```

Greške u povezivanju ukazuju da nedostaje definicija simbola (funkcije, promenljive) potrebna da bi se „spojile” sve reference iz objektnih modula.

Gde (ne) nastaju greške. Sve greške koje se tiču izvornog koda otkrivaju se u: pretprocesiranju, leksičkoj/sintaksičkoj/semantičkoj analizi, ili na linkovanju. U fazama generisanja/optimizacije međukoda i generisanja mašinskog koda (uz ispravan prevodilac) ne bi trebalo da nastaju korisniku prijavljene greške.

Poznavanje ovih faza pomaže u bržem otklanjanju problema: poruka o grešci najčešće jasno upućuje na deo procesa u kome je nastala.

Upozorenja prevodioca (warnings). Pored grešaka, kompilator može izdati upozorenja – program se i dalje prevodi, ali upozorenja ukazuju na potencijalne propuste.

- **Dodela umesto poređenja**: `if (a = 9) ...`

```
primer.c:4: warning: suggest parentheses around assignment
            used as truth value
```

- **Neiskorišćena promenljiva**: `if (9 == 9) ...` (a promenljiva a stoji neupotrebljena)

```
primer.c:3: warning: unused variable ‘a’
```

- **Deljenje nulom (na nivou konstante)**: `if (a / 0) ...`

```
primer.c:4: warning: division by zero
```

Standard i deljenje nulom. Celobrojno deljenje nulom ima nedefinisano ponašanje. U primeru iznad, kompilator može samo da upozori; u toku izvršavanja, većina sistema signalizira grešku.

Dobra praksa: tretirati upozorenja kao obavezna za rešavanje. U GCC-u, zajednička upozorenja se uključuju opcijom `-Wall`.

```
gcc -Wall -o program program.c
```

Primer organizacije programa u više datoteka

Kao ilustraciju programa sastavljenog iz **više datoteka** i **više jedinica prevođenja**, razmotrićemo malu korisničku biblioteku za rad sa pozitivnim razlomcima. Primer je namerno projektovan tako da pokrije što više koncepata iz ovog poglavlja (u realnim projektima neka rešenja mogu biti drugačija).

Biblioteka obezbeđuje:

- tip za reprezentovanje razlomaka `RAZLOMAK`,
- funkcije za kreiranje, skraćivanje i sabiranje razlomaka,
- mehanizam za izveštavanje grešaka preko nabrojivog tipa `RAZLOMAK_GRESKA` (`OK`, `IMENILAC_0`) i pratećih poruka `razlomak_poruka`, uz globalnu promenljivu `razlomak_greska` koja čuva ishod poslednje operacije.

Datoteka `razlomak.h` je javni interfejs biblioteke i sadrži deklaracije funkcija i promenljivih koje su na raspolaganju njenim korisnicima.

```c
#ifndef __RAZLOMAK_H_
#define __RAZLOMAK_H_

typedef struct razlomak {
  unsigned brojilac;
  unsigned imenilac;
} RAZLOMAK;

typedef enum {OK, IMENILAC_0} RAZLOMAK_GRESKA;
extern RAZLOMAK_GRESKA razlomak_greska;
```

```
extern char* razlomak_poruka[];

RAZLOMAK napravi_razlomak(unsigned br, unsigned im);
RAZLOMAK skrati_razlomak(RAZLOMAK r);
RAZLOMAK saberi_razlomke(RAZLOMAK r1, RAZLOMAK r2);

#endif
```

- Za funkcije nije potrebno pisati `extern` (podrazumevano imaju spoljašnju povezanost).
- Za globalne objekte (`razlomak_greska`, `razlomak_poruka`) `extern` jasno naglašava da su ovde samo deklaracije, a definicije su u implementaciji.
- Uobičajeni čuvari zaglavlja (eng. *include guards*) sprečavaju višestruko uključivanje iste `.h` datoteke i posledične greške. Svaka datoteka zaglavlja u tom slučaju treba da ima sledeći opšti oblik:

  ```
  #ifndef IME
  #define IME
  /* ... sadrzaj ... */
  #endif
  ```

  Prilikom prvog uključivajna ove datoteke, definiše se simboličko ime `IME` i zbog toga naredna uključivanja iste datoteke ignorišu celokupan njen sadržaj.

Implementacija biblioteke sadržana je u datoteci `razlomak.c`.

```
#include "razlomak.h"

char* razlomak_poruka[] = {
  "Operacija uspesno sprovedena",
  "Greska: imenilac je nula!"
};

static const RAZLOMAK NULA = {0, 0};
RAZLOMAK_GRESKA razlomak_greska;

static int nzd(unsigned a, unsigned b) {
    return (b == 0) ? a : nzd(b, a % b);
}

RAZLOMAK skrati_razlomak(RAZLOMAK r) {
  if (r.imenilac == 0) {
    razlomak_greska = IMENILAC_0; return NULA;
  }
  unsigned n = nzd(r.brojilac, r.imenilac);
  r.brojilac /= n;
  r.imenilac /= n;
  razlomak_greska = OK;
```

```
  return r;
}

RAZLOMAK napravi_razlomak(unsigned br, unsigned im) {
  RAZLOMAK rez;
  rez.brojilac = br;
  rez.imenilac = im;
  if (im == 0) {
    razlomak_greska = IMENILAC_0; return NULA;
  }
  return skrati_razlomak(rez);
}

RAZLOMAK saberi_razlomke(RAZLOMAK r1, RAZLOMAK r2) {
  if (r1.imenilac == 0 || r2.imenilac == 0) {
    razlomak_greska = IMENILAC_0; return NULA;
  }
  return napravi_razlomak(
     r1.brojilac*r2.imenilac + r1.imenilac*r2.brojilac,
     r1.imenilac*r2.imenilac);
}
```

- Uključujemo **zaglavlje** da prevodilac uporedi **deklaracije** i **definicije**. Iako ovaj korak nije uvek neophodan, smatra se dobrom praksom.
- `razlomak_poruka` i `razlomak_greska` su **globalne definicije**. (Poruke su pokazivači na literale; u strogoj varijanti moglo bi i `const char*`.)
- `NULA` i pomoćna funkcija `nzd` (najveći zajednički delilac) su **„privatni“** za ovaj modul: zato su označeni sa `static` (unutrašnja povezanost).
- Funkcije interfejsa obrađuju greške postavljanjem `razlomak_greska` i, u slučaju greške, vraćaju `NULA` (ovde `{0,0}` predstavlja „neispravan“ razlomak).

Program koji demonstrira upotrebu biblioteke dat je u zasebnoj datoteci `program.c`.

```
#include <stdio.h>
#include "razlomak.h"

int main(void) {
  unsigned a, b;
  RAZLOMAK r1, r2, r;

  scanf("%u%u", &a, &b);
  r1 = napravi_razlomak(a, b);
  if (razlomak_greska != OK) {
    printf("%s\n", razlomak_poruka[razlomak_greska]);
    return 1;
  }

  scanf("%u%u", &a, &b);
```

```
  r2 = napravi_razlomak(a, b);
  if (razlomak_greska != OK) {
    printf("%s\n", razlomak_poruka[razlomak_greska]);
    return 1;
  }

  r = saberi_razlomke(r1, r2);
  if (razlomak_greska != OK)
    printf("%s\n", razlomak_poruka[razlomak_greska]);

  printf("Rezultat je: %u/%u\n", r.brojilac, r.imenilac);
  return 0;
}
```

Program učitava dva razlomka, sabira ih i ispisuje rezultat. Na svakom koraku proverava `razlomak_greska`; u slučaju greške ispisuje odgovarajuću poruku iz `razlomak_poruka`. Isto zaglavlje `razlomak.h` bez problema se uključuje u obe jedinice prevođenja.

Odgovarajuća `Makefile` datoteka može da bude:

```
program : program.o razlomak.o
        gcc -o program program.o razlomak.o

program.o : program.c razlomak.h
        gcc -c -O3 -Wall program.c

razlomak.o : razlomak.c razlomak.h
        gcc -c -O3 -Wall razlomak.c
```

- Završni cilj je `program` i on zavisi od oba objektna modula.
- Svaki `.o` se ponovo gradi samo ako je izmenjena njegova `.c` datoteka ili datoteke od kojih zavisi (inkrementalna gradnja).
- `-Wall` uključuje uobičajena upozorenja; `-O3` uključuje jaku optimizaciju.

1.3.9 Organizacija izvršivog programa

Ako je od izvornog programa uspešno generisana **izvršiva verzija**, operativni sistem može da je pokrene (npr. navođenjem imena programa u komandnoj liniji). Pokretanje uključuje učitavanje programa sa spoljne memorije u radnu memoriju, dodelu adresnog prostora, eventualno dinamičko povezivanje sa *runtime* (rantajm) bibliotekom i početne inicijalizacije; zatim izvršavanje može da počne. U ovoj fazi mogu se pojaviti greške koje nije bilo moguće detektovati pri prevođenju i povezivanju.

Važno ograničenje: izvršivi program je **vezan za ciljni operativni sistem** i **arhitekturu**. Tipični primer: 64-bitna aplikacija ne može da se izvrši na 32-bitnom sistemu.

Izvršno okruženje — operativni sistem i rantajm biblioteka

Kompajlirani i povezani program ne izvršava se samostalno. On sadrži pozive ka rutinama niskog nivoa (I/O, memorija, inicijalizacija) koje se obezbeđuju kroz **rantajm biblioteku** (eng. *C runtime*

library). Pri učitavanju programa OS pokreće **dinamičko povezivanje**: simboli iz izvršnog fajla (pozivi funkcija, reference na globalne objekte) uparuju se sa implementacijama u rantajm biblioteci koja je prisutna u memoriji.

Granica između „standardne biblioteke C" i „rantajm biblioteke" u praksi nije stroga i zavisi od operativnog sistema. U operativnom sistemu Linux, funkcije C rantajm biblioteke smatraju se i delom operativnog sistema i moraju da budu podržane. U Windows okruženju C runtime se isporučuje uz kompajler (a delovi se po potrebi mogu statički ugraditi u izvršni fajl kako bi se izbegla dependencija na ciljnom sistemu).

Tipičan modul runtime-a je istorijski obeležen kao `crt0` („C Run-Time zero"), čija je uloga da pripremi proces: otvaranje standardnih tokova, inicijalizaciju segmenata, pripremu argumenata za `main`, poziv `main`, kao i završne radnje po povratku. Ulazna tačka procesa je obično funkcija (npr. `_start`) koja poziva `main` nakon pripreme okruženja.

Vrste i organizacija podataka u izvršnom programu

Tipovi u jeziku C su **koncepti na nivou izvornog jezika** (namenjeni ljudima i statičkoj proveri). Nakon prevođenja, oni se svode na **mašinski podržane reprezentacije**: cele brojeve i brojeve u pokretnom zarezu. U mašinskom kodu ne postoje „etikete tipova"; prevodilac je već odabrao konkretne instrukcije za odgovarajuće operande (npr. posebne instrukcije za celobrojno sabiranje naspram sabiranja realnih vrednosti).

Veličine osnovnih tipova i širina pokazivača zavise od **ciljne arhitekture** i **konvencije modela podataka**. Danas su uobičajeni 32-bitni i 64-bitni sistemi. Na 64-bitnom OS-u uobičajeno je izvršavati i 32-bitne programe; obrnuto nije moguće. „32-bitni softver" tipično znači da koristi 32-bitni adresni prostor ($\approx$ 4 GiB), „64-bitni softver" koristi 64-bitni.

Na 64-bitnim platformama postoje definisane konvencije koje određuju koji se tipovi proširuju na 64 bita.

tip / model	32-bit	**ILP64**	**LP64**	**LLP64**
`char`	8	8	8	8
`short`	16	16	16	16
`int`	32	**64**	32	32
`long`	32	**64**	**64**	32
`long long`	64	64	64	**64**
pokazivač (`*`)	32	**64**	**64**	**64**

- **LP64** (većina UNIX/Linux sistema, GCC/Clang na x86-64): `long` i *pointer* su 64-bitni; `int` ostaje 32-bitni.
- **LLP64** (Windows na x64, MSVC): `long` ostaje 32-bitni; `long long` i *pointer* su 64-bitni.
- **ILP64** susreće se ređe (npr. specijalizovani HPC sistemi): `int`, `long` i *pointer* su 64-bitni.

GCC podrazumevano generiše kod u istoj širini kao i sistem na kojem se prevođenje obavlja. Po potrebi, moguće je eksplicitno zatražiti 32-bitni ili 64-bitni izlaz (ukoliko je odgovarajući toolchain instaliran):

```
# 32-bitni izlaz
gcc -m32 -o program32 program.c

# 64-bitni izlaz
gcc -m64 -o program64 program.c
```

Zaglavlje <stdint.h> uvodi tipove sa **tačno određenom širinom**, npr.:

- int32_t, uint32_t — tačno 32 bita,
- int64_t, uint64_t — tačno 64 bita,
- intptr_t, uintptr_t — celobrojni tip iste širine kao **pokazivač** (korisno za kastovanje pokazivača u integer i obrnuto).

Preporučuje se upotreba ovih tipova u situacijama kada je važno precizno odrediti broj bita koji se koriste (npr. u binarnim formatima, mrežnim protokolima i interfejsima sa uređajima).

Organizacija memorije

Način organizovanja i korišćenja memorije u toku izvršavanja programa zavisi od operativnog sistema. Ovde opisujemo uobičajenu sliku koja pokriva širok spektar platformi (uz poneko pojednostavljenje).

Kada se izvrši program učita u radnu memoriju računara, biva mu dodeljena određena memorija i započinje njegovo izvršavanje. Dodeljena memorija organizovana je u nekoliko delova koje zovemo *segmenti* ili *zone*:

- **segment koda** (eng. *code/text segment*),
- **segment podataka** (eng. *data segment*),
- **stek segment** (eng. *stack segment*),
- **hip segment** (eng. *heap segment*).

U nastavku razmatramo prva tri; hip (dinamička alokacija) obrađujemo kasnije (vidi o dinamičkoj memoriji).

Veza sa životnim vekom:

- promenljive **statičkog** životnog veka - tipično u **segmentu podataka**,
- promenljive **automatskog** životnog veka - u **stek segmentu**,
- promenljive **dinamičkog** životnog veka - u **hip segmentu**.

Segment koda

U **segmentu koda** nalazi se **mašinski kod** programa: instrukcije svih funkcija, uključujući i one iz biblioteka koje su statički povezane. Na mnogim sistemima, ako je pokrenuto više instanci istog programa, sve one dele isti primerak koda (read-only mapiranje), dok svaka instanca ima sopstvene podatke i stek. Informacija „gde je izvršavanje stiglo" čuva se zasebno za svaki proces/nit.

Segment podataka

U **segmentu podataka** čuvaju se promenljive **statičkog životnog veka** (najčešće *globalne*), kao i **konstantni podaci** (npr. literali niski). Svaka instanca programa ima svoj segment podataka. Pogledajmo naredni primer.

```c
#include <stdio.h>

int a;                /* globalna - u segmentu podataka */

int main(void) {
    int b;            /* lokalna automatska - u steku segmentu*/
    static double c; /* lokalna statička - u segmentu podataka */
    printf("Zdravo\n"); /* literal niske -
                           u segmentu podataka (read-only) */
    return 0;
}
```

U ovom primeru a i c se nalaze u segmentu podataka; literal `"Zdravo\n"` takođe (obično u njegovom read-only delu). Ako se ista niska koristi na više mesta, standard ne propisuje da li će imati jednu ili više fizičkih kopija.

Stek segment

Stek (često se naziva i *programski stek*) čuva sve podatke koji opisuju **trenutna izvršavanja funkcija**. Svaki aktivni poziv funkcije dobija svoj **stek okvir** (eng. *stack frame*), koji obično sadrži:

- argumente funkcije (kada se prosleđuju preko steka),
- lokalne automatske promenljive,
- privremene međurezultate,
- adresu povratka (gde nastaviti posle `return`),
- (često) pokazivač na okvir pozivaoca.

Stek je **LIFO** struktura (eng. *last-in-first-out*): novi okvir se dodaje na vrh, a uklanja se takođe sa vrha. Okvir nastaje pri ulasku u funkciju i „nestaje" pri izlasku iz nje.

Ilustracija poziva:

- dok se izvršava `main()`: vrh steka sadrži okvir `main`;
- `main()` pozove `f()`: novi okvir za `f` postaje vrh;
- `f()` završi: uklanja se njen okvir, vrh se vraća na okvir `main`.

Vrh steka	Vrh steka	
↓	↓	Vrh steka
`main()`,	`f()`,	↓
	njeni argumenti,	`main()`,

njeni argumenti, lokalne promenljive, ...	lokalne promenljive, ... `main()`, njeni argumenti, lokalne promenljive, ...	njeni argumenti, lokalne promenljive, ...

Ovaj mehanizam prirodno podržava **rekurziju**: svaki rekurzivni poziv dobija sopstveni okvir (i sopstvene kopije lokalnih automatskih promenljivih).

Veličina steka je obično ograničena (ovo su pravila operativnog sistema i linkera). Zato ne treba smeštati velike objekte na stek. Sledeći primeri pokazuju zašto:

```
/* Primer A: rizik od prekoračenja steka (ogroman niz na steku) */
int main(void) {
    int a[1000000];  /* može izazvati „stack overflow" */
    /* ... */
}

/* Primer B: bezbednije - veliki niz izvan steka
  (globalan ili statičan) */
int a[1000000];     /* segment podataka */

int main(void) {
    /* ... */
}
```

Na mnogim sistemima primer A može srušiti program pri pokretanju, dok primer B prolazi jer se memorija rezerviše u segmentu podataka. (Na nekim alatnim lancima veličina steka se može podešavati opcijama linkera/okruženja.)

Prenos argumenata u funkciju preko steka

Ilustrujmo kako izgleda prenos argumenata funkciji na sledećem jednostavnom primeru:

```
int f(int a[], int n) {
  int i, s = 0;
  for (i = 0; i < n; i++)
     s += ++a[i];
  return s;
}

int main() {
  int a[] = {1, 2, 3};
  int s;
  s = f(a, sizeof(a)/sizeof(int));
  printf("s = %d, a = {%d, %d, %d}\n",
```

```
          s, a[0], a[1], a[2]);
  return 0;
}
```

Početno stanje steka

U funkciji `main` deklarisane su dve lokalne promenljive: niz `a` i ceo broj `s`. Na početku izvršavanja funkcije `main`, na steku postoji samo njen stek okvir:

```
main:
  114: ...
  112: ?     <- s
  108: 3
  104: 2
  100: 1     <- a
```

Pretpostavljamo da se sve lokalne promenljive čuvaju na steku. U praksi, kompilator može neke smestiti u registre procesora.

Nakon poziva funkcije `f`

Poziv `f(a, 3)` dodaje novi stek okvir iznad `main`-ovog. Argumenti se inicijalizuju: niz se prenosi kao adresa prvog elementa, dok `n` dobija vrednost 3.

```
f:
  134: ...
  130: 0     <- s
  126: ?     <- i
  122: 3     <- n
  118: 100   <- a (adresa niza)

main:
  114: ...
  112: ?     <- s
  108: 3
  104: 2
  100: 1     <- a
```

Tokom izvršavanja petlje

Stanje se menja u skladu sa izračunavanjima. Na primer, posle prve iteracije:

```
f:
  134: ...
  130: 2     <- s
  126: 0     <- i
  122: 3     <- n
  118: 100   <- a

main:
  114: ...
  112: ?     <- s
```

```
  108: 3
  104: 2
  100: 2      <- a
```

Kasnije, kada je `i = 1`:

```
f:
  134: ...
  130: 5      <- s
  126: 1      <- i
  122: 3      <- n
  118: 100    <- a

main:
  114: ...
  112: ?      <- s
  108: 3
  104: 3
  100: 2      <- a
```

I na kraju petlje (`i = 2`):

```
f:
  134: ...
  130: 9      <- s
  126: 2      <- i
  122: 3      <- n
  118: 100    <- a

main:
  114: ...
  112: ?      <- s
  108: 4
  104: 3
  100: 2      <- a
```

Primetimo da originalni niz a u funkciji `main` menja vrednosti jer se u funkciju prenosi adresa početka niza, a ne njegova kopija.

Završetak funkcije `f`

Kada `f` završi, njen stek okvir izgleda ovako:

```
f:
  134: ...
  130: 9      <- s
  126: 3      <- i
  122: 3      <- n
  118: 100    <- a
```

Povratna vrednost `s = 9` obično se upisuje u registar procesora (`eax` ili `rax`), odakle se prenosi u promenljivu `s` u `main`-u. Nakon povratka, stek se vraća u stanje:

```
main:
  114: ...
  112: 9      <- s
  108: 4
  104: 3
  100: 2      <- a
```

Na kraju se poziva `printf`. Na stek se dodaje još jedan okvir, u koji se smeštaju:

- adresa konstantne niske (iz segmenta podataka),
- kopije vrednosti promenljivih `s`, `a[0]`, `a[1]`, `a[2]`.

Posle ispisa, uklanja se okvir `printf`, a zatim i okvir `main`, čime se završava izvršavanje programa.

Implementacija rekurzije

Rekurzija predstavlja situaciju u kojoj jedna funkcija poziva samu sebe — direktno ili indirektno.

Razmotrimo, kao primer, funkciju koja rekurzivno izračunava faktorijel broja (vrednost faktorijela, naravno, može da se izračuna i iterativno, bez korišćenja rekurzije):

```c
#include <stdio.h>

int faktorijel(int n) {
  if (n <= 0)
    return 1;
  else
    return n * faktorijel(n - 1);
}

int main() {
   int n;
   while (scanf("%d", &n) == 1)
      printf("%d! = %d\n", n, faktorijel(n));
   return 0;
}
```

Ako se funkcija `faktorijel` pozove sa argumentom 5, na steku poziva će se formirati *pet nezavisnih stek okvira*, po jedan za svaku instancu funkcije.

- Svaka instanca ima drugačiju vrednost argumenta `n`.
- Iako je aktivno pet instanci funkcije, u memoriji postoji samo **jedan primerak izvršnog koda** (u *segmentu koda*).
- Svaki stek okvir pamti za svoju instancu dokle je stiglo izvršavanje, tj. koja je naredba u segmentu koda tekuća.

Primer organizacije objektnih modula i izvršivog programa

U ovom delu se detaljno razmatra proces kompilacije i povezivanja kroz analizu sadržaja objektnog modula i izvršnog programa za sledeći minimalni primer. Polazimo od pretpostavke da se koristi GCC na 32-bitnom Linux sistemu (na 64-bitnoj arhitekturi određeni aspekti adresiranja i rasporeda sekcija će se razlikovati).

```
#include <stdio.h>
int n = 10, sum;

int main(void) {
   int i;
   for (i = 0; i < n; i++)
        sum += i;
   printf("sum = %d\n", sum);
   return 0;
}
```

U ovom programu ceo izvorni kod predstavlja jednu jedinicu prevođenja. Za dobijanje izvršnog programa potrebno je:

1. prevesti datoteku u objektni modul, i
2. povezati ga sa odgovarajućim objektima iz standardne biblioteke.

Objektni modul. Kompilacijom komandom:

```
gcc -c program.c
```

dobija se datoteka `program.o`. Sadržaj objektnog modula može se analizirati pomoću alata `nm` ili `objdump`. Na primer:

```
nm --format sysv program.o
```

rezultuje tabelom simbola:

```
Name    | Value    | Class | Type    | Size     | Section
main    | 00000000| T     | FUNC    | 00000050 | .text
n       | 00000000| D     | OBJECT  | 00000004 | .data
printf  |          | U     | NOTYPE  |          | *UND*
sum     | 00000004| C     | OBJECT  | 00000004 | .bss
```

Iz ove tabele uočava se:

- `main` je funkcija u segmentu koda (`.text`) i to na njegovom samom početku (na šta ukazuje vrednost 00000000).
- `n` je inicijalizovana globalna promenljiva u segmentu podataka (`.data`) i inicijalizovan je (na šta ukazuje klasa `D`.

- printf je simbol čija je definicija nepoznata (na šta ukazuje oznaka sekcije *UND*, i klase U).
- sum je neinicijalizovana globalna promenljiva, koja se nalazi u posebnom delu segmenta podataka (.bss).

Lokalne automatske promenljive, kao što je i, ne pojavljuju se u tabeli simbola.

Mašinski kod funkcije main**.** Disasembliranjem (objdump -d program.o) može se videti generisani mašinski kod:

```
00000000 <main>:
   0:   55                      push   ebp
   1:   89 e5                   mov    ebp,esp
   3:   83 e4 f0                and    esp,0xfffffff0
   6:   83 ec 20                sub    esp,0x20
   9:   c7 44 24 1c 00 00 00    mov    DWORD PTR [esp+0x1c],0x0
  10:   00
  11:   eb 16                   jmp    29 <main+0x29>
  13:   8b 15 00 00 00 00       mov    edx,DWORD PTR ds:0x0
  19:   8b 44 24 1c             mov    eax,DWORD PTR [esp+0x1c]
  1d:   01 d0                   add    eax,edx
  1f:   a3 00 00 00 00          mov    ds:0x0,eax
  24:   83 44 24 1c 01          add    DWORD PTR [esp+0x1c],0x1
  29:   a1 00 00 00 00          mov    eax,ds:0x0
  2e:   39 44 24 1c             cmp    DWORD PTR [esp+0x1c],eax
  32:   7c df                   jl     13 <main+0x13>
  34:   a1 00 00 00 00          mov    eax,ds:0x0
  39:   89 44 24 04             mov    DWORD PTR [esp+0x4],eax
  3d:   c7 04 24 00 00 00 00    mov    DWORD PTR [esp],0x0
  44:   e8 fc ff ff ff          call   45 <main+0x45>
  49:   b8 00 00 00 00          mov    eax,0x0
  4e:   c9                      leave
  4f:   c3                      ret
```

Adrese u generisanom kodu su relativne (npr. adrese skokova su sve date u odnosu na početak funkcije main) i neke od njih će nakon povezivanja i tokom izvršavanja biti promenjene. Na primer, u instrukciji 29: mov eax,ds:0x0 vrši se kopiranje određene vrednosti iz memorije u registar eax. Trenutni kod govori da je to sadržaj sa adrese 0 (u odnosu na početak segmenta podataka ds), ali u izvršivom programu to neće ostati tako. Zadatak linkera je da na ovom mestu adresu 0 zameni adresom promenljive n (jer je zadatak ove instrukcije upravo da vrednost promenljive n prepiše u registar eax). Slično tome, na mesto poziva funkcije printf u mašinskom kodu generisan je poziv 44: call 45 <main+0x45> koji nije ispravan i nema smisla jer ukazuje upravo na adresu argumenta tog poziva (koji je na 44). Zadatak linkera je da ovaj poziv korektno razreši. Spisak stavki koje linker treba da razreši mogu da se vide komandom objdump -r program.o (tzv. relokacije) i sam kod nema nikakvo smisleno značenje dok se u obzir ne uzme tabela relokacije i dok se sve navedene stavke ne razreše (što je zadatak linkera).

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE       VALUE
```

```
00000015 R_386_32   sum
00000020 R_386_32   sum
0000002a R_386_32   n
00000035 R_386_32   sum
00000040 R_386_32   .rodata
00000045 R_386_PC32 printf
```

Iz tabele relokacija može se uočiti da linker ima sledeće zadatke:

- da na adresu 45 u segmentu koda (`.text`) upiše korektnu adresu mašinskog koda funkcije `printf`, čime poziv postaje ispravan;
- da razreši adresu promenljive `n` koja se koristi na mestu 2a u mašinskom kodu (trenutno je pretpostavljeno da se nalazi na adresi 0 u segmentu podataka, što ne mora biti slučaj u konačnom programu);
- da razreši adresu promenljive `sum` na tri mesta na kojima joj se pristupa (15, 20 i 35);
- da razreši adresu početka segmenta `.rodata` (read-only data) u kome se nalazi konstantna niska `"sum = %d\n"`, koja se koristi na mestu 40 u mašinskom kodu;
- da postavi pravu adresu segmenta koda (`.text`) u zaglavlju izvršne datoteke (`.eh_frame`).

Analiza generisanog mašinskog koda. Iako detaljna analiza mašinskog koda nije od presudnog značaja za svakodnevno programiranje u jeziku C, ona može biti korisna jer pokazuje kako se mehanizmi izvršnog okruženja, često opisani samo na apstraktan način, zaista realizuju u praksi.

Generisani mašinski kod podrazumeva da **stek u memoriji raste od viših ka nižim adresama** (eng. *downward growing*). U svakom trenutku:

- registar `esp` sadrži adresu vrha steka,
- registar `ebp` čuva adresu početka stek okvira tekuće funkcije (kako bi okvir mogao da se ukloni po završetku funkcije).

Prolog funkcije. Prve četiri instrukcije funkcije `main` predstavljaju njen *prolog*. Njegova svrha je priprema novog stek okvira:

- na stek se smešta adresa početka starog okvira,
- adresa početka novog okvira postaje trenutna vrednost registra `esp`,
- vrh steka se pomera (ka manjim adresama) kako bi se rezervisao prostor za lokalne promenljive i pomoćne podatke.

U ovom slučaju promenljiva `i` nalazi se upravo u steku, na adresi `esp+0x1c` nakon što je novi okvir formiran.

Petlja i telo funkcije.

- Naredba u liniji 9 inicijalizuje promenljivu `i` na vrednost 0.
- Nakon toga kontrola prelazi na proveru uslova u liniji 29. Najpre se vrednost promenljive `n` kopira u registar `eax`, zatim se poredi sa trenutnom vrednošću `i` (koja se nalazi u steku na adresi `esp+0x1c`).
- Ako je `i < n`, izvršava se telo petlje (instrukcije 13–23) i zatim korak inkrementacije (instrukcija 24).

Telo petlje se može opisati ovako:

1. vrednost promenljive `sum` kopira se u registar `edx`,
2. vrednost promenljive `i` kopira se u registar `eax`,
3. sabiranjem `eax + edx` dobija se novi rezultat,
4. rezultat se vraća u memoriju, na adresu promenljive `sum`.

Instrukcija u liniji 24 povećava vrednost `i` za jedan, a zatim se ponavlja uslovni skok.

Kada uslov više nije ispunjen, prelazi se na deo koda gde se poziva funkcija `printf` (instrukcije 34–48). Mehanizam izgleda ovako:

- na stek se upisuju argumenti poziva u obrnutom redosledu: najpre vrednost `sum` (preko registra `eax`), zatim adresa format niske `"sum = %d\n"`.
- instrukcija `call` smešta adresu povratka na stek i prenosi izvršavanje na adresu funkcije `printf`.

U trenutnoj verziji objektnog modula ove adrese su još uvek privremene (upisana vrednost `0`) i razrešavaju se tek tokom povezivanja.

Epilog funkcije. Funkcija `main` vraća povratnu vrednost tako što je upisuje u registar `eax` (linija 49). Nakon toga slede instrukcije `leave` i `ret`, koje čine *epilog* funkcije:

- `leave` vraća vrednost registra `esp` na početnu vrednost i restauriše adresu starog stek okvira,
- `ret` preuzima adresu povratka sa vrha steka i prenosi kontrolu pozivaocu.

Uloga linkera. Važno je naglasiti da u fazi **prevođenja** svaka referenca na spoljašnje funkcije, promenljive i konstante ostaje **nerazrešena**. Njihovo razrešavanje obavlja se tek u fazi povezivanja.

- Ako ulazna datoteka sadrži poziv funkcije ili upotrebu promenljive koja nije definisana (već samo deklarisana), objekat će se uspešno generisati, ali od njega samog neće biti moguće napraviti izvršni program.
- Linker tada pretražuje sve priložene objektne module (uključujući i standardne biblioteke) kako bi pronašao odgovarajuću definiciju.

Ako su sve reference uspešno razrešene:

- pozivi funkcija upisuju se sa njihovim stvarnim adresama,
- promenljive i konstante vezuju se za konkretne memorijske lokacije,
- i svi objektni moduli povezuju se u jedinstven izvršni program.

Rantajm biblioteka. Kod dobijen povezivanjem obično i dalje nije u potpunosti autonoman. Razlog je u tome što funkcije standardne biblioteke koriste niz rutina niskog nivoa koje se **ne ugrađuju statički** u program, već se pozivaju iz tzv. **rantajm biblioteke** tokom izvršavanja. Na taj način izvršni program ostaje manji i efikasniji.

U navedenom primeru povezivanje se obavlja komandom:

```
gcc -o program program.o
```

Dobija se izvršna datoteka `program`. Analiza (`nm`, `objdump`) pokazuje:

- u izvršnom programu sada postoji veliki broj dodatnih simbola (npr. `_start`, `printf@@GLIBC_2.0`),
- u funkciji `main` sve sporne adrese su razrešene,
- poziv `printf` vodi ka funkciji `printf@plt`, koja zatim dinamički poziva `printf@@GLIBC_2.0` iz GLIBC biblioteke.

Drugim rečima, poziv funkcije `printf` postaje ispravan tek u fazi **dinamičkog povezivanja**, na početku izvršavanja programa.

1.3.10 Greške u fazi izvršavanja

U fazi pisanja izvornog koda, kao i tokom generisanja izvršnog programa, nije uvek moguće predvideti sve greške koje se mogu javiti pri samom izvršavanju. Program može:

- davati pogrešne rezultate (najčešće zbog nepravilno odabranog algoritma ili grešaka u njegovoj implementaciji), ili
- izazvati prekid izvršavanja usled grešaka koje nastaju u toku rada.

Ove greške mogu biti posledica nedovoljnog predviđanja mogućih situacija od strane programera, ali i zavisiti od uslova izvršavanja i okruženja.

Nedozvoljeni pristup memoriji. Jedna od najčešćih grešaka u fazi izvršavanja jeste pokušaj nedozvoljenog pristupa memoriji. Ova greška nastaje:

- kada se pokušava pristupiti memoriji koja nije dodeljena programu, ili
- kada se pokuša upis u deo memorije koji je dodeljen samo za čitanje (*read-only*).

U ovakvim situacijama operativni sistem detektuje problem i šalje signal koji obično dovodi do prekida programa, uz poruku:

```
Segmentation fault
```

Najčešći uzroci:

- pristup elementu van granica niza (npr. ako je niz deklarisan kao `int a[10];`, a pokušava se pristupiti elementu `a[1000]`),
- dereferenciranje nealociranog pokazivača (npr. `NULL` pokazivača),
- prepunjenje steka (*stack overflow*) usled prevelikog broja funkcijskih poziva (obično nastalog neuspešno zaustavljenom rekurzijom ili definisanjem prevelikih lokalnih automatskih promenljivih).

Odgovornost za ove greške je uglavnom na programeru.

Deljenje nulom. Ako tokom izvršavanja dođe do pokušaja celobrojnog deljenja nulom, program se prekida uz prijavu:

```
Floating point exception
```

Ova greška nastaje zato što autor programa nije predvideo mogućnost deljenja nulom niti obezbedio mehanizam zaštite.

Druge greške u toku rada. Greške u fazi izvršavanja mogu nastati i u komunikaciji sa spoljnim uređajima i perifernim resursima. Primer je pokušaj čitanja iz datoteke koja ne postoji na disku ili pristup drugom nedostupnom resursu.

1.4 Pokazivači

U jeziku C, pokazivači imaju veoma značajnu ulogu i teško je napisati iole kompleksniji program bez njihove upotrebe. Dodatne mogućnosti u radu sa pokazivačima daje **dinamička alokacija memorije**. Programer, u radu sa pokazivačima, ima veliku slobodu i širok spektar mogućnosti. To otvara prostor za efikasno programiranje, ali i za veliki broj potencijalnih grešaka.

1.4.1 Adrese i pokazivači

Memorija računara organizovana je u niz uzastopnih bajtova. Više bajtova može se tretirati kao jedinstven podatak — na primer, 4 uzastopna bajta (ili 8, zavisno od sistema) mogu se tretirati kao celobrojni tip.

Pokazivači (*engl. pointer*) su tip podataka u C-u čije su vrednosti memorijske adrese.

- Na 16-bitnim sistemima adrese zauzimaju 2 bajta,
- na 32-bitnim 4 bajta,
- na 64-bitnim 8 bajtova.

Iako su adrese suštinski celi brojevi, pokazivački tipovi se razlikuju od celobrojnih i ne mogu se mešati sa njima.

Za svaki tip podataka (osnovni ili korisnički) postoji odgovarajući pokazivački tip. Pokazivači implicitno čuvaju informaciju o tipu na koji ukazuju (osim tipa `void*`).

> **Napomena:** Informacija o tipu postoji samo tokom kompilacije. Tokom izvršavanja programa, pokazivač zauzima onoliko bajtova koliko je potrebno da se smesti adresa.

Deklaracija pokazivača

Pokazivač na `int` zapisuje se kao `int *`. Razmak nije bitan — zvezdica se vezuje za identifikator:

```
int *p1;
int* p2;
int* p3, p4;
```

- `p1`, `p2` i `p3` su pokazivači na `int`.
- `p4` je običan `int`, jer se zvezdica vezuje samo za jedan identifikator.

Operator & – operator referenciranja

Pokazivačka promenljiva može sadržati adresu promenljive ili elementa niza. Za to se koristi unarni operator & (*adresni operator*).

Primer:

```
int a = 10, *p;
p = &a;
```

- a dobija vrednost 10,
- p dobija adresu promenljive a,
- kažemo da p pokazuje na a.

Operator * – dereferenciranje

Operator * (*dereferenciranje*) primenjuje se na pokazivač i vraća sadržaj lokacije na koju on pokazuje. Dereferencirani pokazivač može biti i l-vrednost, što znači da se može menjati vrednost podataka na koje ukazuje.

Primer:

```
int a = 10, *p;
p = &a;
*p = 5;
```

Nakon izvršavanja:

- p pokazuje na a,
- vrednost promenljive a postaje 5.

Dereferencirani pokazivač se može koristiti svuda gde i tip na koji ukazuje. Na primer:

```
*p = *p + a + 3;
```

Pokazivači i celobrojne vrednosti

Pokazivački i celobrojni tipovi nisu isto:

```
int *pa, a;
pa = a;   // neispravno (implicitna konverzija, vodi do grešaka)
a = pa;   // takođe neispravno
pa = 1234; // pogrešno dodeljivanje proizvoljne adrese
```

U ovim primerima, kompilacija prolazi uz upozorenje ali upis proizvoljnog broja na mesto adrese će veoma verovatno prouzrokovati prekid rada programa prilikom pokušaja pristupa toj adresi.

Moguće su eksplicitne konverzije:

```
a = (int)pa;
pa = (int*)a;
```

ali treba ih koristiti samo u retkim i dobro opravdanim situacijama.

Jedina celobrojna vrednost koja se može koristiti i kao pokazivačka vrednost je vrednost **0**. Moguće je dodeliti nulu pokazivačkoj promenljivoj i porediti pokazivač sa nulom. Pokazivač koji ima vrednost `0` ne pokazuje ni na šta smisleno i ne sme se dereferencirati (dovodi do greške — *segmentation fault*).

Umesto `0` obično se koristi simbolička konstanta `NULL` (definisana u `<stdio.h>`).

Konverzije pokazivača

Moguće je konvertovati jedan pokazivački tip u drugi:

```
int a;
char* p = (char*)&a;
```

ili čak:

```
int a;
char* p = &a;  // upozorenje
```

Pri tome se konvertuje samo adresa, a ne i podaci na koje se pokazuje, što često može dovesti do neželjenih efekata, te stoga ovakve konverzije pokazivača nećemo često koristiti.

> **Važno:** konverzije pokazivača na podatke u pokazivače na funkcije nisu dozvoljene i dovode do nedefinisanog ponašanja.

Tip `void*`

Za "opšti" pokazivač koristi se tip `void*`.

- Može se konvertovati u bilo koji drugi pokazivački tip.
- Ne može se direktno dereferencirati, jer tip i veličina podataka nisu poznati.
- Pre dereferenciranja, `void*` mora biti konvertovan u konkretan tip.
- Nad `void*` pokazivačima nije dozvoljena aritmetika pokazivača.

`const` i pokazivači

Kombinovanjem ključne reči `const` i pokazivača dobijamo različita značenja:

Deklaracija	Značenje	Šta je konstantno
`const int* p`	Pokazivač na **konstantan int**	Vrednost
`int *const p`	**Konstantan pokazivač** na promenljivu tipa `int`	Pokazivač
`const int *const p`	**Konstantan pokazivač** na **konstantan int**	Oboje

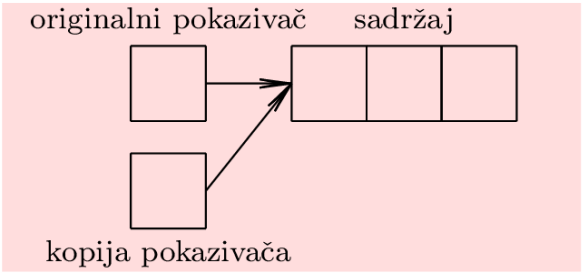
Trik za čitanje: čitaj unazad — „pokazivač na int koji je konstantan" ili „konstantan pokazivač na int".

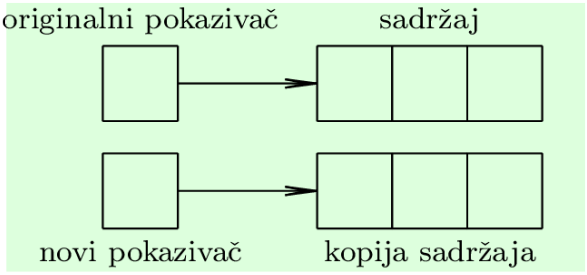
Plitko i duboko kopiranje

Ako se vrednost pokazivača p dodeli pokazivaču q, onda oba pokazuju na isti sadržaj. To se zove **plitko kopiranje**.

Za razliku od toga, **duboko kopiranje** podrazumeva da se sadržaj kopira na novu adresu, a q usmeri na tu kopiju.

Plitko kopiranje je čest uzrok grešaka jer nakon plitkog kopiranja oba pokazivača ukazuju isti sadržaj i izmenom sadržaja preko jednog pokazivača, možda neočekivano, biva izmenjen sadržaj na koji ukazuje i drugi pokazivač.





Ilustracija plitkog i dubokog kopiranja

Poravnanje u memoriji

Korišćenjem pokazivača moguće je steći uvid u način smeštanja promenljivih u memoriju računara. Podaci u memoriji često moraju biti poravnati. Na primer, tip `int` (4 bajta) mora počinjati na adresi deljivoj sa 4.

To je važno kod konverzija pokazivača:

- `char*` može pokazivati bilo gde,
- ali `int*` mora biti poravnat na adresu deljivu sa 4,
- pa eksplicitna konverzija može dovesti do problema na sistemu.

Na isti sadržaj u memoriji može se pristupiti na više načina, npr:

```
int a = 10, *p = &a;
```

Ovde su a i `*p` **alijasi** (druga imena za istu memorijsku lokaciju), a ova pojava se naziva *alijasovanje (preklapanje)* (engl. *aliasing*).

Alijasovanje otežava analizu i optimizaciju koda. Zato standard C uvodi **pravilo striktnog alijasovanja** (*strict aliasing rule*):

- pokazivači različitih tipova ne smeju ukazivati na isti sadržaj (izuzeci: `char*`, `void*`, delimično pokazivači na strukture/unije).

Primer koji krši ovo pravilo:

```
float f = 3.5f;
printf("%x\n", *( (unsigned*)&f ));
```

Ispravan način je korišćenje **unije**:

```
union { float x; unsigned y; } u;
u.x = 3.5f;
printf("%x", u.y);
```

I sledeći primer ilustruje pravila striktnog alijasovanja.

```
int a;
void f(double *p) {
   a = 1;
   *p = 1.234;
   g(a);
}
```

Pod pravilima **striktnog alijasovanja**, može se pretpostaviti da se izmenom vrednosti `*p` u prethodnom kodu **ne vrši posredno izmena** promenljive `a` (jer su `*p` i `a` različitih tipova i ne mogu biti alijasovani i preklopljeni u memoriji).

Zbog toga kompajler može da izvrši **optimizaciju koda** koja pretpostavlja da se funkciji g uvek šalje vrednost 1.

Ali ako bi se pokušao poziv:

```
f((double*)&a);
```

pokušana je izmena promenljive a. Ovakav poziv dovodi do alijasovanja dva objekta. Iako je sintaksički dopuštena konverzija pokazivača, kompajler upozorava da se time narušava pravilo striktnog alijasovanja. To može dovesti do neispravnog ponašanja koda zbog optimizacija.

Naredni primer pokazuje kako poravnanje bajtova utiče na strukture. Kompajler često umeće dodatne bajtove unutar strukture kako bi obezbedio da svako polje počne na adresi koja zadovoljava zahteve poravnanja. Time se strukture mogu povećati u odnosu na prost zbir veličina svojih polja. Sledeći primer ilustruje razliku u rasporedu polja:

```
struct A {
    char c;   // 1 bajt
    int  x;   // 4 bajta
    char d;   // 1 bajt
};

struct B {
    char c;   // 1 bajt
    char d;   // 1 bajt
    int  x;   // 4 bajta
};
```

```
printf("sizeof(struct A) = %zu\n", sizeof(struct A));
printf("sizeof(struct B) = %zu\n", sizeof(struct B));
```

Na većini 32-bitnih i 64-bitnih sistema, rezultat će biti:

```
sizeof(struct A) = 12
sizeof(struct B) = 8
```

Razlog je sledeći:

- U **struct A**, između polja c i x dodaju se 3 bajta popune da bi x počeo na adresi deljivoj sa 4; struktura zatim dobija još 3 bajta popune na kraju.
- U **struct B**, polja c i d stoje uzastopno, pa je potrebno samo 2 bajta popune pre polja x.

Raspored polja u strukturi stoga utiče na njenu ukupnu memorijsku veličinu.

Pre uvođenja ovog pravila, kompajleri su veoma restriktivno vršili optimizaciju koda. Uvođenjem pravila striktnog alijasovanja omogućene su **mnogo agresivnije optimizacije**, što dovodi do znatno bržeg koda — pod uslovom da programer poštuje ovo pravilo.

1.4.2 Pokazivači i argumenti funkcija

U jeziku C argumenti funkcija se prenose po vrednosti (uz napomenu da u slučaju nizova, ne prenosi se ceo niz, već samo adresa njegovog početka.). To znači da promenljiva koja se koristi kao argument funkcije ostaje **nepromenjena** nakon poziva funkcije.

Neuspešana razmena vrednosti promenljivih:

```
void razmeni(int a, int b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

Ako pozovemo funkciju na sledeći način:

```
razmeni(x, y);
```

gde su x i y promenljive tipa int, dešava se sledeće:

- kreira se stek okvir za funkciju swap,
- obezbeđuje se prostor za argumente a i b,
- vrednosti x i y se kopiraju u te promenljive,
- funkcija radi samo sa kopijama.
- **Zaključak:** Nakon povratka iz funkcije, vrednosti x i y ostaju **nepromenjene**.

Ispravana funkcija razmeni sa pokazivačima:

```
void razmeni(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}
```

Sada argumenti nisu `int`, već `int*`. Poziv mora izgledati ovako:

```
razmeni(&x, &y);
```

Tokom izvršavanja:

- kreira se stek okvir za `swap`,
- obezbeđuje se prostor za pokazivačke argumente `pa` i `pb`,
- vrednosti `&x` i `&y` se kopiraju u te promenljive,
- funkcija koristi kopije pokazivača, ali preko njih menja originalne vrednosti `x` i `y`.

Oprez: Argumenti se i dalje **prenose po vrednosti** (kopiraju se pokazivači), ali se zahvaljujući dereferenciranju menjaju originalne promenljive.

Napomena: Ovaj isti mehanizam koristi i funkcija `scanf`.

Prenos pokazivača može se iskoristiti i da funkcija vrati **više različitih vrednosti**.

Primer: količnik i ostatak pri deljenju dva cela broja (ne koristeći pritom operatore \ i %):

```
#include <stdio.h>

void deljenje(unsigned a, unsigned b,
              unsigned* pk, unsigned* po) {
  *pk = 0;
  *po = a;
  while (*po >= b) {
    (*pk)++;
    *po -= b;
  }
}

int main() {
  unsigned k, o;
  deljenje(14, 3, &k, &o);
  printf("%d %d\n", k, o);  // ispisuje 4 2
  return 0;
}
```

Ovde funkcija `deljenje` vraća i količnik (`k`) i ostatak (`o`) preko pokazivača.

1.4.3 Pokazivačka aritmetika

Pokazivačka aritmetika se razlikuje od običnog izračunavanja. Izraz `p+1` ne znači „dodaj 1 na adresu“, već:

> **dodaj veličinu jednog objekta tipa na koji pokazivač `p` ukazuje**.

- Ako je `p` pokazivač na `int`, onda se `p+1` razlikuje od `p` za `sizeof(int)` bajtova.
- Ako `p` sadrži adresu 100 i `sizeof(int) == 4`, onda je `p+3 = 100 + 3*4 = 112`.

Dozvoljene operacije:

- `p + n` → pomeraj unapred za `n` elemenata.
- `p - n` → pomeraj unazad za `n` elemenata.
- `p1 - p2` → razlika pokazivača (u elementima, ne u bajtovima).
- `++p`, `p++`, `--p`, `p--` → pomeranje pokazivača.
- **Nije dozvoljeno** sabirati dva pokazivača (`p1 + p2`).

> **Napomena:** Kada oduzimamo dva pokazivača, rezultat je celobrojna vrednost tipa `ptrdiff_t`, tj. razlika u **broju elemenata**, a ne bajtova.

Moguće je i porediti pokazivače i mogu se porediti samo pokazivači **istog tipa**:

- `p1 < p2` → tačno ako `p1` ukazuje na „raniji“ element niza od `p2`.
- `p1 == p2` → tačno ako ukazuju na isti element.

Unarni operatori `&` i `*` imaju viši prioritet od binarnih aritmetičkih operatora. Zato je značenje izraza `*p+1` zbir sadržaja lokacije na koju ukazuje `p` i vrednosti 1 (a ne sadržaj na adresi `p+1`).

Drugi česti izvor grešaka je primena unarnog operatora `*` i prefiksnih ili postfiksnih operatora za inkrementiranje ili dekrementiranje. Pregled redosleda izvršavanja dat je u tabeli.

Izraz	Značenje	Šta se menja?
`++*p`	Prvo se dereferencira `p`, pa se vrednost na toj adresi uveća.	Menja se **vrednost** (*p).
`(*p)++`	Dereferencira se `p`, koristi se trenutna vrednost, pa se vrednost uveća.	Menja se **vrednost** (*p).
`*p++`	Vraća vrednost na koju `p` pokazuje, a zatim se `p` pomera na sledeći element.	Menja se **adresa** (p).
`*(p++)`	Isto što i `*p++`, samo eksplicitno kroz zagrade.	Menja se **adresa** (p).

> **Oprez:** Sitni detalji prioriteta operatora lako zbunjuju, zato je često bolje koristiti zagrade radi jasnoće.

1.4.4 Pokazivači na funkcije

Do sada smo funkcije koristili na klasičan način – kao gotove blokove koda koje pozivamo po imenu. Ipak, u programiranju često želimo da funkciju **prosledimo kao argument** drugoj funkciji, da bismo omogućili različita ponašanja bez ponavljanja koda.

U jezicima poput C++ ili Pythona ovo se često rešava pomoću *funktora*, *lambda izraza*, a mogu se koristiti i *šabloni* da se smanji ponavljanje koda.

U jeziku C takav mehanizam ne postoji direktno, odnosno u jeziku C funkcije ne mogu direktno:

- da se prosleđuju kao argumenti drugim funkcijama,
- da se vraćaju kao rezultati funkcija,
- niti da se dodeljuju promenljivama.

Ali, ovo ponašanje se može postići sa pokazivačima na funkcije. Pokazivač na funkciju je, praktično, **adresa na kojoj funkcija počinje u memoriji**, a time i mogućnost da tu funkciju pozovemo posredno, bez direktnog navođenja njenog imena. U narednim primerima pokazaćemo kako se ovakvi pokazivači deklarišu i koriste.

Primer: bez pokazivača na funkcije

Razmotrimo program u kojem svaka funkcija transformiše elemente jednog niza i rezultat upisuje u drugi niz:

```c
#include <stdio.h>

void inc1(int a[], int n, int b[]) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = a[i] + 1;
}

void mul2(int a[], int n, int b[]) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = a[i] * 2;
}

void even0(int a[], int n, int b[]) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = a[i] % 2 == 0 ? 0 : a[i];
}

void print(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
     printf("%d ", a[i]);
  putchar('\n');
}
```

```
#define N 8
int main() {
  int a[N] = {1, 2, 3, 4, 5, 6, 7, 8}, b[N];

  inc1(a, N, b);   print(b, N);
  mul2(a, N, b);   print(b, N);
  even0(a, N, b);  print(b, N);

  return 0;
}
```

Sve tri funkcije (`inc1`, `mul2`, `even0`) rade istu stvar u opštem smislu: kopiraju elemente niza `a` u niz `b`, ali pri tome primenjuju različitu transformaciju.

Ovo sugeriše da bi bilo korisno napraviti *jednu opštu funkciju* kojoj se kao argument prosleđuje operacija transformacije.

Primer: sa pokazivačima na funkcije

```
#include <stdio.h>

void map(int a[], int n, int b[], int (*f)(int)) {
  int i;
  for (i = 0; i < n; i++)
     b[i] = (*f)(a[i]);
}

int inc1(int x)    { return x + 1; }
int mul2(int x)    { return 2 * x; }
int parni0(int x)  { return x % 2 == 0 ? 0 : x; }

void ispisi(int a[], int n) {
  int i;
  for (i = 0; i < n; i++)
     printf("%d ", a[i]);
  putchar('\n');
}

#define N 8
int main() {
  int a[N] = {1, 2, 3, 4, 5, 6, 7, 8}, b[N];

  map(a, N, b, &inc1);   ispisi(b, N);
  map(a, N, b, &mul2);   ispisi(b, N);
  map(a, N, b, &parni0); ispisi(b, N);

  return 0;
}
```

Ovde funkcija `map` prima pokazivač na funkciju tipa `int (*)(int)` – tj. funkciju koja prima jedan `int` i vraća `int`. Time je zajednički postupak izdvojen, a transformacija prosleđena kao parametar.

Deklaracije pokazivača na funkcije

Deklaracija pokazivača na funkciju ima oblik:

```
double *a(double, int);     // a je funkcija
double (*b)(double, int);   // b je pokazivač na funkciju
```

- a je funkcija koja prima `(double, int)` i vraća `double*`.
- b je pokazivač na funkciju koja prima `(double, int)` i vraća `double`.

Zagrade oko `*b` su **obavezne**, jer inače deklaracija menja značenje.

Operacije sa pokazivačima na funkcije

Najčešće operacije su:

- **referenciranje** funkcije (`&f`)
- **dereferenciranje** pokazivača (`(*p)()`).

Međutim, u praksi se oba znaka često izostavljaju, pa je dovoljno navesti samo ime funkcije ili ime pokazivača. Na primer:

```
map(a, N, b, inc1);     // umesto &inc1
b[i] = f(a[i]);         // umesto (*f)(a[i])
```

Zaključak: Pokazivači na funkcije omogućavaju da funkcije tretiramo kao podatke – prosleđujemo ih, čuvamo u nizovima i pravimo opštije algoritme.

1.5 Pokazivači i nizovi, dinamička alokacija memorije

1.5.1 Pokazivači i nizovi

Postoji čvrsta veza između pokazivača i nizova. Operacije nad nizovima mogu se iskazati i u terminima pokazivača, i u praksi se ova dva pristupa često ne razlikuju.

Deklaracija i veličina niza

```
int a[10];
```

- Deklariše niz od 10 elemenata tipa `int`.
- Izraz `sizeof(a)` ima vrednost `10 * sizeof(int)`.
- Elementi `a[0] ... a[9]` su u memoriji poređani uzastopno.

Tokom kompilacije ime niza (`a`):

- dobija informaciju o adresi prvog elementa,
- tipu elemenata (`int`),
- broju elemenata (10).

Napomena: Informacije o tipu i dužini postoje samo u fazi kompilacije, ali ne zauzimaju memoriju u toku izvršavanja.

Imena nizova vs. pokazivači

- Pokazivači su **izmenljive l-vrednosti**.
- Imena nizova to **nisu** — ime niza uvek ukazuje na isti prostor rezervisan samo za elemente niza.

Izraz a nije tipa pokazivača, ali se često ponaša kao da jeste.

- U većini slučajeva ime niza tipa `T` se **implicitno konvertuje** u pokazivač tipa `T*`. Vrednosti a odgovara pokazivač na prvi element niza
- Izuzetak su situacije kada se a koristi kao argument operatora `sizeof` ili `&`.

Dakle:

- `a` $\rightarrow$ implicitno se konvertuje u `int*` (pokazivač na prvi element),
- `a + 1` $\rightarrow$ pokazivač na drugi element,
- `a[i]` `*(a+i)`,
- `&a[i]` `a+i`.

Oprez: Kompajler **ne proverava granice niza**! Sintaktički je dozvoljeno `a[100]`, `*(a-100)` i slično, ali to dovodi do nepredvidivog ponašanja i često do prekida rada programa.

Indeksiranje pokazivača

Ako je `p` pokazivač (na primer, `int* p`), moguće je koristiti indeksni pristup:

```
p[3];
```

To je potpuno ekvivalentno izrazu:

```
*(p+3);
```

Definicija: Bez obzira da li je `x` niz ili pokazivač, uvek važi:

- `x[n]` `*(x+n)`
- `&x[n]` `x+n`

Ako je `sizeof(int) == 4` i `p` pokazuje na adresu 100:

- `p[3]` je element na adresi `100 + 3*4 = 112` (o pokazivačkoj aritmetici će biti reči kasnije).
- Isto važi i za `*(p+3)`.

Ovako tesna veza pokazivača i nizova daje puno prednosti (na primer, iako funkcije ne primaju niz već samo pokazivač na prvi element, implementacija tih funkcija može to da zanemari i sve vreme da koristi indeksni pristup kao da je u pitanju niz, a ne pokazivač, što je korišćeno u ranijim poglavljima). Međutim, ovo je i čest izvor grešaka, najčešće prilikom alokacije memorije. Na primer,

```
int a[10];
int *b;

a[3] = 5;  // ispravno
b[3] = 8;  // sintaksički prolazi, ali greška pri izvršavanju
```

- Niz `a` je pravilno alociran i elementu na poziciji 3 se može pristupiti.
- Pokazivač `b` nije inicijalizovan niti mu je dodeljena validna adresa.

> **Oprez:** Kod `b[3]` kompajler neće prijaviti grešku, on bi najverovatnije doveo do greške prilikom izvršavanja programa.

Zato, svaki put kada se koristi operator indeksnog pristupa, potrebno je osigurati ili proveriti da je memorija alocirana na odgovarajući način (jer kompilator ne vrši takve provere).

Razlika između `a` i `&a`

- `a` ima tip `int [10]`, koji se po potrebi može konvertovati u `int*`.
- `&a` ima tip `int (*)[10]` (pokazivač na niz od 10 `int`-ova).

U narednom primeru ilistruje se ova razlika:

```
int a[10];
int *b[10];
int (*c)[10];

c = &a;   // ispravno
(*c)[0] == a[0];  // tačno
```

Promenljiva	Značenje	Zauzima prostor za
a	Niz od 10 `int`-ova	10 * sizeof(int)
b	Niz od 10 pokazivača na `int`	10 * sizeof(int*)
c	Jedan pokazivač na niz od 10 `int`-ova (`int (*)[10]`)	sizeof(int*)

> **Oprez:** Ako se `c` deklaruje kao `int (*c)[5];`, dodela `c = &a;` proći će uz upozorenje, ali tipovi se ne poklapaju i to treba izbegavati.

Nizovi kao argumenti funkcija

Niz se ne može preneti kao argument funkcije. Umesto toga, kao argument funkcije se može navesti ime niza i time se prenosi samo **pokazivač na prvi element**.

Primer deklaracije funkcije:

```
int f(int x[]);
int f(int *x);  // ekvivalentno
```

Ovim se u funkciju kao argument prenosi (kao i uvek — po vrednosti) samo adresa početka niza, ali ne i informacija o dužini niza. Kako se kao argument nikada ne prenosi čitav niz, već samo adresa početka, moguće je umesto adrese početka proslediti i pokazivač na bilo koji element niza, kao i bilo koji drugi pokazivač odgovarajućeg tipa. Na primer, ukoliko je a ime niza i ako funkcija f ima prototip `int f(int x[]);`:

```
f(a);        // adresa prvog elementa
f(&a[2]);    // adresa trećeg elementa
f(a+2);      // ekvivalentno prethodnom
```

Napomena: Funkcija nema informaciju o broju elemenata niza nakon prosleđene adrese — programer mora da obezbedi tu informaciju.

Vezu pokazivača i nizova i pokazivačku aritmetiku ilustruje naredni jednostavni primer.

```
#include <stdio.h>
int main() {
  int a[] = {8, 7, 6, 5, 4, 3, 2, 1};
  int *p1, *p2, *p3;

  // Pristup elementima niza
  printf("%d %d %d\n", a[0], a[3], a[5]);

  // Inicijalizacija pokazivača (ime niza konvertuje se u pokazivač)
  p1 = a; p2 = &a[3]; p3 = a + 5;
  printf("%d %d %d\n", *p1, *p2, *p3);

  // Pokazivači koriste nizovsku sintaksu
  printf("%d %d %d\n", p1[1], p2[2], p3[-1]);

  // Pokazivačka aritmetika
  p1++; --p2; p3 += 2; // a++ nije dozvoljeno
  printf("%d %d %d %lu\n", *p1, *p2, *p3, p3 - p1);

  // Prioriteti * i ++
  *p1++;   printf("%d ", *p1);
  *(p1++); printf("%d ", *p1);
  (*p1)++; printf("%d\n", *p1);
```

```
  return 0;
}
```

Izlaz programa:

```
8 5 3
8 5 3
7 3 4
7 6 1 6
6 5 6
```

1.5.2 Pokazivači i niske

Doslovne tj. **konstantne niske** (eng. *string literal*), na primer `"informatika"`, u memoriji su realizovane slično kao nizovi karaktera: karakteri su poređani redom na uzastopnim memorijskim lokacijama i iza njih se nalazi završna nula (`'\0'`) koja označava kraj niske.

Konstantne niske se uvek čuvaju u **segmentu podataka**. Kada se u pozivu funkcije navede konstantna niska (na primer, u pozivu `printf("%d", x);`), funkcija umesto same niske zapravo dobija **adresu početka konstantne niske** u segmentu podataka.

Razmotrimo deklaraciju:

```
char *p = "informatika";
```

- Ako se ova deklaracija nalazi u okviru neke funkcije, u steku se rezerviše prostor samo za promenljivu `p`, ali ne i za samu nisku `"informatika"` – ona se čuva u segmentu podataka, a `p` ukazuje na nju.
- Ako je deklaracija globalna, promenljiva `p` se takođe čuva u segmentu podataka i ukazuje na `"informatika"`.

U oba slučaja:

- `p` ukazuje na početak niske,
- `p[0] == 'i'`, `p[1] == 'n'`, i tako dalje.

Pokušaj da se menja sadržaj niske (npr. `p[0] = 'x';`) prolazi fazu prevođenja, ali pri izvršavanju dovodi do nedefinisanog ponašanja (najčešće do greške i prekida programa). Sa druge strane, promena pokazivača `p` je dozvoljena (npr. `p++;`). Dakle, `p` jeste izmenljiva l-vrednost, ali `p[0]` nije.

Razlika nastaje ako se koristi niz:

```
char a[] = "informatika";
```

- Kreira se niz `a` dužine 12 (uključujući `'\0'`).
- Dozvoljena je promena vrednosti `a[0]`, `a[1]`, …, `a[9]`.

- Nije dozvoljena promena pokazivača a – dakle, a nije izmenljiva l-vrednost, ali a[0] jeste.

Primer različitih implementacija standardne funkcije `strlen`, koja vraća dužinu niske.

Prva verzija:

```
size_t strlen(const char *s) {
   size_t n;
   for (n = 0; *s != '\0'; s++)
      n++;
   return n;
}
```

Druga verzija (uz pomoć pokazivačkog oduzimanja):

```
size_t strlen(const char *s) {
   const char* t = s;
   while (*s)
      s++;
   return s - t;
}
```

Zbog neposredne veze nizova i pokazivača, `strlen` se može pozvati:

- nad konstantnom niskom → `strlen("informatika");`
- nad imenom niza → `strlen(a);`
- nad pokazivačem → `strlen(p);`

Primer funkcije `strcpy`

Funkcija `strcpy` kopira sadržaj jedne niske u drugu. Bitno je razumeti da **nije dovoljno prekopirati samo pokazivač**, jer bi to značilo da oba pokazivača ukazuju na istu memoriju (plitko kopiranje).

Neispravni primeri:

```
char s[] = "abcd";
char *t;
t = s;   // samo usmeravanje pokazivača, nema kopiranja!
```

```
char s[] = "abcd";
char t[5];
t = s;   // nije dopušteno dodeljivati nizove!
```

Ispravno – korišćenje `strcpy`:

```
char s[] = "abcd";
char t[5];
strcpy(t, s);
```

Prva verzija implementacije `strcpy`:

```
void strcpy(char *s, const char *t) {
   while ((*s = *t) != '\0') {
      s++;
      t++;
   }
}
```

Oba pokazivača uvećavaju se za po jedan (operatorom inkrementiranja), sve dok drugi pokazivač ne dođe do završne nule, tj.~do kraja niske (kako se argumenti prenose po vrednosti, promenljivama `s` i `t` se može menjati vrednost, a da to ne utiče na originalne pokazivače ili na nizove čije su adrese prosleđene kao argumenti).

Kraća verzija (inkrement u okviru petlje): Inkrementiranje koje se koristi u navedenom kodu, može se izvršiti (u postfiksnom obliku) i u okviru same `while` petlje.

```
void strcpy(char *s, const char *t) {
   while ((*s++ = *t++) != '\0');
}
```

Najkraća verzija (bez eksplicitnog poređenja sa nulom): Konačno, poređenje sa nulom može biti izostavljeno jer svaka ne-nula vrednost ima istinitosnu vrednost *tačno*.

```
void strcpy(char *s, const char *t) {
   while (*s++ = *t++);
}
```

> U svim verzijama: `const` u definiciji drugog argumenta (`const char *t`) obezbeđuje da se sadržaj niske iz koje se kopira **ne može menjati**.

1.5.3 Nizovi pokazivača i višedimenzioni nizovi

Kao što je već pomenuto, **ime niza tipa** `T` (osim kada se koristi kao argument operatora `sizeof` ili `&`) konvertuje se u **pokazivač na** `T`.

Ovo važi i za **višedimenzione nizove**:

- ime niza tipa `T a[d1][d2]...[dn]` konvertuje se u pokazivač na `(n-1)`-dimenzioni niz, tj. tipa `T (*)[d2]...[dn]`.

Ova konverzija se dešava, na primer, kada se višedimenzioni niz prenosi kao argument funkciji.

Razmotrimo deklaraciju:

```
int a[10][20];
```

- Svaki izraz `a[0]`, `a[1]`, `a[2]`, ... označava niz od 20 elemenata tipa `int` → tip je `int[20]`.
- Po potrebi, ovaj tip se implicitno konvertuje u `int*`.
- `a[0]` sadrži adresu elementa `a[0][0]`, opštije `a[i]` sadrži adresu `a[i][0]`.

Važno:

- vrednost `a` se (osim u slučaju `sizeof` ili `&`) konvertuje u tip `int (*)[20]` (pokazivač na niz od 20 elemenata).
- izrazi `&a[0]`, `&a[1]`, ... su istog tipa → `int (*)[20]`.

Razlika:

- `a` i `a[0]` imaju **istu vrednost** (adresu početnog elementa),
- ali različite tipove: `a` → `int (*)[20]`, dok `a[0]` → `int*`.

Pokazivačka aritmetika važi u skladu sa tim:

- `a + i` je jednako `a[i]`
- `a[0] + i` je jednako `&a[0][i]`.

Dvodimenzioni niz naspram niza pokazivača

Uporedimo sledeće deklaracije:

```
int a[10][20];
int *b[10];
```

- `a[3][4]` i `b[3][4]` su sintaksički ispravni izrazi za element tipa `int`.
- Ali postoji ključna razlika:

Dvodimenzioni niz `a`:

- rezervisano je tačno 200 lokacija (`10 * 20`) za `int`,
- elementi su smešteni **kontinuirano** u memoriji,
- pristup elementu `a[v][k]` svodi se na računicu `20 * v + k`.

Niz pokazivača `b`:

- alocirano je samo 10 pokazivača (ali ne i podaci na koje ukazuju),
- pokazivače je potrebno eksplicitno inicijalizovati (statički ili dinamički),
- u memoriji će biti prostor za podatke na koje pokazuju + prostor za same pokazivače,
- pokazivači **ne moraju pokazivati** na uzastopne lokacije.

Prednost niza pokazivača je fleksibilnost: različiti pokazivači mogu upućivati na nizove različite dužine, ili čak na `NULL`.

Primer: Razmotrimo primer niza koji treba da sadrži imena meseci.

Rešenje 1 – dvodimenzioni niz:

```
char meseci[][10] = {
   "Greska", "Januar", "Februar", "Mart", "April",
   "Maj", "Jun", "Jul", "Avgust", "Septembar",
   "Oktobar", "Novembar", "Decembar"
};
```

- Svi nizovi imaju istu dužinu (10), iako stvarna imena zauzimaju manje mesta.
- Ostatak memorije ostaje neiskorišćen.

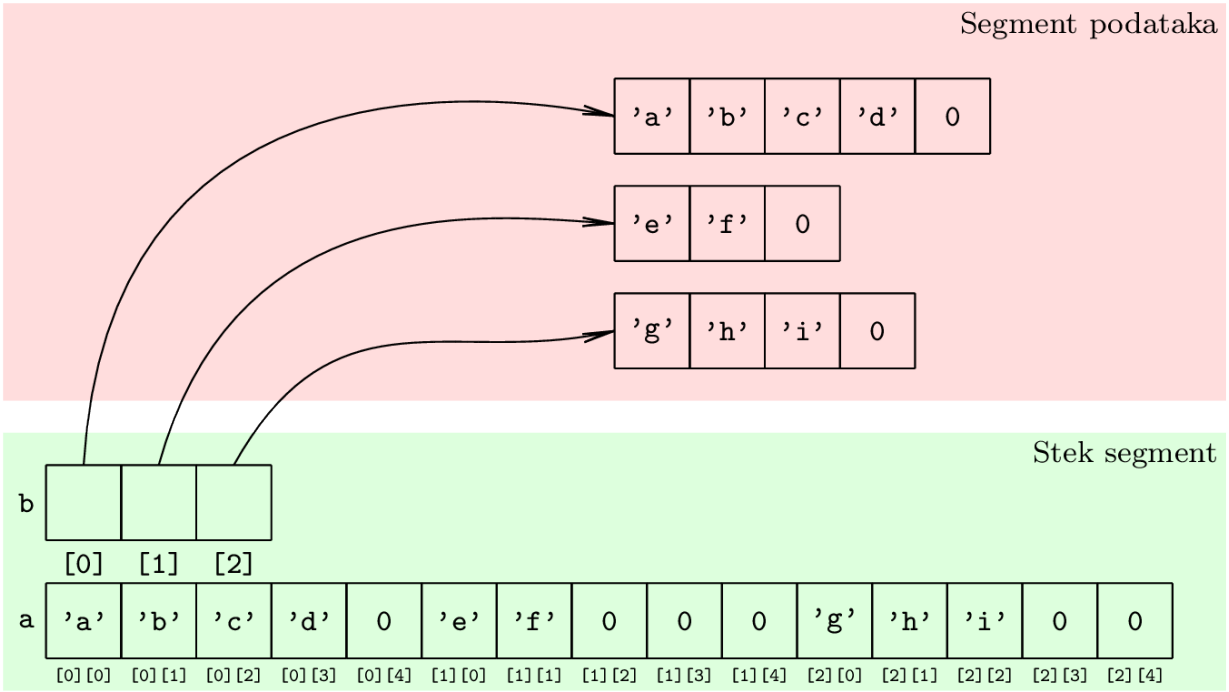
Rešenje 2 – niz pokazivača:

```
char *meseci[] = {
   "Greska", "Januar", "Februar", "Mart", "April",
   "Maj", "Jun", "Jul", "Avgust", "Septembar",
   "Oktobar", "Novembar", "Decembar"
};
```

- Niz pokazivača ukazuje na konstantne niske u segmentu podataka.
- Nije potrebno navoditi broj elemenata jer je dat inicijalizator.
- Memorija je iskorišćena efikasnije.

Slika ?? ilustruje sličan primer - memoriju koja odgovara nizovima a i b deklarisanim u okviru neke funkcije:

```
char  a[][5] = {"abcd", "ef", "ghi"};
char* b[]    = {"abcd", "ef", "ghi"};
```



%

% *Niz pokazivača i dvodimenzioni niz u memoriji*

1.5.4 Pokazivači i strukture

Moguće je definisati i **pokazivače na strukture**. Kada se članovima strukture pristupa preko pokazivača, umesto kombinacije operatora `*` i `.` može se koristiti **operator** `->`. Operator `->` ima **najviši prioritet**.

Na primer, umesto:

```
(*pa).imenilac
```

može se pisati:

```
pa->imenilac
```

što koristimo u ovom primeru:

```
struct razlomak *pa = &a;
printf("%d/%d", pa->brojilac, pa->imenilac);
```

Plitko i duboko kopiranje struktura

Prilikom dodele struktura, vrši se plitko kopiranje pokazivača koji se u njima nalaze. Neka su `a` i `b` promenljive istog tipa strukture koja ima član pokazivačkog tipa `p`.

Prilikom dodele struktura (`a = b;`), vrši se **plitko kopiranje**:

- kopiraju se vrednosti svih članova, uključujući i pokazivače,
- ali se **ne kopira sadržaj memorije** na koji pokazivač ukazuje.

Dakle, posle izraza:

```
a = b;
```

- i `a.p` i `b.p` ukazuju na istu adresu (onu na koju je ukazivao `b.p`).
- sadržaj na koji ukazuje `b.p` nije kopiran.

Ovo se često koristi (jer štedi memoriju), ali može dovesti do grešaka. U praksi je često potrebna implementacija **dubokog kopiranja** – kada se pravi i nova kopija memorije na koju pokazivač ukazuje. To, međutim, zahteva dodatni trud programera.

Strukture kao argumenti funkcija

Strukture se **prenose po vrednosti** kada se proslede funkcijama. Pošto strukture mogu zauzimati značajan prostor, češće se prenosi njihova **adresa** (pokazivač na strukturu). Na primer,

```
void saberi_razlomke(struct razlomak *pa, struct razlomak *pb,
                     struct razlomak *pc)
{
    pc->brojilac = pa->brojilac*pb->imenilac +
                   pa->imenilac*pb->brojilac;
    pc->imenilac = pa->imenilac*pb->imenilac;
}
```

Ova funkcija računa zbir razlomaka pa i pb i rezultat upisuje u pc.

1.5.5 Dinamička alokacija memorije

U većini realnih aplikacija, u trenutku pisanja programa **nije moguće precizno predvideti memorijske zahteve**. Potrebna količina memorije često zavisi od **interakcije sa korisnikom** — tek tokom izvršavanja programa korisnik svojim akcijama određuje koliko će memorije biti neophodno (na primer, koliko elemenata niza će se koristiti).

Ponekad je moguće odrediti **gornje ograničenje** na veličinu potrebne memorije, ali ni to rešenje nije idealno. Ako je ograničenje **premalo**, program neće moći da obradi veće ulaze. Ako je pak **preveliko**, zauzeće više memorije nego što je stvarno potrebno.

Rešenje ovih problema je **dinamička alokacija memorije**, koja omogućava da program **tokom izvršavanja** od operativnog sistema zatraži određenu količinu memorije. Kada ta memorija više nije potrebna, program treba da je **oslobodi** i tako je **vrati operativnom sistemu**.

Proces alokacije i oslobađanja memorije obavlja se pomoću **funkcija iz standardne biblioteke jezika C** i odgovarajućih **poziva rantajm biblioteke**. Tokom izvršavanja programa, sistem vodi evidenciju o **raspoloživim** i **zauzetim blokovima memorije**.

Funkcije standardne biblioteke za rad sa dinamičkom memorijom

Standardna biblioteka jezika C omogućava **dinamičko upravljanje memorijom** pomoću nekoliko funkcija, koje su deklarisane u zaglavlju `<stdlib.h>`. Prostor za dinamički alociranu memoriju nalazi se u delu memorije koji se naziva **hip** (eng. *heap*).

Funkcija `malloc`

```
void *malloc(size_t n);
```

Funkcija `malloc` alocira blok memorije veličine `n` bajtova i vraća **adresu početka bloka** kao pokazivač tipa `void*`. Ovaj generički pokazivač se zatim **pretvara u pokazivač odgovarajućeg tipa** — onog koji odgovara podacima koji će se čuvati u toj memoriji.

Primer:

```
int* p = (int*) malloc(10 * sizeof(int));
```

Ova naredba alocira prostor za niz od 10 celih brojeva i vraća pokazivač p koji pokazuje na prvi element niza.

U jeziku C konverzija pokazivača tipa `void*` u konkretan tip može biti **eksplicitna** (kao u primeru) ili **implicitna**. Mišljenja o tome koji pristup je bolji su podeljena — eksplicitna konverzija povećava čitljivost, dok implicitna skraćuje kod.

> Napomena: U jeziku **C++** implicitna konverzija nije dozvoljena; pokazivač mora biti eksplicitno konvertovan.

Ako sistem **ne može da obezbedi traženu količinu memorije**, funkcija `malloc` vraća vrednost `NULL`. Tada program treba da proveri rezultat alokacije i reaguje u skladu s tim:

```
int* p = malloc(n * sizeof(int));
if (p == NULL) {
    /* prijaviti grešku ili pokušati oporavak */
}
```

U nekim implementacijama (posebno na Linuxu), `malloc` može vratiti pokazivač i kada memorije zapravo nema dovoljno. U tom slučaju, pokušaj kasnijeg pristupa toj memoriji dovodi do **prekida rada programa**. Zbog toga se proveravanje rezultata `malloc` poziva smatra **dobrom praksom**.

> **Napomena:** Memorija dobijena funkcijom `malloc` nije inicijalizovana — njen sadržaj je neodređen.

Tip `size_t`

Argument funkcije `malloc` ima tip `size_t`, koji predstavlja **nenegativan celobrojni tip** dovoljan da predstavi veličinu bilo kog objekta u memoriji. To je ujedno i tip koji vraća operator `sizeof`. Iako nije isti kao `unsigned int`, među njima postoji trivijalna konverzija.

Funkcija `calloc`

```
void *calloc(size_t n, size_t size);
```

Funkcija `calloc` alocira prostor za **n objekata** veličine `size` bajtova. Za razliku od `malloc`, sva alocirana memorija je **inicijalizovana na nulu**.

Ako alokacija ne uspe, `calloc` vraća `NULL`. Dobijeni pokazivač, kao i kod `malloc`, obično se konvertuje u pokazivač konkretnog tipa podataka:

```
double* a = (double*) calloc(100, sizeof(double));
```

Funkcije `malloc` i `calloc` su **generičke** – mogu se koristiti za alokaciju memorije za podatke bilo kog tipa. Nakon uspešne alokacije, novodobijena memorija se koristi isto kao niz, ali **nema ime** — zato kažemo da su dinamički objekti *neimenovani*.

Posle svakog poziva `malloc` ili `calloc` neophodno je proveriti da li je funkcija vratila `NULL`. Ako se pokazivač dereferencira bez te provere, program može da se sruši zbog **dereferenciranja** `NULL` **pokazivača**.

```
int* p = malloc(n*sizeof(int));
if (p == NULL)
  /* pokusati oporavak od greske ili prijaviti gresku */
```

U situacijama gde se pretpostavlja da će alokacija biti uspešna, provera se ponekad zameni makroom `assert`:

```
int* p = malloc(n * sizeof(int));
assert(p != NULL);
```

Oslobađanje memorije – funkcija `free`

U trenutku kada dinamički alociran blok memorije više nije potreban, poželjno je osloboditi ga. To se postiže funkcijom `free`:

```
void free(void* p);
```

Funkcija `free` oslobađa memoriju na koju ukazuje pokazivač p. Važno je razumeti da se **oslobađa prostor na koji pokazuje pokazivač**, a ne sam pokazivač.

Pokazivač p mora da pokazuje na blok memorije koji je ranije alociran pomoću `malloc` ili `calloc`. Oslobađanje memorije koja nije prethodno alocirana na taj način dovodi do **nedefinisanog ponašanja** (često i do rušenja programa).

Takođe, nije dozvoljeno:

- koristiti memoriju nakon što je oslobođena (tzv. *dangling pointer*),
- dva puta pozvati `free` nad istim pokazivačem.

Redosled oslobađanja memorije **ne mora** da odgovara redosledu alokacije.

Ako program ne oslobodi dinamički alocirane blokove pre završetka, operativni sistem će ih automatski osloboditi po zatvaranju procesa. Ipak, **ne treba se oslanjati na to** — dobra praksa je da se **sva dinamički alocirana memorija eksplicitno oslobodi** čim prestane da bude potrebna.

Primer upotrebe funkcija za dinamičku alokaciju memorije

Upotrebu funkcija `malloc` i `free` ilustruje sledeći program. Program unosi niz celih brojeva čiji broj elemenata **nije unapred poznat**. Broj elemenata se unosi sa standardnog ulaza tokom izvršavanja programa, a zatim se niz ispisuje u **obrnutom poretku**.

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int n, i, *a;

    /* Unos broja elemenata */
    scanf("%d", &n);

    /* Alokacija memorije */
```

```
    a = malloc(n * sizeof(int));
    if (a == NULL) {
        printf("Greška prilikom alokacije memorije\n");
        return 1;
    }

    /* Unos elemenata */
    for (i = 0; i < n; i++)
        scanf("%d", &a[i]);

    /* Ispis elemenata u obrnutom poretku */
    for (i = n - 1; i >= 0; i--)
        printf("%d ", a[i]);

    /* Oslobađanje memorije */
    free(a);

    return 0;
}
```

U ovom programu broj elemenata se ne zna unapred, pa se memorija alocira **nakon unosa vrednosti** `n`. Na kraju, memorija se mora eksplicitno osloboditi pozivom funkcije `free`.

Promena veličine alociranog bloka — funkcija `realloc`

U nekim situacijama potrebno je **promeniti veličinu već alociranog bloka memorije**. To se postiže funkcijom `realloc`, čiji je prototip:

```
void *realloc(void *memblock, size_t size);
```

- Prvi parametar `memblock` je pokazivač na prethodno alocirani blok memorije.
- Drugi parametar `size` predstavlja novu veličinu u bajtovima.

Funkcija vraća pokazivač tipa `void*` na **realocirani blok memorije**, ili `NULL` ako zahtev ne može biti ispunjen.

Princip rada funkcije `realloc`:

- Kada se traži **smanjivanje veličine** postojećeg bloka, zahtev uvek uspeva.
- Ako se traži **uvećanje**, funkcija pokušava da proširi blok u istom delu memorije (ako ima dovoljno slobodnog prostora neposredno iza njega).
- U suprotnom, `realloc` **mora da pronađe novo mesto u memoriji** koje može da primi ceo blok nove veličine. U tom slučaju, sadržaj starog bloka se **kopira** na novo mesto, a stari blok se automatski oslobađa. Ova operacija može biti **vremenski zahtevna**, naročito za velike blokove podataka.

Ako je vrednost argumenta `size` jednaka nuli, ponašanje funkcije nije potpuno definisano standardom — funkcija može da vrati `NULL` ili isti pokazivač koji je zadat kao prvi parametar. Poput `malloc`, ni `realloc` **ne inicijalizuje** novododeljenu memoriju.

Primer: postepeno proširivanje niza

Sledeći primer prikazuje upotrebu funkcije `realloc` za dinamičko proširivanje niza tokom unosa podataka. Program učitava cele brojeve dok se ne unese vrednost −1. Pošto se broj elemenata **ne zna unapred**, memorijski prostor se mora **povećavati tokom rada programa**. Kako česta realokacija može biti neefikasna, u narednom programu se izbegava realokacija prilikom unosa svakog sledećeg elementa, već se vrši nakon unošenja svakog stotog elementa.

```
#include <stdio.h>
#include <stdlib.h>

#define KORAK 100
int main() {
   int* a = NULL;      /* Niz je u pocetku prazan */
   int duzina = 0;     /* broj popunjenih elemenata niza */
   int alocirano = 0; /* koliko elemenata moze biti smesteno */
   int i;

   do {
      printf("Unesi ceo broj (-1 za poslednji broj): ");
      scanf("%d", &i);

      /* Ako nema vise slobodnih mesta, vrsi se prosirivanje */
      if (duzina == alocirano) {
         alocirano += KORAK;
         a = realloc(a, alocirano*sizeof(int));
         if (a == NULL) return 1;
      }
      a[duzina++] = i;
   } while (i != -1);

   /* Ispis elemenata */
   printf("Uneto je %d brojeva. Alocirano je %d bajtova\n",
      duzina, alocirano*sizeof(int));
   printf("Brojevi su : ");
   for (i = 0; i<duzina; i++)
      printf("%d ", a[i]);

   /* Oslobadjanje memorije */
   free(a);

   return 0;
```

U gore navedenom primeru koristi se konstrukcija:

```
a = realloc(a, alocirano*sizeof(int));
```

U nekim slučajevima ova konstrukcija može da bude neadekvatna ili opasna. Naime, ukoliko zahtev za proširenje memorijskog bloka ne uspe, vraća se vrednost `NULL`, upisuje u promenljivu `a` i time

se gubi jedina veza sa prethodno alociranim blokom (i on, na primer, ne može ubuduće da bude oslobođen).

Ovaj pristup je jednostavan, ali ne i optimalan — česta realokacija može biti **neefikasna**. Zato se u praksi obično primenjuje **eksponencijalna strategija rasta**: na početku se niz često realocira, ali se svakim proširenjem njegova veličina **udvostručuje**. Na taj način broj realokacija opada logaritamski u odnosu na broj elemenata, što značajno povećava efikasnost.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    int* a = NULL;      /* Niz je u početku prazan */
    int duzina = 0;     /* broj popunjenih elemenata niza */
    int alocirano = 0;  /* koliko elemenata može biti smešteno */
    int x;

    do {
        printf("Unesi ceo broj (-1 za poslednji broj): ");
        scanf("%d", &x);

        /* Ako nema više slobodnih mesta, vrši se proširivanje */
        if (duzina == alocirano) {
            /* Ako je ovo prvo alociranje */
            alocirano = (alocirano == 0) ? 1 : alocirano * 2;

            int* temp = realloc(a, alocirano * sizeof(int));
            if (temp == NULL) {
                printf("Greška prilikom alokacije memorije\n");
                free(a);
                return 1;
            }
            a = temp;
        }

        a[duzina++] = x;

    } while (x != -1);

    /* Ispis rezultata */
    printf("\nUneto je %d brojeva. Alocirano je %d bajtova.\n",
           duzina, alocirano * (int)sizeof(int));

    printf("Brojevi su: ");
    for (int i = 0; i < duzina; i++)
        printf("%d ", a[i]);

    /* Oslobađanje memorije */
    free(a);
```

```
    return 0;
}
```

U ovom primeru, korišćena je konstrukcija:

```
    int* temp = realloc(a, alocirano * sizeof(int));
    if (temp == NULL) {
        printf("Greška prilikom alokacije memorije\n");
        free(a);
        return 1;
    }
    a = temp;
```

gde se koristi privremena promenljiva za čuvanje rezultata funkcije realloc. Na ovaj način, ukoliko realokacija ne uspe, originalni pokazivač a zadržava svoju prethodnu vrednost, čime se obezbeđuje siguran pristup ranije alociranoj memoriji i mogućnost njenog kontrolisanog oslobađanja.

Amortizovana analiza složenosti i realokacija dinamičkih nizova

U nekim situacijama se izvesne operacije ponavljaju puno puta tokom izvršavanja programa. U mnogim situacijama možemo da dopustimo da pojedinačno izvršavanje neke operacije traje i malo duže, ako smo sigurni da više izvršavanja te operacije u zbiru neće trajati predugo (ako postoji dovoljno izvršavanja te operacije koja će trajati kratko). Analiza ukupne dužine trajanja većeg broja operacija naziva se amortizovana analiza složenosti. Amortizovana cena izvršavanja n operacija podrazumeva količnik njihove ukupne dužine izvršavanja i broja n.

Kada govorimo o dinamičkim nizovima, prirodno je napraviti paralelu sa strukturom `std::vector` iz C++-a. Iako je formalno deo standardne biblioteke C++, njegova implementacija se oslanja na iste mehanizme koje u jeziku C možemo ostvariti ručnim korišćenjem `malloc` i `realloc`. `std::vector` *ne proširuje kapacitet aritmetički*, već *geometrijski*. Tipični multiplikator u realnim implementacijama je 1.5 (gcc i clang, dok MSVC koristi multiplikator 2). U nastavku pokazujemo zašto je izabrana geometrijska strategija.

Aritmetička strategija

Aritmetička strategija pretpostavlja da se prilikom svake realokacije kapacitet uvećava za fiksni broj elemenata k. Neka se niz postepeno puni do n elemenata. Pratimo ukupan broj upisa:

- kada prvi put proširujemo niz, dobijamo kapacitet k i upisujemo k elemenata;
- sledeća realokacija daje kapacitet $2k$, i tom prilikom se kopira k postojećih elemenata;
- zatim upisujemo novih k elemenata;
- naredna realokacija povećava kapacitet na $3k$, i kopiramo $2k$ elemenata;
- upisujemo k novih elemenata;
- sledeća realokacija proširuje na $4k$, kopira se $3k$ elemenata;
- i tako dalje, sve dok ne upišemo n elemenata.

Dakle, zbir svih upisa (novih elemenata + elemenata kopiranih tokom realokacije) ima oblik:

$$\underbrace{k}_{\text{upis}} + \underbrace{k}_{\text{upis}} + \underbrace{k}_{\text{upis}} + \underbrace{2k}_{\text{kopiranje}} + \underbrace{k}_{\text{upis}} + \underbrace{3k}_{\text{kopiranje}} + \underbrace{k}_{\text{upis}} + \underbrace{4k}_{\text{kopiranje}} + \ldots$$

Odnosno, redosled operacija je: $k + k + k + 2k + k + 3k + k + 4k + k + 5k + \ldots$. Ukupan broj realokacija je $\frac{n}{k}$ za niz dužine n. Ukupno kopiranje je zato:

$$k \cdot \frac{n}{k} + k \cdot (1 + 2 + 3 + \cdots + \tfrac{n}{k}) = n + k \cdot \frac{(\frac{n}{k})(\frac{n}{k} + 1)}{2} = \frac{n^2}{2k} + \frac{3n}{2}.$$

Dakle, ukupan broj upisa asimptotski je jednak $\frac{n^2}{2k}$ tj. $O(n^2)$. Amortizovana cena jedne operacije je asimptotski jednaka $O(n)$.

Geometrijska strategija

Druga strategija može biti **geometrijska**, i ona podrazumeva da se svaki put veličina niza poveća q *puta*, za neki faktor $q > 1$. Pretpostavimo da je početna veličina niza m_0. Dakle, nakon početne alokacije možemo da upišemo m_0 elemenata.

Nakon toga se vrši prva realokacija u kojoj se veličina niza povećava na $q \cdot m_0$ elemenata, i pri tome se prepisuje m_0 postojećih elemenata. Potom se upisuju preostali elementi — ukupno $q \cdot m_0 - m_0$.

U narednoj realokaciji veličina niza povećava se na $q^2 \cdot m_0$, i tada se kopira $q \cdot m_0$ elemenata. Posle toga se upisuje još $q^2 \cdot m_0 - q \cdot m_0$ elemenata. Postupak se nastavlja po istom principu.

Dakle, ukupan broj upisa u niz jednak je:

$$m_0 + m_0 + (qm_0 - m_0) + qm_0 + (q^2 m_0 - qm_0) + \cdots = m_0 + qm_0 + q^2 m_0 + \ldots$$

Posle r realokacija, ukupan broj upisa je:

$$m_0(1 + q + q^2 + \cdots + q^r) = m_0 \cdot \frac{q^{r+1} - 1}{q - 1}.$$

Ako pretpostavimo da je niz potpuno popunjen posle r realokacija, onda važi $n = m_0 q^r$. Prema tome ukupan broj operacija potrebnih za popunjavanje niza jednak je:

$$m_0 \cdot \frac{q \cdot n/m_0 - 1}{q - 1} = \frac{qn - m_0}{q - 1}.$$

Asimptotski, dakle:

- ukupan broj operacija je **O(n)**,
- amortizovana cena jedne operacije dodavanja je **O(1)**.

Konstantan faktor jednak je $\frac{q}{q-1}$ i on postaje sve manji kako q raste. Zaista, sa većim q realokacije su ređe, ali se cena plaća većim angažovanjem memorije, tj. manjom popunjenošću alociranog prostora.

Amortizovana analiza složenosti nam pokazuje da sa geometrijskom strategijom dobijamo najbolji mogući odnos između broja realokacija i brzine operacije dodavanja.

Greške u radu sa dinamičkom memorijom

Dinamički alocirana memorija omogućava veliku fleksibilnost i efikasno korišćenje resursa, ali je istovremeno i čest izvor grešaka koje mogu biti teške za otkrivanje i otklanjanje. U nastavku su opisane najčešće vrste grešaka pri radu sa dinamičkom memorijom.

Curenje memorije (*memory leak*)

Jedna od najopasnijih grešaka u radu sa dinamičkom memorijom je **curenje memorije** — situacija u kojoj program izgubi informaciju o lokaciji dinamički alociranog, a neoslobođenog bloka memorije. U tom slučaju, program više **nema način da oslobodi taj blok**, pa on ostaje zauzet do završetka izvršavanja programa.

Primer:

```
char* p;
p = malloc(1000);
/* ... */
p = malloc(5000);
```

U ovom primeru, najpre je alocirano 1000 bajtova memorije, a pokazivač `p` ukazuje na taj blok. Nakon ponovne alokacije od 5000 bajtova, vrednost pokazivača `p` se menja, čime se gubi referenca na prvobitni blok od 1000 bajtova koji nikada nije oslobođen. Taj deo memorije ostaje trajno izgubljen za program.

Curenje memorije je naročito opasno ako se dešava **unutar petlji**. U takvim slučajevima, program može postepeno gubiti memoriju sve dok se ne iscrpe svi raspoloživi resursi. Posledice mogu biti ekstremno usporavanje programa (usled tzv. *swappinga*, kada sistem premešta sadržaj memorije na disk) ili potpuno **nasilno prekidanje izvršavanja**.

Još jedan problem je to što se curenje memorije često **ne uočava odmah**. Programi se obično testiraju na malim ulazima i kratko vreme, pa problem postaje vidljiv tek kada program duže radi ili obradi velike količine podataka. Zbog toga su curenja memorije često uzrok teških i skupih grešaka u realnim sistemima.

Za otkrivanje ovih problema koriste se **specijalizovani alati** poznati kao *memory leak profilers*, poput alata **Valgrind**, **AddressSanitizer**, i drugih, koji omogućavaju detaljno praćenje svih alokacija i oslobađanja memorije.

Pristup oslobođenoj memoriji (*dangling pointer*)

Nakon poziva `free(p)`, memorija na koju je pokazivač `p` ukazivao više ne pripada programu i **ne sme se koristiti**. Međutim, poziv `free` **ne menja vrednost pokazivača** — on i dalje sadrži staru adresu.

Ukoliko programer naknadno pokuša da koristi taj pokazivač, može doći do nepredvidivih rezultata: novi poziv `malloc` može dodeliti isti deo memorije, ali i ne mora. Zbog toga je pristup oslobođenoj memoriji **nedefinisano ponašanje** koje se često ne manifestuje odmah, što dodatno otežava otkrivanje greške.

Zbog toga se preporučuje da se pokazivač **odmah nakon oslobađanja** postavi na `NULL`:

```
free(p);
p = NULL;
```

Tako se svaki sledeći pokušaj korišćenja oslobođene memorije prepoznaje kao greška tokom izvršavanja — obično uz poruku tipa *segmentation fault*.

Višestruko oslobađanje memorije (*double free*)

Ako se ista memorijska lokacija oslobodi **više od jednom**, ponašanje programa postaje nedefinisano i može dovesti do pada programa ili ozbiljnih bezbednosnih propusta. Zato se svaki pokazivač nakon `free(p)` mora smatrati nevažećim i ne sme se ponovo koristiti u pozivu `free`.

Oslobađanje neispravnog pokazivača

Funkciji `free(p)` se sme proslediti **isključivo pokazivač** koji je prethodno vraćen od funkcija `malloc`, `calloc` ili `realloc`. Prosleđivanje pokazivača koji ne pokazuje na početak takvog bloka može dovesti do nepredvidivih rezultata.

Na primer:

```
free(p + 10); /* Ne oslobađa „sve osim prvih 10 elemenata"! */
```

Ova naredba ne oslobađa „ostatak" bloka i može izazvati ozbiljne greške, uključujući pad programa ili oštećenje internog stanja alokatora memorije.

Prekoračenje i potkoračenje bafera (*buffer overflow / underflow*)

Nakon dinamičke alokacije, pristup memoriji je dozvoljen **samo unutar granica** dodeljenog bloka. Upis ili čitanje van granica može ozbiljno narušiti rad programa.

Upis van granica je posebno opasan jer se iza alociranog bloka često čuvaju pomoćne informacije koje koristi sam sistem za upravljanje memorijom (alokator). Njihovo oštećenje može dovesti do pada programa, korupcije podataka ili kompromitovanja bezbednosti sistema. Slični problemi se mogu javiti i pri *potkoračenju* (upisu pre početka bloka).

Fragmentisanje memorije

Čak i kada u programu nema curenja memorije, višestruke alokacije i oslobađanja tokom dužeg rada mogu dovesti do **fragmentisanja memorije**.

U takvom stanju, iako u memoriji postoji dovoljno slobodnog prostora ukupno, on je **rascepkan** na mnogo malih, nepovezanih delova. Zbog toga novi zahtev za većim blokom memorije može da ne uspe, jer ne postoji **kontinuirani segment** dovoljne veličine.

Na primer:

- memorija 100101011000011101010110 sadrži ukupno 12 slobodnih bajtova, ali nijedan niz od 5 uzastopnih bajtova,
- memorija 111111111111111100000000 ima samo 8 slobodnih bajtova, ali omogućava uspešnu alokaciju od 5 bajtova.

Operativni sistem ne može automatski „presložiti" memoriju na hipu, jer bi to zahtevalo promenu svih pokazivača koji ukazuju na postojeće objekte.

Moguće mere ublažavanja fragmentacije:

- izbegavati česte male alokacije,
- po mogućnosti **alocirati veće blokove** i interno ih deliti,
- koristiti **pool alokatore** (*memory pools*), koji unapred rezervišu veće delove memorije i njima upravljaju efikasnije.

Hip i dinamički životni vek

Kao što je objašnjeno u poglavlju o organizaciji memorije, program koristi više memorijskih segmenata: **segment koda**, **segment podataka**, **stek** i **hip**.

Hip segment predstavlja deo memorije namenjen **dinamičkoj alokaciji** — funkcije `malloc`, `calloc` i `realloc` vraćaju adrese koje se nalaze upravo u ovom segmentu.

Objekti alocirani na hipu su neimenovani i njima se pristupa isključivo preko pokazivača. Njihov životni vek traje od trenutka alokacije do trenutka eksplicitnog oslobađanja memorije pozivom `free`. Zbog toga se kaže da ovi objekti imaju **dinamički životni vek**.

Tipično, hip se nalazi iza segmenta podataka, a raste ka većim adresama memorije (eng. *upward growing*), dok se stek širi u suprotnom smeru (*downward growing*). Ako se hip i stek u nekom trenutku „sretnu", operativni sistem detektuje **prekoračenje memorijskog prostora programa** i prekida njegovo izvršavanje.

1.6 Pregled standardne biblioteke

Jezik C ima mali broj naredbi i mnoge funkcionalnosti obezbeđene su kroz funkcije *standardne biblioteke* (engl. *standard library*). Standardna biblioteka obezbeđuje funkcije, ali i makroe i definicije tipova za potrebe rada sa datotekama, niskama, alokacijom memorije i slično. Standardna C biblioteka nije jedna, konkretna biblioteka koja se koristi za povezivanje sa programima napisanom u jeziku C, već ona pre predstavlja standard koji mora da poštuje svaki C prevodilac. Kako je implementirana standardna biblioteka zavisi od konkretnog sistema.

Programski interfejs za pisanje aplikacija (API) za standardnu biblioteku dat je u vidu datoteka zaglavlja, od kojih svaka sadrži deklaracije funkcije, definicije tipova i makroa. Skupu petnaest datoteka zaglavlja propisanih standardom ANSI C (C89), dokument *Normative Addendum 1* (NA1) dodao je tri datoteke 1995. godine. Verzija C99 uvela je još šest datoteka zaglavlja, a verzija C11 još pet (od kojih tri mogu da ne budu podržane od implementacija, a da se te implementacije i dalje smatraju standardnim).
U tabeli ispod dat je pregled svih dvadeset devet zaglavlja.
Ove datoteke, zajedno sa izvornim kodom programa bivaju uključene u proces kompilacije (korišćenjem pretprocesora i njegove direktive `#include`). Pored ovih datoteka, standardna biblioteka sadrži i definicije funkcija koje obično nisu raspoložive u izvornom, već samo u prevedenom, objektnom (mašinskom) obliku, spremnom za povezivanje sa objektnim modulima programa korisnika. Ovaj (veći) deo standardne biblioteke biva uključen u fazi povezivanja (i ne obrađuje se tokom kompilacije u užem smislu, već samo tokom povezivanja).

Datoteka zaglavlja	Od	Deklaracije i definicije koje sadrži
<assert.h>	C89	Makro (ili funkcija) assert.
<complex.h>	C99	Funkcije za rad sa kompleksnim brojevima.
<ctype.h>	C89	Funkcije za klasifikovanje i konvertovanje karaktera, nezavisno od karakterske tabele koja se koristi (ASCII ili neke druge).
<errno.h>	C89	Podrška za rad sa kodovima grešaka koje vraćaju bibliotečke funkcije.
<fenv.h>	C99	Funkcije za kontrolu okruženja za brojeve u pokretnom zarezu.
<float.h>	C89	Konstante koje specifikuju svojstva brojeva u pokretnom zarezu zavisna od implementacije.
<inttypes.h>	C99	Proširena podrška za celobrojne tipove fiksne širine.
<iso646.h>	NA1	Makroi za opisivanje standardnih tokena.
<limits.h>	C89	Konstante koje specifikuju svojstva celih brojeva zavisna od implementacije.
<locale.h>	C89	Funkcije za lokalizaciju.
<math.h>	C89	Često korišćene matematičke funkcije.
<setjmp.h>	C89	Makroi setjmp i longjmp.
<signal.h>	C89	Funkcije za upravljanje signalima.
<stdalign.h>	C11	Podrška za ravnanje objekata.
<stdarg.h>	C89	Podrška za funkcije sa promenljivim brojem parametara.
<stdatomic.h>	C11	Podrška za atomičke operacije nad podacima deljenim između niti.
<stdbool.h>	C99	Tip bool.
<stddef.h>	C89	Nekoliko raznorodnih tipova i konstanti (uključujući NULL i size_t).
<stdint.h>	C99	Podrška za celobrojne tipove fiksne širine.
<stdio.h>	C89	Deklaracije osnovnih ulazno/izlaznih funkcija.
<stdlib.h>	C89	Deklaracije funkcija za konverzije, generisanje pseudoslučajnih brojeva, dinamičku alokaciju memorije, prekid programa itd.
<stdnoreturn.h>	C11	Podrška za specifikovanje funkcija bez povratka.
<string.h>	C89	Funkcije za obradu niski.
<tgmath.h>	C99	Matematičke funkcije generičkog tipa.
<threads.h>	C11	Funkcije za upravljanje nitima.
<time.h>	C89	Funkcije za rad sa datumima i vremenom.
<uchar.h>	C11	Tipovi i deklaracije za rad sa Unicode karakterima.
<wchar.h>	NA1	Funkcije za rad sa niskama karaktera veće širine.
<wctype.h>	NA1	Funkcije za klasifikovanje i konvertovanje karaktera veće širine.

Tabela: Pregled zaglavlja standardne biblioteke

U daljem tekstu će biti ukratko opisane neke od funkcija deklarisanih u nekim od ovih zaglavlja. Najkompleksnije zaglavlje stdio.h biće opisano u narednoj glavi, posvećenoj ulazu i izlazu C programa.

1.6.1 Zaglavlje `string.h`

```
size_t strlen  (const char* str);

char*  strcpy  (char* destination, const char* source);
char*  strncpy (char* destination, const char* source,
                size_t num);
char*  strcat  (char* destination, const char* source);
char*  strncat (char* destination, const char* source,
                size_t num);

int    strcmp  (const char* str1, const char* str2);
int    strncmp (const char* str1, const char* str2,
                size_t num);

char*  strchr  (const char* str, int character);
char*  strrchr (const char* str, int character);
char*  strstr  (const char* str1, const char* str2);

size_t strspn  (const char* str1, const char* str2);
size_t strcspn (const char* str1, const char* str2);
char*  strpbrk (char* str1, char* str2);

char*  strtok  (char * str, const char* delimiters);
```

Opisi funkcija:

- `strlen`

  Funkcija `strlen` izračunava dužinu prosleđene niske karaktera (terminalna nula se ne računa).

- `strcpy`

  Funkcija `strcpy` kopira drugu navedenu nisku u prvu (pretpostavljajući da u prvoj ima dovoljno mesta za sve karaktere druge, uključujući i terminalnu nulu). Funkcija `strncpy` takođe vrši kopiranje, ali se dodatnim parameterom `n` kontroliše najveći broj karaktera koji može biti kopiran (u slučaju da je niska koja se kopira kraća od broja `n` ostatak se popunjava nulama, a ukoliko je duža od broja `n` terminalna nula se ne dodaje automatski na kraj). Korišćenje funkcije `strncpy` smatra se bezbednijim od `strcpy` jer je moguće kontrolisati mogućnost prekoračenja bafera.

- `strcat`

  Funkcija `strcat` nadovezuje drugu navedenu nisku na prvu, pretpostavljajući da prva niska ima dovoljno prostora da smesti i sve karaktere druge, uključujući i njenu terminalnu nulu (terminalna nula prve niske se briše). Funkcija `strncat` dodatnim parametrom `n` kontroliše najveći broj karaktera druge niske koji će biti nadovezan na prvu.

- `strchr`

  Funkcija `strchr` proverava da li data niska sadrži dati karakter i vraća pokazivač na prvu poziciju na kojoj je karakter nađen ili `NULL` ako karakter nije nađen. Funkcija `strrchr` radi slično, ali vraća pokazivač na poslednje pojavljivanje karaktera.

- `strpbrk`

  Funkcija `strpbrk` vraća pokazivač na prvo pojavljivanje nekog karaktera iz druge navedene niske u prvoj navedenoj niski ili `NULL` pokazivač ako takvog karaktera nema.

- `strstr`

  Funkcija `strstr` proverava da li je druga navedena niska podniska prve. Ako jeste, vraća se pokazivač na prvo njeno pojavljivanje unutar prve niske, a ako nije, vraća se pokazivač `NULL`.

- `strspn`

  Funkcija `strspn` vraća dužinu početnog dela prve navedene niske koji se sastoji samo od karaktera sadržanih u drugoj navedenoj niski. Slično, funkcija `strcspn` vraća dužinu početnog dela prve navedene niske koji se sastoji samo od karaktera koji nisu sadržani u drugoj navedenoj niski.

- `strtok`

  Niz poziva funkcije `strtok` služe da podele nisku u tokene koji su niske uzastopnih karaktera razdeljene karakterima navedenim u drugoj niski. U prvom pozivu funkcije kao prvi argument navodi se niska koja se deli, a u drugom navodi se `NULL`.

  Na primer, kod:

```
char str[] ="- Ovo, je jedna niska.", delims[] = " ,.-";
char *s = strtok (str, delims);
while (s != NULL) {
  printf ("%s\n", s);
  s = strtok (NULL, delims);
}
```

  ispisuje:

```
Ovo
je
jedna
niska
```

1.6.2 Zaglavlje `stdlib.h`

```
void *malloc(size_t n);
void *calloc(size_t n, size_t size);
void *realloc(void *memblock, size_t size);
void free(void* p);

int rand(void);
void srand(unsigned int);

int system(char* command);
void exit(int);
```

Već je navedeno da se u zaglavlju `stdlib.h` nalaze deklaracije funkcija `malloc`, `calloc`, `realloc` i `free` koje služe za dinamičku alokaciju i oslobađanje memorije. Pored njih, najznačajnije funkcije ovog zaglavlja su i sledeće.

- `rand`

  Funkcija `int rand(void);` generiše pseudo-slučajne cele brojeve u intervalu od 0 do vrednosti `RAND_MAX` (koja je takođe definisana u zaglavlju `<stdlib.h>`). Termin *pseudo-slučajni* se koristi da se naglasi da ovako dobijeni brojevi nisu zaista slučajni, već su dobijeni specifičnim izračunavanjima koja proizvode nizove brojeva nalik na slučajne. Funkcija `rand()` vraća sledeći pseudo-slučajan broj u svom nizu.

  Pseudo-slučajni brojevi u pokretnom zarezu iz intervala `[0,1)` mogu se dobiti na sledeći način:

  ```
  ((double) rand() / (RAND_MAX+1.0))
  ```

  Funkcija `rand()` može biti jednostavno upotrebljena za generisanje pseudo-slučajnih brojeva u celobrojnom intervalu `[n,m]`:

  ```
  n+rand()%(m-n+1)
  ```

  Ipak, bolja svojstva (bolju raspodelu) imaju brojevi koji se dobijaju na sledeći način:

  ```
  n+(m-n+1)*((double) rand() / (RAND_MAX+1.0))
  ```

- `srand`

  Funkcija `rand` niz pseudo-slučajnih brojeva generiše uvek počev od iste podrazumevane vrednosti (koja je jednaka `1`). To znači da će se u svakom pokretanju programa dobiti isti niz pseudo-slučajnih brojeva. Funkcija `void srand(unsigned);` postavlja vrednost koja se u sledećem pozivu funkcije `rand` (a time, indirektno, i u svim narednim) koristi u generisanju tog niza.

  Za bilo koju vrednost funkcije `srand`, čitav niz brojeva koji vraća funkcija `rand` je uvek isti. To je pogodno za ponavljanje (na primer, radi debagovanja) procesa u kojima se koriste pseudo-slučajni brojevi.

  Često se kao argument u pozivu funkcije `srand` navodi trenutno vreme (npr. `srand(time(NULL))`, gde se koristi funkcija `time` iz zaglavlja `<time.h>`) da bi se postiglo da se pri svakom pokretanju programa dobije različit niz vrednosti.

- `system`

  Prototip funkcije `system` je `int system(char* command);` Pozivom `system(komanda)`, izvršava se navedena komanda `komanda` (potencijalno sa argumentima) kao sistemski poziv kojim se izvršava komanda operativnog sistema ili neki drugi program (u okviru tekućeg programa, a ne iz komandne linije). Nakon toga, nastavlja se izvršavanje programa.

  Na primer, pozivom `system("date");` aktivira se program `date` koji ispisuje tekući datum i koji je često prisutan na različitim operativnim sistemima.

  - Ako je argument funkcije `system` vrednost `NULL`, onda se samo proverava da li je raspoloživ komandni interpretator koji bi mogao da izvršava zadate komande. U ovom slučaju, funkcija `system` vraća ne-nulu, ako je komandni interpretator raspoloživ i nulu inače.
  - Ako argument funkcije `system` nije vrednost `NULL`, onda funkcija vraća istu vrednost koju je vratio komandni interpretator (obično nulu ukoliko je komanda izvršena bez grešaka). Vrednost `-1` vraća se u slučaju greške (na primer, komandni interpretator nije raspoloživ, nema dovoljno memorije da se izvrši komanda, lista argumenata nije ispravna).

- `exit`

  Funkcija `void exit(int)` zaustavlja rad programa (bez obzira iz koje funkcije je pozvana) i vraća svoj argument kao povratnu vrednost programa. Obično povratna vrednost nula označava uspešno izvršenje programa, a vrednost ne-nula ukazuje na neku grešku tokom izvršavanja.

  Često se kao argumenti koriste i simboličke konstante `EXIT_SUCCESS` za uspešno izvršavanje i `EXIT_FAILURE` za neuspešno izvršavanje.

  Naredba `exit(e);` u okviru funkcije `main` ekvivalentna je naredbi `return e;`. Funkcija `exit` automatski poziva `fclose` za svaku datoteku otvorenu u okviru programa (više o funkciji `fclose` biće rečeno u glavi o ulazu i izlazu C programa).

1.6.3 Zaglavlje `ctype.h`

```
int isalpha(int c); int isdigit(int c);
int isalnum(int c); int isspace(int c);
int isupper(int c); int islower(int c);
int toupper(int c); int tolower(int c);
```

Zaglavlje `ctype.h` sadrži deklaracije nekoliko funkcija za ispitivanje i konvertovanje karaktera. Sve funkcije navedene u nastavku imaju argument tipa `int` i imaju `int` kao tip povratne vrednosti.

- `isalpha(c)` vraća ne-nula vrednost ako je `c` slovo, nulu inače.
- `isupper(c)` vraća ne-nula vrednost ako je slovo `c` veliko, nulu inače.
- `islower(c)` vraća ne-nula vrednost ako je slovo `c` malo, nulu inače.
- `isdigit(c)`$<>$ vraća ne-nula vrednost ako je `c` cifra, nulu inače.
- `isalnum(c)` vraća ne-nula vrednost ako je `c` slovo ili cifra, nulu inače.
- `isspace(c)` vraća ne-nula vrednost ako je `c` belina (razmak, tab, novi red, itd), nulu inače.
- `toupper(c)` vraća karakter `c` konvertovan u veliko slovo ili, ukoliko je to nemoguće — sam karakter `c`.
- `tolower(c)` vraća karakter `c` konvertovan u malo slovo ili, ukoliko je to nemoguće — sam karakter `c`.

1.6.4 Zaglavlje `math.h`

```
double sin(double); double asin(double);
double cos(double); double acos(double);
double tan(double); double atan(double);
double atan2(double, double);
double log(double); double log10(double); double log2(double);
double exp(double); double pow(double, double);
double sqrt(double); double fabs(double);
double floor(double); double ceil(double);
double trunc(double); double round(double);
```

Zaglavlje `math.h` sadrži deklaracije više od dvadeset često korišćenih matematičkih funkcija. Svaka od njih ima jedan ili dva argumenta tipa `double` i ima `double` kao tip povratne vrednosti.

- `sin(x)` vraća vrednost funkcije $sin(x)$, smatra se da je x zadato u radijanima.
- `asin(x)` vraća vrednost funkcije $arcsin(x)$ u radijanima (ako je `x $\notin$ [-1, 1]`, obično se dobija rezultat `NaN`).
- `cos(x)` vraća vrednost funkcije $cos(x)$, smatra se da je x zadato u radijanima.
- `acos(x)` vraća vrednost funkcije $arccos(x)$ u radijanima (ako je `x $\notin$ [-1, 1]`, obično se dobija rezultat `NaN`).
- `tan(x)` vraća vrednost funkcije $tg(x)$, smatra se da je x zadato u radijanima.
- `atan(x)` vraća vrednost funkcije $arctan(x)$ u radijanima iz intervala `[-$\pi/2$, $\pi/2$]`.
- `atan2(y, x)` vraća vrednost koja odgovara uglu u odnosu na x-osu tačke `(x, y)`. Ugao je u radijanima iz intervala `(-$\pi$, $\pi$]`. Vrednost nije definisana za koordinatni početak.
- `log(x)` vraća vrednost ln(x).
  - ako je $x > 0$, izračunava se vrednost logaritma
  - ako je $x = 0$, rezultat je obično $-inf$
  - ako je $x < 0$, rezultat je obično NaN. Logaritam $log_a b$ (za $a, b > 0$, $a \neq 1$) može se izračunati kao $\frac{log(b)}{log(a)}$.
- `log10(x)` vraća vrednost $log_{10}(x)$ (ponašanje za $x \leq 0$ isto kao za $log(x)$).
- `log2(x)` vraća vrednost $log_2(x)$ (ponašanje za $x \leq 0$ isto kao za $log(x)$).
- `exp(x)` vraća vrednost e^x.
- `pow(x,y)` vraća vrednost x^y.
- `sqrt(x)` vraća vrednost $\sqrt{(x)}$.
  - ako je $x \geq 0$, izračunava se vrednost korena

  - ako je $x < 0$, obično se dobija NaN. Vrednost $\sqrt[y]{x}$ može se izračunati kao `pow(x, 1/n)`.

- `fabs(x)` vraća apsolutnu vrednost od `x`.
- `floor(x)` vraća najveći ceo broj (zapisan u vidu tipa `double`) koji je manji od ili jednak `x`.
- `ceil(x)` vraća najmanji ceo broj (zapisan u vidu tipa `double`) koji je veći od ili jednak `x`.
- `trunc(x)` vraća vrednost `x` sa odsečenim decimalnim delom.
- `round(x)` vraća ceo broj (zapisan u vidu tipa `double`) najbliži vrednosti `x`.

Pored funkcija u ovom zaglavlju su definisane mnoge važne matematičke konstante. Na primer:

- `M_PI` ima vrednost broja
- `M_E` vrednost broja e
- `M_SQRT2` vrednost $\sqrt{2}$, itd.

1.6.5 Zaglavlje `assert.h`

```
void assert(int);
```

- `assert` prima jedan logički uslov, predstavljen celobrojnim izrazom i koristi se u fazi razvoja programa da ukaže na moguće greške. Najčešće se `assert` realizuje kao makro, mada, iz ugla onoga ko ga poziva, nije previše značajno da li je u pitanju makro ili funkcija.

  Ukoliko je pri pozivu `assert(izraz)` taj logički uslov nije ispunjen tj. ako je vrednost celobrojnog izraza `izraz` jednaka nuli, onda će na standardni tok za greške (`stderr`) biti ispisana poruka nalik sledećoj:

  `Assertion failed: expression, file filename, line nnn`

  i biće prekinuto izvršavanje programa.

  Ukoliko je definisano simboličko ime `NDEBUG` (direktivom `#define`) pre nego što je uključeno zaglavlje `<assert.h>`, pozivi `assert` se ignorišu.

  Obično se `assert` koristi u toku razvoja programa, u takozvanoj *debug* (verziji za debagovanje). Kada je program spreman za korišćenje, proizvodi se *release* verzija i tada se pozivi `assert` ignorišu (jer je tada obično definisano ime `NDEBUG`).

  U fazi razvoja programa, upozorenja koja generiše `assert` ukazuju na krupne propuste u programu, ukazuju da tekući podaci ne zadovoljavaju neki uslov koji je podrazumevan, te pomažu u ispravljanju tih propusta.

  `assert` se ne koristi da ukaže na greške u fazi izvršavanja programa koje nisu logičke (na primer, neka datoteka ne može da se otvori), već samo na one logičke greške koje ne smeju da se dogode.

  U završnoj verziji programa, pozivi `assert` imaju i dokumentacionu ulogu — oni čitaocu programa ukazuju na uslove za koje je autor programa podrazumevao da moraju da važe u nekoj tački programa.

1.7 Ulaz i izlaz programa

Jezik C je dizajniran kao mali jezik i ulazno/izlazne operacije nisu direktno podržane samim jezikom, već specijalizovanim funkcijama iz standardne biblioteke jezika (koja je prisutna u svakom C okruženju). Pošto su ove funkcije deo standarda jezika C, mogu se koristiti u programima uz garantovanu prenosivost programa između različitih sistema. Svaka izvorna datoteka u kojoj se koriste ulazno/izlazne funkcije, trebalo bi da uključi standardno zaglavlje `<stdio.h>`.

1.7.1 Standardni tokovi

Standardna biblioteka implementira jednostavan model tekstualnog ulaza i izlaza. Ulaz i izlaz se modeluju tzv. *tokovima* (engl. *stream*) podataka (obično pojedinačnih bajtova ili karaktera). *Standardni ulaz* (`stdin`) obično čine podaci koji se unose sa tastature. Podaci koji se upućuju na *standardni izlaz* (`stdout`) se obično prikazuju na ekranu. Pored standardnog izlaza, postoji i *standardni izlaz za greške* (`stderr`) na koji se obično upućuju poruke o greškama nastalim tokom rada programa i koji se, takođe, obično prikazuje na ekranu.

Unix je definisao `stdin`, `stdout` i `stderr` još 70-ih, a ta ideja je toliko elegantna da se zadržala praktično nepromenjena i danas u svim jezicima i okruženjima.

U mnogim okruženjima, moguće je izvršiti preusmeravanje (redirekciju) standardnog ulaza tako da se, umesto sa tastature, karakteri čitaju iz neke datoteke. Na primer, ukoliko se program pokrene sa

```
./prog < infile
```

onda program `prog` čita karaktere iz datoteke `infile`, umesto sa tastature.

Takođe, mnoga okruženja omogućavaju da se izvrši preusmeravanje (redirekcija) standardnog izlaza u neku datoteku. Na primer, ukoliko se program pokrene sa

```
./prog > outfile
```

onda program `prog` upisuje karaktere u datoteku `outfile`, umesto na ekran.

Ukoliko bi se poruke o greškama štampale na standardni izlaz, zajedno sa ostalim rezultatima rada programa, onda, u slučaju preusmeravanja standardnog izlaza u datoteku, poruke o greškama korisniku ne bi bile prikazane na ekranu i korisnik ih ne bi video. Ovo je glavni razlog uvođenja standardnog izlaza za greške koji se obično prikazuje na ekranu. Moguće je izvršiti redirekciju i standardnog izlaza za greške u datoteku, na primer:

```
./prog 2> errorfile
```

1.7.2 Ulaz i izlaz pojedinačnih karaktera

Najjednostavniji mehanizam čitanja sa standardnog ulaza je čitanje jednog po jednog karaktera korišćenjem funkcije `getchar`:

```
int getchar(void);
```

Funkcija `getchar` vraća sledeći karakter sa ulaza, svaki put kada se pozove, ili `EOF` kada dođe do kraja toka podataka. Simbolička konstanta `EOF` je definisana u zaglavlju `<stdio.h>`. Njena vrednost je obično -1, ali umesto ove konkretne vrednosti ipak treba koristiti ime `EOF`. Funkcija `getchar` (kao i još neke srodne funkcije) umesto tipa `char` koristi tip `int` koji je dovoljno širok da u njega mogu

da se smeste kako ASCII vrednosti od 0 do 127, tako i specijalna vrednost EOF, tj. -1. Ako bi povratni tip funkcije getchar bio char, a povratna vrednost u nekom konkretnom slučaju jednaka EOF (tj. konstantna vrednost -1), na sistemima na kojima je tip char podrazumevano neoznačen, vrednost EOF bi se konvertovala u 255, pa bi poređenje getchar() == EOF bilo netačno (jer bi sa leve strane bila vrednost 255 koja bi pre poređenja sa −1 bila promovisana u tip int).

Funkcija getchar() najčešće se realizuje tako što karaktere uzima iz privremenog bafera koji se puni čitanjem jedne po jedne linije ulaza. Dakle, u interaktivnom radu sa programom, getchar() neće imati efekta sve dok se ne unese prelazak u novi red ili oznaka za kraj toka podataka.

Najjednostavniji mehanizam pisanja na standardni izlaz je pisanje jednog po jednog karaktera korišćenjem funkcije putchar:

```
int putchar(int);
```

Funkcija putchar(c) štampa karakter c na standardni izlaz, a vraća karakter koji je ispisala ili EOF ukoliko je došlo do greške.

Kao primer rada sa pojedinačnim karakterima, razmotrimo program koji prepisuje standardni ulaz na standardni izlaz, pretvarajući pri tom velika u mala slova.

```
#include <stdio.h>
#include <ctype.h>

int main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

Funkcija tolower deklarisana je u zaglavlju <ctype.h> i prevodi karaktere velikih slova u karaktere malih slova, ne menjajući pri tom ostale karaktere.

„Funkcije“ poput getchar i putchar iz <stdio.h> i tolower iz <ctype.h> su u mnogim implementacijama zapravo makroi (zasnovani na drugim funkcijama iz standardne biblioteke), čime se izbegava dodatno vreme i prostor potreban za realizaciju funkcijskog poziva.

Linijski izlaz

Bibliotečka funkcija puts:

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

ispisuje nisku na koju ukazuje s na standardni izlaz, dodajući pri tom oznaku za kraj reda. Završna nula se ne ispisuje. Funkcija puts vraća EOF u slučaju da je došlo do greške, a nenegativnu vrednost inače.

1.7.3 Formatirani izlaz — printf

Funkcija printf već je korišćena u prethodnom tekstu. Njen prototip je:

```
int printf(const char *format, arg1, arg2, ...);
```

Funkcija printf prevodi vrednosti osnovnih tipova podataka u serije karaktera i usmerava ih na standardni izlaz. Funkcija vraća broj odštampanih karaktera. Slučajevi korišćenja ove funkcije iz prethodnih glava jesu najuobičajeniji, ali svakako nisu potpuni. Pozivi funkcija putchar i printf mogu biti isprepletani — izlaz se vrši u istom redosledu u kojem su ove funkcije pozvane.

Funkcija printf prevodi, formatira i štampa svoje argumente na standardni izlaz pod kontrolom date „format niske". Format niska sadrži dve vrste objekata: obične karaktere, koji se doslovno prepisuju na standardni izlaz i specifikacije konverzija od kojih svaka uzrokuje konverziju i štampanje sledećeg uzastopnog argumenta funkcije printf. Svaka specifikacija konverzije počinje karakterom % i završava se karakterima konverzije. Između % i karaktera konverzije moguće je da se redom nađu:

- Znak minus (-), koji prouzrokuje levo poravnanje konvertovanog argumenta.
- Broj koji specifikuje najmanju širinu polja. Konvertovani argument se štampa u polju koje je barem ovoliko široko. Ukoliko je potrebno, polje se dopunjava razmacima sa leve (odnosno desne strane, ukoliko se traži levo poravnanje).
- Tačka ., koja odvaja širinu polja od preciznosti.
- Broj za preciznost, koji specifikuje najveći broj karaktera koje treba štampati iz niske u slučaju štampanja niske ili broj tačaka iza decimalne tačke u slučaju broja u pokretnom zarezu ili najmanji broj cifara u slučaju celog broja.
- Karakter koji označava modifikator dužine (na primer, h označava da ceo broj treba da se štampa kao short).

Konverzioni karakteri

Konverzioni karakteri su prikazani u narednoj tabeli. Ukoliko se nakon % navede pogrešan konverzioni karakter, ponašanje je nedefinisano.

Karakter	Tip	Štampa se kao
d,i	int	dekadni broj
ld,li	long	dekadni broj
lld,lli	long long	dekadni broj
lu	unsigned long	dekadni broj
o	int	neoznačeni oktalni broj (bez vodećeg simbola 0)
x,X	int	neoznačeni heksadekadni broj (bez vodećih 0x ili 0X), korišćenjem abcdef ili ABCDEF za 10, ..., 15
u	int	neoznačeni dekadni broj
lu	long	neoznačeni dekadni broj
llu	long long	neoznačeni dekadni broj
c	int	pojedinačni karakter

Karakter	Tip	Štampa se kao
`s`	`char *`	štampa karaktere niske do karaktera `'\0'` ili broja karaktera navedenog u okviru preciznosti
`f`	`double`	`[-]m.dddddd`, gde je broj d-ova određen preciznošću (podrazumevano 6)
`e,E`	`double`	`[-]m.dddddde+/-xx` ili `[-]m.ddddddE+/-xx`, gde je broj d-ova određen preciznošću (podrazumevano 6)
`g,G`	`double`	koristi %e ili %E ako je eksponent manji od -4 ili veći ili jednak preciznosti; inače koristi `%f`. Završne nule i završna decimalna tačka se ne štampaju
`p`	`void *`	pokazivač (reprezentacija zavisna od implementacije)
`%`		nijedan argument se ne konvertuje; štampa %

Za navedene konverzione karaktere postoje i modifikatori dužine: `h` (za celobrojne tipove, za `short`), `l` (za celobrojne tipove, za `long`), `L` (za brojeve u pokretnom zarezu, za `long double`). Tako, na primer, sekvenca `hd` može se koristiti za ispisivanje podatka tipa `short int`, sekvenca `ld` može se koristiti za ispisivanje podatka tipa `long int`, a sekvenca `Lf` za ispisivanje podatka tipa `long double`.

Primetimo da za format `float` ne postoji zaseban konverzioni karakter. To je zbog toga što se svi argumenti tipa `float` implicitno konvertuju u tip `double` prilikom poziva funkcije `printf` (jer je `printf` funkcija sa promenljivim brojem argumenata kod kojih je ova promocija podrazumevana). Počevši od standarda C99 pored formata `%f` dopušteno je navoditi i `%lf` (kompilator GCC to i zahteva za tip `double`).

Širina polja ili preciznost se mogu zadati i kao `*`, i tada se njihova vrednost određuje na osnovu vrednosti narednog argumenta (koji mora biti `int`). Na primer, da se odštampa najviše `max` karaktera iz niske `s`, može se navesti:

```
printf("%.*s", max, s);
```

Prilikom poziva funkcije `printf` na osnovu prvog argumenta (format niske) određuje se koliko još argumenata sledi i koji su njihovi tipovi. Ukoliko se ne navede odgovarajući broj argumenata ili se navedu argumenti neodgovarajućih tipova, dolazi do greške. Takođe, treba razlikovati naredna dva poziva:

```
printf(s);          /* pogresno ako s sadrzi karakter % */
printf("%s", s);    /* bezbedno */
```

Primeri:

Prikaz celih brojeva:

```
int count = -9234;
printf("Celobrojni formati:\n"
       "\tDekadni: %d  Poravnat: %.6d  Neoznacen: %u\n",
       count, count, count);
printf("Broj %d prikazan kao:\n\tHex: %Xh C hex: 0x%x Oct: %o",
        count, count, count, count );
```

```
Celobrojni formati:
   Dekadni: -9234  Poravnat: -009234  Neoznacen: 4294958062
Broj -9234 prikazan kao:
   Hex: FFFFDBEEh C hex: 0xffffdbee Oct: 37777755756
```

Prikaz konstanti u različitim osnovama:

```
printf("Cifre 10 predstavljaju:\n");
printf("\tHex: %i  Octal: %i  Decimal: %i\n", 0x10, 010, 10);
```

```
Cifre 10 predstavljaju:
   Hex: 16  Octal: 8  Decimal: 10
```

Prikaz karaktera:

```
char ch = 'h';
printf("Karakteri u polju date sirine:\n%10c%c\n", ch, ch);
```

```
Karakteri u polju date sirine:
         hh
```

Prikaz brojeva u pokretnom zarezu:

```
double fp = 251.7366;
printf("Brojevi u pokretnom zarezu:\n\t%f %.2f %e %E\n", fp, fp, fp, fp);
```

```
Realni brojevi:
   251.736600 251.74 2.517366e+002 2.517366E+002
```

Prikaz niski: Ovaj primer prikazuje navođenje širine polja, poravnanja i preciznosti prilikom štampanja niski. U narednoj tabeli dat je efekat različitih format niski na štampanje niske `"hello, world"` (12 karaktera). Dvotačke su stavljene radi boljeg ilustrovanja efekta raznih formata.

```
:%s:          :hello, world:
:%10s:        :hello, world:
:%.10s:       :hello, wor:
:%-10s:       :hello, world:
:%.15s:       :hello, world:
:%-15s:       :hello, world   :
:%15.10s:     :     hello, wor:
:%-15.10s:    :hello, wor     :
```

1.7.4 Formatirani ulaz — `scanf`

Funkcija `scanf` je ulazni analogon funkcije `printf`. Funkcija `scanf` čita karaktere sa standardnog ulaza, interpretira ih na osnovu specifikacije navedene format niskom i smešta rezultat na mesta određena ostalim argumentima:

```
int scanf(const char *format, ...);
```

Opis format niske dat je u nastavku. Ostali argumenti moraju biti pokazivači i određuju lokacije na koje se smešta konvertovani ulaz. Kao i u slučaju funkcije `printf`, ovde će biti dat samo prikaz najčešćih načina korišćenja ove funkcije. Funkcija `scanf` prestaje sa radom kada iscrpi svoju format nisku ili kada neki deo ulaza ne može da se uklopi u shemu navedenu format niskom. Funkcija vraća broj uspešno uklopljenih i dodeljenih ulaznih podataka. U slučaju kraja toka podataka, funkcija vraća `EOF`. Ova vrednost je različita od vrednosti 0 koja se vraća u slučaju da tekući karakter sa ulaza ne može da se uklopi u prvu specifikaciju zadatu format niskom. Svaki naredni poziv funkcije `scanf` nastavlja tačno od mesta na ulazu na kojem je prethodni poziv stao.

Format niska sadrži specifikacije konverzija kojima se kontroliše konverzija teksta pročitanog sa ulaza. Format niska može da sadrži:

- Praznine ili tabulatore koji se ne zanemaruju.
- Obične karaktere (ne %), za koje se očekuje da se poklope sa sledećim ne-belinama sa standardnog ulaza.
- Specifikacije konverzija, koje se sastoje od karaktera %, opcionog karaktera `*` koji sprečava dodeljivanje, opcionog broja kojim se navodi maksimalna širina polja, kao i od konkretnog konverzionog karaktera.
- Specifikacija konverzije određuje konverziju narednog ulaznog polja. Obično se rezultat ove konverzije upisuje u promenljivu na koju pokazuje naredni ulazni argument. Ipak, ako se navede karakter `*`, konvertovana vrednost se preskače i nigde ne dodeljuje. Ulazno polje se definiše kao niska karaktera koji nisu beline. Ono se prostire ili do narednog karaktera beline ili do širine polja, ako je ona eksplicitno zadana. Ovo znači da funkcija `scanf` može da čita ulaz i iz nekoliko različitih linija sa standardnog ulaza, jer se prelasci u novi red tretiraju kao beline.
- Konverzioni karakter upućuje na željenu interpretaciju ulaznog polja.

U narednoj tabeli navedeni su konverzioni karakteri.

Karakter	Tip	Ulazni podatak
`d`	`int*`	dekadni ceo broj
`i`	`int*`	ceo broj, može biti i oktalan (ako ima vodeću 0) ili heksadekadan (vodeći 0x ili 0X)
`o`	`int*`	oktalni ceo broj (sa ili bez vodeće nule)
`u`	`unsigned*`	neoznačeni dekadni broj
`x`	`int*`	heksadekadni broj (sa ili bez vodećih 0x ili 0X)
`c`	`char*`	karakter, naredni karakter sa ulaza se smešta na navedenu lokaciju; uobičajeno preskakanje belina se ne vrši u ovom slučaju; Za čitanje prvog ne-belog karaktera, koristi se %1s
`s`	`char*`	niska (bez navodnika), ukazuje na niz karaktera dovoljno dugačak za nisku uključujući i terminalnu `'\0'` koja se automatski dopisuje
`e,f,g`	`double*`	broj u pokretnom zarezu sa opcionim znakom, opcionom decimalnom tačkom i opcionim eksponentom
%		doslovno %; ne vrši se dodela

Slično kao kod funkcije `printf`, ispred karaktera konverzije `d`, `i`, `o`, `u`, i `x` moguće je navesti i karakter `h` koji ukazuje da se u listi argumenata očekuje pokazivač na `short`, karakter `l` koji ukazuje da se u listi argumenata očekuje pokazivač na `long` ili karakter `L` koji ukazuje da se u listi argumenata očekuje pokazivač na `long double`.

Konverzioni karakteri `d` i `i` imaju isto ponašanje u okviru funkcije `printf`, ali delom različito u okviru funkcije `scanf`. Naime, konverzioni karakter `d` označava da se očekuje označeni ceo broj zapisan u dekadnom sistemu, dok konverzioni karakter `i` označava da se očekuje ceo broj zapisan u dekadnom, oktalnom ili heksadekadnom sistemu (na primer, `10`, `012` ili `0xA`).

Kao što je već rečeno, argumenti funkcije `scanf` moraju biti pokazivači. Najčešći oblik pogrešnog korišćenja ove funkcije je:

```
scanf("%d", n);
```

umesto

```
scanf("%d", &n);
```

Ova greška se obično ne otkriva tokom kompilacije.

Funkcija `scanf` ignoriše beline u okviru format niske. Takođe, preskaču se beline tokom čitanja ulaznih vrednosti. Pozivi funkcije `scanf` mogu biti pomešani sa pozivima drugih funkcija koje čitaju standardni ulaz. Naredni poziv bilo koje ulazne funkcije će pročitati prvi karakter koji nije pročitan tokom poziva funkcije `scanf`.

Dužina niske pročitane od strane funkcije `scanf` može biti ograničena dodatnim parametrom. Na primer, sledeći kod:

```
scanf("%3s", s);
```

obezbeđuje da funkcija neće pročitati više od tri karaktera (završna nula se ne broji). Bez ovakve provere funkcija `scanf` je, kao i funkcija `gets`, opasna za korišćenje. Standard C11 uvodi i funkciju `scanf_s` (koja, ipak nije podržana od strane svih prevodioca).

Primeri

U narednom primeru, korišćenjem funkcije `scanf` implementiran je jednostavni kalkulator:

```
#include <stdio.h>

int main()
{
    double sum, v;

    sum = 0.0;
    while (scanf("%f", &v) == 1)
        printf("\t%.2f\n", sum += v);
    return 0;
}
```

Datume sa ulaza koji su u formatu `25 Dec 1988`, moguće je čitati korišćenjem:

```
int day, year;
char monthname[20];

scanf("%d %s %d", &day, monthname, &year);
```

Ispred monthname ne treba navesti simbol &, jer je ime niza već pokazivač. Doslovni karakteri se mogu pojaviti u okviru format niske i u tom slučaju se oni očekuju i na ulazu. Tako, za čitanje datuma sa ulaza koji su u formatu 12/25/1988, moguće je koristiti:

```
int day, month, year;
scanf("%d/%d/%d", &month, &day, &year);
```

1.7.5 Ulaz iz niske i izlaz u nisku

Funkcija printf vrši ispis formatiranog teksta na standardni izlaz. Standardna biblioteka jezika C definiše funkciju sprintf koja je veoma slična funkciji printf, ali rezultat njenog rada je popunjavanje niske karaktera formatiranim tekstom. Prototip funkcije je:

```
int sprintf(char *string, const char *format, arg1, arg2, ...);
```

Ova funkcija formatira argumente arg1, arg2, ... na osnovu format niske, a rezultat smešta u nisku karaktera prosleđenu kao prvi argument, podrazumevajući da je ona dovoljno velika da smesti rezultat.

Slično funkciji sprintf koja vrši ispis u nisku karaktera umesto na standardni izlaz, standardna biblioteka jezika C definiše i funkciju sscanf koja je analogna funkciji scanf, osim što ulaz čita iz date niske karaktera, umesto sa standardnog ulaza.

```
int sscanf(const char* input, char* format, ...);
```

Navedimo primer korišćenja ove funkcije. Da bi se pročitao ulaz čiji format nije fiksiran, najčešće je dobro pročitati celu liniju sa ulaza, a zatim je analizirati korišćenjem sscanf. Da bi se sa ulaza pročitao datum koji je zadat u jednom od nekoliko mogućih formata, moguće je koristiti sledeći kod:

```
while (getline(line, sizeof(line)) > 0) {
  if (sscanf(line, "%d %s %d", &day, monthname, &year) == 3)
      printf("valid: %s\n", line); /* 25 Dec 1988 */
  else if (sscanf(line, "%d/%d/%d", &month, &day, &year) == 3)
      printf("valid: %s\n", line); /* 12/25/1988 */
  else
      printf("invalid: %s\n", line); /* pogresan oblik */
}
```

1.7.6 Ulaz iz datoteka i izlaz u datoteke

Do sada opisane funkcije za ulaz i izlaz uglavnom su radile sa **standardnim ulazom i izlazom**. Njima program automatski ima pristup preko operativnog sistema. Iako je pomoću *preusmeravanja* bilo moguće povezati program sa lokalnim datotekama, taj mehanizam je ograničen — ne omogućava istovremeno čitanje (ili pisanje) više datoteka niti kombinaciju datoteke i standardnog ulaza.

Zbog toga C nudi **direktnu podršku za rad sa datotekama**, bez zavisnosti od preusmeravanja. Sve potrebne deklaracije nalaze se u zaglavlju `<stdio.h>`.

Tekstualne i binarne datoteke

Svaka datoteka je niz bajtova, ali razlikujemo **tekstualne** i **binarne** datoteke:

- **Tekstualne datoteke** čiji sadržaj u principu čine vidljivi karakteri, sa dodatkom oznake kraja reda i horizontalnog tabulatora.

  Tipično se obrađuju *liniju po liniju*. Standard C uvodi *tekstualni mod* kako bi sakrio razlike između sistema (npr. Windows koristi `\r\n` kao oznaku kraja reda, a Linux `\n`). Programeru se uvek prikazuje samo `\n`.

  > Primer: ako na Windowsu upišemo `\n`, u datoteku će biti zapisano `\r\n`.

- **Binarne datoteke** (npr. slike `.jpg`, arhive `.zip`) mogu sadržati bajtove svih vrednosti 0–255. Kod njih nema konverzije — bajtovi se čitaju i upisuju doslovno.

U jezicima sa Unicode podrškom razlika je još izraženija, jer se više bajtova može mapirati u jedan znak.

Pristupanje datoteci

Da bi se pristupilo datoteci, bilo za čitanje, bilo za pisanje, potrebno je izvršiti određenu vrstu povezivanja datoteke i programa. Za ovo se koristi bibliotečka funkcija `fopen`:

```
FILE* fopen(const char* filename, const char* mode);
```

- `filename` — ime datoteke (npr. `"datoteka.txt"`)
- `mode` — način otvaranja:
  - `"r"` — čitanje (greška ako datoteka ne postoji)
  - `"w"` — pisanje (kreira novu ili briše postojeću datoteku)
  - `"a"` — dopisivanje (dodaje na kraj datoteke)
  - `"r+"`, `"w+"`, `"a+"` — kombinacija čitanja i pisanja
  - dodavanje `"b"` — binarni mod (npr. `"rb"`, `"wb"`)

Ako otvaranje ne uspe, `fopen` vraća `NULL`.

Funkcija vraća pokazivač na strukturu tipa `FILE` (definisanu pomoću `typedef`), koju program koristi pri svim narednim operacijama sa datotekom. Ova struktura sadrži informacije o datoteci koje će biti korišćene prilikom svakog pristupa datoteci. Ove informacije mogu da uključe adresu početka *bafera* (*prihvatnik*, eng.~buffer) kroz koji se vrši komunikacija sa datotekom, tekuću poziciju u okviru bafera, informaciju o tome da li se došlo do kraja datoteke, da li je došlo do greške i slično. Programer ne mora i ne bi trebalo da direktno koristi ove informacije, već je dovoljno da čuva pokazivač na strukturu `FILE` i da ga prosleđuje svim funkcijama koje treba da pristupaju datoteci.

Prilikom pokretanja programa, operativni sistem automatski otvara tri toka podataka (kao da su datoteke):

```
FILE* stdin;   // standardni ulaz
FILE* stdout;  // standardni izlaz
FILE* stderr;  // izlaz za greške
```

Zatvaranje datoteka

Za zatvaranje se koristi funkcija `fclose`:

```
int fclose(FILE *fp);
```

- prekida vezu između programa i datoteke,
- prazni bafer i osigurava da je sadržaj upisan na disk,
- oslobađa memoriju zauzetu za strukturu `FILE`.

Iako se datoteke automatski zatvaraju kada program završi, **dobra praksa** je ručno zatvoriti svaku datoteku čim prestane da bude potrebna. Ovo je posebno važno jer operativni sistemi ograničavaju broj istovremeno otvorenih datoteka.

Ulaz i izlaz pojedinačnih karaktera

Nakon što se otvori datoteka, iz nje se čita ili se u nju piše sadržaj. To mogu biti i pojedinačni karakteri.

Funkcija `getc` vraća naredni karakter iz datoteke određene prosleđenim `FILE` pokazivačem ili `EOF` u slučaju kraja datoteke ili greške:

```
int getc(FILE *fp);
```

Slično, funkcija `putc` upisuje dati karakter u datoteku određenu prosleđenim `FILE` pokazivačem i vraća upisani karakter ili `EOF` u slučaju greške. Iako su greške prilikom izlaza retke, one se mogu javiti (npr. kada se popuni hard disk), pa bi trebalo proveravati i ovakve povratne vrednosti:

```
int putc(int c, FILE *fp);
```

Slično kao i `getchar` i `putchar`, i `getc` i `putc` mogu biti definisani kao makroi, a ne funkcije.

Primer: Kopiranje datoteke karakter po karakter

Naredni program kopira sadržaj datoteke `ulazna.txt` u datoteku `izlazna.txt` čitajući i pišući pritom pojedinačne karaktere. Iako ispravan, ovaj način kopiranja je veoma neefikasan:

```
#include <stdio.h>

/* filecopy:  copy file ifp to file ofp */
void filecopy(FILE *ifp, FILE *ofp)
{
     int c;
     while ((c = getc(ifp)) != EOF)
         putc(c, ofp);
}

int main()
{
     FILE *ulaz, *izlaz;

     ulaz = fopen("ulazna.txt","r");
     if(ulaz == NULL)
         return -1;

     izlaz = fopen("izlazna.txt","w");
     if(izlaz == NULL)
         return -1;

     filecopy(ulaz,izlaz);

     fclose(ulaz);
     fclose(izlaz);
     return 0;
}
```

Funkcija `ungetc`

Funkcija `ungetc` „vraća“ karakter u tok datoteke i vraća ga tako da će naredni poziv za čitanje iz iste datoteke vratiti upravo taj karakter. Vraćeni karakter ne mora biti jednak poslednjem pročitanom karakteru. On se smešta u bafer u memoriji (ne u samu datoteku na disku).

```
int ungetc (int c, FILE *fp);
```

Funkcija vraća `EOF` u slučaju greške, a inače vraća prosleđeni karakter.

Primer: Čitanje celog broja

Primer čita karaktere iz datoteke dok su u pitanju cifre i pri tom gradi dekadni ceo broj. Poslednji karakter koji nije cifra „vraća“ se u tok pomoću `ungetc` da bi mogao biti pročitan pri sledećem čitanju:

```
#include <stdio.h>

int readint(FILE* fp) {
  int val = 0, c;
  while (isdigit(c = getc(fp)))
     val = 10*val + (c - '0');
  ungetc(c, fp);
}
```

Provera grešaka i kraja datoteke

Funkcija `feof` vraća vrednost *tačno* (ne-nula) ukoliko se došlo do kraja date datoteke:

```
int feof(FILE *fp);
```

Funkcija `ferror` vraća vrednost *tačno* (ne-nula) ukoliko je došlo do greške u radu sa datotekom:

```
int ferror(FILE *fp);
```

Linijski ulaz i izlaz

Standardna biblioteka definiše i funkcije za rad sa tekstualnim datotekama liniju po liniju.

Funkcija `fgets` čita jednu liniju iz datoteke:

```
char *fgets(char *line, int maxline, FILE *fp);
```

- Čita sledeću liniju (uključujući oznaku kraja reda) iz datoteke `fp` i rezultat smešta u nisku `line`.
- Može da bude pročitano najviše `maxline-1` karaktera.
- Rezultujuća niska završava se karakterom `'\0'`.
- U normalnom toku izvršavanja vraća pokazivač `line`, a u slučaju kraja datoteke ili greške vraća `NULL`.

Funkcija `fputs` ispisuje nisku (koja ne mora da sadrži oznaku kraja reda) u datoteku:

```
int fputs(const char *line, FILE *fp);
```

- Vraća `EOF` u slučaju greške, a neki nenegativan broj inače.

Za razliku od `fgets` i `fputs`, funkcija `gets` briše završni `'\n'`, a funkcija `puts` ga dodaje.

Implementacija funkcija `fgets` i `fputs`

Implementacija bibliotečkih funkcija nije značajno različita od implementacije ostalih funkcija, što ilustruje primer doslovno preuzet iz jedne realne implementacije C biblioteke:

```
/* fgets:  get at most n chars from iop */
char *fgets(char *s, int n, FILE *iop)
{
    register int c;
    register char *cs;

    cs = s;
    while (--n > 0 && (c = getc(iop)) != EOF)
        if ((*cs++ = c) == '\n')
            break;
    *cs = '\0';
    return (c == EOF && cs == s) ? NULL : s;
}

/* fputs:  put string s on file iop */
int fputs(char *s, FILE *iop)
{
    register int c;

    while (c = *s++)
       putc(c, iop);
    return ferror(iop) ? EOF : 0;
}
```

Primer: funkcija `getline`

Korišćenjem funkcije `fgets` jednostavno je napraviti funkciju koja čita liniju po liniju sa standardnog ulaza, vraćajući dužinu pročitane linije, odnosno nulu kada liniju nije moguće pročitati:

```
/* getline:  read a line, return length */
int getline(char *line, int max)
{
     if (fgets(line, max, stdin) == NULL)
        return 0;
     else
        return strlen(line);
}
```

Formatirani ulaz i izlaz

Za formatirani ulaz i izlaz koriste se funkcije `fscanf` i `fprintf`. One su identične funkcijama `scanf` i `printf`, osim što im je prvi argument pokazivač tipa `FILE` koji ukazuje na datoteku iz koje se čita, odnosno u koju se piše. Format niska u ovom slučaju dolazi kao drugi argument.

```
int fscanf(FILE *fp, const char *format, ...);
int fprintf(FILE *fp, const char *format, ...);
```

Rad sa binarnim datotekama

Standardna biblioteka nudi funkcije za direktno čitanje i pisanje bajtova u binarne datoteke. Programeru su na raspolaganju i funkcije za pozicioniranje u okviru datoteka, pa se binarne datoteke mogu koristiti kao neka vrsta memorije sa slobodnim pristupom.

Funkcija `fread` se koristi za čitanje niza slogova iz binarne datoteke, a `fwrite` za pisanje niza slogova u binarnu datoteku:

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

- **ptr** – adresa u memoriji na koju se smeštaju pročitani slogovi, ili gde se nalaze slogovi koji će biti upisani.
- **size** – veličina jednog sloga (u bajtovima).
- **nmemb** – broj slogova.
- **stream** – pokazivač na datoteku.

Obe funkcije vraćaju broj uspešno pročitanih, odnosno upisanih slogova.

Za kretanje kroz datoteku koriste se funkcije `fseek` i `ftell`.

```
int fseek(FILE *stream, long int offset, int origin);
long int ftell(FILE *stream);
```

- `fseek` – pomera poziciju u datoteci za `offset` bajtova, u odnosu na `origin`:
  - `SEEK_SET` – od početka datoteke,
  - `SEEK_CUR` – od trenutne pozicije,
  - `SEEK_END` – od kraja datoteke.
- `ftell` – vraća trenutnu poziciju u datoteci izraženu u broju bajtova od početka.

Primer: Rad sa binarnom datotekom

```
#include <stdio.h>

int main() {
  struct S {
    int broj;
    int kvadrat;
  } s;

  FILE* f;
  int i;
```

```
  if ((f = fopen("junk", "r+b")) == NULL) {
    fprintf(stderr, "Greska prilikom otvaranja datoteke");
    return 1;
  }

  for (i = 1; i <= 5; i++) {
     s.broj = i;
     s.kvadrat = i*i;
     fwrite(&s, sizeof(struct S), 1, f);
  }

  printf("Upisano je %ld bajtova\n", ftell(f));

  for (i = 5; i > 0; i--) {
    fseek(f, (i-1)*sizeof(struct S), SEEK_SET);
    fread(&s, sizeof(struct S), 1, f);
    printf("%3d %3d\n", s.broj, s.kvadrat);
  }

  fclose(f);
}
```

1.7.7 Argumenti komandne linije programa

Jedan način da se određeni podaci proslede programu jeste da se navedu u komandnoj liniji prilikom njegovog pokretanja. Argumenti koji su tako navedeni prenose se programu kao argumenti funkcije `main`.

- `argc` (*argument count*) – broj argumenata komandne linije (uključujući i ime programa).
- `argv` (*argument vector*) – niz niski karaktera koje sadrže argumente. Svaka niska direktno odgovara jednom argumentu.

Naziv programa je uvek u `argv[0]`. Ako je `argc == 1`, to znači da nisu navedeni dodatni argumenti osim imena programa. Dakle:

- `argv[0]` – ime programa
- `argv[1]` … `argv[argc-1]` – argumenti programa
- `argv[argc]` – ima vrednost `NULL`

Identifikatori `argc` i `argv` nisu obavezni – funkcija `main` može biti deklarisana i na sledeći način:

```
int main (int br_argumenata, char* argumenti[]);
```

Jednostavan program koji štampa broj argumenata i sadržaj vektora argumenata:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
   int i;
   printf("argc = %d\n", argc);

   /* Nulti argument uvek je ime programa (na primer, a.out) */
   for (i = 0; i < argc; i++)
       printf("argv[%d] = %s\n", i, argv[i]);
   return 0;
}
```

Ako se program prevede sa:

`gcc -o echoargs echoargs.c`

i pozove sa:

`./echoargs -U zdravo svima "dobar dan"`

ispisaće:

```
argc = 5
argv[0] = ./echoargs
argv[1] = -U
argv[2] = zdravo
argv[3] = svima
argv[4] = dobar dan
```

Česta upotreba argumenata komandne linije jeste zadavanje **opcija** programu. Opcije se obično kodiraju kao karakteri iza `-` (ili `--`, a na Windows sistemima i `/`). Moguće je navesti više opcija zajedno iza jednog `-`.

Na primer:

`./program -a -bcd 134 -ef zdravo`

Programu se zadaju opcije `a`, `b`, `c`, `d`, `e`, `f` i argumenti `134` i `zdravo`.

Primer: ispis opcija

Program koji ispisuje sve pronađene opcije:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
   /* Za svaki argument komandne linije, počevši od argv[1]
      (preskače se ime programa) */
   int i;
   for (i = 1; i < argc; i++) {
      /* Ukoliko i-ti argument počinje crticom */
```

```
        if (argv[i][0] == '-') {
            /* Ispisuju se sva njegova slova od pozicije 1 */
            int j;
            for (j = 1; argv[i][j] != '\0'; j++)
                printf("Prisutna je opcija : %c\n", argv[i][j]);
        }
    }
    return 0;
}
```

Kraće, ali dosta kompleksnije rešenje se intenzivno zasniva na pokazivačkoj aritmetici i prioritetu operatora.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  char c;
  /* Dok još ima argumenata i dok je na poziciji 0 crtica */
  while (--argc > 0 && (*++argv)[0] == '-')
     /* Dok se ne dođe do kraja tekuće niske */
     while (c = *++argv[0])
       printf("Prisutna je opcija : %c\n", c);
  return 0;
}
```

Glava 2

Neki napredni koncepti jezika C++

Jezik **C++** nastao je kao proširenje jezika **C** sa ciljem da omogući efikasno, ali i bezbednije i modularnije programiranje. Dok je C prvenstveno proceduralni jezik fokusiran na strukture podataka i funkcije, C++ uvodi niz **abstrakcijskih mehanizama** koji omogućavaju bolju organizaciju i ponovnu upotrebu koda, kao i snažnije garancije o tačnosti i bezbednosti.

U ovom poglavlju biće prikazane one osobine jezika C++ koje su posebno važne pri izradi efikasnih i pouzdanih programa, a koje dopunjuju znanja stečena u jeziku C. Posebna pažnja posvećena je **tipovima podataka** i **zaključivanju tipova**, **dinamičkoj alokaciji memorije** i savremenim načinima njenog upravljanja putem **pametnih pokazivača**. Zatim se razmatraju **šabloni** kao osnovni mehanizam generičkog programiranja, kao i **semantika premeštanja** koja omogućava da se resursi prenose, a ne kopiraju. Na kraju, biće prikazan **rad sa datotekama** pomoću *stream* biblioteke i ukratko pomenute moderne konstrukcije standardne biblioteke, poput `std::ranges`, koje dodatno pojednostavljuju rad sa kolekcijama podataka.

Cilj poglavlja nije da se obuhvati čitav jezik C++, već da se pruži jasan uvid u one koncepte koji se najčešće koriste u savremenom programiranju i koji prirodno nadograđuju gradivo obrađeno u jeziku C.

2.1 Tipovi, zaključivanje tipova i životni vek objekata

U prethodnim poglavljima prikazani su osnovni tipovi i mehanizmi jezika C. U ovom poglavlju proširujemo tu sliku i bavimo se nekim specifičnostima C++ jezika koje ga razlikuju od C-a, ali i čine znatno snažnijim i bezbednijim.

Posebno ćemo razmotriti zaključivanje tipova pomoću ključne reči `auto`, inicijalizacione liste (`std::initializer_list`), statičke članove klasa, kao i doseg i životni vek objekata. Na kraju, osvrnućemo se i na najčešće korišćene savremene tipove podataka: `std::vector` i `std::string`, koji u velikoj meri zamenjuju tradicionalne C koncepte — nizove i niske.

2.1.1 Zaključivanje tipova pomoću `auto`

U jeziku C ključna reč `auto` označavala je promenljivu sa **automatskim trajanjem** — tj. promenljivu koja se kreira i uništava pri ulasku i izlasku iz bloka. Kako su sve lokalne promenljive podrazumevano automatskog životnog veka, ova reč je u praksi bila suvišna.

U C++ jeziku, međutim, `auto` je dobila **potpuno novo značenje**: ona govori prevodiocu da **sam zaključi tip promenljive** na osnovu izraza na desnoj strani dodele. Drugim rečima, `auto` nije način

da se „sakrije“ tip, već da se on prepusti prevodiocu — čime se smanjuje količina koda i rizik od grešaka pri promeni tipova.

```
auto a = 10;        // int
auto b = 3.14;      // double
auto c = 'x';       // char
auto p = &a;        // int*
auto& r = a;        // int& (referenca)
```

Zaključivanje tipa doprinosi **čistijem i otpornijem kodu**, posebno kada tip izraza nije trivijalan ili se menja prilikom refaktorisanja. Tipičan primer su iteratori ili izrazi koji uključuju šablone:

```
vector<int> v = {1, 2, 3, 4};
for (auto it = v.begin(); it != v.end(); ++it)
    cout << *it << " ";
```

Bez `auto`, tip iteratora (`vector<int>::iterator`) bio bi nepotrebno dug i teško čitljiv. Korišćenjem `auto` kod postaje jasniji i manje podložan greškama ako se promeni tip elementa u vektoru.

Iako `auto` olakšava pisanje, njegova nepažljiva upotreba može dovesti do nejasnoća ili neželjenih konverzija. Na primer:

```
auto zbir = 10 + 5.5;   // tip je double, ne int
auto niz = {1, 2, 3};   // tip je std::initializer_list<int>, ne vektor
```

U drugom primeru, mnogi programeri očekuju da `niz` bude `std::vector<int>`, ali `auto` u ovom kontekstu zaključuje tip kao `std::initializer_list<int>`. Zbog toga je preporučljivo `auto` koristiti samo kada je tip **očigledan iz konteksta** ili kada bi njegovo eksplicitno navođenje **smanjilo čitljivost**.

Zaključivanje tipova u funkcijama

U modernom C++ moguće je da i funkcije koriste zaključivanje tipa pri povratnoj vrednosti:

```
auto saberi(int a, int b) {
    return a + b;  // tip povratne vrednosti se zaključuje kao int
}
```

Ova mogućnost olakšava generičko programiranje i pojednostavljuje deklaracije šablona.

2.1.2 Zagrade {} za inicijalizaciju

Od C++11 standarda uveden je novi, **jedinstveni način inicijalizacije** koji koristi viticaste zagrade ({}). Naziva se *uniformna inicijalizacija* jer omogućava da se na isti način inicijalizuju **svi tipovi objekata** — ugrađeni tipovi, strukture, klase, nizovi, pa čak i objekti sa konstruktorima.

```
int a{10};
double b{3.14};
vector<int> v{1, 2, 3};
```

Ova sintaksa ne samo da ujednačava inicijalizaciju, već **sprečava niz suptilnih grešaka** koje su bile moguće u ranijim oblicima inicijalizacije pomoću = ili ().

Kada se koristi operator =, kompajler po pravilu dozvoljava konverzije koje mogu dovesti do gubitka informacija. Na primer:

```
int x = 3.14;    // dozvoljeno! x postaje 3
```

C++ dopušta ovakvu konverziju, jer automatski odbacuje decimalni deo pri dodeli u `int`. Ovo je legalno, ali često neželjeno.

Upotrebom viticastih zagrada kompajler **zabranjuje implicitne konverzije koje gube informaciju**:

```
int x{3.14};     // greška pri prevođenju
```

Ovakvo ponašanje znatno smanjuje mogućnost grešaka u velikim programima i čini kod sigurnijim.

Inicijalizacija pomoću zagrada () može ponekad izazvati zabunu između **deklaracije funkcije** i **deklaracije promenljive**. Poznat primer je čuvena „*Most Vexing Parse*“ situacija:

```
struct T { T(){} };

T t1();   // Deklaracija funkcije koja vraća T, a ne objekta t1!
T t2{};   // Ispravno: objekat tipa T
```

Korišćenje {} uvek jednoznačno označava inicijalizaciju objekta, a ne deklaraciju funkcije.

Viticaste zagrade mogu se koristiti i kod **objekata sa konstruktorima**, gde omogućavaju eksplicitnu inicijalizaciju svih članova:

```
struct Tacka { int x, y; };

Tacka p{3, 4};    // inicijalizuje x=3, y=4
```

ili kod složenih tipova:

```
vector<pair<int, int>> v{{1, 2}, {3, 4}, {5, 6}};
```

Ovakva inicijalizacija je **kraća i bezbednija** od ručnog pozivanja konstruktora.

Korišćenje viticastih zagrada {} za inicijalizaciju predstavlja **moderni, bezbedni idiom** u C++ jeziku. Time se sprečavaju greške prilikom konverzija tipova, uklanja dvosmislenost u deklaracijama i obezbeđuje jedinstven mehanizam inicijalizacije za sve tipove podataka. Zato se u savremenom C++-u preporučuje:

> **Uvek koristiti zagrade {} za inicijalizaciju, osim kada postoji jasna potreba za drugim oblikom (npr. kod konstrukcije `std::vector` sa brojem elemenata).**

2.1.3 Liste inicijalizacije i `std::initializer_list<T>`

Kada se koristi u klasama, uniformna inicijalizacija (potrebu viticastih zagrada ({}) za inicijalizaciju bilo kog tipa objekta) omogućava konstrukciju objekata pomoću liste vrednosti, koristeći tip `std::initializer_list<T>`.

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

struct Skup {
    Skup(initializer_list<int> lista) {
        cout << "Inicijalizovan sa " << lista.size() << " elemenata: ";
        for (int x : lista) cout << x << " ";
        cout << endl;
    }
};

int main() {
    Skup s = {1, 3, 5, 7};
}
```

Ovakav konstruktor omogućava jednostavnu i čitljivu inicijalizaciju objekata promenljivog broja argumenata, slično kao u jeziku Python ili JavaScript, ali bez gubitka tipne bezbednosti.

> **Napomena:** Uvek se bira konstruktor koji **najbolje odgovara** prosleđenom izrazu. Ako klasa ima i konstruktor sa `initializer_list` i neki drugi, prioritet ima upravo onaj sa `initializer_list`.

2.1.4 Statički članovi klase

U jeziku C ključna reč `static` ima značenje promenljive sa **statičkim trajanjem** — promenljive koja postoji tokom čitavog izvršavanja programa, bez obzira na opseg. U jeziku C++ `static` zadržava to značenje, ali se proširuje i na **članove klase**.

Statički član klase **nije vezan za pojedinačni objekat**, već postoji samo u **jednoj instanci** zajedničkoj za sve objekte te klase. Takvi članovi su pogodni za praćenje zajedničkog stanja svih objekata.

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

class Brojac {
public:
    static int ukupan_broj;  // zajednički za sve instance

    Brojac() { ukupan_broj++; }
};

int Brojac::ukupan_broj = 0; // definicija statičkog člana izvan klase
```

```
int main() {
    Brojac a, b, c;
    cout << "Kreirano objekata: " << Brojac::ukupan_broj << endl;
}
```

Rezultat:

```
Kreirano objekata: 3
```

Statičkim članovima se najčešće pristupa pomoću imena klase (`Brojac::ukupan_broj`), mada im je dozvoljen pristup i preko pojedinačnih objekata.

2.1.5 Doseg i životni vek objekata

Doseg određuje deo programa u kojem je ime vidljivo, dok **životni vek** označava koliko dugo objekat postoji u memoriji. Ova dva pojma nisu ista, iako su u C-u i C++-u najčešće povezani.

U okviru klasa, članovi mogu imati tri nivoa pristupa:

- `public` – dostupni svuda,
- `protected` – dostupni unutar klase i njenih izvedenih klasa,
- `private` – dostupni samo unutar klase.

Podrazumevani nivo pristupa u `class` je `private`, dok je u `struct` — `public`.

U pogledu životnog veka, C++ zadržava sve mehanizme poznate iz C-a: promenljive mogu biti **lokalne (automatske)**, **statičke** ili **dinamičke** (na hipu, pomoću `new`). Glavna razlika je u tome što C++ proširuje koncept upravljanja životnim vekom i omogućava automatsko oslobađanje resursa kroz **destruktore** i **RAII**.

2.1.6 Izuzeci i rukovanje greškama (`try–catch`)

Važno je pomenuti i mehanizam koji C++ uvodi za **bezbedno rukovanje greškama tokom izvršavanja programa** — sistem **izuzetaka** (*exceptions*). Za razliku od jezika C, gde se greške obično označavaju povratnim vrednostima funkcija, C++ omogućava da se greška „baci“ i kasnije „uhvati“ u odgovarajućem delu programa. Ovo pojednostavljuje kod i sprečava da se logika programa preopteretі proverama svake moguće greške.

Osnovna ideja je jednostavna: ako tokom izvršavanja dođe do greške (npr. otvaranje nepostojeće datoteke, neuspešna alokacija memorije, deljenje nulom i sl.), funkcija može da **„baci“ izuzetak** pomoću ključne reči `throw`. Taj izuzetak se zatim **„hvata“** u odgovarajućem `catch` bloku, dok deo koda koji može da izazove grešku stoji u okviru `try` bloka.

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

int podeli(int a, int b) {
    if (b == 0)
        throw runtime_error("Deljenje nulom!");
    return a / b;
```

```
}

int main() {
    try {
        cout << podeli(10, 2) << endl;    // 5
        cout << podeli(7, 0) << endl;     // baca izuzetak
    }
    catch (const exception& e) {           // hvata sve standardne izuzetke
        cerr << "Greška: " << e.what() << endl;
    }

    cout << "Program se normalno nastavlja." << endl;
}
```

Rezultat:

```
5
Greška: Deljenje nulom!
Program se normalno nastavlja.
```

Blok `try` sadrži „rizičan“ kod, dok `catch` definiše šta treba uraditi ako se desi izuzetak. Ako se izuzetak ne uhvati, program se prekida uz dijagnostičku poruku.

> **Napomena:** Mehanizam izuzetaka koristi se i u standardnoj biblioteci — npr. `new` baca `std::bad_alloc` ako nema memorije, `vector::at()` baca `std::out_of_range`, a `fstream` objekti mogu preći u stanje greške. U ovoj knjizi izuzetke ćemo pominjati samo informativno, ali je važno da se prepozna njihova uloga u savremenom C++ kodu.

2.1.7 Poređenje nizova i vektora

U jeziku C nizovi imaju **fiksnu veličinu**, poznatu u trenutku kompajliranja. To ih čini efikasnim, ali i ograničenim: njihova veličina se ne može menjati, a rad sa dinamičkim nizovima zahteva ručno upravljanje memorijom (`malloc`, `realloc`, `free`).

U jeziku C++ za tu svrhu se koristi `std::vector`, koji automatski upravlja dinamičkom memorijom i raste po potrebi.

Osobina	C niz	`std::vector<T>` u C++-u
Veličina	Fiksna	Dinamička, može da se menja
Inicijalizacija	`int a[3] = {1,2,3};`	`vector<int> a = {1,2,3};`
Pristup elementima	indeks `a[i]`	indeks `a[i]` (bezbedniji, sa proverenim granicama pomoću `at()`)
Broj elemenata	ručno računanje	`a.size()`
Dodavanje elemenata	nije podržano	`a.push_back(x)`
Oslobađanje memorije	ručno	automatski u destruktoru

Vektori predstavljaju **bezbedniju i moderniju alternativu nizovima** — oni brinu o sopstvenoj memoriji, proširuju se automatski, i mogu da se kopiraju ili prosleđuju funkcijama bez posebne brige o veličini.

2.1.8 Poređenje niske i stringa

Slična razlika postoji između *niski znakova* (`char[]` ili `char*`) u jeziku C i klase `std::string` u jeziku C++.

U C-u niska je samo niz karaktera koji završava nul-znakom (`'\0'`), dok je u C++-u `std::string` **klasa** koja obuhvata sve operacije nad tekstom — čitanje, spajanje, pretraživanje, upoređivanje, itd.

Osobina	C niska (`char*`)	`std::string` u C++
Skladištenje	niz karaktera + `'\0'`	dinamička memorija, automatsko upravljanje
Dužina	`strlen(s)`	`s.size()` ili `s.length()`
Spajanje	`strcat(dest, src)`	`s1 + s2`
Poređenje	`strcmp(s1, s2)`	`s1 == s2`, `s1 < s2`
Bezbednost	lako prekoračenje bafera	automatska provera i upravljanje memorijom

Zahvaljujući `std::string`, rad sa tekstom u C++-u postaje prirodan i čitljiv, bez potrebe za ručnim upravljanjem pokazivačima i nul-znakovima.

2.1.9 Const i reference

U modernom C++-u referenca (`T&`) omogućava prosleđivanje objekta bez kopiranja, a `const` garantuje da funkcija neće menjati prosleđeni argument. To je idiomatičan način za efikasno i bezbedno prosleđivanje „velikih“ objekata (npr. `std::string`, `std::vector`).

```
#include <string>
#include <iostream>

void ispisi(const std::string& s) {   // nema kopiranja, nema izmene
    std::cout << s << "\n";
}

void uvecaj(int& x) {                  // menja prosleđeni argument
    ++x;
}

int main() {
    std::string poruka = "MATF";
    ispisi(poruka);          // prosleđivanje kao const ref

    int a = 5;
    uvecaj(a);               // a postaje 6
}
```

Pravilo prakse:

- „Velike“ objekte čitati kao `const T&`.
- „Male“ trivijalne tipove (`int`, `double`) slati po vrednosti.
- Kada funkciji treba „vlasništvo“ (npr. da zadrži objekat), koristiti *move* ili pametne pokazivače (više o tome kasnije).

2.1.10 Preopterećenje operatora

Preopterećenje operatora omogućava da korisnički tipovi (npr. čvorovi, tačke) dobiju prirodno ponašanje: poređenje, sabiranje, ispis i sl. U strukturama podataka se najčešće preopterećuju ==, < (recimo, za uređenost u mapi, skupu), kao i operator ispisa <<.

```
#include <iostream>

struct Tacka {
    int x, y;
};

// uređenost: prvo po x, zatim po y
bool operator<(const Tacka& a, const Tacka& b) {
    return (a.x < b.x) || (a.x == b.x && a.y < b.y);
}

bool operator==(const Tacka& a, const Tacka& b) {
    return a.x == b.x && a.y == b.y;
}

// više o tokovima ispisa na kraju poglavlja
// ispis u obliku (x,y)
std::ostream& operator<<(std::ostream& out, const Tacka& t) {
    return out << "(" << t.x << "," << t.y << ")";
}
```

Napomena: Operatore definišite **van** klase kada je moguće.

2.1.11 Funkcijski objekti (funktori)

Funkcijski objekat (funktor) je objekat koji se, zahvaljujući preopterećenju operatora `operator()`, može koristiti kao funkcija i može da nosi sopstveno stanje ili pravila rada (npr. kriterijum za poređenje).

Primer — uređenje tačaka po `y` koordinati (opadajuće) za upis u skup:

```
struct Tacka { int x, y; };

struct VeceY {
    bool operator()(const Tacka& a, const Tacka& b) const {
```

```
        if (a.y != b.y) return a.y > b.y; // opadajuće po y
        return a.x < b.x;                 // zatim rastuće po x
    }
};

// skup tacaka, uređen „pravilom" VeceY
std::set<Tacka, VeceY> S;

S.insert({1, 5});
S.insert({0, 7});
S.insert({3, 5});
```

Struktura `VeceY` predstavlja **funkcijski objekat (funktor)**, tj. tip čiji se objekti mogu koristiti kao funkcije. To je omogućeno definisanjem **operatora poziva funkcije**, `operator()`, unutar strukture.

Deklaracija:

```
bool operator()(const Tacka& a, const Tacka& b) const
```

znači da se, kada se objekat tipa `VeceY` „pozove“ sa dva argumenta tipa `Tacka`, izvršava telo ovog operatora i vraća logičku vrednost (`bool`).

- `(const Tacka& a, const Tacka& b)` – operator prima dve tačke po **referenci na konstantu**. Time se izbegava kopiranje (radi efikasnosti), ali se obezbeđuje da funkcija **ne može menjati** te objekte (zbog `const`);
- završni `const` – označava da sam operator **ne menja stanje** objekta `VeceY`. To znači da operator može biti pozvan i nad konstantnim objektima, npr. `const VeceY cmp; cmp(t1, t2);`.

2.2 Dinamička alokacija memorije u C++

U ranijim poglavljima videli smo kako se u jeziku C dinamički prostor u memoriji može obezbediti funkcijama `malloc`, `calloc`, `realloc` i `free`. Ove funkcije omogućavaju da program tokom izvršavanja zatraži onoliko memorije koliko mu je stvarno potrebno, ali upravljanje takvom memorijom u potpunosti je **odgovornost programera**. To znači da je programer dužan da ručno oslobađa memoriju, proverava uspešnost alokacije i vodi računa o mogućim greškama poput curenja memorije ili pristupa oslobođenom prostoru.

Jezik C++ zadržava sve ove mogućnosti, ali uvodi **sopstvene mehanizme** koji čine rad sa dinamičkom memorijom **bezbednijim, jednostavnijim i tip-bezbednim**. Umesto funkcija iz `<stdlib.h>`, C++ koristi **operatore** `new` **i** `delete`, koji su integrisani u sam jezik i omogućavaju da se, pored alokacije prostora, automatski pozovu i **konstruktori** i **destruktori** objekata. Na taj način, proces alokacije i oslobađanja memorije postaje deo **objektno-orijentisanog modela** jezika i usklađen je sa njegovim osnovnim principima: **enkapsulacijom**, **sigurnošću tipova** i **RAII idiomom** (*Resource Acquisition Is Initialization*).

U ovom delu upoznaćemo se sa načinom na koji `new` i `delete` funkcionišu, njihovim razlikama u odnosu na C-funkcije.

- Operator `new` istovremeno **alocira** prostor i **inicijalizuje** objekat, pozivajući njegov **konstruktor**.
- `new` vraća pokazivač **odgovarajućeg tipa** — nije potrebna konverzija sa `void*`, što povećava bezbednost i čitljivost koda.
- Kada se objekat više ne koristi, `delete` poziva njegov **destruktor** i zatim oslobađa memoriju.

Primer:

```
struct Vektor2D {
    double x, y;
    Vektor2D(double x, double y) : x(x), y(y) {}
};

int main() {
    Vektor2D* p = new Vektor2D(1.0, 2.0);  // konstruktor se automatski poziva
    delete p;                               // poziva se destruktor i oslobađa me
}
```

Prednost u odnosu na `malloc` i `free`

U C-u funkcija `malloc` samo rezerviše prostor u memoriji, ali **ne poziva konstruktor** objekta. Zbog toga se može dogoditi da objekat ne bude pravilno inicijalizovan, ili da ručno oslobađanje pomoću `free` ne pozove njegov destruktor. U sledećem jednostavnom primeru vidimo kako korišćenje `malloc` može dovesti do **curenja memorije**, jer konstruktor i destruktor nisu pozvani.

```
struct Broj {
    int* vrednost;

    Broj(int x) {
        vrednost = new int(x);
        std::cout << "Konstruktor: alocirana vrednost " << *vrednost << "\n";
    }

    ~Broj() {
        delete vrednost;
        std::cout << "Destruktor: oslobođena memorija\n";
    }
};

int main() {
    // Pogrešno: malloc ne poziva konstruktor ni destruktor
    Broj* a = (Broj*) malloc(sizeof(Broj));
    a->vrednost = new int(5);   // ručna inicijalizacija
    std::cout << "Objekat napravljen malloc-om\n";
    free(a);                    // destruktor se ne poziva → curenje memorije!

    std::cout << "\n";
```

```
    std::cout << "Objekat napravljen new\n";
    // Ispravno: new i delete pozivaju konstruktor i destruktor
    Broj* b = new Broj(5);
    delete b;
}
```

Izlaz programa:

```
Objekat napravljen malloc-om

Objekat napravljen new
Konstruktor: alocirana vrednost 5
Destruktor: oslobođena memorija
```

U prvom slučaju, memorija dodeljena pomoću `malloc` nije „svesna" svog tipa — ne zna da u sebi sadrži pokazivač koji takođe treba osloboditi. Destruktor se ne poziva, pa `*vrednost` ostaje zauzeta do kraja programa. Nasuprot tome, `new` i `delete` automatski pozivaju konstruktor i destruktor i time **sprečavaju curenje memorije**.

Zbog ovakvih situacija, u C++-u se preporučuje **da se** `malloc` **i** `free` **uopšte ne koriste**, osim u retkim slučajevima kada se radi sa starim C bibliotekama.

Dinamička alokacija nizova

Za alokaciju većeg broja objekata koristi se operator `new[]`, a oslobađanje takvih nizova mora se vršiti sa `delete[]`. Ovim operatorima C++ obezbeđuje da se konstruktor i destruktor svakog elementa niza automatski pozovu.

```
#include <iostream>

struct Tacka {
    Tacka() { std::cout << "+"; }
    ~Tacka() { std::cout << "-"; }
};

int main() {
    Tacka* niz = new Tacka[5];   // poziva se 5 konstruktora
    delete[] niz;                // poziva se 5 destruktora
}
```

```
Izlaz iz programa:
++++++-----
```

> **Napomena:** `new` i `delete` uvek moraju biti pravilno upareni. Korišćenje `delete` umesto `delete[]` nad nizom objekata predstavlja **nedefinisano ponašanje**. Slično, `delete[]` nad pojedinačnim objektom nije dozvoljen.

Operator `new` može da se koristi u više oblika, u zavisnosti od načina inicijalizacije:

```
int* a = new int;       // neinicijalizovana vrednost
int* b = new int();     // inicijalizovano na 0
int* c = new int(42);   // inicijalizovano na 42
```

Za nizove, ako se želi da svi elementi budu 0, koristi se:

```
int* niz = new int[100](); // value-inicijalizacija svih elemenata
```

Šta se dešava kada nema dovoljno memorije?

Ako `operator new` ne može da obezbedi traženi prostor, on **baca izuzetak** tipa `std::bad_alloc`. Zato se preporučuje upotreba `try–catch` blokova kada postoji rizik od iscrpljivanja memorije.

Primer:

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

void f() {
    vector<char*> v;
    try {
        for (;;) {
            char* p = new char[10000];  // pokušaj alokacije
            v.push_back(p);             // čuvamo adresu da ne izgubimo referenc
            p[0] = 'x';                 // koristimo memoriju
        }
    } catch (bad_alloc&) {
        cerr << "Nema više memorije!\n";
    }
}
```

Bez obzira na količinu raspoložive memorije, ovaj program će kad-tad izazvati izuzetak `bad_alloc`, ali je ponašanje određeno try-catch blokom.

RAII

U jeziku C++ postoji princip poznat kao **RAII** (*Resource Acquisition Is Initialization*), koji podrazumeva da se svaki resurs (memorija, fajl, mrežna veza i slično) „veže“ za neki objekat — a njegovo oslobađanje obavlja se automatski u destruktoru tog objekta. Na taj način, čak i ako dođe do greške ili izuzetka, destruktori se pozivaju i svi resursi se pravilno oslobađaju, bez potrebe za ručnim `delete`.

U tom kontekstu često se pominju **pravila 3/5/0**, koja ukazuju kada klasa treba da definiše posebne funkcije za kopiranje i brisanje: ako klasa upravlja dinamičkom memorijom ili nekim resursom, mora imati sopstveni **destruktor**, **kopirni konstruktor** i **operator dodele** (*pravilo 3*), a u modernom

C++-u dodaju se i verzije za **pomeranje** objekata, **move konstruktor** i **move operator dodele** (*pravilo 5*). Ako se, međutim, koriste standardni tipovi kao što su `std::vector`, `std::string` ili pametni pokazivači, programer ne mora da piše ništa od toga (*pravilo 0*) jer ovi tipovi već automatski primenjuju RAII princip.

U jeziku C++ ne preporučuje se ručno menjanje veličine dinamički alociranih blokova pomoću funkcija kao što je `realloc`. Umesto toga, treba koristiti **standardne kontejnere** — kao što su `std::vector` za dinamičke nizove, `std::string` za rad sa tekstom ili druge strukture (`std::deque`, `std::map` i dr.) — koji sami upravljaju alokacijom i proširivanjem memorije na bezbedan i efikasan način.

2.3 Pametni pokazivači

2.3.1 Otkrivanje curenja memorije

Jedna od posledica neispravnosti programa može da bude i curenje memorije. Curenje memorije je problem koji je često teško primetiti (sve dok ima memorije na raspolaganju) i locirati u izvornom kodu. Postoji više programa koji mogu pomoći u proveravanju da li u programu postoji curenje memorije i u lociranju mesta u programu koje je odgovorno za to. Jedan od takvih programa je `valgring` (videti i poglavlje **??**) koji ima alatku `memcheck` sa ovom svrhom.

Razmotrimo sledeći jednostavan program.

```
#include <iostream>

int main()
{
  int *p;
  p = malloc(1000);
  if (p == NULL)
    return -1;
  p = malloc(1000);
  if (p == NULL)
    return -1;
  else
    free(p);
  return 0;
}
```

Drugim pozivom funkcije `malloc` biće obrisana prethodna vrednost pokazivača `p`, čime će biti izgubljena mogućnost pristupanja bloku koji je alociran prvim pozivom funkcije `malloc`, tj. doći će do curenja memorije.

Ukoliko je navedeni program kompiliran u *debag* modu i ukoliko je izvršiva verzija nazvana `mojprogrma`, alat `valgrind` može se, za detektovanje curenja memorije, pozvati na sledeći način:

```
valgrind --tool=memcheck --leak-check=yes ./mojprogram
```

Curenje memorije će biti uspešno otkriveno i `valgrind` daje sledeći izlaz:

```
==9697== Memcheck, a memory error detector
==9697== Copyright (C) 2002-2011, and GNU GPL'd, by Julian Seward et al.
==9697== Using Valgrind-3.7.0 and LibVEX; rerun with -h for copyright info
==9697== Command: ./mojprogram
==9697==
==9697==
==9697== HEAP SUMMARY:
==9697==     in use at exit: 1,000 bytes in 1 blocks
==9697==   total heap usage: 2 allocs, 1 frees, 2,000 bytes allocated
==9697==
==9697== 1,000 bytes in 1 blocks are definitely lost in loss record 1 of 1
==9697==    at 0x402BE68: malloc (in /usr/lib/valgrind/vgpreload_memcheck-x86-linu
==9697==    by 0x8048428: main (curenje_memorije.c:6)
==9697==
==9697== LEAK SUMMARY:
==9697==    definitely lost: 1,000 bytes in 1 blocks
==9697==    indirectly lost: 0 bytes in 0 blocks
==9697==      possibly lost: 0 bytes in 0 blocks
==9697==    still reachable: 0 bytes in 0 blocks
==9697==         suppressed: 0 bytes in 0 blocks
==9697==
==9697== For counts of detected and suppressed errors, rerun with: -v
==9697== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Alati za otkrivanje curenja memorije su programerima jako korisni, ali oni ipak nisu svemoćni u otkrivanju curenja memorije. Naime, u procesu traženja curenja memorije prati se samo jedno konkretno izvršavanje programa i na neke naredbe koje su odgovorne za curenje memorije u tom konkretnom izvršavanju možda se uopšte neće naići.

Problem curenja memorije u prethodnom primeru ne potiče iz neke složene logike, već iz jednostavne činjenice — programer je **zaboravio da oslobodi prethodno dodeljeni blok memorije**.

Takve greške su uobičajene u programima koji koriste *sirove pokazivače* (`T*`) i funkcije za ručnu alokaciju (`malloc`, `free`, `new`, `delete`). Čak i kada programer u većini slučajeva pravilno oslobađa memoriju, u složenijim programima — sa više izlaza iz funkcije, obrađivanjem izuzetaka ili dinamički kreiranim strukturama — lako može doći do situacija u kojima se memorija:

- nikada ne oslobodi (*curenje memorije*),
- oslobodi više puta (*dvostruko oslobađanje*),
- oslobodi prerano, pa pokazivač ostane „viseći” (*dangling pointer*).

Takvi problemi se često ne uočavaju tokom kompilacije, niti uvek dovode do pada programa, što ih čini posebno opasnim — posledice se mogu ispoljiti tek nakon dužeg rada ili u potpuno drugom delu programa.

Jedan od osnovnih ciljeva savremenog **C++**-a jeste **automatsko upravljanje resursima** (*Resource Management*) uz očuvanje performansi. Ideja je da se vlasništvo nad memorijom (ili drugim resursima — npr. fajl deskriptorima, mrežnim konekcijama, hendlovima datoteka) tretira kao **deo semantike objekta**, a ne kao obaveza programera koji mora ručno da poziva `delete`.

C++ rešava ovaj problem primenom koncepta **RAII** (*Resource Acquisition Is Initialization*) — resurs se „preuzima“ u konstruktoru, a „oslobađa“ u destruktoru. Na taj način, čim objekat izađe iz opsega, memorija se oslobađa automatski, bez mogućnosti da programer to zaboravi ili ponovi više puta.

Klase koje primenjuju ovaj princip na dinamičku memoriju nazivaju se **pametni pokazivači** (*smart pointers*). Oni su, u suštini, *objekti koji se ponašaju kao pokazivači*, ali koji samostalno upravljaju životnim vekom memorije na koju pokazuju.

Najvažniji tipovi pametnih pokazivača u standardnoj biblioteci `<memory>` su:

- `std::unique_ptr` – predstavlja **iskl čivo vlasništvo** (postoji tačno jedan vlasnik),
- `std::shared_ptr` – omogućava **zajedničko vlasništvo** nad istim resursom,
- `std::weak_ptr` – pruža **slabu vezu** ka objektu kojim upravlja `shared_ptr` (koristi se za razbijanje kružnih zavisnosti).

U narednom odeljku biće detaljno prikazano kako se pomoću ovih klasa postiže ista efikasnost kao pri ručnom upravljanju memorijom, ali uz znatno veću pouzdanost, čitljivost i bezbednost programa.

2.3.2 `std::unique_ptr` — pametni pokazivač sa ekskluzivnim vlasništvom

Kao što je već naglašeno, obični pokazivači (`T*`) ne sadrže nikakvu informaciju o **vlasništvu** nad memorijom na koju pokazuju. To dovodi do problema u složenim programima — jedan deo koda može da oslobodi memoriju koju drugi deo i dalje koristi, ili se memorija nikada ne oslobodi, što rezultira *curenjem memorije*.

Kao što ime sugeriše, `std::unique_ptr` predstavlja **jedinstveno vlasništvo** nad objektom. U svakom trenutku postoji tačno jedan aktivni pokazivač koji poseduje dati resurs — bilo da je to dinamički alociran objekat, otvorena datoteka, mrežna konekcija ili neki drugi sistemski resurs. Kada taj pokazivač izađe iz opsega, njegov destruktor automatski poziva `delete` i oslobađa resurs.

1. Objekat tipa `std::unique_ptr<T>` **poseduje** jedan objekat tipa `T`. Kada `unique_ptr` bude uništen, automatski poziva `delete` nad tim objektom.
2. `unique_ptr` **ne može da se kopira** — nema kopirajući konstruktor ni operator dodele kopiranjem. Vlasništvo se može **preneti** pomoću semantike premještanja (`move semantics`).
3. Prilikom uništavanja `unique_ptr` poziva se **brisač** (eng. deleter), koji definiše način oslobađanja resursa. Podrazumevani brisač koristi `delete`, ali se može zameniti prilagođenom funkcijom.
4. `unique_ptr` može da sadrži i **niz objekata**, koristeći specijalizovanu formu `std::unique_ptr<T[]>`.

`unique_ptr` je osnovni i najlakši za razumevanje pametni pokazivač, jer modeluje najjednostavniji odnos: *jedan vlasnik — jedan životni ciklus*. Upravo ta jednostavnost ga čini izuzetno brzim i pogodnim za gotovo svaku situaciju u kojoj vlasništvo nije deljeno.

Primer:

```
#include <iostream>
#include <memory>

int main() {
    std::unique_ptr<int> broj = std::make_unique<int>(42);
```

```
    std::cout << "Vrednost: " << *broj << std::endl;
} // automatski se poziva delete, nema curenja memorije
```

Ovde je:

- `std::make_unique<int>(42)` alocirao memoriju i inicijalizovao je vrednošću 42;
- `broj` poseduje taj objekat;
- po izlasku iz `main`, destruktor `std::unique_ptr` automatski poziva `delete` nad pokazivačem.

Kada `broj` izađe iz opsega, destruktor `unique_ptr` automatski briše objekat na koji pokazuje. Nema potrebe za `delete`, i što je još važnije — ne postoji mogućnost da objekat „ostane zaboravljen" u memoriji.

Funkcija `std::make_unique<T>(args...)` je preporučeni način za kreiranje `unique_ptr` objekata. Prednost je dvostruka: kod je kraći, i ne postoji mogućnost „curenja" resursa između alokacije i konstrukcije pokazivača.

Alternativni (manje bezbedan) način bio bi:

```
std::unique_ptr<int> q(new int(42)); // ispravno, ali opasno u složenim izrazima
```

Razlog je jednostavan: između `new int(42)` i inicijalizacije `q` može se dogoditi izuzetak koji prekida funkciju — i tada bi alocirani `int` ostao zauzet. `make_unique` garantuje da se alokacija i inicijalizacija obave kao jedna nerazdvojiva operacija.

Za razliku od običnih pokazivača, `unique_ptr` **ne može da se kopira**. Time se osigurava da uvek postoji samo jedan vlasnik. Međutim, vlasništvo se može **preneti** pomoću semantike *pomeranja* (`move`):

```
    auto a = std::make_unique<int>(42);   // vlasništvo prelazi u a
    auto b = std::move(a);                // sada b poseduje objekat, a postaje n

    if (!a) std::cout << "a više nije vlasnik.\n";
    std::cout << "Vrednost je: " << *b << std::endl;
```

Ovakav prenos vlasništva je čest obrazac — na primer, pri vraćanju objekta iz funkcije ili dodavanju u kontejner. Semantika pomeranja omogućava izuzetno efikasno prosleđivanje resursa bez kopiranja podataka.

`unique_ptr` se ponaša kao običan pokazivač pri dereferenciranju:

```
auto p = std::make_unique<int>(42);

// ponaša se kao običan pokazivač
std::cout << *p << std::endl;   // ispisuje 42
(*p) += 5;                      // uvećava vrednost na 47
```

Ovde `*p` jednostavno daje pristup broju koji `unique_ptr` poseduje. Za situacije kada funkciji nije potrebno vlasništvo, objekat se prosleđuje **referencom**:

```
void prikazi(const int& x);

prikazi(*p);   // funkcija vidi vrednost, ali ne poseduje resurs
```

Ako je funkciji potrebno da **preuzme** resurs, pokazivač se prosleđuje pomoću `std::move`, čime se vlasništvo prenosi.

Primer: prenos vlasništva kroz funkcije

Sledeći primer pokazuje razliku između prenosa vlasništva i pristupa referencom:

```
std::unique_ptr<int> f(std::unique_ptr<int> p) {
    ++*p;           // uvećavamo vrednost
    return p;       // vraćamo vlasništvo nazad
}

void f2(const std::unique_ptr<int>& p) {
    std::cout << "Vrednost: " << *p << std::endl;
}

void use() {
    std::unique_ptr<int> p = std::make_unique<int>(7);

    // p = f(p);  // greška: unique_ptr se ne može kopirati
    p = f(std::move(p));  // vlasništvo ide u f() i vraća se nazad
    f2(p);                // pristup referencom, bez prenosa vlasništva
}
```

Ovde:

- `f()` preuzima vlasništvo, menja vrednost i vraća ga nazad;
- `f2()` koristi pokazivač samo za čitanje — ne poseduje objekat;
- `std::move(p)` jasno označava prenos vlasništva.

Takav dizajn čini vlasništvo nad memorijom **eksplicitnim** u samom potpisu funkcije, što značajno olakšava analizu i sprečava greške.

Primer sa vektorom:

Jedan od najčešćih načina korišćenja pametnih pokazivača u praksi jeste u kombinaciji sa kontejnerima iz standardne biblioteke. Tipičan primer je **vektor pokazivača** – npr. kada čuvamo dinamički alocirane objekte kojima svaki element vektora „vlasnički" upravlja.

```
#include <iostream>
#include <memory>
#include <vector>

int main() {
    std::vector<std::unique_ptr<int>> niz;

    // Dodavanje novih elemenata u vektor
```

```
    niz.push_back(std::make_unique<int>(10));
    niz.push_back(std::make_unique<int>(20));
    niz.push_back(std::make_unique<int>(30));

    // Pristup vrednostima
    for (const auto& p : niz)
        std::cout << *p << " ";
    std::cout << std::endl;

    // Prenos vlasništva pojedinačnog elementa
    std::unique_ptr<int> treci = std::move(niz[2]);
    if (!niz[2])
        std::cout << "Element na poziciji 2 više nema vlasnika." << std::endl;
}
```

U ovom primeru:

- `std::vector` sadrži **jedinstvene pokazivače** (`std::unique_ptr<int>`), tako da svaki element u vektoru poseduje svoj objekat.
- `std::make_unique<int>(10)` alocira novi ceo broj i predaje vlasništvo elementu vektora.
- Kada vektor izađe iz opsega, svi njegovi elementi se uništavaju, što povlači automatsko oslobađanje svakog dinamički alociranog objekta.

Uobičajena greška početnika bila bi da se koristi `std::vector<int*>`, pa da se zatim ručno poziva `delete` nad svakom adresom. Međutim, takav kod je podložan curenju memorije i nekonzistentnostima.

Primer: skladište objekata

```
#include <iostream>
#include <memory>
#include <vector>

struct Tacka {
    double x, y;
    Tacka(double x, double y) : x(x), y(y) {}
    void prikazi() const { std::cout << "(" << x << ", " << y << ")"; }
};

int main() {
    std::vector<std::unique_ptr<Tacka>> tacke;

    tacke.push_back(std::make_unique<Tacka>(1.0, 2.0));
    tacke.push_back(std::make_unique<Tacka>(3.5, -0.5));
    tacke.push_back(std::make_unique<Tacka>(0.0, 4.2));

    std::cout << "Skup tacaka: ";
    for (const auto& t : tacke) {
        t->prikazi();
        std::cout << " ";
```

```
    }
    std::cout << std::endl;

    // Prenos vlasništva nad jednom tačkom
    auto poslednja = std::move(tacke.back());
    tacke.pop_back(); // uklanja prazan element iz vektora

    std::cout << "Poslednja tacka preneta: ";
    poslednja->prikazi();
    std::cout << std::endl;
}
```

> Kada imamo kolekciju dinamički kreiranih objekata sa jasnim vlasništvom, najbolji izbor je `std::vector<std::unique_ptr<T>>`. Na taj način kontejner postaje „vlasnik seta objekata“, dok se životni vek svakog elementa u potpunosti kontroliše preko RAII mehanizma.

Kada želimo da funkciji predamo pokazivač koji treba da preuzme vlasništvo nad objektom, koristimo **premještanje** (`std::move`).

```
#include <iostream>
#include <memory>

std::unique_ptr<int> napravi() {
    return std::make_unique<int>(5);
}

void koristi(std::unique_ptr<int> p) {
    std::cout << "Unutar koristi(): " << *p << std::endl;
}

int main() {
    std::unique_ptr<int> broj = napravi();
    koristi(std::move(broj));  // vlasništvo prelazi u funkciju koristi
    if (!broj)
        std::cout << "broj više ne poseduje objekat." << std::endl;
}
```

Ovaj primer ilustruje osnovni princip: vlasništvo se može imati **ili na jednom mestu, ili se prenosi**, ali ne duplira.

Korisnički definisan destruktor

Podrazumevani `std::unique_ptr` koristi `delete`, ali se može definisati sopstveni, npr. ako radimo s C-bibliotekama koje koriste `malloc/free`.

```
#include <iostream>
#include <memory>
#include <cstdlib>
```

```
void* c_allocate() {
    void* p = std::malloc(64);
    std::cout << "C alocirao memoriju..." << std::endl;
    return p;
}

void example() {
    std::unique_ptr<void, void(*)(void*)> blok(c_allocate(), std::free);
    std::cout << "Koristimo blok..." << std::endl;
} // automatski poziva std::free(blok)
```

Ovakvi čine `unique_ptr` univerzalnim mehanizmom za upravljanje bilo kojim resursom — fajlovima, mrežnim konekcijama, sinhronizacionim objektima i dr.

Tabela: najvažnije operacije `std::unique_ptr`

Operacija	Efekat i značenje
`unique_ptr up{}`	Konstruktor bez argumenta — pokazivač je `nullptr`.
`unique_ptr up{p}`	Inicijalizuje vlasništvo nad pokazivačem `p`, koristi podrazumevani destruktor (`delete`).
`up = std::move(up2)`	Operator dodele premeštanjem — briše stari objekat, preuzima novi.
`up = nullptr`	Oslobađa resurs i postavlja pokazivač na `nullptr`.
`*up`	Dereferencira pokazivač.
`up->m`	Pristupa članu objekta.
`up.reset()`	Oslobađa resurs i postavlja na `nullptr`.
`up.swap(up2)`	Razmenjuje vlasništvo sa drugim `unique_ptr`.
`swap(up, up2)`	Ekvivalentno prethodnom.
`up == up2`	Poređenje internih pokazivača.
`up < up2`	Ležišta poređenja između pokazivača (adresno poređenje).

Difoltno ponašanje `unique_ptr`-a je da pri uništenju pozove `delete` nad objektom. Međutim, moguće je definisati **prilagođenu funkciju oslobađanja** — korisno kada se upravlja resursima koji nisu alocirani pomoću `new` (na primer, datotekama ili memorijom iz C-a), ali ovde o tome neće biti reći.

Upotreba `unique_ptr` donosi tri ključne prednosti:

1. **Bezbednost** — automatsko oslobađanje memorije i sprečavanje curenja.
2. **Jasna semantika vlasništva** — uvek postoji samo jedan vlasnik.
3. **Efikasnost** — bez dodatnih struktura, referentnih brojača ili atomskih operacija.

Najvažnije ograničenje jeste da se vlasništvo ne može kopirati. To je i namerno: vlasnik mora eksplicitno odlučiti da li prenosi odgovornost pomoću `move` operacije.

2.3.3 `std::shared_ptr`: zajedničko vlasništvo i upravljanje vekom trajanja

Pametni pokazivači uvode disciplinu u C++ programe tako što automatizuju oslobađanje resursa. Dok `std::unique_ptr` predstavlja ekskluzivno vlasništvo (jedan vlasnik – jedan životni ciklus),

`std::shared_ptr` modeluje okolnosti u kojima objekat ima više legitimnih „gazda". Svaka kopija `shared_ptr` doprinosi zajedničkom vlasništvu, a objekat se uništava tek kada poslednja kopija nestane ili bude preusmerena.

Za razliku od `std::unique_ptr`, koji ima tačno jednog vlasnika, `std::shared_ptr` vodi evidenciju o tome koliko pokazivača deli isti objekat pomoću **brojača referenci**. Kada broj padne na nulu, resurs se oslobađa.

```
#include <iostream>
#include <memory>

int main() {
    auto p1 = std::make_shared<int>(10);  // kreira broj 10
    auto p2 = p1;                         // p2 sada deli isti objekat

    std::cout << *p1 << " " << *p2 << std::endl;  // ispisuje: 10 10
    (*p2)++;                                       // menja vrednost kroz p2
    std::cout << *p1 << std::endl;                 // ispisuje: 11
}   // kraj main-a → p2 se briše, zatim p1 → broj 11 se oslobađa
```

U ovom primeru:

- Oba pokazivača (`p1` i `p2`) dele isti objekat u memoriji.
- Objekat će biti obrisan **tek kad nestane i poslednji pokazivač**.

Svaki objekt pod upravom `shared_ptr` dobija pridruženi kontrolni blok. U njemu se čuvaju bar dve vrednosti: **brojač jakih referenci** (koliko `shared_ptr` instanci trenutno pokazuje na objekat) i **brojač slabih referenci** (koliko `std::weak_ptr` posmatra objekat, o njima više kasnije). Kada broj jakih referenci padne na nulu, `shared_ptr` poziva odgovarajući „deleter" (po difoltu `delete`) i objekt se uništava; kontrolni blok može još kratko da potraje dok postoje `weak_ptr` posmatrači, a zatim i on nestaje.

Na nivou performansi valja primetiti da su uvećanja/smanjenja brojača atomske operacije (bezbedne za niti), pa su nešto skuplje od običnih, ali su i dalje veoma brze. Samo dereferenciranje `shared_ptr` je praktično isto kao dereferenciranje „običnog" pokazivača.

Često se `shared_ptr` koristi kada funkcije treba da dele isti resurs bez brige o tome ko ga „poseduje" i kada se oslobađa. U sledećem primeru, oba poziva funkcija dele istu memoriju:

```
void f(std::shared_ptr<int> p) {
    std::cout << "f: " << *p << std::endl;
}

void g(std::shared_ptr<int> p) {
    std::cout << "g: " << *p << std::endl;
}

int main() {
    auto broj = std::make_shared<int>(42);
    f(broj);
    g(broj);
```

```
}   // objekat 42 se briše tek po izlasku iz main-a
```

Svaka funkcija dobija **kopiju** pametnog pokazivača, čime se broj referenci povećava. Kada sve kopije nestanu, broj se automatski oslobađa.

Standardna funkcija `std::make_shared<T>(args...)` stvara **i objekat i kontrolni blok** u jednoj alokaciji. To je i bezbednije (izbegava se `new`):

```
auto p = std::make_shared<std::string>("MATF");
```

Kopiranje `shared_ptr` uvećava brojač referenci i na taj način produžava život objekta. Pomeranje (move) je jeftinije jer samo premešta internu adresu i „prazni" izvorni pokazivač bez menjanja brojača. To sugeriše jednostavno pravilo: kada se želi da **podeli vlasništvo**, kopirati; kada se **prebacuje** vlasništvo, koristiti move semantiku.

U interfejsima funkcija to vodi jasnoj stilistici: ako funkcija **želi** da zadrži objekat u životu posle povratka, neka prima `std::shared_ptr<T>` **po vrednosti**; ako joj objekt treba samo za čitanje u okviru poziva, prosleđuje se **referenca** (`const T&`). Tako se izbegavaaju nepotrebni inkrementi brojača.

Pokazivač `shared_ptr` koristiti samo kada:

- isti objekat treba da postoji dokle god ga **bilo ko** koristi,
- nije jasno koji deo programa je „glavni" vlasnik ili u situaciji kada postoji više niti koji izvršavaju određeni deo programa).

U svim drugim slučajevima — naročito kada postoji jasan jedan vlasnik — bolje je koristiti `std::unique_ptr`. Funkcija `use_count()` vraća koliko `shared_ptr` instanci trenutno pokazuje na isti objekat.

Odlično — to je logičan završetak teme o pametnim pokazivačima. `std::weak_ptr` se uvodi samo kao **dopuna** `shared_ptr`-**u**, pa i tekst o njemu treba da bude kratak, jasan i motivisan praktičnim razlozima. Evo kako bi izgledao deo o `weak_ptr`, u istom akademskom tonu kao prethodna poglavlja i bez nepotrebnog OOP-a.

2.3.4 `std::weak_ptr`: posmatrač bez vlasništva

Pametni pokazivač `std::weak_ptr` ne poseduje objekat, već samo **posmatra** resurs koji je u vlasništvu nekog `std::shared_ptr` pokazivača. Osnovna ideja `std::weak_ptr` pokazivača jeste da omogući pristup objektu bez preuzimanja odgovornosti za njegovo trajanje.

Drugim rečima, `weak_ptr` ne povećava brojač referenci i ne sprečava brisanje objekta — on samo omogućava bezbednu proveru da li objekat još postoji.

Zamislimo jednostavan primer gde dva pokazivača dele jedan objekat, a zatim čuvaju reference jedno na drugo:

```
#include <memory>

struct Cvor {
    std::shared_ptr<Cvor> sledeci;
    std::shared_ptr<Cvor> prethodni;
```

```
};
```

Ako a pokazuje na b, a b nazad na a, formira se **ciklus**. Brojači referenci se nikada ne spuste na nulu i objekti se nikada ne brišu — nastaje **curenje memorije**. Rešenje je jednostavno: jedna od veza ne sme biti vlasnička. Umesto `shared_ptr`, koristi se `weak_ptr`:

```
struct Cvor {
    std::shared_ptr<Cvor> sledeci;
    std::weak_ptr<Cvor>   prethodni;  // ne učestvuje u brojanju
};
```

Sada `prethodni` samo posmatra objekat na koji pokazuje, ali ga ne zadržava u životu.

Pokazivač `weak_ptr` se uvek pravi iz postojećeg pokazivača `shared_ptr`:

```
auto p = std::make_shared<int>(100);
std::weak_ptr<int> w = p;   // w posmatra, ali ne poseduje
```

Pošto `w` ne garantuje da objekat i dalje postoji, ne može se direktno dereferencirati. Pre pristupa vrednosti koristi se metoda `lock()` koja vraća `shared_ptr` ako je objekat još nije obrisan:

```
if (auto s = w.lock()) {     // ako objekat još postoji
    std::cout << *s << std::endl;
} else {
    std::cout << "Objekat više ne postoji.\n";
}
```

U praksi, `std::weak_ptr` ima smisla u složenijim sistemima gde više delova programa dinamički deli iste objekte (npr. u višeslojnim arhitekturama, asinhronim sistemima ili grafovima međusobno povezanih entiteta).

Međutim, u klasičnim strukturama podataka kao što su **liste, stabla ili grafovi** koje se nalaze u okviru jednog modula, upotreba običnog pokazivača (`T*`) je najjednostavnije i sasvim bezbedno rešenje — pod uslovom da je vlasništvo nad čvorovima jasno definisano. Tipično, vlasnički pokazivači (`std::unique_ptr`) čuvaju hijerarhiju strukture, dok se obični pokazivači koriste za ne-vlasničke veze (npr. ka roditelju u stablu ili prethodnom čvoru u listi).

2.4 Šabloni

Šabloni su jedan od najmoćnijih mehanizama C++-a: omogućavaju da isti izvorni kod radi za različite tipove podataka bez gubitka efikasnosti. Suština je **generičko programiranje** — pišu se algoritmi i strukture podataka koji funkcionišu za „bilo koji tip“, pod uslovom da taj tip podržava operacije koje šablon koristi.

Ključna ideja je da tip ili vrednost mogu biti parametar u definiciji klase, funkcije ili alias-tipa. Kompajler zatim, u trenutku prevođenja, generiše konkretan kod za svaku upotrebljenu kombinaciju parametara — potpuno statički i tip-bezbedno (za razliku od makroa, svaka instancijacija prolazi regularnu proveru tipova).

2.4.1 Klasa parametarskog tipa

Pretpostavimo da je potrebno definisati strukturu *Niz* koja može da skladišti elemente različitih tipova, na primer `int`, `double` ili `string`. Bez mehanizma šablona, bilo bi neophodno pisati posebne verzije iste klase za svaki konkretan tip podataka. Upotrebom **šablona klase**, tip elemenata se parametriše, čime se omogućava da jedna jedinstvena definicija klase generiše različite verzije — npr. `Niz<int>`, `Niz<double>` ili `Niz<string>` — u zavisnosti od tipa koji se navede kao parametar.

```
template<typename T>
class Niz {
private:
    T* elementi;
    int duzina;
public:
    explicit Niz(int n) : elementi(new T[n]), duzina(n) {}
    ~Niz() { delete[] elementi; }

    T& operator[](int i) { return elementi[i]; }
    const T& operator[](int i) const { return elementi[i]; }

    int size() const { return duzina; }
};
```

Prefiks `template<typename T>` označava da se definiše **šablon klase** i da će tip `T` biti **parametar šablona**. Nakon uvodne deklaracije, identifikator `T` se koristi na istovetan način kao i svako drugo ime tipa, a opseg važenja identifikatora `T` obuhvata čitavu definiciju klase na koju se prefiks odnosi.

Umesto ključne reči `typename`, može se koristiti i `class`, pa su zapisi `template<typename T>` i `template<class T>` **potpuno ekvivalentni**. U oba slučaja, `T` predstavlja **bilo koji tip**, ne nužno i klasu.

> **Napomena:** U matematičkom smislu, izraz `template<typename T>` može se posmatrati kao formalni način da se iskaže „**za svaki tip T**“. Međutim, klasični C++ standardi ne pružaju opšti mehanizam za precizno navođenje **ograničenja nad tipom T**, tj. nije moguće unutar šablona direktno zapisati uslov oblika „za svaki tip T koji poseduje određena svojstva“. Takva mogućnost uvedena je tek u savremenim verzijama jezika putem **koncepta** (*concepts*, C++20).

Članovi šablona klase deklarišu se i definišu na isti način kao i kod običnih (neparametrizovanih) klasa. Kada se šablon koristi uz konkretne argumente tipova, na primer `Niz<int>` ili `Niz<string>`, kompajler formira **instancu šablona** (engl. *instantiation*), odnosno **specijalizaciju** šablona za te konkretne tipove.

U procesu prevođenja, za svaku instancu koja se zaista koristi u programu generiše se odgovarajući mašinski kod. Taj kod prolazi kroz standardnu proveru tipova, čime se obezbeđuje da je **jednako tip-bezbedan** kao i ručno napisan kod bez šablona.

Na primer:

```
Niz<int> brojevi(10);
Niz<string> reci(5);
```

Kompajler će prilikom upotrebe automatski generisati **dve posebne verzije klase** — `Niz<int>` i `Niz<string>` — koje se dalje ponašaju kao uobičajene klase sa svim pravilima jezika C++. Na sličan način funkcije definisane kao šabloni postaju obične funkcije nakon što se konkretni tipovi parametara odrede u trenutku instanciranja.

2.4.2 Vrednosni parametar u šablonu

Pored tipova, šabloni mogu imati i **vrednosne parametre** — konstante poznate u vreme prevođenja. Takvi parametri se često koriste za predstavljanje veličine niza, broja elemenata u strukturi, dimenzija matrice i sličnih svojstava koja su poznata unapred.

Na primer:

```
template<typename T, int N>
struct Buffer {
    using value_type = T;
    constexpr int size() const { return N; }
    T elementi[N];
    // ...
};
```

U ovom primeru, tip `T` predstavlja tip elemenata, dok vrednosni parametar `N` označava broj elemenata koji će biti smešteni u statičkom nizu. Član `value_type` i funkcija `size()` omogućavaju da se ovi parametri koriste i izvan strukture, bez mogućnosti izmene.

Korišćenjem vrednosnih parametara moguće je definisati strukture čija se veličina određuje u trenutku prevođenja, bez dinamičke alokacije memorije. Na primer:

```
Buffer<char, 1024> globalni;  // globalni bafer od 1024 karaktera
void f() {
    Buffer<int, 10> lokalni;  // lokalni bafer od 10 celih brojeva
    // ...
}
```

Vrednosni parametar šablona mora biti **konstantan izraz** poznat u vreme kompilacije, jer kompajler koristi njegovu vrednost za generisanje konkretne verzije klase.

2.4.3 Dedukcija argumenata šablona (C++17)

U ranijim verzijama jezika C++-a, prilikom korišćenja šablona bilo je neophodno eksplicitno navesti sve tipove parametara. Na primer, kod standardne klase `std::pair` bilo je potrebno pisati:

```
std::pair<int, double> p = {21, 3.2};
```

Od C++17, kompajler može **sam da zaključi tipove** na osnovu argumenata konstruktora, pa se sada može jednostavno pisati:

```
std::pair p = {21, 3.2};  // tip se dedukuje kao pair<int, double>
```

Ovaj mehanizam, nazvan *Class Template Argument Deduction (CTAD)*, važi i za korisnički definisane šablone. Na primer:

```
template<typename T>
class Niz {
public:
    Niz(std::initializer_list<T>) { /* ... */ }
};

Niz v1 = {1, 2, 3};   // dedukuje se kao Niz<int>
Niz v2 = {"a", "b"};  // dedukuje se kao Niz<const char*>
```

Kompajler dakle automatski zaključuje da su tipovi elemenata `int` ili `const char*`, u zavisnosti od vrednosti inicijalizatora. Ako elementi imaju **neusaglašene tipove**, dedukcija nije moguća i dolazi do greške.

> Dedukcija tipova pojednostavljuje sintaksu i smanjuje mogućnost grešaka, ali ne eliminiše potrebu za pažnjom — naročito kod C-stilskih niska (`const char*`), koje se ne tretiraju isto kao `std::string`.

> U slučaju nejednoznačnosti, tip je i dalje moguće eksplicitno navesti, npr. `Niz<std::string>{"a", "b"}`.

2.4.4 Funkcijski šabloni

Funkcijski šablon je „opšti" oblik funkcije koji se konkretizuje za dati tip pri pozivu. Na primer, suma elemenata niza (model „akumulatora"):

```
template<typename T, typename Acc>
Acc sum(const T* a, int n, Acc init) {
    for (int i = 0; i < n; ++i) init += a[i];
        return init;
}

// upotreba
int    ai[] = {1, 2, 3};
double ad[] = {0.5, 1.5};

auto s1 = sum(ai, 3, 0);       // int
auto s2 = sum(ad, 2, 0.0);     // double
```

Kao što je ranije prikazano, funkcijski šablon može biti član klase.

Tipovi šablona (ovde `Acc`, `T`) **dedukuju se** iz argumenata funkcije, pa ih obično nije potrebno navoditi eksplicitno. Ako dedukcija nije moguća ili nije jednoznačna, potrebno je *eksplicitno* navođenje bar nekih parametara.

Kratak primer dvosmislenosti:

```
template<typename T>
T max2(T a, T b);

const int s = 7;
auto a = max2(1, 2);       // T=int
auto b = max2(2.7, 4.9);   // T=double
auto c = max2('a', 1);     // GREŠKA: T=char ili T=int? dvosmisleno
```

Rešenja: eksplicitno navođenje tipa (`max2<int>('a',1)`) ili definisanje odgovarajućih običnih preopterećenja.

2.4.5 Alijasi tipova

U jeziku **C**, uvođenje novog imena za postojeći tip obavlja se pomoću ključne reči `typedef`. Na primer:

```
typedef unsigned long BrojElemenata;
```

Ovim se tipu `unsigned long` dodeljuje novo ime `BrojElemenata`. Upotreba `typedef` je naročito korisna kada se žele pojednostaviti složene deklaracije, na primer pokazivači na funkcije ili strukture sa dugačkim nazivima. Ipak, `typedef` ima ograničenja: ne može se primenjivati na **šablone**, pa je njegova upotreba u C++-u delimično prevaziđena.

Jezik **C++** uvodi moderniji i fleksibilniji način za definisanje alijasa — ključnu reč `using`. Njena osnovna funkcija je ista kao `typedef`, ali ima jednostavniju i dosledniju sintaksu, naročito kada se koristi u kombinaciji sa **šablonima**.

Na primer, sledeće dve deklaracije su ekvivalentne:

```
// stariji C-stil
typedef unsigned long BrojElemenata;

// moderni C++ stil
using BrojElemenata = unsigned long;
```

Prednost `using` izraza je što se čita **u istom smeru kao i dodela vrednosti** — „`BrojElemenata` je isto što i `unsigned long`“ — dok `typedef` zahteva obrnutu, manje intuitivnu interpretaciju. Pored toga, `using` se može koristiti i za definisanje **alijasa šablona**, što `typedef` ne podržava.

Kada se alijasi kombinuju sa šablonima, njihova primena postaje još značajnija. Na primer, svaka standardna kolekcija u C++-u (poput `std::vector`, `std::list`, `std::set`) definiše **unutrašnji alijas**:

```
template<typename T>
class Kolekcija {
public:
    using value_type = T;
    // ...
};
```

To znači da svaki kontejner „u sebi nosi“ informaciju o tome koji tip sadrži. Ova konvencija (`value_type`) omogućava da se generički algoritmi pozivaju nad bilo kojim kontejnerom bez potrebe da se zna konkretan tip elemenata. Na primer:

```
template<typename C>
using Element = typename C::value_type;

template<typename Kontejner>
void ispisi(const Kontejner& c) {
    Element<Kontejner> suma{};
    for (const auto& x : c) suma += x;
    std::cout << "Suma: " << suma << std::endl;
}
```

Linija `using Element = typename C::value_type;` samo uvodi kraće ime (`Element`) za tip elemenata nekog kontejnera `C`. Dakle, ako je:

```
C = std::vector<int>
```

onda `Element<C>` postaje `int`. Ako je:

```
C = std::list<double>
```

onda `Element<C>` postaje `double`.

Alijasi takođe mogu unapred „vezati“ deo parametara šablona, čime se dobijaju novi tipovi izvedeni iz opštih obrazaca. Na primer:

```
template<typename Ključ, typename Vrednost>
class MapaAsocijacija {
    // ...
};

// novi šablon sa unapred određenim tipom ključa
template<typename Vrednost>
using MapaPoNazivu = MapaAsocijacija<std::string, Vrednost>;

MapaPoNazivu<int> brojevi; // ekvivalentno MapaAsocijacija<std::string, int>
```

Naredba `typedef` je istorijski važan deo jezika C, ali u savremenom C++-u gotovo ga u potpunosti zamenjuje `using`.

2.5 Sementika premeštanja

U modernom C++-u pojam *pomeranja* (*move semantics*) uveden je da bi se izbegla nepotrebna kopiranja podataka. Funkcija `std::move` ne pomera ništa fizički — ona samo **označava da je objekat spreman da preda svoje podatke drugom**. Najčešće se koristi sa **pametnim pokazivačima** i **kontejnerima** da bi se izbeglo kopiranje i prenosilo vlasništvo.

Drugim rečima, `std::move(x)` ne kopira, ne briše i ne pomera sadržaj promenljive `x`; ono samo govori prevodiocu:

> "od ovog trenutka, `x` se može tretirati kao privremeni objekat (rvalue) iz koga se može preuzeti vrednost".

Ova oznaka omogućava da se u nekim situacijama (npr. kod pametnih pokazivača) vlasništvo nad resursom **prenese**, umesto da se duplira.

Najjednostavniji primer gde je `std::move` neophodan jeste prenos vlasništva kod `std::unique_ptr`-a.

```
    auto p1 = std::make_unique<int>(10);

    // auto p2 = p1;        //  greška: unique_ptr ne može da se kopira
    auto p2 = std::move(p1); //  prenos vlasništva

    if (!p1) std::cout << "p1 više nije vlasnik.\n";
    std::cout << "Vrednost: " << *p2 << std::endl;
```

Pozivom `std::move(p1)` prevodilac zna da `p1` više ne mora da ostane vlasnik i da se njegov resurs može „predati" promenljivoj `p2`. Objekat `p1` ostaje validan, ali više ne pokazuje ni na jedan resurs (njegova vrednost postaje `nullptr`).

2.5.1 U čemu je razlika između kopiranja i pomeranja

Bez `std::move`, prenos bi se obavljao kopiranjem:

```
int a = 5;
int b = a;  // kopira vrednost 5
```

Kod jednostavnih tipova ovo je beznačajno, ali kod složenih tipova — kao što su vektori ili stringovi — kopiranje može biti skupo, jer zahteva dupliranje memorije i elemenata. Pomeranje, naprotiv, **preuzima postojeće resurse** i dodeljuje ih novom objektu bez kopiranja sadržaja.

Na sledećem primeru sa stringovima vidi se razlika:

```
    std::string s1 = "tekst";
    std::string s2 = std::move(s1);  // s2 preuzima sadržaj, s1 postaje prazan

    std::cout << "s1: " << s1 << ", s2: " << s2 << std::endl;
```

Ispis:

```
s1: , s2: tekst
```

Dakle, objekat s2 je preuzeo resurs koji je ranije pripadao s1, a s1 je ostao u „praznom“ ali validnom stanju.

Kada se koristi `std::move` * Koristi se kada želimo **da prenesemo vlasništvo** (npr. kod `unique_ptr`, `std::vector`, `std::string`). * Ne koristi se kod tipova gde kopiranje nije skupo (npr. `int`, `char`). * Ne koristi se nad **konstantnim objektima** (`const`), jer se iz njih ne može „pomeriti“ sadržaj — konstanta se ne može menjati.

`std::move` se često koristi pri pozivu funkcija, kada se želi efikasno proslediti veliki objekat (npr. vektor) koji više nije potreban u trenutnom opsegu.

```cpp
// Funkcija prima vektor po vrednosti - napraviće kopiju argumenta
void prikazi(std::vector<int> podaci) {
    std::cout << "Broj elemenata: " << podaci.size() << std::endl;
}

// Funkcija koja kreira i vraća vektor
std::vector<int> napravi_vektor(int n) {
    std::vector<int> v;
    for (int i = 0; i < n; ++i)
        v.push_back(i);
    return v;  // vraćanje bez std::move - automatsko pomeranje (RVO)
}

int main() {
    std::vector<int> niz = napravi_vektor(5);

    // poziv bez std::move → kopiranje vektora
    prikazi(niz);

    // poziv sa std::move → sadržaj se prenosi, niz ostaje prazan
    prikazi(std::move(niz));

    std::cout << "Posle std::move, niz ima " << niz.size() << " elemenata.\n";
}
```

- Funkcija `prikazi` prima argument **po vrednosti**, što znači da će napraviti kopiju. Ako pošaljemo `niz` bez `std::move`, sadržaj se kopira (svih 5 elemenata).
- Kada pozovemo `prikazi(std::move(niz));`, objekat `niz` se tretira kao *privremeni (rvalue)* i resursi se **prenose**, umesto da se kopiraju. Tako se ceo vektor „predaje“ funkciji bez kopiranja podataka.
- Nakon toga, `niz` ostaje u praznom, ali validnom stanju.

Ispis:

```
Broj elemenata: 5
Broj elemenata: 5
Posle std::move, niz ima 0 elemenata.
```

2.6 Rad sa datotekama u jeziku C++

U jeziku C rad sa datotekama obavlja se pomoću pokazivača tipa `FILE*` i funkcija kao što su `fopen`, `fscanf` i `fprintf`. U jeziku C++ za tu svrhu postoje **stream** objekti, koji omogućavaju rad sa datotekama na način potpuno analogan radu sa standardnim ulazom i izlazom (`cin`, `cout`). Prednost stream pristupa je u tome što su ovi objekti **tip-bezbedni**, **jednostavni za upotrebu**, i mogu se **prosleđivati funkcijama** kao i svaki drugi objekat.

2.6.1 Vrste file streamova

U zaglavlju `<fstream>` definisane su tri osnovne klase:

- `std::ifstream` – za **čitanje** iz datoteke (*input file stream*),
- `std::ofstream` – za **pisanje** u datoteku (*output file stream*),
- `std::fstream` – za **istovremeno čitanje i pisanje** (*file stream*).

Njihova upotreba je vrlo slična radu sa `cin` i `cout` objektima.

2.6.2 Pisanje u datoteku

Sledeći primer ilustruje jednostavno pisanje u tekstualnu datoteku:

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

int main() {
    ofstream izlaz("rezultati.txt");  // otvara datoteku za pisanje

    if (!izlaz) {                     // provera da li je datoteka otvorena
        cerr << "Greška: ne mogu da otvorim datoteku!\n";
        return 1;
    }

    izlaz << "Prvi red" << endl;
    izlaz << "Drugi red" << endl;
    izlaz.close();                    // zatvara datoteku (može i automatski)
}
```

Datoteka se otvara konstruktorom objekta `ofstream`, a zatvara metodom `close()`. Kao i kod `cout`, operator `<<` koristi se za formatirano ispisivanje podataka.

2.6.3 Dodavanje podataka (režim *append*)

Podrazumevano, `ofstream` otvara datoteku u **režimu pisanja** i pri tome briše njen prethodni sadržaj. Ako želimo da se novi podaci **dodaju na kraj postojeće datoteke**, potrebno je otvoriti datoteku u režimu *append* pomoću zastavice `ios::app`.

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

int main() {
    ofstream izlaz("rezultati.txt", ios::app);  // otvara datoteku za dodavanje
    if (!izlaz) {
        cerr << "Greška: ne mogu da otvorim datoteku!\n";
        return 1;
    }

    izlaz << "Novi red koji se dodaje na kraj datoteke" << endl;
    izlaz.close();
}
```

U ovom slučaju, datoteka se ne prepisuje, već se novi tekst upisuje **na kraj** postojećeg sadržaja. Pored ovog, ima i drugih režima, na primer `ios::binary` – otvaranje u binarnom režimu (bez konverzije znakova novog reda).

> **Napomena:** Režim otvaranja može se kombinovati pomoću operatora `|`, na primer:
>
> ```
> ofstream izlaz("rezultati.txt", ios::binary | ios::app);
> ```

2.6.4 Čitanje iz datoteke

Čitanje iz datoteke vrši se na isti način kao čitanje sa standardnog ulaza (`cin`):

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

int main() {
    ifstream ulaz("rezultati.txt");   // otvara datoteku za čitanje
    if (!ulaz) {
        cerr << "Greška: ne mogu da otvorim datoteku!\n";
        return 1;
    }

    string red;
    while (getline(ulaz, red))        // čita red po red
```

```
        cout << red << endl;

    ulaz.close();
}
```

Kao što se vidi, rad sa `ifstream` i `ofstream` potpuno je analogan radu sa `cin` i `cout`, samo što podaci dolaze iz (ili odlaze u) datoteku.

Napomena (provera uspešnosti otvaranja datoteke):

Kada se datoteka otvara pomoću objekta `ofstream`, `ifstream` ili `fstream`, može doći do greške — na primer:

- ako datoteka ne postoji (pri čitanju),
- ako program nema dozvolu pisanja u direktorijum,
- ako je putanja pogrešna ili uređaj nije dostupan.

U takvim situacijama stream prelazi u **neispravno stanje**, što znači da dalji upis ili itanje neće biti moguće. Provera `if (!izlaz)` zapravo proverava **stanje toka**: operator `!` vraća `true` ako je tok u grešci (ako je `failbit` ili `badbit` postavljen).

Prednosti u odnosu na 'FILE*'

U odnosu na C pristup, file streamovi imaju nekoliko važnih prednosti:

- nije obavezno eksplicitno pozvati metodu `close()`; naime, svaki od ovih objekata ima destruktor koji se automatski poziva kada objekat izađe iz svog opsega (npr. na kraju funkcije). Destruktor tada sam zatvara datoteku i oslobađa sve povezane resurse.
- podržavaju **preopterećene operatore** `<<` **i** `>>`, pa je čitanje i pisanje podataka jednostavnije i čitljivije;
- mogu se **prosleđivati funkcijama kao reference** (npr. `void upisi(std::ostream& out)`), čime se omogućava da ista funkcija radi i sa datotekom i sa standardnim izlazom.

2.6.5 Prenos streamova funkcijama

U sledećem primeru prikazano je kako ista funkcija može da ispisuje podatke bilo u datoteku, bilo na standardni izlaz, u zavisnosti od prosleđenog argumenta:

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

void ispis(ostream& out, int n) {
    out << "Rezultat: " << n << endl;
}

int main() {
    ofstream fajl("rez.txt");
    if (!fajl) return 1;

    ispis(cout, 42);   // ispis na ekran
    ispis(fajl, 42);   // ispis u datoteku
```

```
}
```

Ovo pokazuje još jednu prednost streamova — isti programski kod radi sa različitim izvorima i odredištima podataka, bez potrebe da se menja logika funkcije.

2.6.6 String streamovi

Pored datoteka, standardna biblioteka omogućava i **streamove nad tekstualnim stringovima**, definisane u zaglavlju `<sstream>`:

- `std::istringstream` — čitanje iz stringa,
- `std::ostringstream` — pisanje u string,
- `std::stringstream` — dvosmerni tok.

String streamovi omogućavaju da se podaci formatiraju ili analiziraju **kao da se čitaju i pišu u fajl**, ali u stvari sve ostaje u memoriji, unutar običnog stringa. To je naročito korisno kada želimo da:

- pripremimo tekstualni rezultat koji se kasnije koristi u programu (npr. za ispis, logovanje ili GUI),
- parsiramo podatke iz stringa kao da dolaze sa ulaza.

Primer 1: formatiranje brojeva i teksta u stringu

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

int main() {
    double temperatura = 23.6789;
    int dan = 5;
    string mesec = "april";

    ostringstream oss;  // tok koji piše u string

    oss << "Datum: " << dan << ". " << mesec
        << ", temperatura: " << fixed << setprecision(1)
        << temperatura << "°C";

    string poruka = oss.str();  // dobijeni string

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

Izlaz:

```
Datum: 5. april, temperatura: 23.7°C
```

Ovim pristupom lako se formira složen tekst sa različitim tipovima podataka, bez potrebe za `sprintf`, ručnim konkatenacijama (+) ili konverzijama tipova.

Primer 2: čitanje vrednosti iz stringa

Druga česta upotreba je **parsiranje** — kada neki string sadrži brojeve koje želimo da izdvojimo.

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

int main() {
    string linija = "42 3.14 256";
    istringstream iss(linija);

    int a; double b; int c;
    iss >> a >> b >> c;  // čitanje iz stringa kao sa ulaza

    cout << "a = " << a << ", b = " << b << ", c = " << c << endl;
}
```

`istringstream` omogućava jednostavno čitanje brojeva iz stringa, bez ručnih parsiranja i bez rizika od grešaka u konverziji.

2.7 Ranges (C++20)

Jedna od najvažnijih novina uvedenih u standardu **C++20** je biblioteka **Ranges**, koja predstavlja značajno unapređenje u radu sa kolekcijama podataka. Ideja *ranges* biblioteke jeste da objedini **sekvencu elemenata** i **operacije koje se nad njom primenjuju**, čineći rad sa podacima izražajnijim i bezbednijim.

U klasičnom pristupu C++-a funkcije iz zaglavlja `<algorithm>` (poput `sort`, `count_if`, `transform`) zahtevaju prosleđivanje *početnog i krajnjeg iteratora*. U *ranges* pristupu, umesto parova iteratora, algoritmi direktno primaju **kolekciju (range)** — što kod čini čitljivijim i smanjuje mogućnost grešaka.

Svi algoritmi iz `<algorithm>` imaju svoje *ranges* ekvivalente u zaglavlju `<ranges>`.

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

int main() {
    vector<int> brojevi = {1, 2, 3, 4, 5, 6};

    // Filtriranje i transformacija pomoću ranges pipeline-a
    auto kvadrati_parnih =
        brojevi
```

```
        | views::filter([](int x){ return x % 2 == 0; })
        | views::transform([](int x){ return x * x; });

    for (int x : kvadrati_parnih)
        cout << x << " ";
}
```

Rezultat:

4 16 36

Ovaj zapis se može posmatrati kao **"lanac transformacija" (pipeline)**: najpre se biraju samo parni brojevi (`filter`), zatim se svaki kvadrira (`transform`), a rezultat se može direktno koristiti u `for` petlji — bez pravljenja privremenih vektora ili ručnih petlji.

> **Napomena (kompajliranje programa sa Ranges bibliotekama):**
>
> Podrška za *ranges* uvedena je u standardu **C++20**, pa programi koji koriste ove konstrukcije moraju biti prevedeni uz odgovarajuću opciju kompajlera.
>
> Na primer:
>
> ```
> g++ -std=c++20 primer.cpp -o primer
> ```
>
> Ako se program prevodi bez ove opcije, kompajler neće prepoznati zaglavlje `<ranges>` niti izraze poput `views::filter` i `ranges::sort`.

Prednosti *ranges* pristupa:

- kraći i čitljiviji kod, bez ručnih iteratora,
- mogućnost **komponovanja operacija** (`filter | transform | take | sort`),
- **lenja evaluacija** — elementi se obrađuju po potrebi, bez nepotrebnog kopiranja,
- bolja integracija sa STL kontejnerima i modernim C++ stilom.

Primer:

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

int main() {
    vector<int> v = {3, 8, 1, 5, 2, 9, 4};

    // Izdvoj prvih 3 broja većih od 3
    for (int x : v
                | views::filter([](int x){ return x > 3; })
                | views::take(3))
        cout << x << " ";
}
```

Rezultat:

```
8 5 9
```

Biblioteka <ranges> ne donosi samo *poglede* (eng. views), već i **modernu verziju klasičnih algoritama** iz <algorithm>. Oni se mogu primenjivati direktno na kolekcijama, bez navođenja početnog i krajnjeg iteratora.

```cpp
#include <iostream>
#include <vector>
#include <ranges>
using namespace std;

int main() {
    vector<int> brojevi = {7, 2, 9, 4, 1, 5};

    ranges::sort(brojevi);  // sortiranje bez iteratora

    cout << "Sortirani brojevi: ";
    for (int x : brojevi)
        cout << x << " ";
}
```

Rezultat:

```
Sortirani brojevi: 1 2 4 5 7 9
```

Kao i u primeru sa *pogledima*, rad je jednostavniji i čitljiviji: nije potrebno pisati `sort(brojevi.begin(), brojevi.end());`, jer `ranges::sort` sam prepoznaje granice kolekcije. Na isti način dostupne su i druge funkcije: `ranges::find`, `ranges::count`, `ranges::max`, `ranges::reverse`, i mnoge druge.

Biblioteka **ranges** donosi savremen i deklarativan način rada sa kolekcijama podataka. Ona omogućava da se operacije nad nizovima izraze jednostavno, slično kao u funkcionalnim jezicima, bez ručnog rukovanja iteratorima i petljama. Zbog čitljivosti i efikasnosti, *ranges* postaju deo idiomatskog C++ stila u modernim programima.

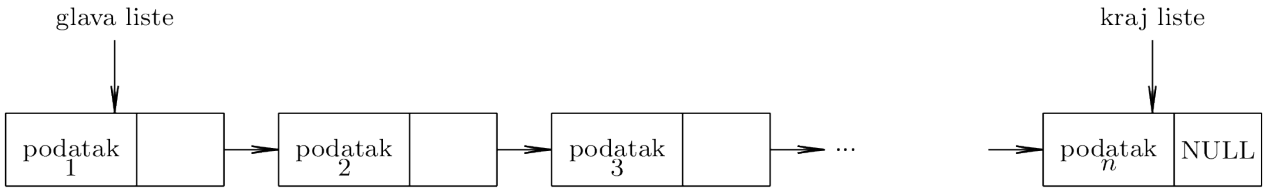
Glava 3

Sekvencijalne strukture podataka

3.1 Jednostruko povezane liste

Jednostruko povezana lista je dinamička struktura u kojoj su pojedinačni elementi povezani pokazivačima. U ovoj strukturi postoji veza pokazivačem (i to samo u jednom smeru) samo između susednih elemenata. U jeziku C ne postoji tip koji opisuje povezane liste, već se one kreiraju koristeći druge tipove i, najčešće, pokazivače.

Jednostruko povezanu listu čine elementi (čvorovi liste) od kojih svaki sadrži podatak izabranog tipa i pokazivač. Svaki pokazivač pokazuje na jedan (sledeći) element liste i ti pokazivači povezuju elemente u jednu celinu — listu. Izuzetak je poslednji čvor čiji pokazivač ima vrednost `NULL` (ili `nullptr` u C++). Ovakva reprezentacija omogućava da se čvorovi fizički nalaze bilo gde u hipu, dok logički redosled određuju isključivo veze između čvorova.



Slika 1: Ilustracija povezane liste

Listi se obično pristupa preko pokazivača na njen početni element, tj. na *početak* ili *glavu* liste. Zato se pokazivač na početak liste mora čuvati i održavati tokom svih operacija koje mogu da ga promene, jer omogućava pristup svim elementima liste. Iz sličnih razloga, ponekad se čuva i pokazivač na poslednji element liste, tj. na njen *kraj*.

Odnos povezanih lista i nizova u teoriji i praksi

Više podataka (elemenata) istog tipa može se objediniti pomoću povezane liste, statički alociranih nizova ili dinamički alociranih nizova. Ove strukture se razlikuju po načinu alokacije memorije, složenosti osnovnih operacija i realnim performansama na savremenom hardveru.

- **Veličina i proširenje**
  - *Niz* zauzima jedan uzastopni blok memorije. Statički niz mora imati veličinu poznatu u fazi kompilacije, dok dinamički niz zahteva dostupnost dovoljno velikog bloka slobodne

memorije prilikom alokacije. Veličina dinamičkog niza se može menjati samo funkcijom `realloc`, i to samo na kraju niza.
  - *Lista* se sastoji od čvorova rasutih po hipu; svaki čvor se alocira posebno, pa broj elemenata liste može se smanjivati i povećavati tokom izvršavanja programa jednostavnim dodavanjem ili brisanjem pojedinačnih elemenata.
  - Kod *liste* veliki broj malih alokacija usporava program i fragmentiše memoriju. To dalje dovodi do loše iskorišćenosti keš memorije, čvorovi su udaljeni, pa CPU učestalo povlači podatke iz glavne memorije.
  - *Lista* ima dodatni utrošak memorije po čvoru – osim samog podatka, mora se čuvati i pokazivač - na primer, 8B na 64-bitnim sistemima (ako imamo 10^6 elemenata tipa `int` niz će zauzeti 4MB, a lista 16MB).

- **Pristup i modifikacije**
  - *Niz* omogućava direktan pristup `i`-tom elementu u `O(1)` jer su smešteni u uzastopnim memorijskim lokacijama. Ali, umetanje ili brisanje elementa može zahtevati pomeranje velikih blokova memorije (`O(n)`).
  - Elementi *liste* su raspoređeni proizvoljno, pa je za pristup `i`-tom elementu potrebno `O(n)` prateći pokazivače, ali umetanje ili brisanje čvora na početku ili iza poznatog čvora je `O(1)` i ne utiče na ostale elemente.
  - Spajanje celih pod-*lista* može da se uradi u `O(1)` (korisno u određenim algoritmima).
  - Nepredvidiva raspodela elemenata *liste* u memoriji otežava paralelizaciju, tj. optimizacije koje se postižu korišćenjem više niti.

Danas se povezane liste najviše isplate kad je *ključno* često umetanje/brisanje (npr. realizacija reda prioriteta sa spajanjem, implementacija heš-tabele sa ulančavanjem). U većini drugih slučajeva dinamički ili statički niz (ili `std::vector`) daće bolje ukupne performanse.

Karakteristika	Niz (statički/dinamički)	Jednostruko povezana lista
Memorijski raspored	Kontinuiran	Raspršen po hipu
Promena veličine	Teško / samo `realloc`	Lako — dodavanjem/brisanje čvorova
Pristup i-tom elementu	`O(1)`	`O(n)`
Umetanje/brisanje na početku	`O(n)`	`O(1)`
Umetanje/brisanje posle datog čvora	`O(n)`	`O(1)`
Keš lokalnost	Odlična	Slaba
Memorijsko zauzeće	Minimalan	Dodatni pokazivač po čvoru

Jednostruko povezana lista u standardnoj biblioteci C++: `std::forward_list`

U standardnoj biblioteci C++ postoji kolekcija `std::forward_list<T>` koja predstavlja *jednostruko povezanu listu*. Za njeno korišćenje potrebno je uključiti zaglavlje `<forward_list>`.

Operacija	Opis	Vremenska složenost
`begin()`	postavlja iterator na prvi element liste	`O(1)`
`end()`	postavlja iterator na kraj liste, tj. na „poziciju jedan iza poslednjeg"	`O(1)`
`insert_after(it, x)`	Umeće element **posle** čvora na koji iterator `it` pokazuje.	`O(1)`
`erase_after(it)`	Briše element **posle** čvora na koji iterator `it` pokazuje.	`O(1)`
`push_front(x)` / `pop_front()`	umetanje/brisanje na elemenata početku liste	`O(1)`

Iterator `std::forward_list<T>` podržava samo operator ++.

Elementi povezane liste

Tip elementa ili čvora liste obično se definiše na sledeći način:

```
struct cvor {
  <type> podatak;
  struct cvor* sledeci;
};
```

U navedenoj definiciji, `<type>` označava tip podatka koji element sadrži. To može biti proizvoljan tip (moguće i struktura), a može biti i pokazivač `void*` ako se želi generička lista koja se lako može prilagoditi nekom konkretnom tipu. Element liste može da sadrži i više od jednog podatka (i, naravno, ti podaci ne moraju da budu istog tipa). Član strukture `struct cvor* sledeci;` je dozvoljen u okviru same definicije strukture `struct cvor`, jer se zna koliko memorije zahteva (onoliko koliko zahteva i bilo koji drugi pokazivač).

Koristeći `typedef` može se uvesti ime strukture koje ne sadrži ključnu reč `struct` (element `sledeci` ipak mora da se deklariše tipom `struct cvor`):

```
typedef struct cvor {
  <type> podatak;
  struct cvor* sledeci;
} cvor;
```

Dozvoljeno je da ime strukture i novo ime tipa budu jednaki (`cvor` u ovom primeru). Novo ime tipa za `struct cvor` moglo je da bude uvedeno i pre i nakon definicije strukture (ali se novo ime ne može koristiti pre naredbe `typedef`):

```
typedef struct _cvor cvor;

struct _cvor {
  <type> podatak;
  cvor* sledeci;
```

```
};
```

ili

```
struct _cvor {
  <type> podatak;
  struct _cvor* sledeci;
};

typedef struct _cvor cvor;
```

Ponekad se imenuje tip pokazivača na čvor liste:

```
typedef struct _cvor* pCvor;
struct _cvor {
  <type> podatak;
  pCvor sledeci;
};
```

Veličina čvora nije jednaka samo veličini podatka. Uvek postoji dodatni trošak zbog pokazivača `sledeci`. Na 64-bitnoj arhitekturi pokazivač zauzima 8 bajtova, pa čvor koji čuva `int` (4 bajta) ukupno zauzima najmanje 16 bajtova zbog poravnanja memorije. Zato liste troše znatno više memorije od nizova.

Radi preglednosti, u primerima u `C` u nastavku biće smatrano da je element liste deklarisan na sledeći način (tj. da ima samo jedan podatak, podatak tipa `int`):

```
typedef struct cvor {
    int podatak;
    struct cvor *sledeci;
} cvor;
```

Elementi liste kreiraju se pojedinačno i onda spajaju sa tekućom povezanim listom. Novi element liste može biti kreiran sledećom funkcijom:

```
cvor *novi_cvor(int podatak)
{
  cvor *novi = malloc(sizeof(cvor));
  if (novi == NULL)
    return NULL;
  novi->podatak = podatak;
  novi->sledeci = NULL;
  return novi;
}
```

Zapis `novi->podatak` je kraći zapis za `(*novi).podatak` (isto važi za bilo koji pokazivač na strukturu, dakle umesto `(*a).podatak` može se pisati `a->podatak`).

Ukoliko element liste sadrži više podataka, onda bi navedena funkcija za kreiranje elementa imala po jedan parametar za svaki od njih.

Ako element liste treba da sadrži nisku, moguće je da ta niska bude predstavljena nizom karaktera koji se čuva u elementu liste. Tada je potrebno znati gornje ograničenje za dužinu niske (računajući i završnu nulu), kao u narednom primeru:

```
typedef struct {
  char podatak[MAX];
  struct cvor *sledeci;
} cvor;
```

Prilikom alokacije niske, tada bi umesto dodele, podatak bilo potrebno inicijalizovati funkcijom `strcpy`:

```
cvor *novi_cvor(const char *podatak)
{
  ...
  strcpy(novi->podatak, podatak);
}
```

Druga mogućnost je da se u elementu povezane liste čuva samo pokazivač, a da se niz karaktera alocira i dealocira nezavisno od liste:

```
typedef struct {
  char *podatak;
  struct cvor *sledeci;
} cvor;
```

Onda je, prilikom kreiranja novog elementa liste, moguće izvršiti bilo plitko, bilo duboko kopiranje (videti prvi deo ove knjige, ??). U prvom slučaju, sa plitkim kopiranjem, kreiranje novog elementa liste može imati sledeću formu:

```
cvor *novi_cvor(char *podatak)
{
  ...
  novi->podatak = podatak;
}
```

Sa ovim pristupom treba biti obazriv jer je moguće da se istom nizu karaktera pristupa preko više pokazivača (što ponekad može biti opravdano). Plitko kopiranje uglavnom ima smisla ako se podaci u čvorovima liste samo čitaju a ne menjaju se. Tada ima smisla primeniti kvalifikator `const` na tip argumenta čime se dodatno osiguravamo od slučajnih promena podataka unutar funkcije (ako bi tip argumenta bio `const char *podatak`, onda bi kompilator prijavio grešku pri svakom pokušaju modifikacije tog podatka unutar funkcije). Sa druge strane, ne postoji garancija da se podaci koji se čuvaju u listi neće promeniti iz neke druge funkcije. Naime, funkciji koja prima podatak tipa `const char*` može se proslediti i pokazivač tipa `char*`, a sadržaj na koji on ukazuje može se (negde drugde) nesmetano menjati.

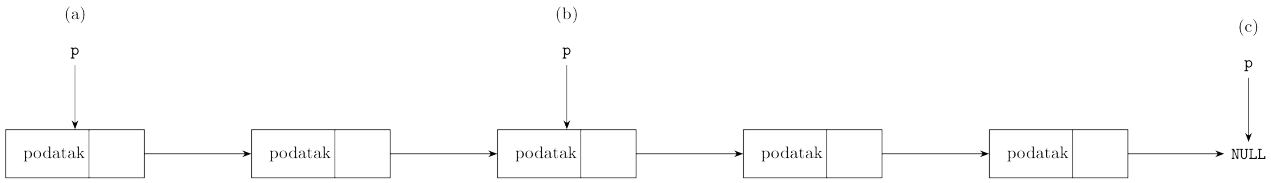
U drugom slučaju (sa dubokim kopiranjem) potrebno je alocirati potreban prostor i iskopirati nisku (na primer, funkcijama `malloc` i `strcpy` ili, kraće, funkcijom `strdup`), a prilikom uklanjanja čvorova liste, potrebno je ukloniti i ovako alocirane kopije (funkcijom `free`):

```
cvor *novi_cvor(const char *podatak)
{
  ...
  novi->podatak = strdup(podatak);
}
```

Sentineli

Kod rada sa jednostruko povezanim listama, veliki broj operacija zavisi od položaja pokazivača u listi. Analizom takvih operacija uočava se da pokazivač može zauzimati jedan od tri osnovna položaja:

1. *na prvom elementu liste* (početak),
2. *između dva susedna elementa* (unutrašnjost),
3. *iza poslednjeg elementa* (kraj).



Prikaz tri moguća položaja pokazivača nad jednousmernom listom: (a) položaj na prvom elementu liste, (b) položaj na unutrašnjem elementu, i (c) položaj iza poslednjeg elementa liste (pokazivač na `NULL`).

U klasičnim implementacijama algoritama nad listama, ova tri slučaja zahtevaju odvojena uslovna grananja. Takav kod otežava održavanje, povećava verovatnoću grešaka i umanjuje mogućnost za elegantna proširenja koda.

U klasičnim implementacijama svaki od ova tri položaja zahteva poseban tretman u kodu. Na primer, umetanje na početak liste ne sledi isti obrazac kao umetanje iza proizvoljnog čvora, a brisanje prvog elementa zahteva posebno grananje. Ovi izuzeci čine kod sklonim greškama i otežavaju proširenje strukture.

Radi pojednostavljenja uvodi se **sentinel-čvor** (naziva se još i *čuvarski* ili *lažni čvor*). Sentinel-čvor je fiktivni čvor koji ne sadrži korisne podatke i ne predstavlja stvarni element liste. On služi isključivo kao tehnički objekat koji obeležava početak (ili kraj) liste i pojednostavljuje algoritme nad listom. Kod jednostruko povezane liste sentinel zauzima položaj *pre* prvog stvarnog elementa; njegovo polje `sledeći` upućuje na prvi element ili je `NULL` (tj. `nullptr`) ako je lista prazna.

Uvođenjem ovakvog fiktivnog čvora eliminišu se posebni slučajevi pri umetanjima i brisanjima na početku: sve operacije postaju uniformne, jer se nikada direktno ne manipuliše „stvarnim“ prvim elementom, već isključivo poljem `sledeci` sentinela. Jedina mana ovog pristupa je u korišćenju dodatne memorije za jedan čvor što je zanemarljivo.

Implementacija `std::forward_list` ne koristi sentinel-čvor, ali implementacija `std::list` ih koristi i o tome više kasnije.

Implementacija koncepta sentinel-čvor u okviru jezika C:

```
/* Formiranje prazne liste: vraća pokazivač na sentinel */
cvor *inicijalizuj_listu(void)
{
    // sentinel - nema korisnu vrednost
    cvor *s = malloc(sizeof(cvor));
    //neuspešna alokacija memorije
    if (s == NULL) return NULL;
    //u suprotnom, sentinel pokazuje na NULL jer
    //lista je prazna
    s->sledeci = NULL;
    return s;
}
```

Osnovne operacije za rad sa povezanim listama

Kao što je već rečeno, ne postoji tip koji opisuje listu, već se listi može pristupiti preko pokazivača na njen početak, odnosno na sentinel-čvor. Dodavanje elementa na početak liste može se implemenirati na sledeći način (SLIKA):

```
void dodaj_cvor(cvor *sentinel, cvor *novi)
{
  novi->sledeci = sentinel->sledeci;
  sentinel->sledeci = novi;
}
```

Kao što je već istaknuto, prvi "pravi" element liste se nalazi na polju `sledeći` datog sentinel-čvora. Dodavanje novog čvora neposredno iza sentinela predstavlja logičko dodavanje na početak liste. Funkcija može da vrati i podatak o nekakvom statusu (a ne `void`):

```
typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA
} status;

/* ... */

status dodaj_cvor(cvor *sentinel, cvor *novi)
{
  novi->sledeci = sentinel->sledeci;
  sentinel->sledeci = novi;
```

```
  return OK;
}
```

Dodatno, možemo napraviti funkciju koja vrši i alokaciju čvora, pre nego što ga uveže u listu:

```
status dodaj_cvor(cvor *sentinel, int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
  novi->sledeci = sentinel->sledeci;
  sentinel->sledeci = novi;
  return OK;
}
```

Navedena funkcija ima složenost **O(1)**.

Idiomska forma za prolazak kroz sve elemente povezane liste u C je:

```
for (p = sentinel->sledeci; p != NULL; p = p->sledeci)
  ...
```

Koristeći ovu formu, novi član može da se doda i na kraj liste, na primer sledećom funkcijom:

```
status dodaj_na_kraj(cvor *sentinel, int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;

  cvor *p;
  novi->sledeci = NULL;

  for (p = sentinel; p->sledeci != NULL; p = p->sledeci)
    ;

  p->sledeci = novi;

  return OK;
}
```

Složenost prethodne funkcije je reda **O(n)**, gde je n broj elemenata liste. Naglasimo da se ovakve funkcije nikada ne koriste zbog svoje neefikasnosti: ukoliko je potrebna opisana funkcionalnost, onda je potrebno promeniti samu definiciju strukture kako bi se omogućilo efikasno izvršavanje. Na primer, dodavanje elementa na kraj liste može da se izvrši i u vremenu **O(1)**, ali tada je potrebno održavanje i pokazivača na poslednji element liste:

```
status dodaj_na_kraj(cvor *sentinel, cvor **pokazivac_na_kraj, int podatak)
{
    cvor *novi = novi_cvor(podatak);
    if (novi == NULL)
        return NEDOVOLJNO_MEMORIJE;

    novi->sledeci = NULL;

    (*pokazivac_na_kraj)->sledeci = novi;
    *pokazivac_na_kraj = novi;

    return OK;
}
```

Navedena funkcija podrazumeva da je zadat pokazivač na pokazivač na kraj liste, kako bi se sam pokazivač na kraj mogao menjati. Čak i kada je lista prazna, pokazivač na početak liste se neće menjati jer to je specijalan sentinel-čvor. U praksi, ako se često dodaje na kraj liste, isplati se održavati i dodatni pokazivač na poslednji element, čime se složenost ove operacije spušta sa $O(n)$ na $O(1)$. Primer korišćenja ove funkcije je prikazan u narednom primeru:

```
cvor *lista = inicijalizuj_listu();
cvor *kraj = lista;  /* na početku, kraj je sentinel */

dodaj_na_kraj(lista, &kraj, 42);
dodaj_na_kraj(lista, &kraj, 7);
```

Analogno se može implementirati i da funkcija koja novi element dodaje na početak liste, a održava i pokazivač na kraj:

```
status dodaj_cvor(cvor *sentinel, cvor **pokazivac_na_kraj, int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;

  novi->sledeci = sentinel->sledeci;
  sentinel->sledeci = novi;

  if (*pokazivac_na_kraj == sentinel)
    *pokazivac_na_kraj = novi;

  return OK;
}
```

Sadržaj liste može se ispisati sledećom funkcijom:

```
void ispisi_elemente_liste(cvor *sentinel)
{
  cvor* tekuci;
  for (tekuci = sentinel->sledeci; tekuci != NULL; tekuci = tekuci->sledeci)
    printf("%d ", tekuci->podatak);
}
```

ili njenom rekurzivnom verzijom:

```
void ispisi_elemente_liste_rek(cvor *tekuci)
{
    if (tekuci == NULL)
        return;

    printf("%d ", tekuci->podatak);
    ispisi_elemente_liste_rek(tekuci->sledeci);
}
```

Rekurzivno se mogu implementirati i druge gore navedene funkcije, ali te verzije nisu mnogo jednostavnije od iterativnih. Međutim, rekurzivna verzija ispisa elemenata može se lako prepraviti da ispisuje elemente liste u obratnom poretku:

```
void ispisi_listu_obratno_rek(cvor *tekuci)
{
    if (tekuci == NULL)
        return;

    ispisi_listu_obratno_rek(tekuci->sledeci);
    printf("%d ", tekuci->podatak);
}
```

Obratite pažnju da obe funkcije pozivamo sa `ispisi_elemente_liste_rek(sentinel->sledeci);`, što je neophodno da se ne bi ispisala vrednost sentinel čvora.

> **Napomena:** Rekurzivno rešenje može biti neefikasno jer formira onoliko stek-okvira koliko ima elemenata, što može dovesti do prekoračenja steka. U takvim situacijama poželjno je izbeći rekurziju.

Početni element liste može se pročitati i osloboditi primenom sledeće funkcije, koja kroz argument vraća podatak iz izbrisanog elementa:

```
status obrisi_sa_pocetka(cvor *sentinel, int *podatak)
{
   if (sentinel->sledeci == NULL)
       return NEMA_ELEMENATA;

   cvor *p = sentinel->sledeci;
   *podatak            = p->podatak;
```

```
    sentinel->sledeci   = p->sledeci;
    free(p);
    return OK;
}
```

Ako je potrebno održavati i pokazivač na kraj liste, funkcija može izgledati ovako:

```
status obrisi_sa_pocetka(cvor *sentinel,
                         cvor **pokazivac_na_kraj,
                         int *podatak)
{
    if (sentinel->sledeci == NULL)
        return NEMA_ELEMENATA;

    cvor *p = sentinel->sledeci;
    *podatak          = p->podatak;
    sentinel->sledeci = p->sledeci;
    free(p);

    //ako je lista prazna, postavlja se pokazivac na kraj na sentinel
    if (sentinel->sledeci == NULL)
        *pokazivac_na_kraj = sentinel;

    return OK;
}
```

Složenost brisanja prvog elementa je $\mathbf{O(1)}$.

Brisanje elemenata iz jednostruko povezane liste sa sentinel-čvorom predstavlja ključnu operaciju nad ovom strukturom podataka. Pri tome razlikujemo dva konceptualna nivoa:

1. **brisanje jednog elementa** (npr. prvog),
2. **brisanje svih elemenata**, tj. **pražnjenje** liste,
3. **uništavanje liste** (uključujući i sentinel-čvor).

U svim slučajevima, zahvaljujući sentinel-čvoru na početku liste, operacije su jednostavnije i ne zahtevaju posebne slučajeve za praznu listu.

Kod operacija uklanjanja svih čvorova, sentinel-čvor ostaje. Zahvaljujući sentinel-čvoru, ova operacija ne zahteva izmenu pokazivača na početak liste — on ostaje stabilan:

```
void obrisi_listu(cvor *sentinel)
{
    cvor *tekuci = sentinel->sledeci;

    while (tekuci != NULL) {
        cvor *p = tekuci;
        tekuci  = tekuci->sledeci;
        free(p);
    }
```

```
    sentinel->sledeci = NULL;  /* lista je sada prazna */
}
```

U ovom kodu ključno je da se pokazivač `tekuci` unapred pomeri na sledeći element, jer bi u suprotnom oslobađanje trenutnog čvora uništilo informaciju o nastavku liste. Složenost operacije je **O(n)**.

Ako algoritam koristi i pokazivač na poslednji element, i njega je potrebno ažurirati:

```
void obrisi_listu(cvor *sentinel, cvor **pokazivac_na_kraj)
{
    cvor *tekuci = sentinel->sledeci;

    while (tekuci != NULL) {
        cvor *p = tekuci;
        tekuci  = tekuci->sledeci;
        free(p);
    }

    sentinel->sledeci  = NULL;
    *pokazivac_na_kraj = sentinel;   /* kraj je sada sentinel */
}
```

Rekurzivna implementacija omogućava elegantan opis algoritma, ali je nepraktična za veoma duge liste zbog mogućeg prekoračenja steka. Iako je ovde prikazana i ta verzija, u praksi se ne koristi. Pomoćna funkcija briše samo *prave* elemente; sentinel se ne oslobađa:

```
static void obrisi_elemente(cvor *tekuci)
{
    if (tekuci == NULL)
        return;

    obrisi_elemente(tekuci->sledeci);
    free(tekuci);
}
```

A glavna funkcija za pražnjenje liste glasi:

```
void obrisi_listu(cvor *sentinel)
{
    obrisi_elemente(sentinel->sledeci);
    sentinel->sledeci = NULL;
}
```

> **Napomena:** Iako je rekurzija konceptualno jasna, preporučuje se iterativno rešenje zbog ograničene veličine steka u realnim sistemima.

Ukoliko je sentinel-čvor alociran dinamički (npr. u funkciji `inicijalizuj_listu`), po završetku upotrebe liste neophodno ga je takođe osloboditi. Pošto se pri tome menja i sam pokazivač na listu, funkcija prirodno prima **pokazivač na pokazivač**:

```
void unisti_listu(cvor **psentinel)
{
    if (*psentinel == NULL)
        return;     /* lista već uništena */

    obrisi_listu(*psentinel);  /* obriši sve prave čvorove */

    free(*psentinel);          /* oslobodi sentinel-čvor */
    *psentinel = NULL;         /* poništi pokazivač ka listi */
}
```

Time se kompletna struktura povezane liste trajno uklanja.

Za razliku od nizova, u jednostruko povezanoj listi nije moguće izvršiti direktan pristup elementu sa zadatim indeksom u vremenu **O(1)**, jer čvorovi nisu smešteni u uzastopnim memorijskim lokacijama. Umesto toga, pretraga se obavlja **sekvencijalnim prolaskom** kroz listu, počevši od prvog elementa.

Zbog toga je pretraga u jednostruko povezanoj listi složenosti **O(n)**, gde je n broj elemenata liste: u najgorem slučaju potrebno je obići sve čvorove.

Kod liste sa sentinel-čvorom, pretraga uvek počinje od `sentinel->sledeci`, jer sentinel ne sadrži korisnu vrednost.

Najjednostavniji oblik pretrage traži prvi čvor čije je polje `podatak` jednako zadatoj vrednosti. Ako je takav čvor pronađen, funkcija vraća njegovu adresu; u suprotnom vraća `NULL`:

```
cvor *nadji_prvi(cvor *sentinel, int vrednost)
{
    for (cvor *p = sentinel->sledeci; p != NULL; p = p->sledeci)
        if (p->podatak == vrednost)
            return p;

    return NULL;   /* nema elementa sa zadatom vrednošću */
}
```

Ovakva funkcija je korisna kad je potrebno samo pročitati ili izmeniti podatak u pronađenom čvoru (na primer, povećati neki brojač, ažurirati strukturu, itd).

Za operaciju **brisanja** čvora u jednostruko povezanoj listi neophodno je imati pristup i čvoru koji mu prethodi, kako bi se ažurirale veze. Zbog toga se često koristi pretraga koja vraća **prethodni** element u odnosu na onaj koji tražimo.

Ako se traži prvi čvor čiji je `podatak == vrednost`, možemo implementirati funkciju koja vraća pokazivač na **prethodni** čvor; ako takav element ne postoji, vraća se `NULL`:

```
cvor *nadji_prethodni(cvor *sentinel, int vrednost)
{
    cvor *prev = sentinel;
```

```
    cvor *curr = sentinel->sledeci;

    while (curr != NULL) {
        if (curr->podatak == vrednost)
            return prev;   /* prev je prethodni u odnosu na curr */

        prev = curr;
        curr = curr->sledeci;
    }

    return NULL;          /* nema traženog elementa */
}
```

Ovakva funkcija je naročito pogodna za brisanje elementa u **O(1)** nakon što je pronađen njegov prethodnik:

```
status obrisi_prvi_sa_vrednoscu(cvor *sentinel, int vrednost)
{
    cvor *prev = nadji_prethodni(sentinel, vrednost);
    if (prev == NULL)
        return NEMA_ELEMENATA;  /* nema elementa sa datom vrednošću */

    cvor *za_brisanje   = prev->sledeci;
    prev->sledeci       = za_brisanje->sledeci;
    free(za_brisanje);

    return OK;
}
```

Ovde je sama operacija brisanja, nakon pronalaska prethodnog čvora, složenosti **O(1)**, dok je ukupna složenost (zbog pretrage) i dalje **O(n)** u najgorem slučaju.

U složenijim programima korisno je definisati pretragu ne samo po konkretnoj vrednosti, već po proizvoljnom uslovu. To se može postići prosleđivanjem pokazivača na funkciju (predikat) koja proverava da li dati čvor zadovoljava uslov:

```
typedef int (*predikat)(cvor *);  /* vraća nenultu vrednost ako uslov važi */

cvor *nadji_prvi_po_uslovu(cvor *sentinel, predikat uslov)
{
    for (cvor *p = sentinel->sledeci; p != NULL; p = p->sledeci)
        if (uslov(p))
            return p;

    return NULL;
}
```

Na primer, ako želimo da pronađemo prvi element sa parnom vrednošću:

```
int je_paran(cvor *c)
{
    return (c->podatak % 2 == 0);
}

/* ... */

cvor *prvi_paran = nadji_prvi_po_uslovu(sentinel, je_paran);
```

Sledeći kod ilustruje upotrebu svih navedenih funkcija (... označava prethodno definisane funkcije):

```
## include <stdio.h>
## include <stdlib.h>

/* ... */

int main()
{
  int i;
  cvor *l = inicijalizuj_listu();
  for (i = 0; i < 20; i++)  {
    if (i % 2 != 0) {
      if (dodaj_cvor(l, i) != OK) {
        obrisi_listu(l);
        return -1;
      }
    } else {
      if (dodaj_na_kraj(l, i) != OK) {
        obrisi_listu(l);
        return -1;
      }
    }
  }

  ispisi_elemente_liste(l);
  unisti_listu(&l);

  return 0;
}
```

```
19 17 15 13 11 9 7 5 3 1 0 2 4 6 8 10 12 14 16 18
```

U povezanu listu moguće je umetnuti element na bilo koje mesto, ali ta operacija nije složenosti **O(1)** ako je poznat samo početak liste. Ukoliko je poznat pokazivač na element iza kojeg treba umetnuti novi element, ta operacija može da se izvrši u vremenu **O(1)**. Na primer:

```
void ubaci_iza_elementa(cvor *element, cvor *novi)
{
```

```
  novi->sledeci = element->sledeci;
  element->sledeci = novi;
}
```

SLIKA

Za ubacivanje elementa *ispred* datog, u $\mathbf{O(1)}$, može da se primeni trik: novi se ubacuje iza datog, a zatim im se zamene podaci:

```
void ubaci_ispred_elementa(cvor *element, cvor *novi)
{
  novi->sledeci = element->sledeci;
  element->sledeci = novi;
  swap(&(element->podatak), &(novi->podatak));
}
```

Sa ovom funkcijom treba biti obazriv jer joj ne možemo proslediti sentinel-čvor koji i nije "pravi" čvor liste. Naime, ako bi funkciji prosledili pokazivač na sentinel čvor onda bi nakon zamene, ubačeni čvor posmatrali kao sentinel, a sam sentinel bi postao prvi "pravi" čvor liste.

Za brisanje elementa u $\mathbf{O(1)}$ potrebno je imati pokazivač na prethodni element. Ako treba obrisati poslednji element, čak i poznajući adresu poslednjeg čvora, mora se obići lista da bi se našao prethodni čvor, što je $\mathbf{O(n)}$ — zbog čega se takva operacija obično izbegava.

Dodajmo da je moguće umetnuti element na pravo mesto u sortiranoj listi (pod pretpostavkom da je definisan poredak nad podacima). Rekurzivna verzija izgleda ovako:

```
static void ubaci_sortirano_rek(cvor *prev, cvor *curr, cvor *novi)
{
    if (curr == NULL || novi->podatak <= curr->podatak) {
        novi->sledeci = curr;
        prev->sledeci = novi;
        return;
    }
    ubaci_sortirano_rek(curr, curr->sledeci, novi);
}

void ubaci_sortirano(cvor *sentinel, cvor *novi)
{
    ubaci_sortirano_rek(sentinel, sentinel->sledeci, novi);
}
```

Rekurziju je poželjno izbeći, pa iterativna verzija glasi:

```
void ubaci_sortirano(cvor *sentinel, cvor *novi)
{
    cvor *prev = sentinel;
    cvor *curr = sentinel->sledeci;

    while (curr != NULL && curr->podatak < novi->podatak) {
```

```
        prev = curr;
        curr = curr->sledeci;
    }

    novi->sledeci = curr;
    prev->sledeci = novi;
}
```

Ovaj pristup se može iskoristiti za sortiranje liste algoritmom *insertion sort*, ali za bolje performanse treba koristiti algoritme poput *merge sort*. Dve sortirane liste mogu se jednostavno objediniti (merge) u treću, sortiranu listu:

```
static cvor *ucesljaj_rek(cvor *a, cvor *b)
{
    if (a == NULL) return b;
    if (b == NULL) return a;

    if (a->podatak <= b->podatak) {
        a->sledeci = ucesljaj_rek(a->sledeci, b);
        return a;
    } else {
        b->sledeci = ucesljaj_rek(a, b->sledeci);
        return b;
    }
}
void ucesljaj(cvor *s1, cvor *s2)
{
    s1->sledeci = ucesljaj_rek(s1->sledeci, s2->sledeci);
    s2->sledeci = NULL;
}
```

Ista funkcionalnost može i iterativno:

```
void ucesljaj(cvor *s1, cvor *s2)
{
    cvor *p = s1;
    cvor *a = s1->sledeci;
    cvor *b = s2->sledeci;

    while (a != NULL && b != NULL) {
        if (a->podatak <= b->podatak) {
            p->sledeci = a;
            a = a->sledeci;
        } else {
            p->sledeci = b;
            b = b->sledeci;
        }
        p = p->sledeci;
    }
```

```
    p->sledeci = (a != NULL) ? a : b;

    s2->sledeci = NULL;
}
```

Opisano učešljavanje omogućava efikasno sortiranje liste algoritmom *merge sort*.

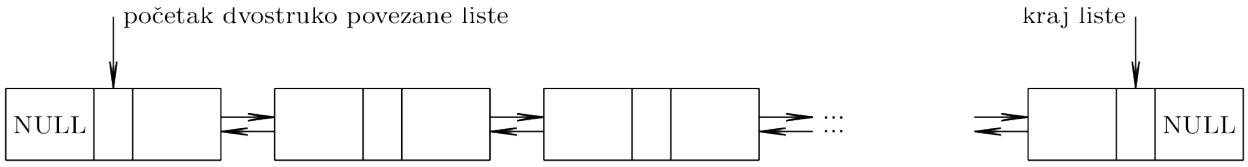
3.2 Dvostruko povezane liste

U jednostruko povezanoj listi svaki element poseduje samo jedan pokazivač – pokazivač na naredni element. To omogućava jednostavno kretanje u jednom smeru, ali je povratno kretanje nepraktično: za pristup prethodnom elementu neophodno je ponovo proći kroz čitavu listu. Dvostruko povezana lista uvodi dodatnu vezu: svaki čvor sadrži pokazivač na svog prethodnika i na svog sledbenika, čime se obezbeđuje efikasno kretanje u oba smera.

Na primer, ako lista čuva karaktere, čvor se može definisati ovako:

```
typedef struct cvor {
    char c;
    struct cvor *prethodni;
    struct cvor *sledeci;
} cvor;
```

Zahvaljujući pokazivačima u oba smera, i umetanje i brisanje čvorova na proizvoljnom mestu u listi mogu se obaviti u vremenu *O(1)*, pod uslovom da je poznat pokazivač na čvor ispred ili iza koga se operacija obavlja. Isto važi za operacije na početku i na kraju liste.



Dvostruko povezana lista

U implementacijama dvostruko povezanih lista često se prethodnik prvog i sledbenik poslednjeg čvora postavljaju na `NULL`. Takav pristup zahteva dodatnu pažnju pri obradi graničnih slučajeva. Mnogo elegantnije rešenje dobija se uvođenjem **sentinel-čvora**, čija se vrednost ne koristi, ali olakšava manipulaciju listom. Sentinel istovremeno predstavlja i logički početak i logički kraj liste: njegov pokazivač `sledeci` pokazuje na prvi element, a `prethodni` na poslednji; u slučaju prazne liste oba pokazivača upućuju na sam sentinel. Na taj način održava se kružna dvostruko povezana lista, ali se sentinel ne posmatra kao stvarni element strukture.

Prisustvo sentinela čini implementaciju značajno jednostavnijom. Svi slučajevi umetanja i brisanja postaju uniformni, bez potrebe za proverama tipa „da li je prethodni element jednak `NULL`. Ovo je idiomatičan pristup i vertikalno se poklapa sa načinom na koji se dvostruko povezana lista često implementira i u standardnoj biblioteci C++.

U nastavku je prikazana osnovna implementacija dvostruko povezane liste sa sentinel-čvorom. Sadržaj zaglavlja:

```
## ifndef DVOSTRUKO_POVEZANA_LISTA_H
## define DVOSTRUKO_POVEZANA_LISTA_H

typedef struct cvor {
    char c;
    struct cvor *prethodni;
    struct cvor *sledeci;
} cvor;

cvor *inicijalizuj_listu(void);
cvor *napravi_cvor(char c, cvor *pre, cvor *sli);

void obrisi_cvor(cvor *cv);

void ubaci_prethodni(cvor *cv, char c);
void ubaci_sledeci(cvor *cv, char c);

void ubaci_na_pocetak(cvor *sentinel, char c);
void ubaci_na_kraj(cvor *sentinel, char c);

int prazna(cvor *sentinel);

void ispisi_listu(cvor *sentinel);
void obrisi_listu(cvor *sentinel);

## endif
```

Prateći izvorni fajl sadrži implementaciju svih navedenih operacija:

```
## include <stdio.h>
## include <stdlib.h>
## include "dvostruko_povezana_lista.h"

cvor *inicijalizuj_listu(void)
{
    cvor *s = malloc(sizeof(cvor));
    if (s == NULL)
        return NULL;

    /* Prazna lista: sentinel pokazuje na sebe */
    s->prethodni = s;
    s->sledeci   = s;
    return s;
}

cvor *napravi_cvor(char c, cvor *pre, cvor *sli)
{
    cvor *novi = malloc(sizeof(cvor));
```

```
    if (novi == NULL)
        return NULL;

    novi->c         = c;
    novi->prethodni = pre;
    novi->sledeci   = sli;
    return novi;
}

void obrisi_cvor(cvor *cv)
{
    cv->prethodni->sledeci = cv->sledeci;
    cv->sledeci->prethodni = cv->prethodni;
    free(cv);
}

void ubaci_prethodni(cvor *cv, char c)
{
    cvor *novi = napravi_cvor(c, cv->prethodni, cv);
    if (novi == NULL)
        return;

    cv->prethodni->sledeci = novi;
    cv->prethodni          = novi;
}

void ubaci_sledeci(cvor *cv, char c)
{
    cvor *novi = napravi_cvor(c, cv, cv->sledeci);
    if (novi == NULL)
        return;

    cv->sledeci->prethodni = novi;
    cv->sledeci            = novi;
}

void ubaci_na_pocetak(cvor *sentinel, char c)
{
    ubaci_sledeci(sentinel, c);
}

void ubaci_na_kraj(cvor *sentinel, char c)
{
    ubaci_prethodni(sentinel, c);
}

int prazna(cvor *sentinel)
{
```

```
    return sentinel->sledeci == sentinel;
}

void ispisi_listu(cvor *sentinel)
{
    for (cvor *p = sentinel->sledeci; p != sentinel; p = p->sledeci)
        putchar(p->c);
    putchar('\n');
}

void obrisi_listu(cvor *sentinel)
{
    while (!prazna(sentinel))
        obrisi_cvor(sentinel->sledeci);
}
```

Ova implementacija obuhvata sve osnovne operacije nad dvostruko povezanom listom: umetanje na početak i kraj liste, umetanje neposredno ispred ili iza određenog čvora, brisanje čvorova i prolazak kroz listu. U svakoj od operacija sentinel ima ključnu ulogu, jer eliminiše specijalne slučajeve i omogućava da sve funkcije budu kratke, čitljive i vremenski efikasne.

3.2.1 Dvostruko povezana lista u standardnoj biblioteci C++: `std::list`

Standardna biblioteka C++ obezbeđuje kolekciju `std::list<T>`, koja predstavlja *dvostruko povezanu listu*. Za njeno korišćenje potrebno je uključiti zaglavlje `<list>`.

Za razliku od `std::forward_list<T>`, koja podržava samo kretanje unapred, `std::list<T>` omogućava:

- kretanje iteratora u oba smera (`++it` i `--it`);
- umetanje i brisanje na proizvoljnoj poziciji u **O(1)**, pod uslovom da je poznat validan iterator na tu poziciju;
- stabilan pokazivač/iterator: element „ne menja adresu“ pri drugim ubacivanjima ili brisanjima (za razliku od `std::vector`).

Implementacija `std::list<T>` koristi **sentinel-čvor** i unutrašnje kružno ulančavanje, slično pristupu koji smo opisali ranije, ali detalji implementacije ostaju sakriveni od korisnika.

Najvažnije operacije prikazane su u tabeli:

Operacija	Opis	Vremenska složeno
`begin()` / `end()`	iteracija od početka do kraja liste	`O(1)`
`push_front(x)` / `push_back(x)`	umetanje na početak/kraj liste	`O(1)`
`pop_front()` / `pop_back()`	uklanjanje sa početka/kraja	`O(1)`
`insert(it, x)`	umetanje pre pozicije `it`	`O(1)`
`erase(it)`	uklanjanje elementa na poziciji `it`	`O(1)`
`splice()`	premeštanje elemenata između dve liste *bez kopiranja*	`O(1)`

Iterator `std::list<T>` je **dvosmerni iterator** i podržava operatore `++` i `--`, što omogućava efikasno kretanje u oba smera.

> **Napomena:** `std::list<T>` ne obezbeđuje slučajni pristup elementima. Operacije koje zahtevaju pozicioniranje po indeksu (`operator[]`, pristup *i*-tom elementu) izostaju jer bi njihova vremenska složenost bila linearna i neodgovarajuća za ovu strukturu.

Kada je poželjna česta manipulacija elementima liste (ubacivanje, brisanje, spajanje podlisti), `std::list<T>` predstavlja idiomatsko i efikasno rešenje u C++.

3.2.2 Primer primene dvostruko povezane liste: linijski editor

Kao ilustraciju praktične upotrebe dvostruko povezane liste razmotrimo jednostavan *linijski editor*. U tipičnim editorima teksta kursor može da se pomera ulevo i udesno, a korisnik može da umeće karaktere na trenutnu poziciju kursora, kao i da briše karaktere levo (taster *backspace*) ili desno (*delete*) od kursora. Kada bi se linija teksta čuvala kao statički niz karaktera, umetanje jednog karaktera u sredinu zahtevalo bi pomeranje svih elemenata koji slede — operacija linearne složenosti. To je neefikasno i nepraktično. Zbog toga se prirodnim rešenjem pokazuje upotreba povezane liste.

Pošto kretanje kursora mora biti moguće u oba smera, lista mora biti dvostruko povezana. Iako je kursor u realnom editoru pozicioniran *između* dva karaktera, u implementaciji je predstavljen pokazivačem na čvor koji se nalazi neposredno **desno** od stvarne pozicije kursora. Ako kursor pokazuje na sentinelu, smatra se da je na *kraju* linije. Ako pokazuje na čvor koji se neposredno nalazi posle sentinela, kursor je na *početku* linije.

Operacije kursora u okviru ove strukture izgledaju ovako:

- Kursor se može pomeriti ulevo ako mu prethodi neki „pravi“ čvor (dakle, ne sentinela).
- Kursor se može pomeriti udesno ako on sam nije sentinela.
- Umetanje karaktera obavlja se ubacivanjem elementa *pre trenutnog kursora*; kursor se ne pomera.
- Brisanje karaktera levo od kursora vrši se samo ako postoji čvor pre kursora; briše se prethodni čvor.
- Brisanje karaktera desno od kursora briše čvor na koji kursor ukazuje, a zatim se kursor pomera nadesno.

Radi jednostavnosti, pretpostavićemo da su sve akcije korisnika zapisane u jednoj niskoj. U pravoj implementaciji editora ove radnje bi se izvršavale kao odgovor na događaje iz okruženja (pritisci tastera, komande, događaji miša i sl.).

Skup podržanih komandi:

Komanda	Opis
`iX`	umetanje karaktera `X`
`<`	pomeranje kursora ulevo
`>`	pomeranje kursora udesno
`b`	brisanje karaktera levo od kursora
`d`	brisanje karaktera desno od kursora

Na primer, niz komandi

```
iAiB<biC>>
```

predstavlja sledeće radnje:

1. unos `A`
2. unos `B`
3. pomeranje kursora ulevo
4. brisanje karaktera `A` (*backspace*)
5. unos karaktera `C`
6. pomeranje kursora udesno dva puta

Stanje linije nakon svake komande može se prikazati ovako (kursor je označen znakom `|`):

Komanda	Objašnjenje	Stanje
`iA`	umetanje `A`	`A |`
`iB`	umetanje `B`	`AB |`
`<`	pomeranje ulevo	`A | B`
`b`	brisanje levo	`| B`
`iC`	umetanje `C`	`C | B`
`>`	pomeranje desno	`CB |`

Na kraju, sadržaj linije je `CB`.

Sledi kompletna implementacija:

```c
## include <stdio.h>
## include "dvostruko_povezana_lista.h"

void obradi_akcije(char akcije[])
{
    /* Linija teksta čuva se u dvostruko povezanoj listi. */

    /* Sentinela predstavlja poziciju "izvan" početka/kraja liste. */
    cvor *sentinela = inicijalizuj_listu();

    /* Kursor: pokazuje na čvor desno od stvarne pozicije kursora. */
    cvor *kursor = sentinela;

    int i = 0;
    while (akcije[i] != '\0') {
        char a = akcije[i++];

        if (a == '<') {
            /* pomeranje ulevo */
            if (kursor->prethodni != sentinela)
                kursor = kursor->prethodni;

        } else if (a == '>') {
```

```
            /* pomeranje udesno */
            if (kursor != sentinela)
                kursor = kursor->sledeci;

        } else if (a == 'i') {
            /* umetanje karaktera */
            char c = akcije[i++];
            ubaci_prethodni(kursor, c);

        } else if (a == 'b') {
            /* brisanje levo od kursora */
            if (kursor->prethodni != sentinela)
                obrisi_cvor(kursor->prethodni);

        } else if (a == 'd') {
            /* brisanje desno od kursora */
            if (kursor != sentinela) {
                cvor *sl = kursor->sledeci;
                obrisi_cvor(kursor);
                kursor = sl;
            }
        }
    }

    ispisi_listu(sentinela);
    obrisi_listu(sentinela);
}

int main(void)
{
    char akcije[] = "iAiB<biC>>";
    obradi_akcije(akcije);
    return 0;
}
```

Ovaj jednostavan primer lepo ilustruje prednosti dvostruko povezane liste u situacijama gde je potrebno često umetanje i brisanje elemenata iz sredine strukture, kao i dvosmerno kretanje kroz podatke. Upravo zbog takvih operacija mnogi realni sistemi za uređivanje teksta oslanjaju se na neku varijantu povezane liste kao internu strukturu.

3.2.3 Dvostruko povezana lista u jeziku C++

U jeziku C++ dvostruko povezana lista može se realizovati kao generička klasa, pri čemu se posebna pažnja posvećuje upravljanju memorijom i pravilnom modelovanju vlasništva nad čvorovima. U ovom pristupu koristi se *veštački čvor* (sentinela), koji pojednostavljuje sve operacije nad listom i eliminiše poseban tretman za umetanje i brisanje na početku ili na kraju liste.

Klasa `lista<T>` sastoji se od tri osnovna dela: definicije čvora, upravljačkih polja liste i interfejsa za rad sa elementima.

Struktura čvora

```
template <typename T>
class lista {
private:
    struct cvor {
        T vrednost;
        std::unique_ptr<cvor> sledeci;
        cvor* prethodni;

        cvor() : vrednost(), sledeci(nullptr), prethodni(nullptr) {}
        explicit cvor(const T& v)
            : vrednost(v), sledeci(nullptr), prethodni(nullptr) {}
    };
```

Čvor čuva vrednost tipa `T` i to je korisnički podatak. Tip `T` je parametar šablona, pa lista može da čuva bilo koji tip — od prostih tipova (`int`, `double`) do složenih korisničkih klasa. Dodatno, čvor čuva i dve veze:

- `std::unique_ptr<cvor> sledeci` predstavlja jedinstveno vlasništvo nad narednim čvorom; oslobađanje memorije je automatsko i lokalizovano,
- `cvor* prethodni` je nevlasnički pokazivač koji omogućava kretanje u suprotnom smeru bez rizika od stvaranja ciklusa vlasništva.

Ovakva kombinacija omogućava pravilno upravljanje memorijom uz očuvanje semantike dvostruko povezane liste.

Pri projektovanju dvostruko povezane liste, prirodno se postavlja pitanje zašto se za prethodnika ne koristi pametni pokazivač (`std::unique_ptr` ili `std::shared_ptr`), već običan pokazivač tipa `cvor*`.

Polje

```
std::unique_ptr<cvor> sledeci;
```

modeluje **jedinstveno vlasništvo**: svaki čvor poseduje tačno jedan sledeći čvor. Ovo dobro odgovara strukturi liste, jer je svaki element „odgovoran“ za oslobađanje sopstvenog nastavka. Pri uklanjanju čvora iz liste, njegov pametni pokazivač se uništava, čime se automatski oslobađa i memorija svih narednih čvorova koji su za njega vezani.

Uvođenje jedinstvenog vlasništva i za prethodni čvor dovelo bi do kontradikcije: jedan objekat ne može imati dva nezavisna vlasnika. Na primer, sledeća definicija je konceptualno pogrešna:

```
struct cvor {
    T vrednost;
    std::unique_ptr<cvor> sledeci;
    std::unique_ptr<cvor> prethodni; // problematično
};
```

U ovakvom modelu pokušava se da svaki čvor istovremeno poseduje i svog prethodnika i svog sledbenika. To je nespojivo sa pojmom jedinstvenog vlasništva: isti objekat bi morao istovremeno da bude u vlasništvu dva pametna pokazivača tipa `std::unique_ptr`, što je nedozvoljeno.

Zbog toga se veza unazad realizuje običnim (nevlasničkim) pokazivačem:

```
cvor* prethodni;
```

Ovaj pokazivač ne učestvuje u oslobađanju memorije i služi isključivo za navigaciju.

Dodatno, na prvi pogled, čini se da bi `std::shared_ptr<cvor>` mogao da reši problem deljenog posedovanja čvora od strane prethodnika i sledbenika. Međutim, takav pristup dovodi do klasičnog problema cikličnih referenci. Razmotrimo sledeću definiciju:

```
struct cvor {
    T vrednost;
    std::shared_ptr<cvor> sledeci;
    std::shared_ptr<cvor> prethodni;
};
```

Ako postoje makar dva čvora međusobno povezana i unapred i unazad, formira se ciklus zajedničkog vlasništva: svaki čvor uvećava brojač referenci onog drugog. Čak i nakon što lista više nije dostupna iz programa, brojači referenci nikada neće dostići nulu, jer se čvorovi međusobno „drže u životu“. Posledica je curenje memorije, iako se sami pametni pokazivači formalno koriste.

Iz tog razloga, `std::shared_ptr` nije pogodan za implementaciju osnovnih struktura podataka koje prirodno formiraju cikluse (liste, grafovi), osim ako se dodatno ne uvode složeniji obrasci (npr. kombinacija `shared_ptr` i `weak_ptr`), što u kontekstu udžbeničke implementacije nije cilj.

Struktura liste

Lista sadrži sledeća polja:

```
    cvor        m_sentinel;
    cvor*       m_poslednji;
    std::size_t m_broj_elemenata;
```

Čvor `m_sentinel` predstavlja veštački čvor koji ne učestvuje u logičkom sadržaju liste. Njegova svrha je da:

- `m_sentinel.sledeci` ukazuje na prvi stvarni element ili na sentinela ako je lista prazna,
- `m_sentinel.prethodni` ukazuje na poslednji stvarni element.

Pokazivač `m_poslednji` omogućava dodavanje na kraj u konstantnom vremenu, dok `m_broj_elemenata` obezbeđuje trenutni uvid u veličinu liste.

Konstrukcija, destrukcija i pravila kopiranja

```
public:
    lista()
        : m_sentinel(), m_poslednji(nullptr), m_broj_elemenata(0)
    {
        m_sentinel.sledeci   = nullptr;
        m_sentinel.prethodni = nullptr;
    }

    ~lista() {
        ocisti();
    }
```

Podrazumevani konstruktor postavlja listu u prazno stanje. Sentinelu nisu pridružene korisničke vrednosti i njegova polja se inicijalizuju na nulu. Destruktor poziva funkciju za oslobađanje svih elemenata:

```
void ocisti() {
    while (m_sentinel.sledeci) {
        m_sentinel.sledeci =
            std::move(m_sentinel.sledeci->sledeci);
    }
    m_poslednji      = nullptr;
    m_broj_elemenata = 0;
}
```

Funkcija uklanja sve čvorove tako što u svakoj iteraciji „odseca“ prvi stvarni čvor liste. Budući da je veza unapred modelovana pametnim pokazivačem `std::unique_ptr<cvor>`, prvi čvor se nalazi u vlasništvu polja `m_sentinel.sledeci`.

U svakoj iteraciji izvodi se operacija:

```
m_sentinel.sledeci = std::move(m_sentinel.sledeci->sledeci);
```

Ovom naredbom vlasništvo nad drugim čvorom u listi prenosi se u `m_sentinel.sledeci`. Prvi čvor time gubi svog vlasnika (njegov `unique_ptr` se zamenjuje drugim), pa se, u skladu sa pravilima rada pametnih pokazivača, njegov destruktor automatski poziva i memorija se oslobađa.

Na taj način lista se u svakoj iteraciji skraćuje za jedan čvor, bez eksplicitnog poziva `delete`, jer `unique_ptr` u potpunosti upravlja oslobađanjem memorije.

Kopirajuće operacije su zabranjene jer lista poseduje `unique_ptr` u čvorovima, čime se sprečava kopiranje strukture na načine koji bi narušili pravila vlasništva. Kako se na ovom mestu fokus stavlja na osnovne strukture podataka i rad sa memorijom, semantika pomeranja (move semantika) nije uvedena kako bi izlaganje ostalo pregledno.

```
    lista(const lista&)            = delete;
    lista& operator=(const lista&) = delete;
```

Iterator dvostruko povezane liste

Pri obilasku liste u jeziku C++ često se koristi konstrukcija oblika `for(p = p->sledeci)`, zasnovana na direktnom radu sa pokazivačima. Međutim, u implementaciji liste koja počiva na pametnim pokazivačima tipa `std::unique_ptr`, takav pristup nije pogodan. Direktna manipulacija pokazivačima može nehotice narušiti strukturu liste, naročito u situacijama kada je pokazivač vlasničkog tipa (poput `unique_ptr`) deo njene unutrašnje reprezentacije. Na primer, svako pomeranje pokazivača kojima se prenosi vlasništvo (`std::move`) može dovesti do nenamerne dezintegracije liste. Na primer, sledeći fragment:

```
for (auto p = m_sentinel.sledeci.get(); p != nullptr; p = p->sledeci.get()) {
    …
}
```

funkcioniše korektno, ali ako bi se (greškom ili nepažnjom) pisalo nešto tipa:

```
m_sentinel.sledeci = std::move(m_sentinel.sledeci->sledeci);
```

u petlji (umesto samo čitanja), cela lista bi se postepeno „uklanjala“ jer se prenosi vlasništvo i brišu čvorovi. Dakle, *sirovi pokazivač („raw pointer“) omogućava pristup internim čvorovima, ali ne obezbeđuje nikakvu zaštitu*: pogrešna operacija može trajno izmeniti listu.

Standardni način rešavanja ovog problema u C++ jeste definisanje iteratora. Iterator obezbeđuje mehanizam za sekvencijalni prolazak kroz strukturu podataka, uz očuvanje njene interne konzistentnosti. Pri tome, korisniku se omogućava idiomatski način korišćenja u duhu STL biblioteke, uključujući konstrukcije oblika:

```
for (auto it = lista.begin(); it != lista.end(); ++it)
    ...
```

što povećava čitljivost, ali i obezbeđuje kontrolisano kretanje kroz listu, bez izlaganja unutrašnjih pokazivača.

U okviru implementacije dvostruko povezane liste iterator se uvodi kao ugnežđena klasa. Njegova struktura definisana je na sledeći način:

```
class iterator {
    cvor*  m_trenutni;
    lista* m_vlasnik;

    iterator(cvor* c, lista* l) : m_trenutni(c), m_vlasnik(l) {}

    friend class lista;
};
```

Iterator čuva dva podatka: pokazivač na tekući čvor (`m_trenutni`) i pokazivač na listu kojoj pripada (`m_vlasnik`). Prvi podatak predstavlja položaj iteratora unutar liste, dok drugi obezbeđuje da se iterator može povezati sa odgovarajućim kontekstom liste, što je važno radi usklađenog ponašanja u operacijama poput poređenja i pomeranja.

Konstruktor iteratora je privatan, a klasa liste je proglašena prijateljskom (`friend`). Ovim se garantuje da se iteratori mogu konstruisati i inicijalizovati isključivo putem operacija koje lista eksplicitno pruža (`begin()`, `end()`), čime se sprečava stvaranje iteratora koji pokazuje na nepostojeće ili tuđe strukture. Takva kontrola konstrukcije predstavlja sastavni deo mehanizma enkapsulacije i osigurava da iteratori uvek odražavaju validno stanje unutar liste.

Polje `m_trenutni` nije tipa `unique_ptr<cvor>`, već običan pokazivač, jer iterator ne poseduje čvor, već ga samo referiše. Upotreba vlasničkih pametnih pokazivača u iteratoru bila bi konceptualno pogrešna: iterator bi time sticao vlasništvo nad delom liste, što bi narušilo jedinstvenost vlasničkog odnosa i dovelo do neispravnog oslobađanja memorije. Naprotiv, vlasništvo ostaje u strukturi liste, dok iterator služi samo kao „pokazivač“, bez prava upravljanja resursom.

Uloga iteratora je, dakle, dvostruka: (1) obezbeđivanje bezbednog pristupa elementima liste kroz interfejs koji ne omogućava nenamerne izmene unutrašnje strukture i (2) omogućavanje upotrebe standardnih programskih obrazaca karakterističnih za C++ kolekcije. Ovakva konstrukcija približava korisnički definisanu listu ponašanju standardnih tipova iz STL-a i čini je pogodnom za opštu upotrebu u različitim programskim kontekstima.

Iterator se ponaša kao apstraktni pokazivač na element liste, omogućavajući prolazak kroz njene čvorove u oba smera. U nastavku su predstavljene osnovne operacije koje iterator omogućava. Iterator obezbeđuje pristup vrednosti skladištenoj u čvoru putem operatora:

```
T& operator*() const { return m_trenutni->vrednost; }
T* operator->() const { return &m_trenutni->vrednost; }
```

Operator $*$ vraća referencu na vrednost u tekućem čvoru, dok operator -> omogućava pristup njenim članovima. Ovi operatori modeluju ponašanje pokazivača, ali pri tome ne izlažu unutrašnju strukturu liste, već pružaju kontrolisan i bezbedan interfejs.

Iterator se može pomeriti na sledeći čvor prefiksnim ili postfiksnim operatorom inkrementa:

```
// prefiksni
iterator& operator++() {
    if (m_trenutni) {
        m_trenutni = m_trenutni->sledeci.get();
    }
    return *this;
}

// postfiksni
iterator operator++(int) {
    iterator tmp(*this);
    ++(*this);
    return tmp;
}
```

Prefiksni operator (`++it`) predstavlja osnovnu operaciju pomeranja na sledeći čvor. Budući da je sledeći čvor povezan pametnim pokazivačem `std::unique_ptr`, iterator preuzima samo sirovi pokazivač putem `get()`, bez narušavanja vlasništva nad memorijom.

Postfiksni operator (`it++`) zadržava prethodno stanje iteratora i vraća njegovu kopiju, dok se original pomera na sledeći čvor. Na ovaj način poštuje se standardna semantika iteratorskih operacija u C++-u.

Za razliku od jednostruko povezanih lista, dvostruko povezana lista omogućava pomeranje iteratora unazad. Operator dekrementa definisan je na sledeći način:

```
iterator& operator--() {
    if (!m_vlasnik) return *this;

    if (m_trenutni == nullptr) {
        // --end() -> prelazak na poslednji element
        m_trenutni = m_vlasnik->m_poslednji;
    } else {
        m_trenutni = m_trenutni->prethodni;
        if (m_trenutni == &(m_vlasnik->m_sentinel)) {
            // --begin() je nedefinisano
            m_trenutni = nullptr;
        }
    }
    return *this;
}

iterator operator--(int) {
    iterator tmp(*this);
    --(*this);
    return tmp;
}
```

Ključni položaji pri izvršavanju:

1. Kada iterator pokazuje na krajnju poziciju (`end()`), operacija `--end()` dovodi do pozicioniranja na poslednji stvarni element liste.
2. U opštem slučaju iterator prelazi na čvor obeležen kao prethodni.
3. Ukoliko bi dekrementacija dovela do izlaska iz opsega („pre početka liste“), rezultat je nedefinisan, u skladu sa pravilima STL iteratora.

Ovakva implementacija omogućava da iterator funkcioniše kao **bidirekcioni iterator**, što odgovara svojstvima dvostruko povezane liste.

Poređenje je zasnovano isključivo na položaju iteratora u listi:

```
bool operator==(const iterator& drugi) const {
    return m_trenutni == drugi.m_trenutni;
}

bool operator!=(const iterator& drugi) const {
    return !(*this == drugi);
}
```

Dva iteratora su jednaka ukoliko ukazuju na isti čvor, što je standardni kriterijum za sve iteratore u C++-u. Ovo omogućava korišćenje iteratora u petljama, algoritmima i generičkom programiranju.

Standardni način obilaska kolekcija u jeziku C++ zasniva se na pojmu *opsega iteracije*, koji se predstavlja parom iteratora: početnim (`begin()`) i završnim (`end()`). U nastavku je prikazano kako se ovi iteratori definišu u okviru dvostruko povezane liste.

```
iterator begin() {
    if (prazna())
        return end();
    return iterator(m_sentinel.sledeci.get(), this);
}
```

Lista koristi **sentinel-čvor** koji ne predstavlja stvarni element kolekcije. Stoga se prvi validni element uvek nalazi u čvoru na koji ukazuje `m_sentinel.sledeci`. Logika metode `begin()` je sledeća:

1. **Ako je lista prazna**, ne postoji početni element; u skladu sa pravilima STL-a, u tom slučaju se kao početak iteracije vraća iterator jednak `end()`.
2. **Ako lista nije prazna**, početni iterator pokazuje na prvi stvarni čvor: $ begin() = iterator(m_sentinel.sledeci.get(), this) $ gde `get()` daje sirovi pokazivač na čvor, dok drugi argument omogućava da iterator zadrži vezu sa listom kojoj pripada.

Na ovaj način `begin()` uvek predstavlja validnu ulaznu tačku za prolazak kroz elemente liste.

Kao i u standardnoj biblioteci C++, završni iterator ne ukazuje ni na jedan stvarni element kolekcije. On predstavlja *poziciju jedan iza poslednjeg elementa* i služi kao granična tačka za zaustavljanje algoritama koji koriste iteratore.

```
iterator end() {
    return iterator(nullptr, this);
}
```

Korišćenjem vrednosti `nullptr` kao položaja iteratora postiže se sledeće:

- završni iterator je lako prepoznatljiv (`it == end()`),
- postaje moguće pravilno definisati operaciju dekrementacije `--end()`, koja se vraća na poslednji element liste,
- iteracija ima formu potpuno analogne STL kontejnerima:

```
for (auto it = lista.begin(); it != lista.end(); ++it) {
    // obrada elementa
}
```

Metode `begin()` i `end()` omogućavaju da se lista koristi u svim generičkim algoritmima (npr. `std::find`, `std::for_each`) i da se uklopi u standardni koncept opsega (*range*). One obezbeđuju da iteracija preko liste bude:

- bezbedna (ne otkriva interne pokazivače strukture),
- konzistentna sa ponašanjem drugih kontejnera u C++ standardnoj biblioteci,
- formalno ispravna čak i u prisustvu sentinela.

Operacije u radu sa dvostruko povezanom listom

Umetanje novog elementa na početak dvostruko povezane liste realizuje se metodom `push_front`, koja umeće čvor neposredno iza sentinela.

```
    void push_front(const T& vrednost) {
        auto novi = std::make_unique<cvor>(vrednost);
        // novi ide odmah iza sentinela
        novi->prethodni = &m_sentinel;
        novi->sledeci   = std::move(m_sentinel.sledeci);

        if (novi->sledeci) {
            novi->sledeci->prethodni = novi.get();
        } else {
            // lista je bila prazna, novi je i poslednji
            m_poslednji = novi.get();
        }

        m_sentinel.sledeci = std::move(novi);
        ++m_broj_elemenata;
    }
```

Ovakav pristup eliminiše potrebu za posebnim tretiranjem prazne liste i omogućava da se umetanje obavlja uvek na isti način. Novi čvor se najpre kreira dinamički, pri čemu `std::unique_ptr` preuzima vlasništvo nad alociranom memorijom. Pošto čvor poseduje pokazivač ka čvoru koji sledi, prenošenje vlasništva pomoću `std::move` omogućava da se stari prvi čvor (ako postoji) automatski priključi iza novog. Zatim se eksplicitno postavljaju pokazivači: novi čvor se vezuje za sentinela kao svog prethodnika, dok njegov sledbenik postaje raniji prvi čvor. Ukoliko lista nije bila prazna, taj raniji prvi čvor mora da dobije novog prethodnika; ukoliko jeste, novi čvor postaje i poslednji čvor u listi. Nakon podešavanja pokazivača, `unique_ptr` u sentinelu preuzima vlasništvo nad umetenim čvorom, čime novi čvor postaje prvi stvarni element liste. Broj elemenata se na kraju uvećava za jedan.

Ovakva konstrukcija obezbeđuje da umetanje bude operacija vremenske složenosti ($O(1)$), uz potpuno automatsko i bezbedno upravljanje memorijom: čvorovi se ne oslobađaju eksplicitno, već njihovo oslobađanje proizilazi iz semantike vlasništva pametnih pokazivača. Pri tome, logička struktura liste ostaje jasna: sentinel zauzima stabilno prvo mesto, dok se čvorovi umeću isključivo iza njega, uz minimalne promene u topologiji liste i bez ikakvih posebnih slučajeva.

Umetanje elementa na kraj liste realizuje se metodom `push_back`, pri čemu se novi čvor dodaje neposredno iza trenutno poslednjeg čvora.

```
    void push_back(const T& vrednost) {
        auto novi = std::make_unique<cvor>(vrednost);
        novi->sledeci = nullptr;

        if (prazna()) {
            // isto kao push_front na praznoj listi
            novi->prethodni      = &m_sentinel;
            m_sentinel.sledeci   = std::move(novi);
```

```
            m_poslednji          = m_sentinel.sledeci.get();
        } else {
            novi->prethodni      = m_poslednji;
            m_poslednji->sledeci = std::move(novi);
            m_poslednji          = m_poslednji->sledeci.get();
        }

        ++m_broj_elemenata;
    }
```

Ključnu ulogu u ovoj proceduri ima pokazivač `m_poslednji`, koji omogućava da se umetanje na kraj izvrši u vremenu (O(1)), bez potrebe za obilaskom cele liste. Novi čvor se najpre kreira i ostavlja bez sledbenika, jer je po definiciji poslednji u lancu. Postupak zatim zavisi od toga da li je lista prazna: ukoliko nema nijedan stvarni element, umetanje je ekvivalentno umetanju na početak — novi čvor se vezuje za sentinela kao za prethodnika, a sentinel preuzima vlasništvo nad njim putem `std::unique_ptr`. U tom slučaju, novi čvor postaje istovremeno i prvi i poslednji u listi.

Ako lista nije prazna, novi čvor se vezuje za prethodni kraj: pokazivač `prethodni` postavlja se na čvor na koji ukazuje `m_poslednji`, dok se njegovo polje `sledeci` postavlja tako što `std::unique_ptr` u poslednjem čvoru preuzima vlasništvo nad novokreiranim čvorom. Nakon toga se pokazivač `m_poslednji` ažurira tako da ukazuje upravo na novi čvor.

Operacija je konstantne vremenske složenosti i ne zahteva obilazak prethodnih čvorova.

Operacija umetanja i brisanja na osnovu iteratora predstavlja najopštiji i najsnažniji način manipulacije dvostruko povezanom listom. U nastavku se pojašnjava logika funkcija `insert` i `erase`:

```
iterator insert(iterator pozicija, const T& vrednost) {
        if (pozicija == end()) {
            push_back(vrednost);
            return iterator(m_poslednji, this);
        }

        cvor* tekuci    = pozicija.m_trenutni;
        cvor* prethodni = tekuci->prethodni; // može biti &m_sentinel

        auto novi = std::make_unique<cvor>(vrednost);
        novi->prethodni = prethodni;

        std::unique_ptr<cvor>* veza;
        if (prethodni == &m_sentinel) {
            veza = &m_sentinel.sledeci;
        } else {
            veza = &prethodni->sledeci;
        }

        novi->sledeci = std::move(*veza);   // stari "sledeći" ide iza novog
        if (novi->sledeci) {
            novi->sledeci->prethodni = novi.get();
        } else {
```

```
            // umetnuli smo na kraj
            m_poslednji = novi.get();
        }

        *veza = std::move(novi);
        ++m_broj_elemenata;

        return iterator(veza->get(), this);
    }

    iterator erase(iterator pozicija) {
        cvor* tekuci = pozicija.m_trenutni;
        if (!tekuci) return end();

        cvor* prethodni = tekuci->prethodni; // može biti &m_sentinel
        cvor* sledeci   = tekuci->sledeci.get();

        std::unique_ptr<cvor>* veza;
        if (prethodni == &m_sentinel) {
            veza = &m_sentinel.sledeci;
        } else {
            veza = &prethodni->sledeci;
        }

        std::unique_ptr<cvor> za_brisanje = std::move(*veza);
        *veza = std::move(za_brisanje->sledeci);

        if (*veza) {
            (*veza)->prethodni = prethodni;
        } else {
            // obrisali smo poslednji element
            if (prethodni == &m_sentinel) {
                m_poslednji = nullptr;
            } else {
                m_poslednji = prethodni;
            }
        }

        --m_broj_elemenata;

        return iterator(sledeci, this);
    }
```

Operacija `insert` umeće novi element neposredno ispred pozicije na koju iterator `pozicija` ukazuje. Poseban slučaj nastaje kada je `pozicija == end()`: tada se radi o umetanju na kraj liste i zadatak se svodi na poziv `push_back`, nakon čega se novi iterator vraća tako da ukazuje na poslednji čvor. U opštem slučaju, iterator `pozicija` ukazuje na čvor `tekuci`, a novi element se umeće pre njega. Potrebno je, stoga, identifikovati njegovog prethodnika, koji se dobija kao `tekuci->prethodni`; taj čvor može biti i sam sentinelski čvor, ako se umetanje vrši na početak liste.

Nakon toga se alocira novi čvor i postavlja mu se pokazivač `prethodni` na već određeni prethodni čvor. Ključni tehnički detalj u ovoj funkciji predstavlja promenljiva `veza` tipa `std::unique_ptr<cvor>*`. Ona ne pokazuje na čvor, već na pokazivač koji „drži“ taj čvor, tj. na vlasnički `unique_ptr` koji povezuje prethodni čvor sa tekućim. Ako je prethodni čvor sentinela, odgovarajuća veza je `&m_sentinel.sledeci`; u suprotnom, veza je `&prethodni->sledeci`. Time se na jedinstven način zadaje mesto na koje treba „ukačiti“ novi čvor, bez dupliranja koda za slučaj početka i unutrašnjosti liste.

Novi čvor tada „preuzima“ dotadašnju vezu: njegovo polje `sledeci` dobija vlasništvo nad starim vrednostima preko `std::move(*veza)`, čime bivši sledbenik prethodnog čvora postaje sledbenik novog. Ako novi čvor zaista ima sledbenika (lista se ne završava tu), tom sledbeniku se ažurira polje `prethodni` tako da ukazuje na novi čvor; u suprotnom, novoumetnuti čvor postaje poslednji i promenljiva `m_poslednji` se ažurira odgovarajuće. Konačno, `*veza = std::move(novi)` uspostavlja vlasničku vezu od prethodnog čvora (ili sentinela) ka novom čvoru, povećava se broj elemenata, a kao rezultat se vraća iterator koji ukazuje na novoumetnuti čvor. Na taj način, umetanje preko iteratora ostaje lokalna operacija vremenske složenosti ($O(1)$), uz pravilno održavanje svih veza unutar liste.

Operacija `erase` uklanja čvor na koji iterator `pozicija` ukazuje i vraća iterator na element koji sledi za obrisanim (ili `end()` ako takav element ne postoji). Ako iterator ne ukazuje ni na jedan čvor (`m_trenutni == nullptr`), briše se „prazna“ pozicija i vraća se `end()`. U suprotnom, čvor koji se uklanja označava se kao `tekuci`. Njegov prethodni čvor dobija se kao `tekuci->prethodni` (što opet može biti sentinela), a sledeći čvor kao `tekuci->sledeci.get()`.

Kao i kod umetanja, uvodi se promenljiva `veza` tipa `std::unique_ptr<cvor>*`, koja opet ne predstavlja sam čvor, već pokazivač na vlasničku vezu koja vodi do njega. Ako je prethodni čvor sentinela, `veza` ukazuje na `m_sentinel.sledeci`; u suprotnom, na `prethodni->sledeci`. Na taj način, `*veza` je upravo `unique_ptr` koji poseduje čvor koji se briše. Čvor se izdvaja iz strukture pomoću:

```
std::unique_ptr<cvor> za_brisanje = std::move(*veza);
*veza = std::move(za_brisanje->sledeci);
```

čime se vlasništvo nad sledećim čvorom prenosi u `*veza`, a čvor označen kao `za_brisanje` ostaje izolovan: više nije u ulančanoj strukturi, ali je još uvek u vlasništvu lokalne promenljive. Kada ova promenljiva izađe iz opsega, njen destruktor oslobađa memoriju obrisanog čvora. Ako novi `*veza` nije prazan, treba ažurirati polje `prethodni` tog čvora tako da ukazuje na prethodni čvor u lancu. Ako, međutim, `*veza` postane `nullptr`, znači da je obrisan poslednji element, pa se promenljiva `m_poslednji` podešava na `nullptr` (ako sada nema nijednog čvora osim sentinela) ili na odgovarajući prethodni čvor.

Broj elemenata liste se smanjuje, a kao rezultat funkcije vraća se iterator koji ukazuje na čvor `sledeci`, tj. na element koji sledi za obrisanim (ili na `end()` ukoliko je brisanjem uklonjen poslednji element). Na taj način, operacija brisanja takođe ostaje lokalna, vremenske složenosti ($O(1)$), uz potpunu korektnost u pogledu vlasništva i oslobađanja memorije.

Primer korišćenja opisane klase

U nastavku se daje jednostavan primer korišćenja generičke dvostruko povezane liste, koji demonstrira umetanje, brisanje i upotrebu iteratora, uključujući i kretanje unazad.

```
## include <iostream>
## include "lista.hpp"

int main()
{
    using dl::lista;

    lista<int> l;

    // ------------------------------
    // Dodavanje elemenata
    // ------------------------------
    l.push_back(10);
    l.push_back(20);
    l.push_front(5);   // lista: 5 10 20

    std::cout << "Pocetna lista: ";
    for (auto it = l.begin(); it != l.end(); ++it)
        std::cout << *it << " ";
    std::cout << "\n";

    // ------------------------------
    // Umetanje elementa
    // ------------------------------
    auto it = l.begin();
    ++it;  // pozicija iza prvog elementa (iza 5)

    l.insert(it, 7);   // lista: 5 7 10 20

    std::cout << "Nakon umetanja 7: ";
    for (auto x : l)   // podržano zahvaljujući iteratoru
        std::cout << x << " ";
    std::cout << "\n";

    // ------------------------------
    // Brisanje elementa
    // ------------------------------
    auto it2 = l.begin();
    ++it2; // pokazuje na 7

    l.erase(it2);   // brise 7 -> lista: 5 10 20

    std::cout << "Nakon brisanja 7: ";
    for (auto x : l)
        std::cout << x << " ";
    std::cout << "\n";

    // ------------------------------
```

```
    // Demonstracija kretanja unazad (--it)
    // ------------------------------
    auto kraj = l.end();
    --kraj;  // sada ukazuje na poslednji element (20)

    std::cout << "Kretanje unazad: ";
    std::cout << *kraj;  // 20

    --kraj;
    std::cout << " " << *kraj;  // 10

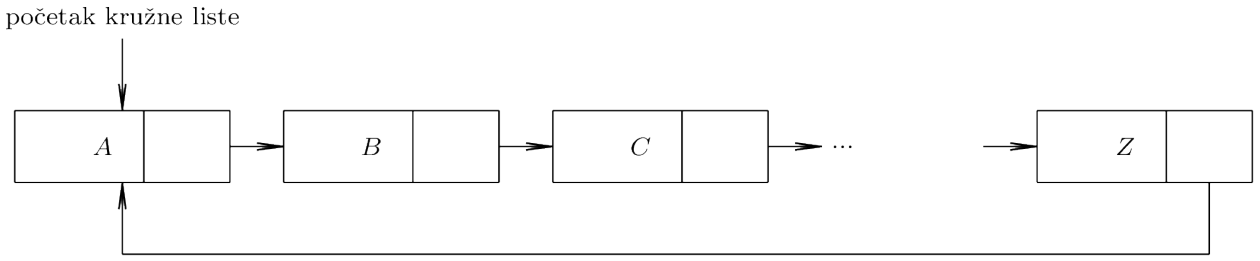
    --kraj;
    std::cout << " " << *kraj;  // 5
    std::cout << "\n";

    return 0;
}
```

3.3 Kružne liste

Često se javlja potreba da se kroz određene podatke prolazi „u krug“, tj. da se nakon obrade poslednjeg podatka ponovo vratimo na prvi. Ako su podaci smešteni u niz, nakon što indeks dostigne dužinu niza možemo ga ponovo postaviti na nulu (ili uvećavanje indeksa vršiti po modulu dužine niza). Međutim, ukoliko se tokom takvog kružnog obilaska javlja potreba da se elementi umeću ili uklanjaju na tekućoj poziciji, niz postaje neprikladna struktura podataka, jer umetanje i brisanje iz sredine niza zahteva pomeranje velikog broja elemenata.

Prirodna struktura podataka za ovakve situacije je **kružna (ciklična, cirkularna) lista**. U kružnoj listi poslednji element ne pokazuje na `NULL`, već na prvi element liste. Na taj način se do bilo kog elementa može doći polazeći od bilo kog drugog elementa, jednostavnim praćenjem pokazivača. U tom smislu, izbor „prvog“ i „poslednjeg“ elementa u kružnoj listi je relativan.



Kružna lista

Na slici ilustrovana je jednostruko povezana kružna lista. Obilazak takve liste moguć je samo u jednom smeru, ali je sasvim analogno moguće definisati i dvostruko povezanu kružnu listu, koja omogućava kretanje u oba smera.

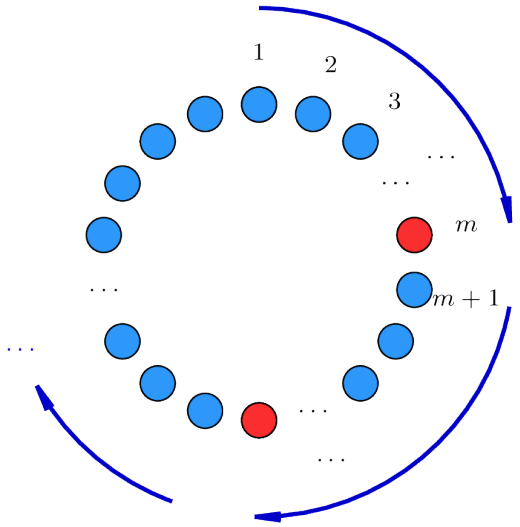
U odnosu na implementacije sa sentinel-čvorovima, kod kružne liste dovoljno je održavati pokazivač na neki „tekući“ element — pošto poslednji element pokazuje na prvi, prirodna „cikličnost“ strukture

već je ugrađena. Upravo zato se u ovom delu ne uvodi sentinel-čvor: ideja kružne liste je dovoljno transparentna i bez dodatnog fiktivnog elementa.

3.3.1 Primer primene strukture kružna lista: plesači

Korišćenje jednostruko povezane kružne liste možemo ilustrovati na zadatku u kome se prirodno javlja potreba za prolaskom kroz elemente neke kolekcije „u krug“, pri čemu se tokom obilaska pojedini elementi uklanjaju sa tekuće pozicije (ništa suštinski ne bi bilo drugačije ni da se povremeno umeću novi elementi na tekuću poziciju).

Grupa od (n) plesača (na čijim kostimima su, u smeru kazaljke na satu, redom ispisani brojevi od (1) do (n)) izvodi plesnu tačku tako što formiraju krug iz koga najpre izlazi (m)-ti plesač (odbrojava se počev od plesača označenog brojem (1), u smeru kazaljke na satu). Preostali plesači ponovo obrazuju krug iz koga opet izlazi (m)-ti plesač (odbrojava se počev od prvog sledećeg za prethodno izbačenog, ponovo u smeru kazaljke na satu). Izlasci iz kruga nastavljaju se sve dok svi plesači ne napuste krug.



Slika 3.1: Ilustracija algoritma izbacivanja plesača.

Zadatak je napisati program koji, za zadate vrednosti (n) i (m), određuje redni broj poslednjeg plesača koji ostaje u krugu. Na primer, za ulaz ($n = 5$) i ($m = 3$), redosled izlazaka je:

```
3 1 5 2 4
```

pa je poslednji plesač onaj sa brojem 4.

Suštinski, ovde se radi o poznatom problemu eliminacije elemenata u ciklusu (poznat i kao Josefusov problem). Kružna lista je prirodan model za ovakvu eliminaciju „u krug“.

Struktura čvora jednostruko povezane liste (koju sada posmatramo kao kružnu) može biti definisana na sledeći način:

```
typedef struct cvor {
    int x;
    struct cvor *sledeci;
```

```
} cvor;
```

Za potrebe ovog zadatka definisaćemo nekoliko pomoćnih funkcija. One su namerno napisane pod pretpostavkom da se koriste nad **nepraznom listom**, kako bi implementacija ostala jednostavna i fokusirana na konkretnu primenu.

Prva funkcija kreira novi čvor sa datom vrednošću i zadatim pokazivačem na sledeći element:

```
cvor *napravi_cvor(int x, cvor *sledeci)
{
    cvor *novi = malloc(sizeof(cvor));
    if (novi == NULL)
        return NULL;

    novi->x       = x;
    novi->sledeci = sledeci;
    return novi;
}
```

Zatim ćemo definisati funkciju za dodavanje novog čvora na kraj **neprazne** liste, pri čemu je poznat pokazivač na trenutni kraj. Funkcija vraća pokazivač na novoubaceni čvor, koji postaje novi kraj liste:

```
cvor *dodaj_na_kraj(cvor *kraj, int x)
{
    assert(kraj != NULL);

    kraj->sledeci = napravi_cvor(x, NULL);
    return kraj->sledeci;
}
```

Za operaciju uklanjanja elemenata koristićemo sledeći trik: umesto da brišemo čvor na koji ukazuje pokazivač `tekuci`, prepisujemo vrednost iz njegovog sledbenika u njega samog, a zatim izbacujemo čvor sledbenika. Posmatrano spolja, efekat je isti kao da je obrisan „tekući“ čvor. Funkcija vraća pokazivač na čvor koji ostaje na toj poziciji:

```
cvor *obrisi_tekuci(cvor *tekuci)
{
    assert(tekuci != NULL);
    assert(tekuci->sledeci != NULL);

    cvor *tmp        = tekuci->sledeci;
    tekuci->x        = tmp->x;
    tekuci->sledeci  = tmp->sledeci;
    free(tmp);

    return tekuci;
}
```

Napomena: u kontekstu kružne liste koja u svakom trenutku ima bar dva čvora, pokazivač `tekuci->sledeci` nikada neće biti `NULL`. Kada u krugu preostane samo jedan čvor, ovaj zadatak se završava i funkcija `obrisi_tekuci` se više ne poziva.

Sa ovako definisanim funkcijama možemo jednostavno formirati kružnu listu sa elementima (1, 2, ..., n), a zatim u svakom od (n-1) koraka:

1. od tekućeg čvora „odbrojati“ (m-1) korak u smeru kazaljke na satu,
2. obrisati čvor na koji se došlo,
3. nastaviti sa sledećeg čvora.

Implementacija funkcije koja vraća preostalog plesača (njegov broj) može izgledati ovako:

```
int preostali_plesac(int n, int m)
{
    cvor *pocetak, *kraj, *tekuci;

    /* formiramo listu sa elementima 1, 2, ..., n */
    pocetak = napravi_cvor(1, NULL);
    if (pocetak == NULL)
        return -1;

    kraj = pocetak;

    for (int i = 2; i <= n; i++) {
        kraj = dodaj_na_kraj(kraj, i);
        if (kraj == NULL)
            return -1;
    }

    /* spajamo kraj i početak: niz postaje kružna lista */
    kraj->sledeci = pocetak;

    /* brojanje kreće od prvog čvora */
    tekuci = pocetak;

    /* dok ne preostane jedan plesač u krugu */
    for (int k = n; k > 1; k--) {
        /* pomeramo se na m-ti čvor od tekućeg */
        for (int i = 0; i < m - 1; i++)
            tekuci = tekuci->sledeci;

        /* brišemo „tekući" čvor i nastavljamo sa sledećeg */
        tekuci = obrisi_tekuci(tekuci);
    }

    /* preostali čvor sadrži broj poslednjeg plesača */
    int rezultat = tekuci->x;
    free(tekuci);
```

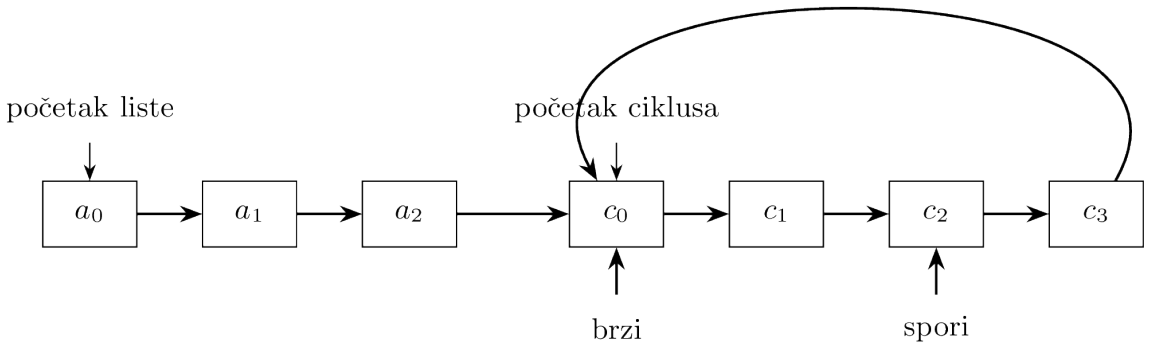
```
    return rezultat;
}
```

Funkcija `preostali_plesac` vraća broj plesača koji ostaje poslednji u krugu. Korišćenjem kružne liste i odgovarajućih pomoćnih funkcija, algoritam je prirodno i pregledno izveden, a kružni karakter problema potpuno je usklađen sa strukturom podataka.

3.3.2 Detekcija ciklusa u jednostruko povezanoj listi (Flojdov algoritam)

U prethodnim odeljcima posmatrali smo kružne liste kao strukturu u kojoj su svi čvorovi deo jednog ciklusa. U praksi se često javlja i sledeći problem: imamo „običnu“ jednostruko povezanu listu u kojoj se greškom (ili namerno) javlja **ciklus** – neki čvor pokazuje na neki prethodni čvor u listi, pa se deo liste ponaša kao kružna struktura.

U praksi se problem detekcije ciklusa u jednostruko povezanim listama javlja znatno češće nego što bi se očekivalo na prvi pogled. Dinamičke strukture podataka zasnovane na pokazivačima (poput lista procesa u operativnim sistemima, redova događaja u serverima ili struktura keš-objekata) mogu usled greške u kodu slučajno formirati ciklus, što obično dovodi do beskonačnog obilaska ili curenja memorije. Sličan problem javlja se i u analizatorima memorije, interpreterima i sistemima za upravljanje memorijom, gde je potrebno efikasno proveriti da li referencirane strukture sadrže zatvorenu putanju. Detekcija ciklusa značajna je i u oblastima kao što su pseudorandom generatori, dinamički sistemi i kriptografski algoritmi, gde iteracije funkcija mogu ući u periodičnost. Zbog svoje jednostavnosti, linearnog vremena i konstantne potrošnje memorije, **Flojdov algoritam** (“kornjača i zec”) postao je standardno rešenje za ovaj problem u širokom spektru praktičnih primena.



U trenutku susreta važi $S = a + kb$,
a položaj unutar ciklusa je $S \equiv a \pmod{b}$

Jednostruko povezana lista sa ciklusom

Tipičan scenario izgleda ovako:

- početni deo liste je linearan (`NULL` se ne dostiže),
- od neke tačke nadalje, pokazivači formiraju ciklus (kružni deo liste).

Zadatak je:

1. najpre utvrditi da li lista sadrži ciklus,
2. a zatim odrediti **prvi čvor ciklusa**, tj. čvor od koga počinje kružni deo.

Na prvi pogled, problem deluje složen: ne znamo ni dužinu „linearnog“ dela, ni dužinu ciklusa, niti gde tačno počinje ciklus. Ipak, postoji vrlo elegantan algoritam složenosti **O(n)** po vremenu i **O(1)** po memoriji, poznat kao Flojdov algoritam „kornjača i zec“ (*Floyd's Tortoise and Hare*).

Posmatrajmo dva pokazivača koji kreću od početka liste:

- „spori“ pokazivač (kornjača) pomera se za **jedan** čvor po koraku,
- „brzi“ pokazivač (zec) pomera se za **dva** čvora po koraku.

Ako lista **nema ciklus**, brzi pokazivač će kad-tad doći do `NULL` i algoritam može da zaključi da ciklus ne postoji.

Ako lista **ima ciklus**, intuitivno je jasno da će se brzi pokazivač u nekom trenutku „zatvoriti“ sa sporim — kao što se automobil koji se kreće većom brzinom po kružnoj stazi kad-tad nađe u istoj tački kao i sporiji automobil.

Teorema 3.3.1. *Neka je data jednostruko povezana lista u kojoj postoji ciklus. Neka su:*

- $a \geq 0$ — *broj čvorova od početka liste do prvog čvora ciklusa,*
- $b \geq 1$ — *dužina ciklusa (broj čvorova u kružnom delu).*

Pokrenimo dva pokazivača iz početka liste:

- *spori pokazivač koji se u svakom koraku pomera za* 1 *čvor,*
- *brzi pokazivač koji se u svakom koraku pomera za* 2 *čvora.*

Tada važi:

1. *Postoji najmanji prirodan broj* t *za koji će se dva pokazivača naći u istom čvoru unutar ciklusa.*
2. *Ako nakon tog susreta jedan pokazivač vratimo na početak liste, a zatim oba pokazivača pomeramo po jedan čvor u svakom koraku, tada će se oni sresti upravo u čvoru koji predstavlja* ***početak ciklusa****.*

Dokaz. Dokaz prvog dela. Nakon t koraka:

- spori pokazivač je prešao t čvorova,
- brzi pokazivač je prešao $2t$ čvorova.

Za $t \geq a$ oba pokazivača nalaze se u ciklusu. Razlika u njihovim pređenim putanjama jednaka je:

$$(2t - t) = t.$$

Pošto se pokazivači kreću unutar ciklusa dužine b, njihova relativna pozicija posmatra se *modularno* u odnosu na b.

Tražimo najmanje t takvo da:

$$t \equiv 0 \pmod{b}.$$

Očigledno je da takvo t postoji (na primer $t = b$), pa će dva pokazivača neminovno doći u isti čvor u ciklusu. To dokazuje postojanje susreta.

Dokaz drugog dela. Neka je t_0 trenutak prvog susreta. U tom trenutku spori pokazivač je prešao:

$$t_0 = a + kb,$$

za neki ceo broj $k \geq 0$, jer je nakon prolaska linearnog dela dužine a obio k punih krugova duž ciklusa dužine b. Pozicija sporog pokazivača u ciklusu je stoga:

$$t_0 - a \equiv 0 \pmod{b},$$

što znači da je njegova udaljenost od početka ciklusa jednaka 0 modulo b, odnosno da se nalazi na istoj „fazi“ ciklusa kao i početak, ali ne i nužno u istom čvoru.

Vraćanjem jednog pokazivača na početak liste i pomeranjem oba za po 1 čvor, u svakom koraku dobijamo sledeće:

- pokazivač koji je krenuo sa početka biće posle a koraka tačno u prvom čvoru ciklusa;
- pokazivač koji je krenuo iz tačke susreta, nakon a koraka prelazi put:

$$(t_0 - a) + a \equiv 0 \pmod{b},$$

pa će se takođe naći u tom istom čvoru. Stoga se pokazivači susreću u jedinstvenoj poziciji — čvoru koji predstavlja početak ciklusa. □

U narednoj funkciji je implementirana ova ideja:

```
cvor *pocetak_ciklusa(cvor *glava)
{
    cvor *spori = glava;
    cvor *brzi  = glava;

    /* Faza 1: tražimo susret unutar ciklusa (ako postoji). */
    while (brzi != NULL && brzi->sledeci != NULL) {
        spori = spori->sledeci;           /* pomeraj 1 */
        brzi  = brzi->sledeci->sledeci;   /* pomeraj 2 */

        if (spori == brzi)
            break;
    }

    /* Ako nismo našli susret, nema ciklusa. */
    if (brzi == NULL || brzi->sledeci == NULL)
        return NULL;

    /* Faza 2: jedan pokazivač ide na početak liste. */
```

```
    spori = glava;

    /* Oba se sada pomeraju po jedan čvor; susret je početak ciklusa. */
    while (spori != brzi) {
        spori = spori->sledeci;
        brzi  = brzi->sledeci;
    }

    return spori;
}
```

- U prvoj fazi dva pokazivača kreću sa početka liste: jedan („spori“) ide po jedan čvor, drugi („brzi“) po dva čvora. Ako lista nema ciklus, „brzi“ će naići na `NULL` i funkcija vraća `NULL`. Ako ciklus postoji, pokazivači će se sigurno sresti u nekom čvoru unutar ciklusa.
- U drugoj fazi jedan pokazivač se vraća na početak liste, dok drugi ostaje u tački susreta. Zatim se oba pomeraju **istom brzinom** (po jedan čvor). Matematička analiza pokazuje da će se posle tačno a koraka (gde je a broj elemenata do početka ciklusa) sresti u čvoru koji je upravo **prvi čvor ciklusa**.

Algoritam pri tom prolazi listu najvećim delom samo jednom i koristi konstantnu količinu dodatne memorije, pa je složenost po vremenu $O(n)$, a po memoriji $O(1)$, što ga čini standardnim rešenjem za detekciju ciklusa u jednostruko povezanim listama.

3.4 Stek

Stek (*stack*) je apstraktna struktura podataka koja funkcioniše po principu **LIFO** (*last in, first out*). Intuicija je jednostavna: zamislimo štap na koji se nižu kompakt diskovi. Disk koji je poslednji stavljen na štap biće prvi uklonjen, a do diska koji je na dnu moguće je doći tek kada se prethodno uklone svi diskovi iznad njega.

Stek može da skladišti elemente različitih tipova, a njegova funkcionalnost se zasniva na dve osnovne operacije, obe projektovane da budu vremenske složenosti $O(1)$:

- **push** – dodaje novi element na vrh steka;
- **pop** – uklanja element sa vrha steka.

Tipična implementacija steka namerno izlaže samo ograničen skup operacija, kako bi se korisniku omogućilo da lako zameni jednu implementaciju drugom bez uticaja na ostatak programa. Pored osnovnih operacija `push` i pop, uobičajeno je implementirati i:

- inicijalizaciju steka,
- brisanje i oslobađanje memorije,
- proveru da li je stek prazan.

Važno je naglasiti da je stek zapravo **adapter** nad postojećom strukturom podataka: on ne određuje način skladištenja elemenata, već uvodi strogo LIFO pravilo pristupa nad nizom, povezanim listama ili drugim sekvencijalnim kontejnerima koji omogućavaju efikasno dodavanje i uklanjanje na jednom kraju.

Stek je izuzetno korisna struktura i pojavljuje se u velikom broju algoritamskih i sistemskih primena, od parsiranja izraza, preko rekurzije, do simulacija i algoritama na grafovima.

3.4.1 Implementacija steka korišćenjem niza

Stek se može implementirati na više načina, a najjednostavniji među njima koristi **dinamički niz**. Moguće je koristiti i statički niz, ali takva implementacija nije fleksibilna jer ne dozvoljava promenu kapaciteta tokom rada programa.

Najpraktičnije je uvesti posebnu strukturu podataka koja enkapsulira sve što je potrebno za rad sa stekom. Funkcije za rad sa stekom tada primaju pokazivač na instancu ove strukture, čime se obezbeđuje efikasnost i jasan interfejs.

Primer zaglavlja `stek_niz.h`:

```
#ifndef __STEK_NIZ_H__
#define __STEK_NIZ_H__

typedef enum {
    OK,
    NEDOVOLJNO_MEMORIJE,
    NEMA_ELEMENATA
} status;

typedef struct {
    int *podaci;   /* pokazivač na dinamički niz elemenata */
    int kapacitet;  /* trenutno alocirani kapacitet */
    int vrh;       /* indeks prvog slobodnog mesta (broj elemenata) */
} Stek;

status inicijalizuj_stek(Stek* stek);
status push(Stek* stek, int podatak);
status pop(Stek* stek, int *podatak);
size_t broj_elemenata(const Stek* stek);
int stek_je_prazan(const Stek* stek);
void obrisi_stek(Stek* stek);

#endif

Implementacija u datoteci `stek_niz.c`:

```c
## include <stdlib.h>
## include "stek_niz.h"

const int INICIJALNI_KAPACITET_STEKA = 100;

status inicijalizuj_stek(Stek* stek)
{
    stek->podaci = malloc(INICIJALNI_KAPACITET_STEKA * sizeof(int));
    if (stek->podaci == NULL)
        return NEDOVOLJENO_MEMORIJE;
```
```

```
    stek->vrh = 0;
    stek->kapacitet = INICIJALNI_KAPACITET_STEKA;
    return OK;
}

status push(Stek* stek, int podatak)
{
    /* Ako je stek pun, dupliramo kapacitet */
    if (stek->vrh == stek->kapacitet) {
        int *p = realloc(stek->podaci, 2 * stek->kapacitet * sizeof(int));
        if (p == NULL)
            return NEDOVLJNO_MEMORIJE;

        stek->kapacitet *= 2;
        stek->podaci = p;
    }

    stek->podaci[stek->vrh++] = podatak;
    return OK;
}

status pop(Stek* stek, int *podatak)
{
    if (stek->vrh == 0)
        return NEMA_ELEMENATA;

    *podatak = stek->podaci[--stek->vrh];
    return OK;
}

size_t broj_elemenata(const Stek* stek)
{
    return stek->vrh;
}

int stek_je_prazan(const Stek* stek)
{
    return stek->vrh == 0;
}

void obrisi_stek(Stek* stek)
{
    free(stek->podaci);
    stek->podaci = NULL;
    stek->kapacitet = 0;
    stek->vrh = 0;
}
```

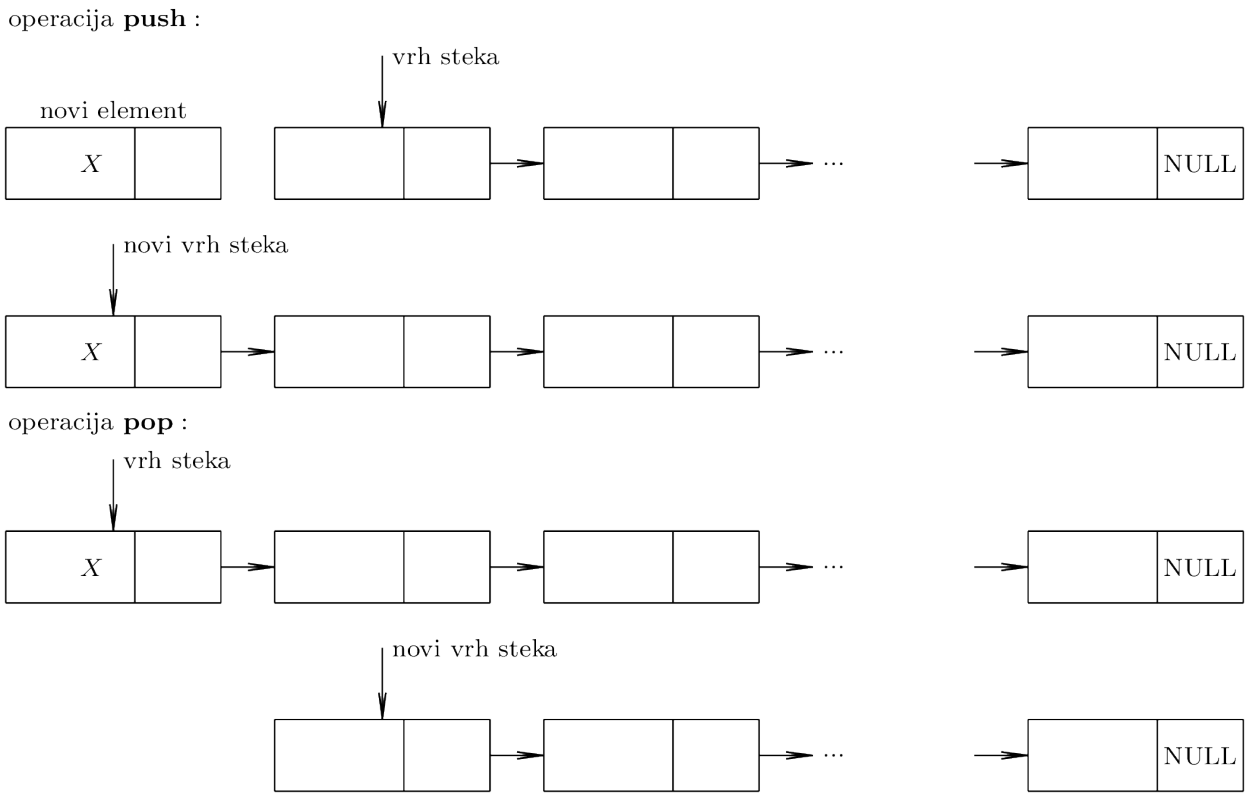
Ovaj pristup daje veoma efikasnu implementaciju: operacije push i pop imaju vremensku složenost

(O(1)) u proseku, dok se realokacija dešava retko i amortizuje se kroz geometrijsko povećavanje kapaciteta.

3.4.2 Implementacija steka korišćenjem povezane liste

Stek se može implementirati i korišćenjem povezane liste. U tom slučaju struktura steka održava pokazivač na njegov **vrh**, odnosno na prvi element liste. Operacije `push` i `pop` prirodno se preslikavaju na dodavanje i uklanjanje čvora sa početka liste, što je upravo mesto gde povezane liste imaju optimalnu efikasnost.

Osnovne operacije nad ovakvim stekom prikazane su na slici **??**.



Ilustracija osnovnih operacija nad stekom implementiranim korišćenjem povezane liste

U implementaciji koja sledi koristi se poseban tip čvora liste, dok se funkcije za rad sa stekom mogu oslanjati na jednostavne pomoćne operacije: kreiranje novog čvora, dodavanje na početak i uklanjanje sa početka liste.

Primer zaglavlja `stek_lista.h`:

```
## ifndef __STEK_LISTA_H__
## define __STEK_LISTA_H__

typedef enum {
    OK,
    NEDOVOLJNO_MEMORIJE,
    NEMA_ELEMENATA
```

```
} status;

typedef struct cvor {
    int podatak;
    struct cvor *sledeci;
} cvor;

typedef struct {
    cvor *vrh;        /* pokazivač na prvi element liste */
    size_t broj_el;   /* ukupan broj elemenata u steku */
} Stek;

status inicijalizuj_stek(Stek* stek);
status push(Stek* stek, int podatak);
status pop(Stek* stek, int *podatak);
size_t broj_elemenata(const Stek* stek);
int stek_je_prazan(const Stek* stek);
void obrisi_stek(Stek* stek);

## endif
```

Implementacija funkcija u datoteci `stek_lista.c`:

```
## include <stdlib.h>
## include "stek_lista.h"

static cvor *novi_cvor(int podatak);
static status dodaj_na_pocetak(cvor **pocetak, int podatak);
static status obrisi_sa_pocetka(cvor **pocetak, int *podatak);

static cvor *novi_cvor(int podatak)
{
    cvor *novi = malloc(sizeof(cvor));
    if (novi == NULL)
        return NULL;

    novi->podatak = podatak;
    novi->sledeci = NULL;
    return novi;
}

static status dodaj_na_pocetak(cvor **pocetak, int podatak)
{
    cvor *novi = novi_cvor(podatak);
    if (novi == NULL)
        return NEDOVOLJNO_MEMORIJE;

    novi->sledeci = *pocetak;
```

```
    *pocetak = novi;
    return OK;
}

static status obrisi_sa_pocetka(cvor **pocetak, int *podatak)
{
    if (*pocetak == NULL)
        return NEMA_ELEMENATA;

    cvor *stari = *pocetak;
    *podatak = stari->podatak;
    *pocetak = stari->sledeci;
    free(stari);
    return OK;
}

status inicijalizuj_stek(Stek* stek)
{
    stek->vrh = NULL;
    stek->broj_el = 0;
    return OK;
}

status push(Stek* stek, int podatak)
{
    status s = dodaj_na_pocetak(&(stek->vrh), podatak);
    if (s == OK)
        stek->broj_el++;
    return s;
}

status pop(Stek* stek, int *podatak)
{
    status s = obrisi_sa_pocetka(&(stek->vrh), podatak);
    if (s == OK)
        stek->broj_el--;
    return s;
}

size_t broj_elemenata(const Stek* stek)
{
    return stek->broj_el;
}

int stek_je_prazan(const Stek* stek)
{
    return stek->vrh == NULL;
}
```

```
void obrisi_stek(Stek* stek)
{
    int x;
    while (pop(stek, &x) == OK)
        ; /* praznimo čvor po čvor */
}
```

U ovoj implementaciji pomoćne funkcije (`novi_cvor`, `dodaj_na_pocetak`, `obrisi_sa_pocetka`) nisu deklarisane u zaglavlju, već se nalaze samo u `stek_lista.c` i označene su kao `static`. Time se jasno naglašava da predstavljaju **interni API** implementacije, koji ne treba izlagati korisniku steka.

Prednost ovakvog rešenja jeste fleksibilnost — čvorovi liste ne moraju da se nalaze u uzastopnim memorijskim lokacijama. Ipak, za razliku od implementacije koja koristi niz, svaka `push` operacija zahteva poziv `malloc`, što značajno usporava rad i čini implementaciju manje efikasnom u većini realnih scenarija.

3.4.3 Generički stek u C++: šablon zasnovan na `std::vector`

U jeziku C generičnost je nezgodna: ili se uvodi pomoćni tip preko `typedef`, ili se koristi tip `void*` i ručno kastovanje. U C++ generičnost je deo jezika – šabloni (`template`) omogućavaju da se isti kod koristi za različite tipove, bez dodatnog troška u izvođenju.

Prirodno je da se generički stek u C++ implementira kao **šablonska klasa** koja interno koristi standardni kontejner, npr. `std::vector`. Vektor već obezbeđuje:

- dinamičko proširivanje memorije,
- efikasne operacije na kraju niza (`push_back`, `pop_back`, `back`),
- pravilno konstruisanje i uništavanje objekata tipa `T`.

Time implementacija steka postaje vrlo jednostavna.

```
## include <vector>
## include <cstddef> // size_t

template <typename T>
class stek {
public:
    using value_type = T;
    using size_type  = std::size_t;

    stek() = default;

    // Da li je stek prazan?
    bool empty() const noexcept {
        return podaci_.empty();
    }

    // Broj elemenata na steku
    size_type size() const noexcept {
```

```
        return podaci_.size();
    }

    // Dodaje element na vrh steka (kopiranjem)
    void push(const T& x) {
        podaci_.push_back(x);
    }

    // Dodaje element na vrh steka (premeštanjem)
    void push(T&& x) {
        podaci_.push_back(std::move(x));
    }

    // Uklanja element sa vrha steka (pretpostavlja se da nije prazan)
    void pop() {
        podaci_.pop_back();
    }

    // Pristup elementu na vrhu steka (može da menja taj element)
    T& top() {
        return podaci_.back();
    }

    // Pristup elementu na vrhu steka (samo za čitanje)
    const T& top() const {
        return podaci_.back();
    }

private:
    std::vector<T> podaci_;
};
```

Ova implementacija koristi `std::vector<T>` kao internu strukturu podataka. Operacije su:

- `push` i `pop` – vremenska složenost (O(1)) u proseku (amortizovano),
- `top` – vremenska složenost (O(1)),
- `size` i `empty` – takođe (O(1)).

Napomena o `push`: postoje dve verzije – jedna prima `const T&` (kopira vrednost), a druga `T&&` (koristi premještanje – *move semantics*). Kod jednostavnih tipova (poput `int`) razlika je zanemarljiva, ali kod složenih tipova (npr. veliki objekti, stringovi, vektori) verzija sa premještanjem omogućava efikasnije dodavanje.

Primer upotrebe:

```
## include <iostream>
## include <string>

int main() {
    stek<int> s1;
```

```
    s1.push(10);
    s1.push(20);

    std::cout << "Vrh steka: " << s1.top() << "\n"; // 20
    s1.pop();
    std::cout << "Novi vrh: " << s1.top() << "\n";   // 10

    stek<std::string> s2;
    s2.push("prvi");
    s2.push("drugi");

    std::cout << "Vrh (string): " << s2.top() << "\n"; // "drugi"
}
```

U odnosu na C-varijante:

- ne koristimo `malloc`, `realloc`, `free`, ni ručno baratanje pokazivačima;
- ne pišemo pomoćne `typedef`-ove za "generički" tip – sama klasa je šablon;
- destruktori i kopirajući/premještajući konstruktori automatski pravilno rukuju resursima.

Drugim rečima, isti kod `stek<T>` radi za `int`, `double`, `std::string`, pa čak i za korisnički definisane tipove, bez dodatnih promena u implementaciji.

3.4.4 Kako je organizovan `std::stack` u STL-u

Standardna biblioteka već sadrži generički stek: `std::stack`. On je definisan kao **adapter nad kontejnerom**:

```
template <class T, class Container = std::deque<T>>
class stack;
```

Dakle:

- `std::stack` ne čuva elemente direktno, već koristi neki drugi kontejner (`Container`),
- podrazumevani kontejner je `std::deque<T>`,
- umesto operacija `push_back`, `pop_back` i `back` kontejnera, izlaže operacije `push`, `pop` i `top`.

Na primer:

```
## include <stack>
## include <vector>

int main() {
    // Podrazumevani stek: koristi std::deque<int> interno
    std::stack<int> s1;
    s1.push(1);
    s1.push(2);

    // Stek koji interno koristi std::vector<int>
```

```
    std::stack<int, std::vector<int>> s2;
    s2.push(3);
    s2.push(4);
}
```

3.4.5 Stek i aritmetički izrazi

Cilj ovog poglavlja je da prikaže kako se stek, kao jednostavna ali izuzetno moćna apstraktna struktura podataka, prirodno primenjuje u obradi aritmetičkih izraza. Računanje vrednosti izraza, njihovo prevođenje u druge oblike zapisa i analiza njihove strukture predstavljaju klasične zadatke koji se javljaju u programiranju, kompajlerima, interpreterskim sistemima, kalkulatorima, analizatorima formula i brojnim softverskim alatima.

Ovi problemi na prvi pogled deluju različito, ali svi dele istu potrebu za uređivanjem i privremenim skladištenjem delova izraza prema pravilima koja odgovaraju režimu LIFO. Upravo tu se stek pokazuje kao idealan adapter. U ovom poglavlju koristićemo generičku C++ implementaciju steka razvijenu ranije i upotrebiti je u tri konteksta: za računanje vrednosti postfiksnih izraza, za prevođenje potpuno zagrađenih infiksnih izraza u postfiksni oblik i za obradu opštih aritmetičkih izraza sa prioritetima i asocijativnošću operatora.

Na ovim primerima jasno se vidi kako jednom definisana apstraktna struktura podataka omogućava rešavanje čitave klase problema na ujednačen i elegantan način.

Vrednost postfiksnog izraza

Jedan od najjednostavnijih i najelegantnijih načina za izračunavanje aritmetičkih izraza zasniva se na upotrebi **postfiksne (obratne poljske) notacije**. Ideja potiče od poljskog logičara Jana Lukašijeviča, koji je pokazao da se aritmetički izrazi mogu zapisati tako da **ne zahtevaju nikakve zagrade**, a da pri tom i dalje ostanu jednoznačni.

U postfiksnoj notaciji binarni operatori se ne zapisuju između operanada (kao u uobičajenom infiksnom zapisu), već **iza njih**. Na primer, izraz

$$3 \cdot ((1 + 2) \cdot 3 + 5)$$

može se zapisati kao:

$$(3\ (((1\ 2\ +)\ 3\ \cdot)\ 5\ +)\ \cdot)$$

Interesantno je da u postfiksnoj notaciji **zagrade uopšte nisu potrebne**, jer se struktura izraza jasno određuje samim redosledom operanada i operatora. Zbog toga se isti izraz može zapisati znatno jednostavnije:

$$3\ 1\ 2\ +\ 3\ \cdot\ 5\ +\ \cdot$$

Ova jednoznačnost je upravo razlog zašto je postfiksni oblik izuzetno pogodan za obradu — ne mora se voditi računa o prioritetu operatora, asocijativnosti ili dubini ugnježđenih zagrada. Sve što je potrebno jeste znati kada je operacija spremna da se primeni: svaki operand se pojavljuje pre svog operatora.

Postfiksni izraz se može izračunati jednostavnim prolaskom kroz zapis **sleva nadesno**, pri čemu se koristi standardni LIFO mehanizam steka:

- ako je pročitan broj $\longrightarrow$ stavlja se na stek,
- ako je pročitan operator $\longrightarrow$ sa steka se skidaju dva broja, operator se primenjuje na njih, a rezultat se vraća na stek.

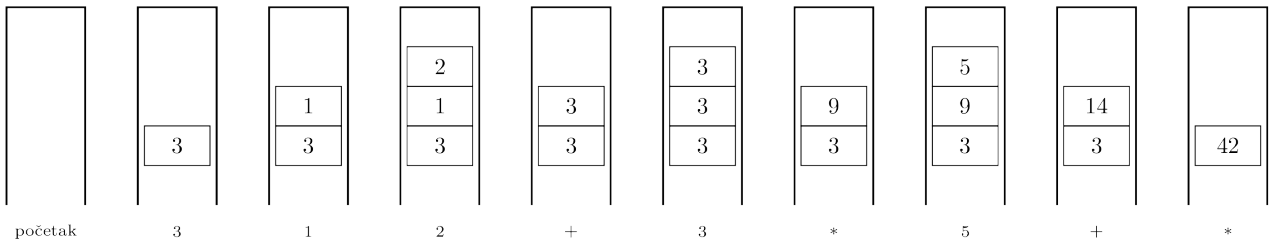
Kada čitav izraz bude obrađen, na steku ostaje tačno jedan element — **vrednost kompletnog izraza**.

Ova procedura neposredno pokazuje snagu steka kao apstraktne strukture podataka: algoritam se oslanja samo na operacije `push`, `pop` i `top`, nezavisno od toga da li je stek realizovan pomoću niza, liste ili standardnog `std::vector<T>` u našoj generičkoj C++ implementaciji. Zbog toga postfiksni oblik predstavlja prirodan i efikasan format za izračunavanje i jedan od ključnih motivacionih primera za upotrebu steka.

Razmotrimo postfiksno zapisan izraz:

$$3\ 1\ 2\ +\ 3\ *\ 5\ +\ *.$$

Na slici **??** prikazano je stanje steka nakon svake obrade jednog simbola: kada se pročita broj, on se stavlja na stek, a kada se pročita operator, sa steka se skidaju dva operanda, primenjuje se operator i rezultat se vraća na stek.



Ilustracija promene steka tokom izracunavanja vrednosti izraza

Pretpostavljamo da je šablonska klasa `stek<T>` već definisana i da je dostupna iz zaglavlja, na primer, `stek.hpp`.

```
## include <string>
## include <sstream>
## include <cctype>
## include "stek.hpp"   // ovde se nalazi šablon class stek<T>

// Proverava da li je dati znak jedan od podržanih operatora
bool je_operator(char c)
{
    return c == '+' || c == '-' || c == '*' || c == '/';
}

// Primena operatora na dva operanda.
// Napomena: pretpostavljamo celobrojno deljenje za '/'.
int primeni_operator(char op, int op1, int op2)
```

```
{
    switch (op) {
    case '+': return op1 + op2;
    case '-': return op1 - op2;
    case '*': return op1 * op2;
    case '/': return op1 / op2;  // oprez, nije obrađeno deljenje nulom
    default:
        // Ovaj slučaj ne bi trebalo da se desi ako
        // izraz sadrži samo dozvoljene operatore.
        return 0;
    }
}

// Računa vrednost postfiksnog izraza zapisanog
// kao niz tokena odvojenih razmacima.
int vrednost_postfiksnog_izraza(const std::string& izraz)
{
    stek<int> st;
    std::istringstream iss(izraz);
    std::string token;

    while (iss >> token) {
        // Ako je token operator dužine 1, tretiramo ga kao operator
        if (token.size() == 1 && je_operator(token[0])) {
            char op = token[0];

            // Skinemo dva operanda sa steka (pretpostavljamo ispravan izraz)
            int op2 = st.top(); st.pop();
            int op1 = st.top(); st.pop();

            int v = primeni_operator(op, op1, op2);
            st.push(v);
        }
        else {
            // U suprotnom, token tretiramo kao ceo broj (može biti više-cifren)
            int vrednost = std::stoi(token);
            st.push(vrednost);
        }
    }

    // Na kraju (ako je izraz ispravan) na steku ostaje tačno jedan element:
    // vrednost kompletnog izraza.
    return st.top();
}
```

Prevođenje potpuno zagrađenog izraza u postfiksni oblik

Sada razmatramo obrnut problem: umesto da polazimo od postfiksnog izraza i računamo njegovu vrednost, polazimo od **infiksnog** izraza (u uobičajenom obliku sa operatorima između operanada) i želimo da ga prevedemo u **postfiksni** oblik.

U ovom delu posmatramo posebno jednostavan, ali značajan slučaj: **potpuno zagrađene** infiksne izraze. Pod tim podrazumevamo da je svaka primena binarnog operatora (+, *, \ ili -) eksplicitno okružena zagradama. Na primer:

- `((3*5)+(7+(2*1)))*4` je potpuno zagrađen izraz,
- `3*5+7` nije potpuno zagrađen (nedostaju zagrade oko `3*5` i oko celog `3*5+7`).

Ovakvi izrazi mogu se opisati sledećom jednostavnom gramatikom:

```
<izraz> ::= <cifra>
          | '(' <izraz> '+' <izraz> ')'
          | '(' <izraz> '*' <izraz> ')'
          | '(' <izraz> '/' <izraz> ')'
          | '(' <izraz> '-' <izraz> ')'
```

Ovde je `<cifra>` jedna decimalna cifra od 0 do 9. Svaki složeniji izraz nastaje kombinovanjem manjih izraza, pri čemu se svaki binarni operator nalazi između dva podizraza i okružen je zagradama.

Potrebno je napisati program koji ispravan infiksni aritmetički izraz, u kome su sve primene binarnih operatora potpuno zagrađene, prevodi u postfiksni oblik. Na primer, za ulaz:

`(((3*5)+(7+(2*1)))*4)`

izlaz treba da bude:

`3 5 * 7 2 1 * + + 4 *`

Ključna ideja je održavanje **steka operatora**:

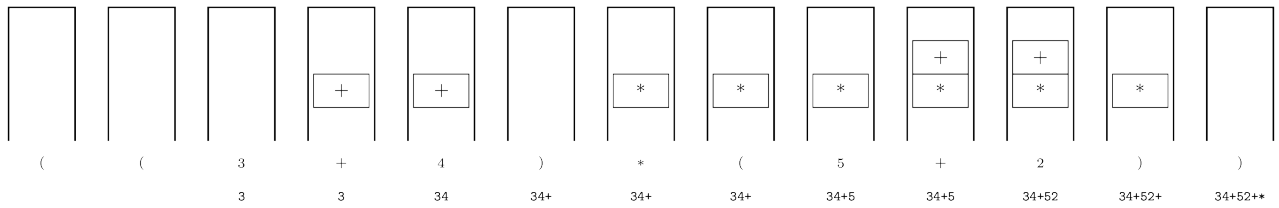
- kada naiđemo na broj (cifru), odmah ga ispisujemo na izlaz (postfiksni izraz);
- kada naiđemo na operator (+ ili *), stavljamo ga na **stek operatora**;
- kada naiđemo na zatvorenu zagradu `)`, znamo da se završila jedna celina oblika `( <izraz> op <izraz> )` pa skidamo operator sa vrha steka i ispisujemo ga.

Otvorene zagrade `(` u ovom posebnom slučaju možemo jednostavno ignorisati: one služe samo da označe strukturu izraza, ali nas ne zanimaju direktno u prevodu, jer je struktura već osigurana pretpostavkom „potpuno zagrađeno“.

Ovako definisan algoritam koristi stek na veoma jednostavan način:

- stek sadrži **samo operatore**,
- svaki operator se na stek stavlja tačno jednom (kada se pročita u infiksnom izrazu),
- svaki operator se sa steka skida tačno jednom (kada naiđe zatvorena zagrada koja završava njegovu „podstrukturu“).

Za ilustraciju, razmotrimo izraz:



Ilustracija sadržaja steka tokom prevođenja potpuno zagrađenog izraza

((3+4)*(5+2))

Rezultat je postfiksni izraz:

34+52+*

Ovo ponašanje jasno pokazuje da je „sporedni kolosek“ za operatore, odnosno stek, upravo struktura koja nam je potrebna da bismo realizovali prevođenje. U narednom koraku možemo na ovu šemu nadograditi i opštiji algoritam u kome nema zahteva za potpuno zagrađivanje i uvodi se prioritet operatora.

Data je implementacija ove ideje, kao i korišćenje definisanih funkcija:

```
string infiks_u_postfiks(const string& izraz)
{
    stek<char> operatori;   // stek operatora
    string postfiks;        // izlaz

    int i = 0;
    int n = (int)izraz.size();

    while (i < n) {
        char c = izraz[i];

        if (isspace((unsigned char)c)) {
            // preskačemo razmake
            i++;
        }
        else if (isdigit((unsigned char)c)) {
            // čitamo ceo broj (može biti više-cifren)
            while (i < n && isdigit((unsigned char)izraz[i])) {
                postfiks += izraz[i];
                i++;
            }
            postfiks += ' ';   // razdvajamo tokene razmakom
        }
        else if (c == '(') {
            // otvorenu zagradu ovde ignorišemo
            i++;
        }
        else if (jeOperator(c)) {
            // operator ide na stek
```

```
            operatori.push(c);
            i++;
        }
        else if (c == ')') {
            // zatvorena zagrada: završava se (E op E)
            char op = operatori.top();
            operatori.pop();
            postfiks += op;
            postfiks += ' ';
            i++;
        }
        else {
            // u okviru zadatka pretpostavljamo ispravan izraz,
            // pa ne obrađujemo dodatne slučajeve
            i++;
        }
    }

    return postfiks;
}

int main()
{
    std::ifstream in("izrazi.txt");
    if (!in) {
        std::cerr << "Greska \n";
        return 1;
    }

    std::string linija;
    while (std::getline(in, linija)) {
        std::string postfiks = infiks_u_postfiks(linija);
        int vrednost = vrednost_postfiksnog_izraza(postfiks);
        std::cout << vrednost << "\n";
    }

    return 0;
}
```

Računanje vrednosti opštih aritmetičkih izraza

U prethodnim odeljcima videli smo kako se stek može iskoristiti za izračunavanje vrednosti postfiksnih izraza, kao i za prevođenje potpuno zagrađenih infiksnih izraza u postfiksni oblik. Sada razmatramo opštiji problem: *izračunavanje vrednosti proizvoljnog infiksnog aritmetičkog izraza** koji može da sadrži brojeve, zagrade i sve standardne binarne operacije +, −, ∗ i / (celobrojno deljenje).

Takvi izrazi mogu se formalno opisati sledećom proširenom BNF gramatikom:

```
<izraz>     ::= <term> { <operacija1> <term> }
<term>      ::= <faktor> { <operacija2> <faktor> }
<faktor>    ::= <broj> | '(' <izraz> ')'
<broj>      ::= <cifra> { <cifra> }
<cifra>     ::= '0' | '1' | … | '9'
<operacija1>::= '+' | '-'
<operacija2>::= '*' | '/'
```

Ova gramatika jasno pokazuje hijerarhiju operatora: * i / imaju viši prioritet od + i −, a zagrade dozvoljavaju lokalnu promenu prioriteta. Zadati izraz je uvek ispravan i ne sadrži razmake.

Da bismo izraz izračunali, najpre ga prevodimo u postfiksni oblik, a zatim vrednost dobijenog postfiksnog izraza izračunavamo pomoću steka operanada, kao i ranije. Međutim, za razliku od potpuno zagrađenih izraza, sada moramo obraćati pažnju na **prioritet operatora** i njihovu **asocijativnost**.

Posmatrajmo izraz oblika:

```
i1 op1 i2 op2 i3
```

gde su `i1`, `i2` i `i3` izrazi (brojevi ili izrazi u zagradama), a `op1` i `op2` operatori. Ključno pitanje je šta raditi kada se pročita `op2`. U tom trenutku:

- postfiksni izlaz već sadrži prevod `i1` i `i2`,
- operator `op1` se nalazi na vrhu steka operatora,
- upravo je pročitan novi operator `op2`.

Pitanje je: *da li treba odmah preuzeti `op1` sa steka i prebaciti ga na izlaz, ili prvo treba „zapamtiti“ `op2` i nastaviti čitanje?*

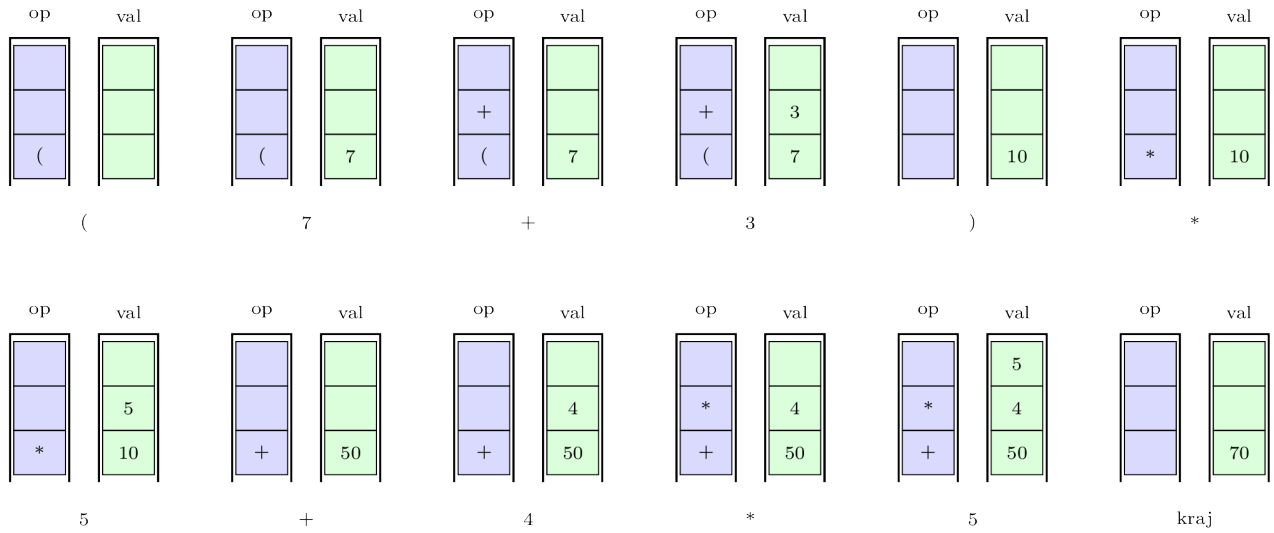
Analizom, dolazi se do sledećeg pravila:

- ako `op1` ima *viši prioritet* od `op2`, onda `op1` prvo ide na izlaz;
- ako `op1` i `op2` imaju *isti prioritet*, a operatori su *levo asocijativni* (što važi za sve aritmetičke operacije u ovom zadatku), tada opet `op1` ide na izlaz;
- u suprotnom, `op2` se stavlja na stek iznad `op1`.

Zagrade se obrađuju posebno:

- otvorena zagrada se jednostavno smešta na stek,
- kada se naiđe na zatvorenu zagradu, operatori se skidaju sa steka i prebacuju na izlaz sve dok se ne naiđe na odgovarajuću otvorenu zagradu.

Ovaj postupak prevođenja infiksnog izraza u postfiksni oblik poznat je kao **Shunting-yard algoritam**, koji je uveo Edsger Dijkstra. Slika procesa često se opisuje železničkom analogijom: izraz „dolazi“ sa desne strane, a operatori se privremeno skladište na sporednom koloseku (steku). Na kolosek se postavlja samo ono što trenutno ne može da pređe direktno na izlaz, a skidanje operatora uvek poštuje pravila prioriteta i asocijativnosti.



Ilustracija promene steka operanada i steka operacija tokom izračunavanja vrednosti izraza

Kada se čitav infiksni izraz pročita, svi preostali operatori sa steka redom se prebacuju na izlaz. Dobijeni postfiksni zapis zatim se izračunava korišćenjem steka operanada, istim algoritmom kao u prethodnom odeljku.

Na ovaj način stek postaje ključna struktura podataka: koristi se najpre za upravljanje operatorima pri prevođenju, a zatim za čuvanje operanada prilikom izračunavanja vrednosti postfiksnog izraza.

Naredni kod ilustruje ovu ideju:

```
// Prioritet operatora
int prioritet(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

// Računanje vrednosti infiksnog izraza sa zagradama
int vrednost_izraza(const std::string& s) {
    stek<int>  vrednosti;
    stek<char> operatori;

    std::size_t i = 0;
    const std::size_t n = s.size();

    while (i < n) {
        char c = s[i];

        if (std::isspace(c)) {
            ++i;
        }
        else if (std::isdigit(c)) {
            // čitanje više-cifrenog broja
```

```
            int broj = 0;
            while (i < n && std::isdigit(s[i])) {
                broj = broj * 10 + (s[i] - '0');
                ++i;
            }
            vrednosti.push(broj);
        }
        else if (c == '(') {
            operatori.push(c);
            ++i;
        }
        else if (c == ')') {
            // primenjuj sve dok se ne naiđe na '('
            while (!operatori.empty() && operatori.top() != '(') {
                char op = operatori.top(); operatori.pop();
                int b = vrednosti.top(); vrednosti.pop();
                int a = vrednosti.top(); vrednosti.pop();
                vrednosti.push(primeni_operator(op, a, b));
            }
            operatori.pop(); // uklanja '('
            ++i;
        }
        else {
            // c je operator +, -, *, /
            while (!operatori.empty() &&
                   operatori.top() != '(' &&
                   prioritet(operatori.top()) >= prioritet(c))
            {
                char op = operatori.top(); operatori.pop();
                int b = vrednosti.top(); vrednosti.pop();
                int a = vrednosti.top(); vrednosti.pop();
                vrednosti.push(primeni_operator(op, a, b));
            }
            operatori.push(c);
            ++i;
        }
    }

    // isprazni ostatak operatora
    while (!operatori.empty()) {
        char op = operatori.top(); operatori.pop();
        int b = vrednosti.top(); vrednosti.pop();
        int a = vrednosti.top(); vrednosti.pop();
        vrednosti.push(primeni(op, a, b));
    }

    return vrednosti.top();
}
```

Ova implementacija koristi standardno rešenje sa **dva steka**:

- *stek operatora* drži `+ - * / (`
- *stek vrednosti* drži sve brojeve
- kada se pročita operator, primenjuju se svi operatori sa vrha steka koji imaju *veći ili isti prioritet*
- kada se pročita `)`, primenjuju se operatori sve dok se ne naiđe na `(`
- kada se pročita broj, on direktno ide na stek vrednosti

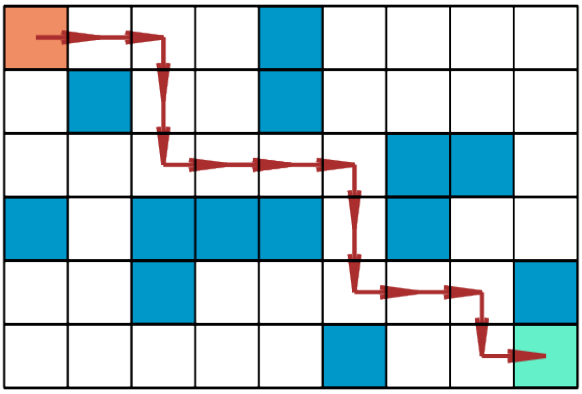
Na kraju, preostali operatori se primene nad preostalim vrednostima.

Ovaj metod je deterministič an, linearan i ne zahteva ponovno parsiranje izraza — baš zato se i koristi u kompajlerima, kalkulatorima i većini evaluatora izraza.

Evo tvoje kompletne sekcije, **prevedene u čist, uredan Markdown**, potpuno spremne za udžbenik. Format prati stil koji si već koristila: jasno, akademski, bez viškova.

3.4.6 Primer primene strukture stek: pretraga u dubinu i ispitivanje da li postoji put na mapi

Razmotrimo problem ispitivanja da li postoji put na mapi. Mapu posmatramo kao pravougaonu mrežu polja. Pojedina polja mogu biti prolazna ili označena kao prepreke. Cilj je utvrditi da li se od početnog polja $(0, 0)$ može stići do ciljnog polja $(m - 1, n - 1)$.



Slika 3.2: Primer mape

U nastavku prikazan pristup koristi *stek* da bi se simulirala **pretraga u dubinu (DFS)** bez rekurzije. Svako polje predstavlja se tipom:

```
typedef struct data {
    int v, k;   /* koordinate polja */
} Data;
```

Ovaj tip se zatim uključuje u zaglavlje generičkog steka kako bi se mogla koristiti implementacija steka čiji su elementi tipa `Data`.

Funkcija `postojiPut`:

1. polazi od polja $(0, 0)$ i stavlja ga na stek,
2. zatim u petlji skida jedno po jedno polje sa steka,
3. ukoliko je izvučeno polje cilj, vraća se vrednost 1,
4. u suprotnom, sva njegova susedna polja koja:
    - nisu van mape,
    - nisu prepreke,
    - nisu već posećena

dodaju se na stek i označavaju kao posećena, 5. ako se stek isprazni, a cilj nije dostignut, put ne postoji.

Susedna polja se određuju pomoću pomoćnog niza `smer`, koji sadrži vektore četiri osnovna pomeraja.

```
## include <stdio.h>
## include "stek_lista.h"

## define MAX_N 50
const int smer[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

int postojiPut(int prepreke[MAX_N][MAX_N], int m, int n)
{
    int poseceno[MAX_N][MAX_N] = {0};
    Stek stek;
    inicijalizuj_stek(&stek);

    poseceno[0][0] = 1;
    Data polje = { 0, 0 };  /* početno polje */
    push(&stek, polje);

    while (!stek_je_prazan(&stek)) {

        pop(&stek, &polje);

        /* da li je ciljno polje? */
        if (polje.v == m - 1 && polje.k == n - 1) {
            obrisi_stek(&stek);
            return 1;
        }

        /* razmatramo susedna polja */
        for (int i = 0; i < 4; i++) {
            int v1 = polje.v + smer[i][0];
            int k1 = polje.k + smer[i][1];

            if (0 <= v1 && v1 < m &&
                0 <= k1 && k1 < n &&
                !poseceno[v1][k1] &&
```

```
                !prepreke[v1][k1])
            {
                Data polje1 = { v1, k1 };
                push(&stek, polje1);
                poseceno[v1][k1] = 1;
            }
        }
    }

    obrisi_stek(&stek);
    return 0;
}

int main()
{
    int m, n;
    char c;

    /* učitavanje dimenzija mape */
    if (scanf("%d %d\n", &m, &n) != 2)
        return -1;

    int prepreke[MAX_N][MAX_N];

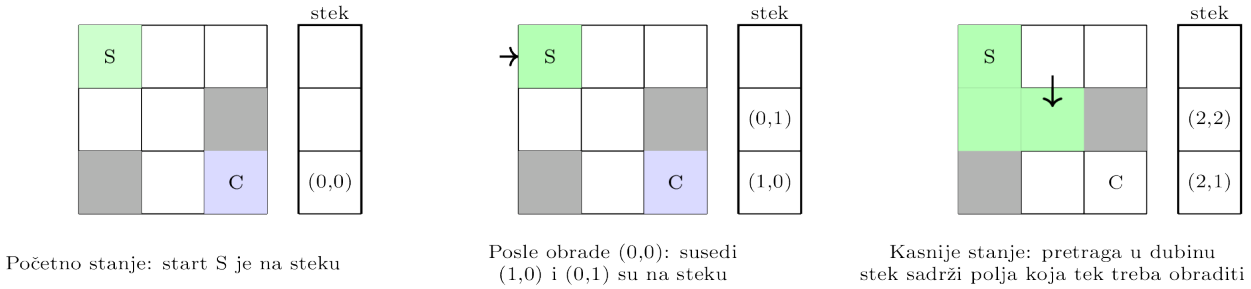
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            c = getchar();
            prepreke[i][j] = (c == '1');  /* '1' označava prepreku */
        }
        getchar(); /* preskoči '\n' */
    }

    if (postojiPut(prepreke, m, n))
        printf("\nPostoji put\n");
    else
        printf("\nNe postoji put\n");

    return 0;
}
```

```
Ulaz:           Izlaz:
3 3             Postoji put
0 0 0
0 0 1
1 0 0
```

```
Ulaz:           Izlaz:
3 3             Ne postoji put
0 0 0
0 1 1
0 1 0
```



Ilustracija stanja steka tokom obilaska mape

Ovaj algoritam predstavlja **sistematičnu pretragu mape**. Redosled prolaska kroz polja određen je strukturom steka — uvek se bira poslednje dodato polje. Ovakav redosled naziva se **pretraga u dubinu (depth-first search, DFS)**.

Ako put postoji, algoritam će ga sigurno otkriti, ali pronađeni put **nije nužno najkraći**. Najkraći put se nalazi pomoću algoritma koji koristi red, odnosno pretragu u širinu (BFS), o čemu će biti reči kasnije.

3.5 Red

Red (*queue*) je linearna struktura podataka koja radi po principu **FIFO** (*first in, first out*). Ponaša se kao red u čekaonici kod zubara. Novi element se dodaje na kraj reda, a uklanja se onaj koji je najduže čekao. Poslednji koji stigne je poslednji koji će biti uslužen. Element reda može sadržati proizvoljan broj podataka i tipova, zavisno od konkretne primene.

Red obezbeđuje dve ključne operacije, obe zahtevane u vremenskoj složenosti **O(1)**:

- **enqueue:** Dodaje element na kraj reda. U nekim implementacijama naziva se i `add`.
- **dequeue:** Uklanja element sa početka reda i vraća ga korisniku. Ponegde se naziva `get`.

Pored osnovnih operacija, tipična implementacija reda obično sadrži i:

- inicijalizaciju strukture,
- brisanje i oslobađanje resursa,
- proveru da li je red prazan,
- očitavanje broja elemenata.

Ove funkcije ne menjaju osnovnu FIFO semantiku, ali omogućavaju da se implementacija lako zameni drugom bez izmene korisničkog koda. Red je pogodan za mnoge algoritamske obrasce i sisteme, o čemu će kasnije u tekstu biti više reči.

3.5.1 Implementacija reda korišćenjem niza

Red je moguće implementirati korišćenjem niza u kojem se elementi smeštaju **kružno**. Kada se dođe do poslednje pozicije u nizu, sledeći element ponovo se smešta na početak. Ovakva implementacija je jednostavna, efikasna i vrlo česta u praksi.

Ako je unapred poznat **maksimalni kapacitet** reda (najveći broj elemenata koji se istovremeno mogu nalaziti u strukturi), tada se može koristiti statički alociran niz ili dinamički alociran niz čija se veličina ne menja tokom rada programa. Upravo takvu, fiksnu implementaciju prikazujemo u nastavku. Postoji i fleksibilnija varijanta u kojoj se niz realocira kada se popuni, ali ona ovde nije obrađena.

Održavamo dva indeksa:

- `pocetak` – pozicija prvog elementa u redu
- `kraj` – prva slobodna pozicija iza poslednjeg elementa

Na početku su oba indeksa jednaka nuli. Ako su jednaki u bilo kom trenutku, red je prazan.

Da bismo razlikovali stanje praznog reda od stanja punog reda, niz alociramo tako da ima **jedno mesto više** od maksimalnog broja elemenata. U nizu veličine `n + 1` može se držati najviše `n` elemenata reda. Red smatramo punim ako indeks `kraj` dolazi neposredno ispred indeksa `pocetak`. U tom stanju nije dozvoljeno dodavanje novog elementa.

Dodavanje (`enqueue`) se vrši tako što se novi element postavi na poziciju `kraj`, a zatim se `kraj` uveća za 1:

- idealno je pisati `kraj = (kraj + 1) % (n + 1)`, ali izbegavamo modul zbog performansi,
- zato koristimo jednostavno grananje: ako smo dostigli kraj niza, vraćamo indeks na 0.

Uklanjanje (`dequeue`) radi simetrično: element se preuzima sa pozicije `pocetak`, a zatim se `pocetak` pomera unapred uz kružno vraćanje na nulu po potrebi.

Broj elemenata u redu dobija se kao razlika: `kraj - pocetak`. Ako je razlika negativna, uvećavamo je za `velicina` niza (`n+1`). Ovo uklanja potrebu za skupim operacijama deljenja po modulu.

Red je:

- prazan ako je broj elemenata 0,
- pun ako je broj elemenata `n`.

Datoteka `red_niz.h`:

```
## ifndef __RED_NIZ_H__
## define __RED_NIZ_H__

typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA,
  PUN_RED
} status;

typedef struct {
```

```
  int pocetak, kraj;
  int velicina;
  int *podaci;
} Red;

status inicijalizuj_red(Red* red, int n);
status enqueue(Red* red, int podatak);
status dequeue(Red* red, int *podatak);
size_t broj_elemenata_reda(const Red* red);
int red_je_prazan(const Red* red);
int red_je_pun(const Red* red);
void obrisi_red(Red* red);

## endif
```

Datoteka red_niz.c:

```
## include <stdlib.h>
## include "red_niz.h"

static int red_je_pun(const Red* red);

status inicijalizuj_red(Red* red, int n)
{
  red->velicina = n + 1;
  red->podaci = malloc(red->velicina*sizeof(int));
  if (red->podaci == NULL)
    return NEDOVOLJNO_MEMORIJE;
  red->pocetak = 0;
  red->kraj = 0;
  return OK;
}

status enqueue(Red* red, int podatak)
{
  if (red_je_pun(red))
    return PUN_RED;
  red->podaci[red->kraj] = podatak;
  red->kraj++;
  if (red->kraj == red->velicina)
    red->kraj = 0;
  return OK;
}

status dequeue(Red* red, int *podatak)
{
  if (red_je_prazan(red))
    return NEMA_ELEMENATA;
```

```
  *podatak = red->podaci[red->pocetak];
  red->pocetak++;
  if (red->pocetak == red->velicina)
    red->pocetak = 0;
  return OK;
}

size_t broj_elemenata_reda(const Red* red)
{
  int razlika = red->kraj - red->pocetak;
  if (razlika < 0)
    razlika += red->velicina;
  return razlika;
}

int red_je_prazan(const Red* red)
{
  return broj_elemenata_reda(red) == 0;
}

int red_je_pun(const Red* red)
{
  return broj_elemenata_reda(red) + 1 == red->velicina;
}

void obrisi_red(Red* red)
{
  free(red->podaci);
  red->velicina = 0;
  red->pocetak = 0;
  red->kraj = 0;
}
```

Primer korišćenja:

```
## include <stdio.h>
## include "red_niz.h"

int main()
{
  Red red;
  int podatak;
  int n = 5;
  if (inicijalizuj_red(&red, n) != OK)
    return -1;

  if (enqueue(&red, 1) != OK)
    obrisi_red(&red);
```

```
  if (enqueue(&red, 2) != OK)
    obrisi_red(&red);

  dequeue(&red, &podatak);
  printf("%d ", podatak);

  if (enqueue(&red, 3) != OK)
    obrisi_red(&red);

  dequeue(&red, &podatak);
  printf("%d ", podatak);

  dequeue(&red, &podatak);
  printf("%d ", podatak);

  return 0;
}
```

```
Izlaz:
1 2 3
```

Funkcija dequeue se poziva samo u situacijama u kojima se zna da red nije prazan, pa se povratni kod operacije ne proverava.

3.5.2 Implementacija reda korišćenjem povezane liste

Red se može implementirati korišćenjem **povezane liste**. Osnovne operacije nad redom — dodavanje na kraj i uklanjanje sa početka — ilustrovane su na slici.

Struktura reda u ovoj implementaciji čuva pokazivače na **početak** i **kraj** liste, kao i **broj elemenata**. Ovaj poslednji podatak je koristan jer određivanje dužine liste direktnim prebrojavanjem zahteva vreme linearno u broju elemenata.

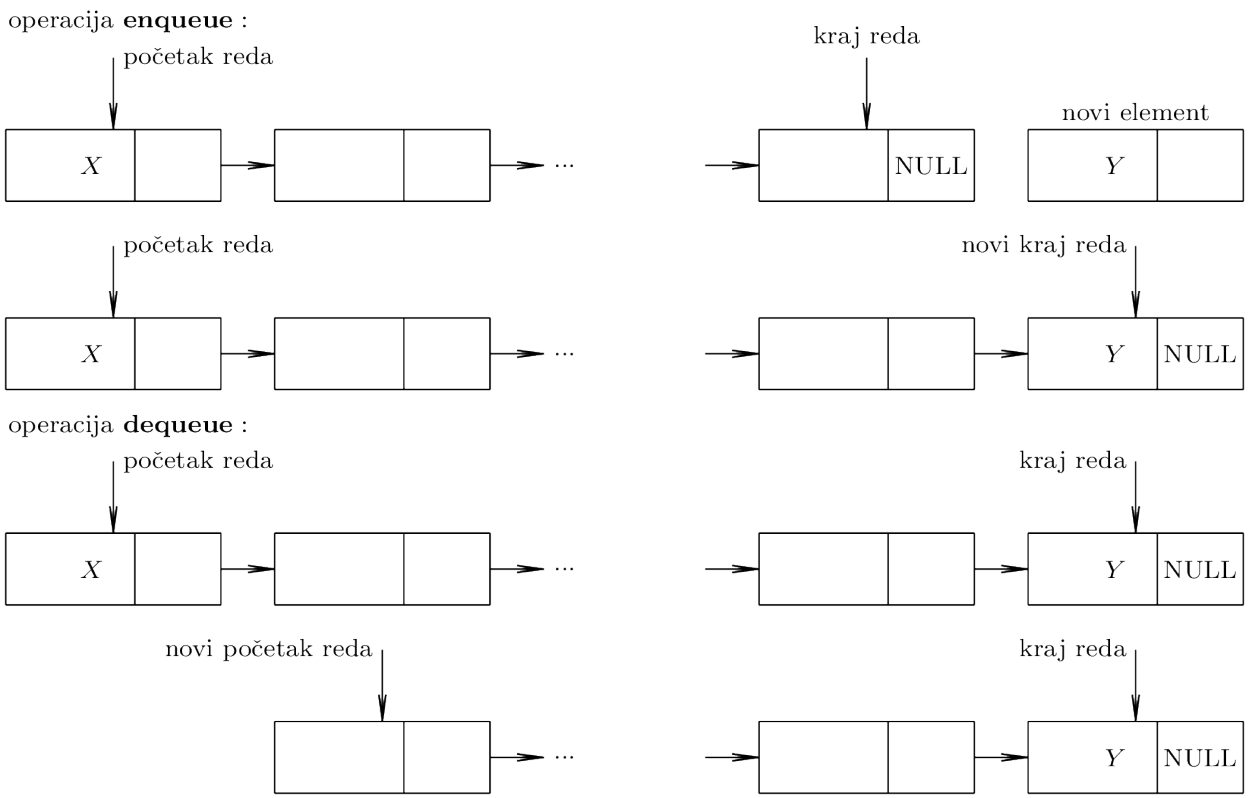
Koristeći opšte pomoćne funkcije za rad sa povezanim listama, funkcije za rad sa redom mogu se organizovati u datoteke `red_lista.h` i `red_lista.c` na sledeći način.

Datoteka `red_lista.h`**:**

```
## ifndef __RED_LISTA_H__
## define __RED_LISTA_H__

typedef enum {
  OK,
  NEDOVOLJNO_MEMORIJE,
  NEMA_ELEMENATA
} status;

typedef struct cvor {
  int podatak;
```



Ilustracija osnovnih operacija nad redom implementiranim kao povezana lista

```
  struct cvor *sledeci;
} cvor;

typedef struct {
  cvor *pocetak;
  cvor *kraj;
  size_t broj_elemenata;
} Red;

status inicijalizuj_red(Red* red);
status enqueue(Red* red, int podatak);
status dequeue(Red* red, int *podatak);
size_t broj_elemenata_reda(const Red* red);
int red_je_prazan(const Red* red);
void obrisi_red(Red* red);

## endif
```

Datoteka `red_lista.c`:

```
## include <stdlib.h>
## include "red_lista.h"

static cvor *novi_cvor(int podatak);
```

```
static status dodaj_na_kraj(cvor **pokazivac_na_pocetak,
                            cvor **pokazivac_na_kraj,
                            int podatak);
static status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                                cvor **pokazivac_na_kraj,
                                int *podatak);
static int red_je_pun(const Red *red);

static cvor *novi_cvor(int podatak)
{
  cvor *novi = malloc(sizeof(cvor));
  if (novi == NULL)
    return NULL;
  novi->podatak = podatak;
  return novi;
}

static status dodaj_na_kraj(cvor **pokazivac_na_pocetak,
                            cvor **pokazivac_na_kraj,
                            int podatak)
{
  cvor *novi = novi_cvor(podatak);
  if (novi == NULL)
    return NEDOVOLJNO_MEMORIJE;
  novi->sledeci = NULL;
  if (*pokazivac_na_pocetak == NULL)
    *pokazivac_na_pocetak = novi;
  else
    (*pokazivac_na_kraj)->sledeci = novi;
  *pokazivac_na_kraj = novi;
  return OK;
}

static status obrisi_sa_pocetka(cvor **pokazivac_na_pocetak,
                                cvor **pokazivac_na_kraj,
                                int *podatak)
{
  cvor *p;
  if (*pokazivac_na_pocetak == NULL)
    return NEMA_ELEMENATA;
  p = *pokazivac_na_pocetak;
  *podatak = (*pokazivac_na_pocetak)->podatak;
  *pokazivac_na_pocetak = (*pokazivac_na_pocetak)->sledeci;
  free(p);

  if (*pokazivac_na_pocetak == NULL)
     *pokazivac_na_kraj = NULL;
  return OK;
```

```
}

status inicijalizuj_red(Red *red) {
  red->pocetak = NULL;
  red->kraj = NULL;
  red->broj_elemenata = 0;
  return OK;
}

status enqueue(Red* red, int podatak)
{
  status s = dodaj_na_kraj(&(red->pocetak), &(red->kraj), podatak);
  if (s == OK)
    red->broj_elemenata++;
  return s;
}

status dequeue(Red* red, int* podatak)
{
  status s = obrisi_sa_pocetka(&(red->pocetak), &(red->kraj), podatak);
  if (s == OK)
    red->broj_elemenata--;
  return s;
}

size_t broj_elemenata_reda(const Red* red)
{
  return red->broj_elemenata;
}

int red_je_prazan(const Red* red)
{
  return broj_elemenata_reda(red) == 0;
}

static int red_je_pun(const Red* red)
{
   return 0;
}

void obrisi_red(Red* red)
{
  while (red->pocetak)  {
    cvor *p = red->pocetak;
    red->pocetak = red->pocetak->sledeci;
    free(p);
  }
}
```

Funkcije koje su označene kao `static` namenjene su samo internom korišćenju u okviru implementacije i ne treba da budu dostupne korisniku reda.

Radi jednostavnosti, funkcija `red_je_pun` uvek vraća netačno, jer se povezana lista može proširivati po potrebi i praktično nema gornjeg ograničenja kapaciteta (osim raspoložive memorije).

Kao i kod implementacije steka pomoću povezane liste, mana ove realizacije je što svaka operacija dodavanja ili uklanjanja elementa zahteva **alokaciju odnosno dealokaciju čvora**, što jeste operacija konstantne složenosti, ali je sporija od manipulacije indeksima u implementaciji zasnovanoj na nizu.

3.5.3 Implementacija reda korišćenjem dva steka

U funkcionalnim programskim jezicima uobičajeno postoje liste kojima se elementi mogu dodavati na početak i uklanjati sa početka. Takve liste pružaju idealnu osnovu za implementaciju **stekova**, ali implementacija **reda** nije jednako direktna.

Ipak, pokazuje se da se red može elegantno implementirati korišćenjem **dva steka**. Ovakav pristup nije tipičan za C, ali je veoma čest u funkcionalnim jezicima.

Elementi reda su raspoređeni u dva steka:

- **ulazni stek** – u njega se elementi dodaju (`enqueue`)
- **izlazni stek** – iz njega se elementi uklanjaju (`dequeue`)

Kada je izlazni stek prazan, a potrebno je ukloniti element iz reda, svi elementi se *prebacuju* sa ulaznog na izlazni stek. Pri tom se redosled inverzuje: element koji je poslednji ubačen u ulazni stek postaje poslednji u redu, tj. smešta se na dno izlaznog steka. Time se obezbeđuje pravilna FIFO semantika.

Ako, na primer, izlazni stek od vrha ka dnu sadrži `5, 6, 2`, a ulazni `3, 7, 4`, tada je sadržaj reda (od prvog do poslednjeg elementa):

`5, 6, 2, 4, 7, 3`

Operacija **dodavanja** je uvek `O(1)`.

Za operaciju **uklanjanja**:

- ako izlazni stek sadrži elemente $\rightarrow$ `O(1)`
- ako je prazan $\rightarrow$ potrebno je prebaciti sve elemente sa ulaznog steka, što je u najgorem slučaju `O(n)`

Međutim, ovaj najgori slučaj je retkost. Svaki element tokom života u redu:

1. jednom se postavlja na ulazni stek
2. jednom prebacuje na izlazni stek
3. jednom uklanja iz izlaznog steka

Dakle, ukupno se nad `n` elemenata obavi `O(3n)` operacija. Prosečno vreme jedne operacije uklanjanja je zato `O(1)` — što je suština **amortizovane analize**.

Datoteka `red_dva_steka.h`**:**

```
## ifndef __RED_DVA_STEKA_H__
## define __RED_DVA_STEKA_H__

## include "stek_niz.h"

typedef struct {
  Stek ulaz, izlaz;
} Red;

status inicijalizuj_red(Red* red);
status enqueue(Red* red, int podatak);
status dequeue(Red* red, int *podatak);
size_t broj_elemenata_reda(const Red* red);
int red_je_prazan(const Red* red);
void obrisi_red(Red* red);

## endif
```

Datoteka `red_dva_steka.c`:

```
## include <stdlib.h>
## include "red_dva_steka.h"

static int red_je_pun(const Red* red);

status inicijalizuj_red(Red* red)
{
  status s;
  if ((s = inicijalizuj_stek(&(red->ulaz))) != OK)
    return s;
  if ((s = inicijalizuj_stek(&(red->izlaz))) != OK)
    return s;
}

status enqueue(Red* red, int podatak)
{
  return push(&(red->ulaz), podatak);
}

status dequeue(Red* red, int *podatak)
{
  status s;
  if (stek_je_prazan(&(red->izlaz))) {
    /* prebacujemo sve elemente sa ulaznog na izlazni stek */
    while (!stek_je_prazan(&(red->ulaz))) {
      int tmp;
      if ((s = pop(&(red->ulaz), &tmp)) != OK)
        return s;
```

```
      if ((s = push(&(red->izlaz), tmp)) != OK)
        return s;
    }
  }
  /* vracamo podatak na vrhu izlaznog steka */
  return pop(&(red->izlaz), podatak);
}

size_t broj_elemenata_reda(const Red* red)
{
   return broj_elemenata(&(red->ulaz)) +
          broj_elemenata(&(red->izlaz));
}

int red_je_prazan(const Red* red)
{
   return (stek_je_prazan(&(red->ulaz)) &&
           stek_je_prazan(&(red->izlaz)));
}

static int red_je_pun(const Red* red)
{
   return 0;
}

void obrisi_red(Red* red)
{
   obrisi_stek(&(red->ulaz));
   obrisi_stek(&(red->izlaz));
}
```

Pretpostavlja se da se stekovi implementirani u `stek_niz.h` i `stek_niz.c` dinamički proširuju po potrebi, pa red nikada nije pun. Zbog toga `red_je_pun` uvek vraća vrednost *netačno* (0).

Da bi se izbeglo preklapanje funkcija sa istim imenom, funkcija za brojanje elemenata reda nazvana je `broj_elemenata_reda`, dok stek koristi naziv `broj_elemenata`.

U ovoj implementaciji se oslanjamo isključivo na stek interfejs, a `status` tip se dobija iz `stek_niz.h`, pa se u ovom modulu ponovo ne definiše.

3.5.4 Primer primene strukture red: poslednjih k učitanih brojeva

Tokom obrade podataka često se susreće situacija u kojoj se unosi dugačak niz vrednosti, ali je u svakom trenutku potrebno poznavati samo **poslednjih** k učitanih elemenata.

Tipičan primer je analiza temperature: dnevne vrednosti osciluju, ali prosečna temperatura tokom prethodnih k dana (recimo 7) daje stabilnije informacije. U takvim problemima nije potrebno čuvati sve podatke; dovoljno je održavati samo poslednjih k učitanih vrednosti, čime se značajno štedi memorija.

Kolekcija poslednjih k elemenata ponaša se kao **red**: najranije učitan element prvi ispada, a novi se dodaje na kraj. Zbog toga se ovaj pristup naziva i **pokretni prozor** (*sliding window*).

Neka je potrebno da se među n učitanih elemenata ispiše poslednjih k. Postupak je sledeći:

1. Učitati element.
2. Ako red već sadrži k elemenata $\rightarrow$ ukloniti jedan (najstariji).
3. Dodati novi element.

Na kraju ispisati preostale elemente u redu — to su poslednjih k vrednosti.

Primer programa:

```
## include <stdio.h>
## include "red_dva_steka.h"

int main() {
  int k, n, i;
  Red red;
  if (scanf("%d %d", &k, &n) != 2 || n<=0 || k<=0)
    return -1;
  if (inicijalizuj_red(&red) != OK)
    return -1;

  for (i = 0; i < n; i++) {
    int x;
    if (scanf("%d", &x) != 1)
      return -1;
    if (broj_elemenata_reda(&red) == k) {
      int tmp;
      dequeue(&red, &tmp);
    }
    if (enqueue(&red, x) != OK)
      return -1;
  }

  printf("\n");
  while (!red_je_prazan(&red)) {
    int x;
    dequeue(&red, &x);
    printf("%d ", x);
  }
  printf("\n");
  obrisi_red(&red);
  return 0;
}
```

```
Za ulaz:
4 10
1 2 3 4 5 6 7 8 9 10
```

```
izlaz je:
7 8 9 10
```

Očekivano — poslednjih 10 unosa, ali uz prozor širine 4, dobijamo upravo poslednja 4 broja.

3.5.5 Primer primene strukture red: pretraga u širinu i pronalaženje puta na mapi

Razmotrimo ponovo problem pronalaženja puta na mapi opisan ranije u poglavljima.

Ovaj problem može se rešiti i korišćenjem **reda**. Kôd je praktično identičan rešenju iz poglavlja o pronalaženju puta pomoću steka, uz jedinu razliku: umesto strukture *stek* koristi se struktura *red*, a umesto operacija `push` i `pop` koriste se `enqueue` i `dequeue`.

I u ovoj varijanti pretraga je sistematična i, ukoliko postoji put između polaznog i ciljnog polja, algoritam će ga uvek pronaći.

Razlika je, međutim, u **redosledu obilaska polja**:

- korišćenje **steka** vodi pretrazi u dubinu (*depth-first search*, DFS),
- korišćenje **reda** daje pretragu u širinu (*breadth-first search*, BFS).

Zahvaljujući BFS-u, algoritam koji koristi red pronalazi **najkraći put** između početnog i ciljnog polja, pod pretpostavkom da je svaki korak iste „cene“, tj. da se kretanje u svim smerovima jednako računa.

To je glavna prednost korišćenja reda u ovom kontekstu — dobijamo korektnu i optimalnu putanju, bez dodatnih složenosti algoritma.

3.6 Dek (red sa dva kraja)

Red sa dva kraja (dek, *deque*) predstavlja strukturu podataka koja objedinjuje osnovne funkcionalnosti steka i reda: dodavanje i uklanjanje elemenata moguće je na oba kraja strukture. U prethodnim poglavljima videli smo da ove operacije mogu da budu veoma efikasne kada se struktura implementira listama. Međutim, liste imaju jedno ograničenje koje ne možemo da zanemarimo: *indeksni pristup elementima nije moguć u konstantnom vremenu*. To isključuje primenu binarne pretrage i uopšte otežava algoritme koji se oslanjaju na brz pristup proizvoljnom elementu.

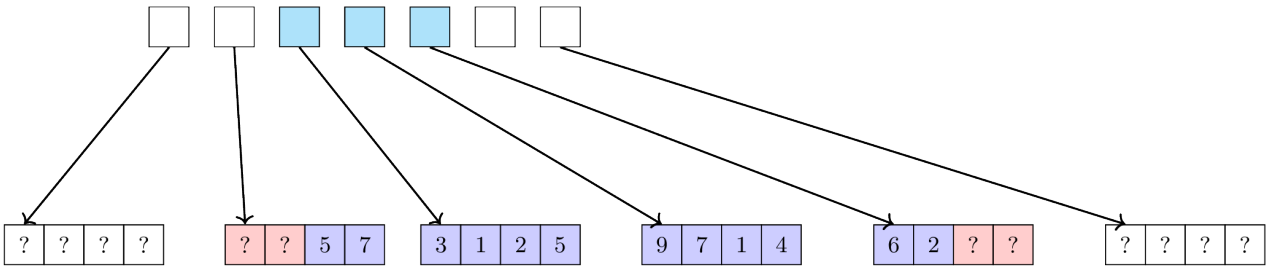
Zbog toga u ovom poglavlju uvodimo praktičnu implementaciju deka zasnovanu na **dinamičkom nizu segmenata fiksne veličine**. Ovaj model kombinuje dobro ponašanje lista na krajevima (O(1) dodavanje/brisanje) sa indeksnim pristupom karakterističnim za nizove (O(1) pristup elementu po indeksu).

3.6.1 Osnovna ideja: niz segmenata

Zamislimo da je dek podeljen na segmente jednake veličine, na primer blokove od 32 ili 64 elementa. Svaki segment je lokalni, kompaktni niz, a sam dek čuva **nivo više** — niz pokazivača na te segmente.

Na primer, niz:

```
5, 7, 3, 1, 2, 5, 9, 7, 1, 4, 6, 2
```



Ilustracija deka

može biti raspoređen u segmentima koji imaju po četiri elementa kao na slici.

Početak deka može biti u sredini drugog segmenta, a kraj u sredini pretposlednjeg — nije neophodno da segmenti budu puni.

Ovo je važan detalj jer struktura ostavlja rezervu sa obe strane kako bi se izbegle česte realokacije.

3.6.2 Primer implementacije deka

U ovoj implementaciji dek (red sa dva kraja) ne posmatramo kao jednu "kontinuiranu" strukturu, već kao **niz segmenata fiksne veličine**. Svaki segment je mali niz koji sadrži deo elemenata deka, dok sam dek održava dodatni nivo: **niz pokazivača na segmente**. Ovakva reprezentacija je standardna za strukture tipa `dek`, jer spaja dve važne osobine: efikasne operacije na krajevima i indeksni pristup elementima.

Neka je `S` fiksna veličina segmenta. Segment sadrži niz elemenata `a[0..S-1]`, ali u opštem slučaju segment nije uvek potpuno popunjen. Zato se popunjenost segmenta opisuje parom indeksa `l` i `d`:

- `l` je **leva granica popunjenog dela** (inkluzivno),
- `d` je **desna granica popunjenog dela** (ekskluzivno).

Drugim rečima, **popunjeni elementi segmenta** nalaze se tačno u intervalu:

$$a[l], a[l+1], \ldots, a[d-1]$$

Segment je prazan ako i samo ako važi `l == d`. U tom slučaju ne postoji nijedan element koji pripada segmentu.

U kodu, segment je definisan ovako:

```
struct Segment {
    int* a;   /* niz elemenata */
    int l;    /* leva granica */
    int d;    /* desna granica */
};
```

Ovaj par indeksa (`l`, `d`) opisuje isključivo **stanje unutar jednog segmenta**. Važno je naglasiti da `l` i `d` ne govore ništa o tome gde se segment nalazi u okviru deka; oni su lokalni za konkretan segment.

Dek kao "mapa" segmenata

Sam dek čuva pokazivače na segmente u nizu seg. Taj niz se često naziva "mapa" (engl. *map*), ali ovde nije reč o mapi u smislu asocijativnog kontejnera, već o **indeksabilnom nizu pokazivača**.

```
typedef struct Segment Segment;

typedef struct Dek {
    Segment** seg;       /* niz pokazivača na segmente */
    int seg_broj;        /* broj slotova u nizu seg */
    int seg_velicina;    /* veličina segmenta S */

    int levo;            /* indeks levog segmenta sa elementima */
    int desno;           /* indeks desnog segmenta sa elementima */

    int velicina;        /* ukupan broj elemenata u deku */
} Dek;
```

Ovde se jasno vide dva nivoa:

1. seg je niz (kontinuirana memorija) čiji su elementi **pokazivači na segmente** (Segment*).
2. Svaki segment (Segment) poseduje sopstveni niz elemenata.

Zbog toga je tip seg upravo Segment**: to je niz pokazivača na prvi element niza čiji su elementi tipa Segment.

Zašto Segment, a ne Segment*?**

Kada bismo imali samo Segment* seg, to bi značilo da dek poseduje **jedan kontinuirani niz segmenata** (npr. seg[0], seg[1], ... su stvarni objekti tipa Segment). Takvo rešenje je nepraktično iz dva razloga:

1. *Realokacija bi bila skupa.* Kada se dek proširuje, morali bismo da realociramo i kopiramo ceo niz Segment struktura. Kod niza pokazivača (Segment**) realokacija znači kopiranje samo pokazivača (tipično 8 bajtova po segmentu), dok sami segmenti ostaju na svojim adresama.
2. *Segmenti i njihovi podaci se ne pomeraju u memoriji.* Segment sadrži pokazivač a ka nizu elemenata. Ako bi se segmenti pomerali, morali bismo pažljivo voditi računa o svim vezama.

Ova ideja je ključna: **dek se širi tako što se širi "mapa" pokazivača, a ne tako što se kopiraju elementi deka**.

Granice deka: levo/desno naspram l/d

U ovoj implementaciji postoje dva različita para granica i njihova uloga mora biti strogo razdvojena:

- levo i desno su *globalne granice*: oni su indeksi u nizu seg i opisuju koji segmenti trenutno sadrže elemente deka.
- l i d su *lokalne granice*: oni su indeksi u nizu, unutar jednog segmenta i opisuju koji deo segmenta je popunjen.

Invarijante koje važe u svakom trenutku

U nastavku su osnovne invarijante koje implementacija održava u svakom trenutku rada:

1. *Segmenti su fiksne veličine.* Svi segmenti koji postoje imaju dužinu `seg_velicina`.
2. *Popunjenost segmenta je interval.* U svakom segmentu važi `0 <= l <= d <= S`. Elementi segmenta koji pripadaju deku čine jedan uzastopni interval `a[l..r-1]`.
3. *Granice deka opisuju opseg segmenata sa elementima.* Ako `velicina > 0`, tada segmenti sa elementima čine kontinuirani blok u mapi: od `seg[levo]` do `seg[desno]` (uključivo). Segmenti van tog opsega ili su `NULL` ili predstavljaju rezervu koja trenutno nije u upotrebi.
4. *Prazan dek je posebno stanje.* Ako je `velicina == 0`, tada dek ne sadrži nijedan element. U tom slučaju `levo` i `desno` su jednaki (ili eventualno susedni, u zavisnosti od dogovorene konvencije), a segmenti na granicama su prazni. Suština je da ne postoji nijedan segment sa popunjenim intervalom.
5. *Broj elemenata je konzistentan.* Vrednost `velicina` jednaka je ukupnom broju elemenata raspoređenih po segmentima između `levo` i `desno`.

Ove invarijante su pravilo koje važi u svakom trenutku: svaka operacija mora da ih sačuva. U narednim odeljcima, prilikom objašnjavanja dodavanja i brisanja elemenata, upravo na njih ćemo se stalno vraćati.

U prikazanoj implementaciji dek skladišti `int`. To pojednostavljuje kod, ali je sama ideja potpuno opšta: umesto `int` može se koristiti bilo koji tip podataka, pod uslovom da se pravilno upravlja kopiranjem i eventualnim destruktorima (što je u C-u odgovornost programera).

Inicijalizacija deka

Pre nego što se nad dekom mogu izvršavati bilo kakve operacije, neophodno je da se on dovede u *konzistentno početno stanje* u kome sve invarijante važe, ali nijedan element još uvek nije prisutan. Tu ulogu ima funkcija `dek_init`.

```
int dek_init(Dek* d, int velicina_segmenta, int broj_pocetnih_segmenata)
{
    if (!d || velicina_segmenta <= 0 || broj_pocetnih_segmenata <= 0) return 0;

    d->seg = (Segment**)malloc((size_t)broj_pocetnih_segmenata * sizeof(Segment*));
    if (!d->seg) return 0;

    for (int i = 0; i < broj_pocetnih_segmenata; i++) d->seg[i] = NULL;

    d->seg_broj = broj_pocetnih_segmenata;
    d->seg_size  = velicina_segmenta;

    /* start u sredini map-a */
    d->levo = d->desno = broj_pocetnih_segmenata / 2;

    d->velicina = 0;
    return 1;
```

```
}
```

Funkcija `dek_init` inicijalizuje praznu strukturu deka tako da je spremna za efikasno dodavanje elemenata i sa levog i sa desnog kraja. Nakon njenog izvršavanja dek ne sadrži nijedan element, ali poseduje unapred alociranu strukturu koja omogućava da se prve operacije dodavanja izvrše bez momentalne realokacije. Prvi korak je alokacija niza pokazivača na segmente:

```
d->seg = (Segment**)malloc(broj_pocetnih_segmenata * sizeof(Segment*));
```

Ovde se ne alociraju sami segmenti, već **samo niz pokazivača** na njih. Svaki element ovog niza inicijalno se postavlja na `NULL`, čime se eksplicitno označava da segment na toj poziciji još uvek ne postoji. Segmenti se alociraju *lenjo* (tek kada su zaista potrebni), što smanjuje početnu potrošnju memorije.

Veličina ovog niza (`broj_pocetnih_segmenata`) predstavlja početni kapacitet mape. U toku rada deka ovaj kapacitet može biti uvećan realokacijom, ali se u inicijalnom stanju bira neka razumna početna vrednost.

Posebno je važno početno postavljanje indeksa `levo` i `desno`:

```
d->levo = d->desno = broj_pocetnih_segmenata / 2;
```

Time se dek "pozicionira" u sredinu mape segmenata. Iako je dek u tom trenutku prazan, ovakav izbor omogućava da se prvi elementi dodaju i na početak i na kraj bez potrebe za trenutnim proširivanjem mape. Drugim rečima, unapred se obezbeđuju *prazni segmenati i levo i desno*, što značajno smanjuje broj realokacija u početnim fazama rada.

U praznom deku važi da su `levo` i `desno` jednaki i da segment na toj poziciji još uvek ne postoji (`seg[levo] == NULL`). To stanje je potpuno konzistentno sa invarijantama: dek nema elemenata, ali zna gde će se prvi segment pojaviti.

Na kraju se postavlja:

```
d->velicina = 0;
```

Polje `velicina` uvek sadrži ukupan broj elemenata u deku. U inicijalnom stanju taj broj je nula, što eksplicitno označava prazan dek.

Ako je inicijalizacija uspešna, funkcija vraća vrednost `1`. U suprotnom (neispravan argument ili neuspešna alokacija memorije) vraća `0`, a dek se smatra nevalidnim.

Uz inicijalizaciju, često su potrebne i jednostavne funkcije koje omogućavaju proveru trenutnog stanja deka.

```
int dek_velicina(const Dek* d)
{
    return d ? d->velicina : 0;
}

int dek_prazan(const Dek* d)
{
    return !d || d->velicina == 0;
```

```
}
```

Dodavanje elementa

Dodavanje elemenata na početak i na kraj deka oslanja se na postojanje slobodnih segmenata sa odgovarajuće strane. Dok god u mapi segmenata postoji slobodan slot levo od indeksa `levo` ili desno od indeksa `desno`, novi segment se može dodati bez ikakvog strukturnog preuređenja. Međutim, u nekom trenutku dek može da dođe u stanje u kome su svi slotovi mape sa jedne strane iscrpljeni. Tada je neophodno **proširiti mapu segmenata**.

Važno je naglasiti da se ovim postupkom *ne proširuje sam dek u smislu elemenata*, već se proširuje samo pomoćna struktura koja omogućava njihovo organizovanje. Drugim rečima, ne kopiraju se elementi deka, niti se menjaju postojeći segmenti; menja se isključivo niz pokazivača koji na njih upućuje.

Proširivanje mape realizovano je funkcijom:

```
static int dek_prosiri(Dek* d, int novi_broj_segmenata)
{
    assert(novi_broj_segmenata > d->seg_broj);

    Segment** nova_mapa = (Segment**)malloc((size_t)novi_broj_segmenata * sizeof(S
    if (!nova_mapa) return 0;

    for (int i = 0; i < novi_broj_segmenata; i++)
        nova_mapa[i] = NULL;

    int shift = (novi_broj_segmenata - d->seg_broj) / 2;

    for (int i = 0; i < d->seg_broj; i++)
        nova_mapa[shift + i] = d->seg[i];

    free(d->seg);
    d->seg = nova_mapa;

    d->levo  += shift;
    d->desno += shift;
    d->seg_broj = novi_broj_segmenata;

    return 1;
}
```

Osnovna ideja realokacije je jednostavna: *postojeći segmenti se zadržavaju i njihove adrese*. Novi, veći niz pokazivača se alocira, a stari pokazivači se premeštaju u njegovu sredinu. Na taj način dek dobija novu rezervu slobodnih slotova i sa leve i sa desne strane.

Ovakav postupak ima dve važne posledice:

- elementi deka se ne kopiraju i ne pomeraju u memoriji,
- realokacija je znatno jeftinija nego kod struktura koje koriste jedan kontinuirani niz elemenata.

Upravo ova razlika objašnjava zašto se dek u praksi često implementira kao "mapa segmenata", a ne kao jedan veliki niz.

Prilikom realokacije stari pokazivači na segmente ne smeštaju se na početak nove mape, već u njen **centralni deo**. Time se postiže ravnoteža: dek dobija približno jednaku količinu slobodnog prostora sa obe strane. Ova simetrična rezerva je ključna za efikasnost, jer sprečava da se već posle nekoliko narednih operacija ponovo javi potreba za proširivanjem.

Upravo zbog toga se izračunava pomeraj `shift`, koji određuje za koliko mesta se postojeći segmenti pomeraju udesno u odnosu na početak nove mape. Svi indeksi segmenata ostaju relativno raspoređeni na isti način; menja se samo njihov apsolutni položaj u mapi.

Pošto se svi segmenti pomeraju za isti broj pozicija, neophodno je da se i indeksi koji opisuju granice deka pomere za isti iznos. Zbog toga se nakon realokacije ažuriraju vrednosti `levo` i `desno`.

Ovo ažuriranje je suštinsko za ispravnost strukture: bez njega bi dek izgubio informaciju o tome u kojim se segmentima nalaze prvi i poslednji elementi. Time bi bile narušene osnovne invarijante reprezentacije.

Realokacija mape segmenata ima složenost proporcionalnu broju segmenata, jer se kopiraju svi pokazivači iz stare mape u novu. Međutim, ova operacija se ne izvršava često. Nakon realokacije dek dobija značajnu rezervu slobodnih slotova, pa se veliki broj narednih dodavanja može izvršiti bez ikakvog strukturnog proširivanja.

Zbog toga, iako pojedinačna realokacija nije konstantna, ukupna cena niza operacija dodavanja na početak i na kraj ostaje **amortizovano konstantna**.

Važno je uočiti da se funkcija `dek_prosiri` ne poziva direktno od strane korisnika deka. Ona je pomoćna funkcija, skrivena iza operacija dodavanja. Time se složenost upravljanja memorijom izoluje unutar implementacije, dok javni interfejs deka ostaje jednostavan.

Dodavanje na početak deka

Dodavanje na početak je jedna od dve osnovne operacije deka. U segmentnoj reprezentaciji cilj je sledeći: novi element treba da postane prvi element deka, uz očuvanje invarijanti da su elementi raspoređeni po segmentima kao uzastopni intervali, i da `levo` uvek pokazuje na segment koji sadrži početak deka.

Da bismo to postigli bez ponavljanja koda, koristi se nekoliko pomoćnih funkcija. Najpre ćemo objasniti kako se segment kreira, zatim kako se segmenti lenjo (po potrebi) alociraju u mapi, pa kako se obezbeđuje prostor na levom kraju, i na kraju sama operacija dodavanja.

Kreiranje segmenta

Segment je mala struktura koja sadrži niz dužine S (ovde `seg_size`) i granice popunjenog intervala `l` i `r`. Funkcija `kreiraj_segment` alocira novi segment i inicijalizuje ga kao prazan, tako što postavlja `l = r = pocetna_pozicija`.

```
static Segment* kreiraj_segment(int seg_size, int pocetna_pozicija)
{
    Segment* s = (Segment*)malloc(sizeof(Segment));
    if (!s) return NULL;

    s->a = (int*)malloc((size_t)seg_size * sizeof(int));
    if (!s->a) {
```

```
        free(s);
        return NULL;
    }

    s->l = pocetna_pozicija;
    s->r = pocetna_pozicija;
    return s;
}
```

Parametar `pocetna_pozicija` određuje *gde se nalazi prazni interval* u segmentu. U ovom poglavlju koristimo uniforman model: segment koji će se koristiti na levom kraju inicijalizuje se sa `pocetna_pozicija = S`, pa je prazan interval `[S, S)`, i prvi upis ulevo ide u `a[S-1]` preko operacije `--l`. Analogno, segment za desni kraj inicijalizuje se sa `pocetna_pozicija = 0`.

Funkcija vraća pokazivač na novi segment ili `NULL` ako alokacija ne uspe (u tom slučaju vodi se računa da se već alocirana memorija oslobodi).

Par pomoćnih funkcija za proveru stanja segmenta su:

```
static int segment_prazan(const Segment* s) { return s->l == s->d; }
static int segment_pun_levo(const Segment* s) { return s->l == 0; }
static int segment_pun_desno(const Segment* s, int S) { return s->d == S; }
```

Lenja alokacija segmenta u mapi

Mapa `seg` je niz pokazivača na segmente, pri čemu mnogi slotovi mogu biti `NULL`. To nije greška, već namerno ponašanje: segmenti se alociraju tek kada postanu potrebni. Ovo pojednostavljuje upravljanje memorijom i izbegava bespotrebnu alokaciju velikog broja segmenata unapred.

```
/* Lazy alokacija segmenta u mapi na poziciji idx, ako je potrebno. */
static int obezbedi_segment(Dek* d, int idx, int pocetna_pozicija)
{
    if (d->seg[idx] != NULL) return 1;
    d->seg[idx] = kreiraj_segment(d->seg_size, pocetna_pozicija);
    return d->seg[idx] != NULL;
}
```

Funkcija garantuje sledeće: posle njenog poziva, slot `seg[idx]` sadrži validan segment ili funkcija prijavljuje neuspeh.

Ovo je tipičan idiom u implementaciji struktura podataka: umesto da se svuda u kodu ponavljaju provere "ako je `NULL` alociraj", uvodi se jedna funkcija koja taj obrazac enkapsulira. Time se smanjuje verovatnoća greške, a glavne operacije (`push_front`, `push_back`, …) ostaju kraće i preglednije.

Dodavanje na početak koristi segmente koji rastu ulevo i to je prikazano pomoćnom funkcijom:

```
/* Obezbedi krajnji levi segment: prazan segment spreman za rast ulevo (l=r=S). *
static int obezbedi_levo_segment(Dek* d, int idx)
{
```

```
    return obezbedi_segment(d, idx, d->seg_velicina);
}
```

Analogno, za desni kraj:

```
/* Obezbedi krajnji desni segment: prazan segment spreman za rast udesno (l=r=0). */
static int obezbedi_desno_segment(Dek* d, int idx)
{
    return obezbedi_segment(d, idx, 0);
}
```

Ove dve funkcije nisu "suvišne": one eksplicitno izražavaju nameru (koji tip segmenta je potreban), pa se u ostatku implementacije više ne razmišlja o tome koji `pocetna_pozicija` treba proslediti.

Pre samog upisa elementa, treba osigurati da se u segmentu `seg[levo]` zaista može izvršiti dodavanje ulevo. Dva su razloga zbog kojih to može biti nemoguće:

1. `seg[levo]` je pun s leve strane (`l == 0`), pa nema mesta za još jedan element u tom segmentu.
2. `levo == 0`, pa se ne može preći u prethodni segment bez proširivanja mape.

Ovaj posao obavlja funkcija `obezbedi_prostor_na_pocetku`.

```
/* Obezbedi da postoji mesta za dodavanje na početak.
 * Nakon poziva: segment seg[levo] postoji i ima mesta ulevo.
 */
static int obezbedi_prostor_na_pocetku(Dek* d)
{
    if (!obezbedi_levo_segment(d, d->levo))
        return 0;

    if (!segment_pun_levo(d->seg[d->levo]))
        return 1;

    if (d->levo == 0) {
        if (!dek_prosiri(d, d->seg_broj * 2))
            return 0;
    }

    d->levo--;
    return obezbedi_levo_segment(d, d->levo);
}
```

Ova funkcija se može čitati kao "ako nema prostora u trenutnom levom segmentu, pomeri se na prethodni segment; ako to nije moguće jer smo na ivici mape, prvo proširi mapu".

Kritična stvar je da `obezbedi_prostor_na_pocetku` ne samo da proverava prostor, već po potrebi menja `levo`. Drugim rečima, ona održava globalnu invarijantu o tome gde se nalazi početak deka: ako se prelazi u novi segment, `levo` mora da počne da pokazuje na njega.

`dek_dodaj_na_pocetak`

Kada su segment i prostor obezbeđeni, upis elementa je konceptualno jednostavan: granica `l` se pomeri ulevo i na novu poziciju se upisuje element. U ovoj implementaciji slučaj "prvi element" nema poseban segment niti drugačiju inicijalizaciju segmenta; jedina razlika je što se posle prvog upisa mora uskladiti da oba kraja deka budu u istom segmentu (`desno = levo`).

```
int dek_dodaj_na_pocetak(Dek* d, int x)
{
    assert(d && d->seg && d->seg_velicina > 0);

    int prazan = (d->velicina == 0);

    if (!obezbedi_prostor_na_pocetku(d))
        return 0;

    Segment* s = d->seg[d->levo];
    s->a[--s->l] = x;

    if (prazan)
        d->desno = d->levo;

    d->velicina++;
    return 1;
}
```

Dodavanje na kraj deka

Dodavanje na kraj deka u potpunosti je simetrično dodavanju na početak. Pošto je mehanizam dodavanja na početak već detaljno razmotren, ovde se zadržavamo na kraćem opisu i isticanju razlika koje su specifične za desni kraj deka.

Kao i ranije, operacija se oslanja na pomoćne funkcije koje obezbeđuju da na kraju deka postoji segment u koji se može izvršiti upis, kao i da se po potrebi dek proširi.

Najpre se definiše pomoćna funkcija koja obezbeđuje postojanje desnog krajnjeg segmenta. Takav segment je inicijalizovan tako da je pogodan za rast udesno, odnosno sa granicama `l = r = 0`.

```
static int obezbedi_desno_segment(Dek* d, int idx)
{
    return obezbedi_segment(d, idx, 0);
}
```

Zatim se definiše funkcija koja obezbeđuje da na kraju deka postoji slobodno mesto za dodavanje novog elementa. Ukoliko je trenutni desni segment popunjen, prelazi se u sledeći segment u mapi. Ako se pritom dođe do kraja mape segmenata, dek se proširuje.

```
static int obezbedi_prostor_na_kraju(Dek* d)
{
    if (!obezbedi_desno_segment(d, d->desno))
```

```
        return 0;

    if (!segment_pun_desno(d->seg[d->desno], d->seg_velicina))
        return 1;

    if (d->desno == d->seg_broj - 1) {
        if (!dek_prosiri(d, d->seg_broj * 2))
            return 0;
    }

    d->desno++;
    return obezbedi_desno_segment(d, d->desno);
}
```

Posle uspešnog poziva ove funkcije važi invarijanta da segment `seg[desno]` postoji i da u njemu ima mesta za dodavanje elementa udesno.

Sama operacija dodavanja na kraj deka tada postaje jednostavna:

```
int dek_dodaj_na_kraj(Dek* d, int x)
{
    assert(d && d->seg && d->seg_velicina > 0);

    int prazan = (d->velicina == 0);

    if (!obezbedi_prostor_na_kraju(d))
        return 0;

    Segment* s = d->seg[d->desno];
    s->a[s->r++] = x;

    if (prazan)
        d->levo = d->desno;

    d->velicina++;
    return 1;
}
```

Ako je dek pre dodavanja bio prazan, nakon upisa prvog elementa neophodno je uskladiti granice tako da i `levo` i `desno` pokazuju na isti segment. U suprotnom slučaju granica `levo` ostaje nepromenjena.

Pomoćne funkcije `kreiraj_segment` i `dek_prosiri` opisane su u prethodnom delu. Suštinska razlika u odnosu na dodavanje na početak sastoji se u tome što se ovde uvećava desna granica segmenta (`r++`) i što se, po potrebi, dek proširuje na desnoj strani mape segmenata. Sve ostale invarijante i razmatranja ostaju analogne.

Uklanjanje elementa sa početka i sa kraja deka

Operacije uklanjanja sa početka i sa kraja deka predstavljaju prirodan par operacijama dodavanja. Njihova uloga je da uklone i vrate prvi, odnosno poslednji element deka, pri čemu je neophodno održati kako lokalne invarijante segmenta, tako i globalne invarijante cele strukture.

U segmentnoj reprezentaciji uklanjanje elementa ne podrazumeva fizičko brisanje vrednosti iz niza, već se svodi na pomeranje odgovarajuće granice popunjenosti segmenta (`l` ili `d`). Time se obezbeđuje da se osnovna operacija izvršava u konstantnom vremenu.

Uklanjanje sa početka deka

Kod uklanjanja sa početka uklanja se element na levoj granici popunjenog intervala u levom krajnjem segmentu. Nakon čitanja elementa, granica `l` se uvećava, čime se popunjeni interval sužava.

```
int dek_ukloni_sa_pocetka(Dek* d)
{
    assert(d && d->seg && d->velicina > 0);
    assert(d->seg[d->levo] != NULL);

    Segment* s = d->seg[d->levo];
    assert(!segment_prazan(s));

    int x = s->a[s->l++];
    d->velicina--;

    /* Ako je segment postao prazan i dek i dalje sadrži elemente,
       oslobodi segment i pređi na naredni. */
    if (segment_prazan(s) && d->velicina > 0 && d->levo < d->desno) {
        segment_unisti(d->seg[d->levo]);
        d->seg[d->levo] = NULL;
        d->levo++;
    }

    /* Ako je dek postao prazan, normalizuj granice. */
    if (d->velicina == 0) {
        d->desno = d->levo;
    }

    return x;
}
```

Ključna odluka u ovoj implementaciji jeste da se segment oslobađa *samo ako je postao prazan i ako u deku i dalje postoje elementi*. Time se izbegava situacija u kojoj bi se uklanjanjem poslednjeg elementa ostalo bez ijednog segmenta na koji granice mogu da ukazuju. Kada dek postane prazan, granice se samo normalizuju, dok se eventualno preostali segment može ostaviti alociran radi ponovne upotrebe.

Uklanjanje sa kraja deka

Uklanjanje sa kraja deka je potpuno simetrično. Element se uklanja sa desne granice popunjenog intervala u desnom krajnjem segmentu, pri čemu se granica `d` najpre umanjuje, a zatim se čita odgovarajući element.

```c
int dek_ukloni_sa_kraja(Dek* d)
{
    assert(d && d->seg && d->velicina > 0);
    assert(d->seg[d->desno] != NULL);

    Segment* s = d->seg[d->desno];
    assert(!segment_prazan(s));

    int x = s->a[--s->d];
    d->velicina--;

    /* Ako je segment postao prazan i dek i dalje sadrži elemente,
       oslobodi segment i pređi na prethodni. */
    if (segment_prazan(s) && d->velicina > 0 && d->levo < d->desno) {
        segment_unisti(d->seg[d->desno]);
        d->seg[d->desno] = NULL;
        d->desno--;
        assert(d->seg[d->desno] != NULL);
    }

    /* Ako je dek postao prazan, normalizuj granice. */
    if (d->velicina == 0) {
        d->levo = d->desno;
    }

    return x;
}
```

Kod obe operacije uklanjanja važno je uočiti sledeće:

- granice `l` i `d` uvek opisuju popunjeni interval segmenta u obliku `[l, d)`;
- uklanjanje sa početka vrši se povećanjem `l`, dok se uklanjanje sa kraja vrši smanjenjem `d`;
- prelazak na sledeći ili prethodni segment dozvoljen je samo ako dek i dalje sadrži elemente;
- kada dek postane prazan, granice `levo` i `desno` se svode na istu vrednost, čime se struktura dovodi u konzistentno početno stanje.

Na ovaj način operacije uklanjanja ostaju jednostavne, efikasne i potpuno usklađene sa prethodno definisanim invarijantama deka.

Pristup elementu na datoj poziciji

Za razliku od lista, dek omogućava **indeksni pristup elementima u konstantnom vremenu**. To je jedna od njegovih ključnih prednosti u odnosu na liste. Funkcija `dek_na_poziciji` vraća element koji se u deku nalazi na poziciji `i`, gde se elementi numerišu od 0 (prvi element) do `velicina - 1` (poslednji element).

Osnovna ideja pristupa zasniva se na tome da se dek konceptualno posmatra kao *kontinuirani niz elemenata*, iako je fizički implementiran kao niz segmenata fiksne veličine.

```
int dek_na_poziciji(const Dek* d, int i)
{
    assert(d && d->seg && 0 <= i && i < d->velicina);
    assert(d->seg[d->levo] != NULL);

    const Segment* levo_seg = d->seg[d->levo];

    /* globalni indeks: krecemo od (levo, levo_seg->l) i pomeramo se za i */
    int base = d->levo * d->seg_velicina + levo_seg->l;
    int g = base + i;

    int seg_idx = g / d->seg_velicina;
    int off     = g % d->seg_velicina;

    assert(seg_idx >= 0 && seg_idx < d->seg_broj);
    assert(d->seg[seg_idx] != NULL);

    return d->seg[seg_idx]->a[off];
}
```

Ključni korak u ovoj funkciji jeste **preslikavanje logičke pozicije `i` u fizičku lokaciju** u strukturi segmenata. To se radi u dva koraka.

Prvo se određuje *globalna pozicija* prvog elementa deka u zamišljenom kontinuiranom nizu. Taj element se nalazi u segmentu sa indeksom `levo`, na poziciji `l` tog segmenta. Njegov globalni indeks je zato jednak:

$$\text{base} = levo \cdot S + l$$

gde je `S` veličina jednog segmenta.

Zatim se traženi element dobija pomeranjem za `i` mesta unapred, pa je njegova globalna pozicija:

$$g = \text{base} + i$$

Kada je globalni indeks g poznat, lako se određuje:

- **indeks segmenta** u kome se element nalazi: $seg_{idx} = g/S$
- **pozicija unutar tog segmenta**: $off = g \bmod S$

Ova dva izraza koriste celobrojno deljenje i ostatak pri deljenju, čime se u konstantnom vremenu dobija tačna lokacija elementa u strukturi.

Svi koraci u funkciji `dek_na_poziciji` izvode se u konstantnom vremenu. Ne vrši se nikakva iteracija kroz segmente, već se traženi element direktno adresira aritmetikom nad indeksima. Zbog toga je vremenska složenost ove operacije **O(1)**.

Važno je naglasiti da se ova prednost ostvaruje upravo zahvaljujući segmentnoj reprezentaciji deka i činjenici da su svi segmenti iste, fiksne veličine. U slučaju klasične povezane liste, ovakav indeksni pristup ne bi bio moguć bez linearne pretrage.

Oslobađanje memorije

Pošto je dek implementiran korišćenjem dinamički alociranih segmenata i niza pokazivača na te segmente, neophodno je obezbediti pravilno oslobađanje svih zauzetih resursa. U suprotnom bi došlo do curenja memorije, naročito u programima koji dek koriste duže vreme ili u petljama.

Oslobađanje se sprovodi na dva nivoa:

1. oslobađanje pojedinačnog segmenta,
2. oslobađanje cele strukture deka.

Oslobađanje segmenta

Svaki segment sadrži dinamički alociran niz elemenata, kao i samu strukturu segmenta. Funkcija `segment_unisti` oslobađa oba nivoa memorije. Poziv funkcije je bezbedan i u slučaju kada je prosleđen pokazivač jednak `NULL`.

```
static void segment_unisti(Segment* s)
{
    if (!s) return;
    free(s->a);
    free(s);
}
```

Ova funkcija se koristi isključivo unutar implementacije deka, jer upravljanje životnim ciklusom segmenata ne treba da bude deo javnog interfejsa strukture.

Oslobađanje celog deka

Funkcija `dek_unisti` oslobađa sve segmente koji su alocirani u mapi deka, zatim oslobađa samu mapu segmenata, i na kraju dovodi strukturu u konzistentno "prazno" stanje.

```
void dek_unisti(Dek* d)
{
    if (!d) return;

    if (d->seg) {
        for (int i = 0; i < d->seg_broj; i++) {
            segment_unisti(d->seg[i]);
            d->seg[i] = NULL;
        }
        free(d->seg);
        d->seg = NULL;
    }

    d->seg_broj = 0;
    d->seg_velicina  = 0;
    d->levo = d->desno = 0;
    d->velicina = 0;
```

```
}
```

Petlja prolazi kroz ceo niz pokazivača na segmente i za svaki od njih poziva `segment_unisti`. Nakon toga se oslobađa i sam niz pokazivača (`seg`). Na kraju se svi brojači i indeksi postavljaju na nulu, čime se sprečava upotreba strukture u nekonzistentnom stanju.

Važno je primetiti da se odgovornost za oslobađanje memorije nalazi isključivo u funkciji `dek_unisti`. Korisnik strukture ne treba (i ne sme) da ručno oslobađa segmente, već samo da pozove ovu funkciju kada dek više nije potreban. Na taj način se postiže jasna podela odgovornosti i smanjuje mogućnost grešaka u radu sa dinamičkom memorijom.

Primer upotrebe deka

Na kraju poglavlja dajemo jednostavan primer programa koji ilustruje korišćenje implementiranog deka. Program obrađuje niz naredbi sa standardnog ulaza i nad dekom izvršava osnovne operacije dodavanja, uklanjanja i indeksnog pristupa.

Primer je namenjen isključivo demonstraciji interfejsa i ponašanja strukture. Radi lakšeg praćenja, dek se inicijalizuje sa malim brojem segmenata i malom veličinom segmenta.

```c
/*
 * Podržane komande:
 *   F x  - dodaj x na početak deka
 *   B x  - dodaj x na kraj deka
 *   f    - ukloni i ispiši element sa početka
 *   b    - ukloni i ispiši element sa kraja
 *   p p  - ispiši element na poziciji p (0-indeksirano)
 */

int main(void)
{
    Dek d;

    /* inicijalno 2 segmenta, svaki segment veličine 4
       (namerno mali brojevi radi lakšeg testiranja) */
    if (!dek_init(&d, 4, 2)) {
        fprintf(stderr, "Greška: neuspešna inicijalizacija deka.\n");
        return 1;
    }

    char cmd;
    while (scanf(" %c", &cmd) == 1) {

        if (cmd == 'F') {
            int x;
            scanf("%d", &x);
            assert(dek_dodaj_na_pocetak(&d, x));

        } else if (cmd == 'B') {
            int x;
```

```
            scanf("%d", &x);
            assert(dek_dodaj_na_kraj(&d, x));

        } else if (cmd == 'f') {
            assert(!dek_prazan(&d));
            int x = dek_ukloni_sa_pocetka(&d);
            printf("%d\n", x);

        } else if (cmd == 'b') {
            assert(!dek_prazan(&d));
            int x = dek_ukloni_sa_kraja(&d);
            printf("%d\n", x);

        } else if (cmd == 'p') {
            int p;
            scanf("%d", &p);
            assert(0 <= p && p < dek_velicina(&d));
            printf("%d\n", dek_na_poziciji(&d, p));
        }
    }

    dek_unisti(&d);
    return 0;
}
```

Program pokazuje tipičan način korišćenja deka: najpre se struktura inicijalizuje, zatim se nad njom vrše operacije u skladu sa ulaznim komandama, a na kraju se zauzeta memorija pravilno oslobađa. Sve operacije koriste javni interfejs deka, čime se jasno razdvaja upotreba strukture od njene unutrašnje implementacije.

3.7 Hip i red sa prioritetom

Red sa prioritetom (engl. *priority queue*) je apstraktna struktura podataka u kojoj svaki element ima pridruženi prioritet, a operacije se ne izvršavaju po redosledu dolaska, već u skladu sa prioritetom. Tipično se razmatraju sledeće osnovne operacije:

- umetanje novog elementa u red,
- pristup elementu sa najvećim (ili najmanjim) prioritetom,
- uklanjanje elementa sa najvećim (ili najmanjim) prioritetom.

U zavisnosti od primene, red sa prioritetom može biti **maksimalni** (uvek se izdvaja element sa najvećim prioritetom) ili **minimalni** (uvek se izdvaja element sa najmanjim prioritetom).

Jedan prirodan način da se red sa prioritetom implementira jeste korišćenjem **pretraživačkog binarnog stabla**. Ako se koristi balansirano pretraživačko stablo (npr. AVL stablo ili crveno-crno stablo), sve osnovne operacije mogu se realizovati u logaritamskoj vremenskoj složenosti.

U takvoj implementaciji važi sledeće:

- umetanje nove vrednosti u stablo obavlja se u složenosti $O(\log n)$,
- element sa najvećim prioritetom odgovara maksimalnom elementu u stablu i može se pronaći spuštanjem udesno sve dok je to moguće,
- uklanjanje tog elementa takođe se izvršava u složenosti $O(\log n)$,
- dodatno, moguće je pretražiti stablo i ukloniti proizvoljnu vrednost, opet u $O(\log n)$.

Ovakav pristup je vrlo fleksibilan i moćan, ali ta fleksibilnost dolazi uz određenu cenu: implementacija je složenija, a konstantni faktori u složenosti često su veći nego što je neophodno ako se struktura koristi isključivo kao red sa prioritetom.

Ako se fokusiramo *isključivo* na tri osnovne operacije reda sa prioritetom — umetanje elementa, pristup elementu sa ekstremnim prioritetom i njegovo uklanjanje — tada se otvara mogućnost za efikasniju i jednostavniju implementaciju.

U tom slučaju se i dalje koristi binarno stablo, ali *ne pretraživačko*. Umesto toga, stablo je organizovano prema drugačijim pravilima i čini strukturu podataka koja se naziva **hip** (engl. *heap*).

Hip ne omogućava efikasno pretraživanje proizvoljnih elemenata, ali zato omogućava izuzetno efikasno upravljanje elementom sa najvećim ili najmanjim prioritetom, što ga čini idealnim za implementaciju reda sa prioritetom.

3.7.1 Osnovna svojstva hipa

Maks-hip (engl. *max-heap*) je binarno stablo koje zadovoljava sledeća dva uslova:

1. **Strukturni uslov (potpunost):** Svi nivoi stabla, osim eventualno poslednjeg, potpuno su popunjeni, dok se elementi na poslednjem nivou popunjavaju sleva nadesno.

2. **Uslov uređenosti:** Vrednost svakog čvora veća je ili jednaka vrednostima u njegovoj deci.

Analogno se definiše **min-hip**, sa razlikom da se zahteva da je vrednost svakog čvora manja ili jednaka vrednostima u njegovoj deci.

Neposredna posledica ovih uslova jeste da se u maks-hipu **maksimalni element uvek nalazi u korenu stabla**, dok se u min-hipu u korenu nalazi minimalni element. Time se pristup elementu sa ekstremnim prioritetom svodi na konstantnu vremensku složenost.

Pored efikasnog pristupa ekstremnom elementu, hip omogućava i:

- umetanje novog elementa u složenosti $O(\log n)$,
- uklanjanje korena (maksimalnog ili minimalnog elementa) u složenosti $O(\log n)$,

pri čemu se oba postupka oslanjaju na lokalne transformacije duž visine stabla.

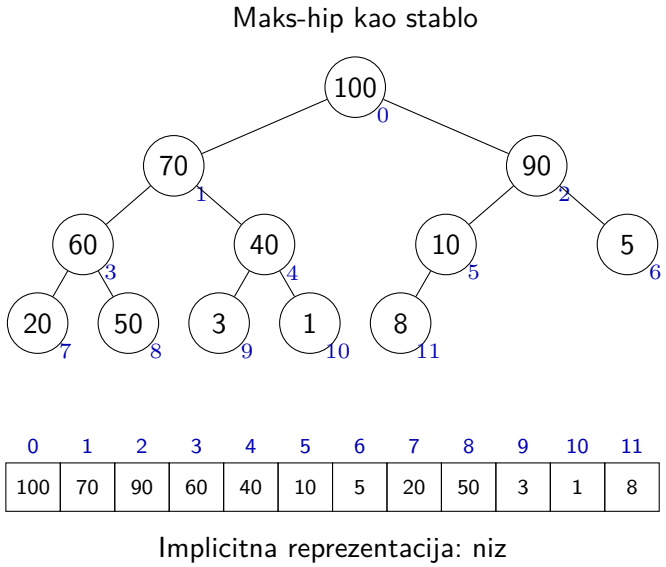
Zbog ovih osobina, hip predstavlja *standardno i najčešće korišćeno rešenje* za implementaciju reda sa prioritetom u praksi, kako u sistemskom programiranju, tako i u standardnim bibliotekama savremenih programskih jezika.

3.7.2 Primer implementacije maks-hipa

Razmotrimo implementaciju *maks-hipa*. Ideja je da se čvorovi hipa u niz upisuju redom po nivoima, sleva nadesno.

Koren stabla smešta se na poziciju 0 u nizu, njegova dva deteta na pozicije 1 i 2, zatim elementi narednog nivoa na pozicije 3, 4, 5 i 6, i tako dalje. Na taj način, struktura binarnog stabla u potpunosti je određena indeksima elemenata u nizu.

Na slici je prikazan jedan maks-hip i njegov odgovarajući niz, koji predstavlja upravo ovakav raspored čvorova po nivoima.



Slika 3.3: Primer hipa.

Hip mora zadovoljiti **uslov potpunosti**: svi nivoi stabla, osim eventualno poslednjeg, moraju biti potpuno popunjeni, dok se na poslednjem nivou čvorovi popunjavaju sleva nadesno, bez "rupa".

Upravo ovaj uslov omogućava da se hip prirodno i efikasno mapira na niz. Zahvaljujući tome, u implementaciji nije potrebno čuvati eksplicitne veze između čvorova — sve relacije roditelj–dete dobijaju se jednostavnom indeksnom aritmetikom.

Neka se čvor hipa nalazi na poziciji `i` u nizu. Tada važe sledeće relacije:

- **levo dete** se nalazi na poziciji $2i + 1$,
- **desno dete** se nalazi na poziciji $2i + 2$,
- **roditelj čvora** se nalazi na poziciji

$$\left\lfloor \frac{i-1}{2} \right\rfloor.$$

Ove formule direktno proizlaze iz načina na koji se elementi hipa smeštaju u niz i važe za svaki čvor osim korena.

Jedno od ključnih svojstava maks-hipa jeste da se **najveći element uvek nalazi u korenu stabla**, odnosno na poziciji 0 u nizu. Ovo svojstvo lako se dokazuje indukcijom po visini stabla, imajući u vidu da je vrednost u svakom čvoru veća ili jednaka vrednostima u njegovoj deci.

Zbog toga je operacija pristupa maksimalnom elementu izuzetno efikasna — vreme potrebno za nalaženje maksimuma iznosi $O(1)$.

Struktura podataka za maks-hip

U implementaciji maks-hipa koristi se sledeća struktura podataka:

```
typedef struct {
    int*    a;        /* niz elemenata */
    size_t  broj;     /* broj elemenata u hipu */
    size_t  kapacitet;/* alocirani kapacitet niza */
} Hip;
```

Struktura `Hip` enkapsulira sve informacije potrebne za rad sa hipom. Binarno stablo se ne čuva eksplicitno pomoću čvorova i pokazivača, već je **implicitno predstavljeno nizom** celobrojnih vrednosti.

Pokazivač `a` upućuje na dinamički alociran niz u kojem su smešteni elementi hipa. Polje `broj` označava trenutni broj elemenata u hipu, dok `kapacitet` predstavlja veličinu alociranog prostora u nizu. Razdvajanje ova dva pojma omogućava efikasno ubacivanje novih elemenata bez česte realokacije memorije.

Indeksne relacije u implicitnoj reprezentaciji

Pošto je hip smešten u niz, odnosi roditelj–dete između čvorova stabla određeni su isključivo njihovim indeksima. U implementaciji se koriste sledeće pomoćne funkcije:

```
static size_t roditelj(size_t i) { return (i - 1) / 2; }
static size_t levo_dete(size_t i) { return 2 * i + 1; }
static size_t desno_dete(size_t i) { return 2 * i + 2; }
```

Inicijalizacija hipa

Pre prve upotrebe hip mora biti inicijalizovan:

```
void hip_inicijalizuj(Hip* h) {
    h->a = NULL;
    h->broj = 0;
    h->kapacitet = 0;
}
```

Inicijalizacijom se hip dovodi u dosledno početno stanje: niz elemenata još nije alociran, broj elemenata je nula, a kapacitet niza jednak nuli. Dinamička memorija će biti alocirana tek prilikom prvog ubacivanja elementa.

Čišćenje i uništavanje hipa

Tokom rada može biti potrebno ukloniti sve elemente iz hipa, ili potpuno osloboditi zauzetu memoriju.

```
void hip_ocisti(Hip* h) {
    if (!h) return;
    h->broj = 0;
}
```

Funkcija `hip_ocisti` uklanja sve elemente iz hipa tako što postavlja broj elemenata na nulu, ali **ne oslobađa alocirani niz**. Ovo omogućava da se hip ponovo koristi bez dodatnih troškova alokacije.

```
void hip_unisti(Hip* h) {
    if (!h) return;
    free(h->a);
    h->a = NULL;
    h->broj = 0;
    h->kapacitet = 0;
}
```

Funkcija `hip_unisti` oslobađa svu dinamički alociranu memoriju i vraća hip u stanje pre inicijalizacije. Ovu funkciju je potrebno pozvati kada se hip više neće koristiti.

Osnovne informativne operacije

Za potrebe kontrole toka programa često je neophodno znati koliko elemenata hip sadrži ili da li je prazan.

```
size_t hip_velicina(const Hip* h) {
    return h ? h->broj : 0;
}

int hip_je_prazan(const Hip* h) {
    return !h || h->broj == 0;
}
```

Ove funkcije ne menjaju sadržaj hipa i izvršavaju se u konstantnom vremenu. One predstavljaju osnovni interfejs za proveru stanja strukture podataka.

Umetanje elementa u maks-hip

Umetanje novog elementa u maks-hip zasniva se na očuvanju dva osnovna svojstva strukture:

- oblik stabla se ne menja,
- uslov hipa se, ukoliko je potrebno, popravlja lokalnim razmenama.

Novi element se najpre upisuje **na kraj niza**, odnosno na prvo slobodno mesto na poslednjem nivou potpunog binarnog stabla. Time se obezbeđuje da *uslov potpunosti* ostane zadovoljen u svakom trenutku.

Nakon ovog upisa, međutim, *uslov maks-hipa ne mora biti ispunjen*. Naime, može se desiti da je novoubačeni element veći od svog roditelja, čime se narušava zahtev da je vrednost u svakom čvoru veća ili jednaka vrednostima u njegovoj deci. Zbog toga je, po potrebi, neophodno izvršiti dodatnu popravku strukture.

Pošto se hip realizuje pomoću *dinamičkog niza*, pre upisa novog elementa proverava se da li u nizu postoji dovoljno slobodnog prostora. Ako kapacitet nije dovoljan, niz se realocira na veći kapacitet. Pri tome se kapacitet uvećava eksponencijalno (najčešće dupliranjem), čime se postiže dobra amortizovana efikasnost pri većem broju uzastopnih umetanja.

Važno je naglasiti sledeće:

- realokacija memorije može da ne uspe,
- u tom slučaju se postupak umetanja prekida,
- sadržaj hipa ostaje *neizmenjen*, čime se čuva konzistentnost strukture.

```
static int obezbedi_kapacitet(Hip* h, size_t potreban) {
    if (potreban <= h->kapacitet) return 1;

    size_t novi = (h->kapacitet == 0) ? 8 : h->kapacitet;
    while (novi < potreban) {
        novi *= 2;
    }

    int* p = (int*)realloc(h->a, novi * sizeof(int));
    if (!p) return 0;

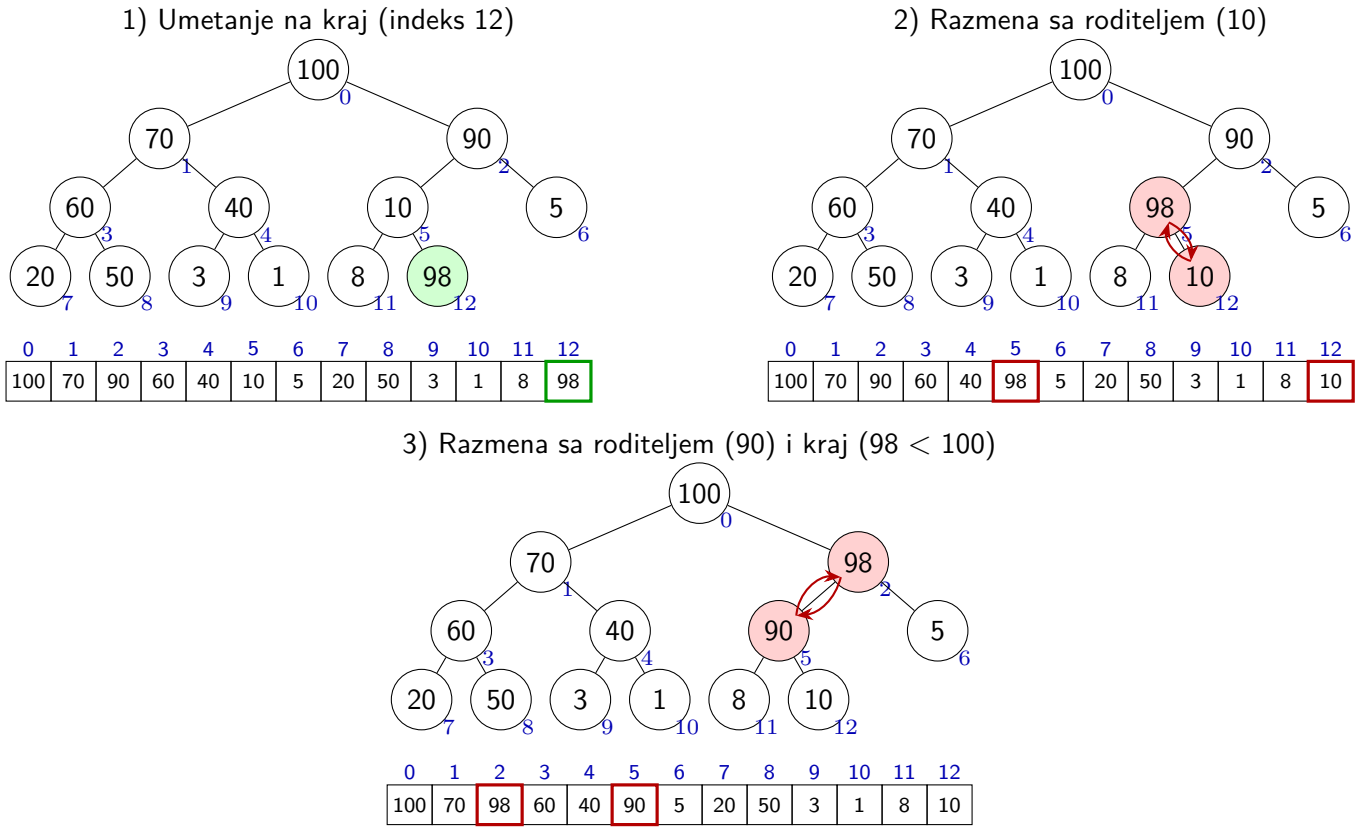
    h->a = p;
    h->kapacitet = novi;
    return 1;
}
```

Nakon upisa novog elementa na kraj niza, hip se po potrebi popravlja **pomeranjem elementa naviše**. Ako je novoubačeni element veći od svog roditelja, vrši se njihova razmena.

Važno je uočiti da se nakon ove razmene narušavanje uslova hipa ne širi na ceo hip, već ostaje ograničeno isključivo na putanju ka korenu. Pre razmene, novoubačeni element se nalazi u listu, pa samim tim nema decu i ne može narušiti uslov hipa u odnosu na svoju decu. Istovremeno, roditelj novog elementa je pre umetanja već zadovoljavao uslov hipa u odnosu na *svoje drugo dete*, kao i u odnosu na ostatak podstabla.

Nakon razmene dolazi do sledećih promena:

- novoubačeni element prelazi na roditeljsku poziciju,
- na toj poziciji postaje roditelj istom podstablu koje je prethodno pripadalo starom roditelju,
- pošto je veći od starog roditelja, a stari roditelj je bio veći ili jednak svojoj deci, sledi da je novoubačeni element **veći ili jednak i toj deci**.



Slika 3.4: Ilustracija umetanja u hip i pomeranje elementa naviše.

Zbog toga se uslov maks-hipa u odnosu na decu na toj strani ne kvari. Jedino mesto na kojem uslov hipa može biti narušen posle razmene jeste odnos novog elementa sa njegovim *novim roditeljem*, koji se nalazi jedan nivo iznad.

Razmene se zatim ponavljaju na isti način. Postupak se završava u jednom od sledećih slučajeva:

- element stigne do korena hipa,
- element postane manji ili jednak svom roditelju.

Ovakav postupak ima *induktivno-rekurzivnu strukturu*: nakon svake razmene, podstablo ispod razmatranog elementa ostaje ispravan hip, dok eventualno narušavanje može postojati samo iznad njega. Time se problem u svakom koraku svodi na isti problem, ali na stablu manje visine.

Sama funkcija umetanja ima sledeći oblik:

```
int hip_ubaci(Hip* h, int x) {
    if (!h) return 0;

    if (!obezbedi_kapacitet(h, h->broj + 1))
        return 0;

    h->a[h->broj] = x;
    pomeri_navise(h, h->broj);
    h->broj++;
```

```
    return 1;
}
```

Pomeranje naviše realizuje se iterativno:

```
static void pomeri_navise(Hip* h, size_t i) {
    while (i > 0) {
        size_t r = roditelj(i);
        if (h->a[r] >= h->a[i]) break;
        zameni(&h->a[r], &h->a[i]);
        i = r;
    }
}
```

U svakom koraku novoubačeni element se poredi sa svojim roditeljem i, ukoliko je veći, razmenjuje se sa njim i pomera za tačno jedan nivo naviše. Kako se u jednoj iteraciji prelazi upravo jedan nivo u stablu, ukupan broj takvih iteracija ograničen je visinom hipa.

Za hip koji sadrži n elemenata, visina stabla iznosi

$$h = \lfloor \log_2 n \rfloor,$$

pošto je hip potpuno binarno stablo. Zbog toga se u najnepovoljnijem slučaju može izvršiti najviše $O(\log n)$ razmena. Svaka pojedinačna razmena i odgovarajuće poređenje izvode se u konstantnom vremenu, pa ukupno vreme potrebno za umetanje novog elementa u maks-hip iznosi $O(\log n)$.

Brisanje elementa iz maks-hipa

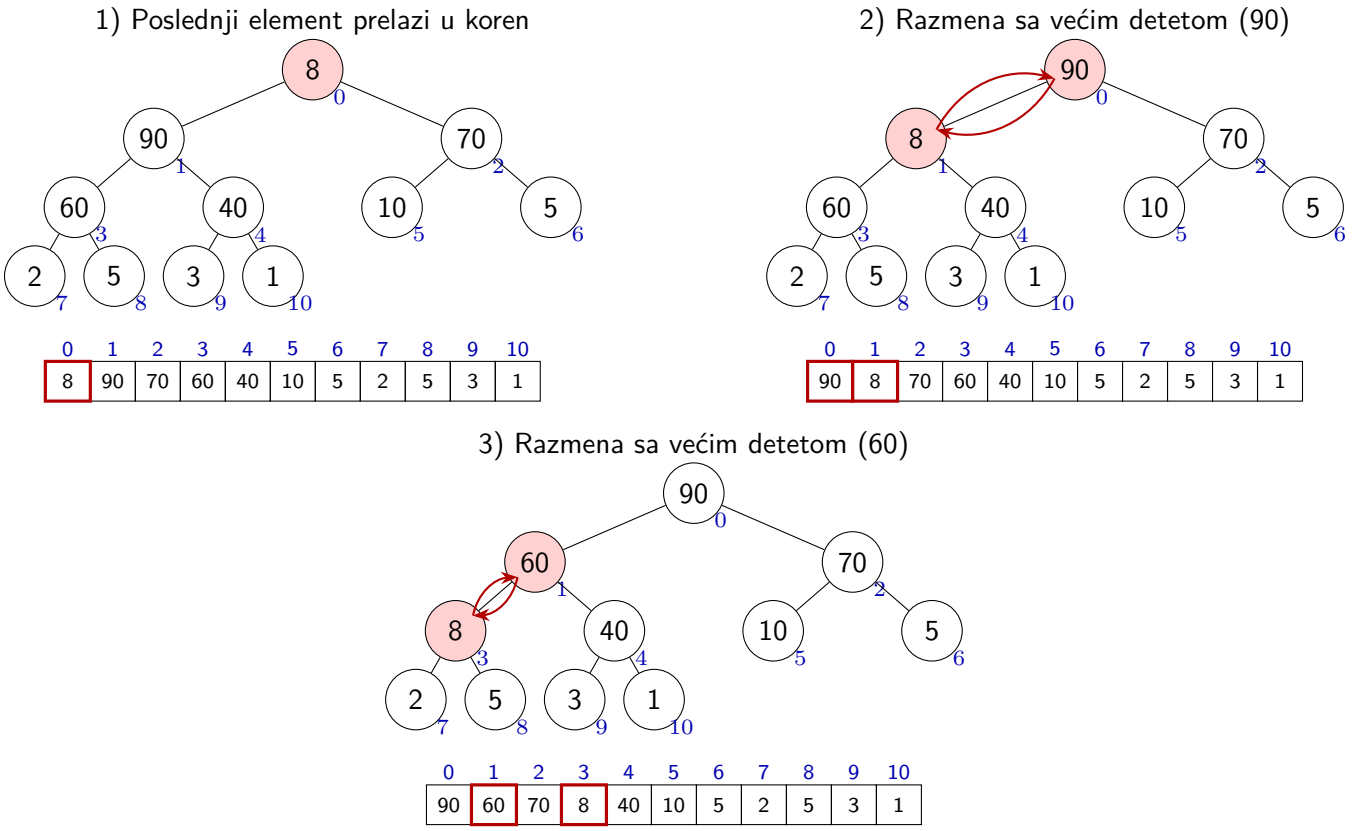
Izbacivanje elementa sa najvećim prioritetom iz maks-hipa zasniva se na simetričnom principu u odnosu na umetanje. Element sa najvećom vrednošću uvek se nalazi u korenu hipa, odnosno na početku niza, pa se upravo on uklanja.

Pošto se oblik stabla ne sme narušiti, uklanjanje korena ne može se izvršiti prostim izbacivanjem tog elementa. Umesto toga, **poslednji element u nizu premešta se na poziciju korena**. Time se zadržava uslov potpunosti stabla, jer se uklanja upravo desni list na najnižem nivou, a stablo ostaje kompaktno.

Nakon ovog premeštanja, međutim, *uslov maks-hipa može biti narušen*. Element koji je premešten u koren može biti manji od jednog ili oba svoja deteta, čime se narušava zahtev da je vrednost u svakom čvoru veća ili jednaka vrednostima u njegovoj deci. Zbog toga je neophodno izvršiti popravku strukture pomeranjem elementa naniže.

Popravka se sprovodi na sledeći način. Posmatra se čvor u kome se nalazi element i njegova deca. Ukoliko je vrednost u tom čvoru manja od vrednosti nekog deteta, **bira se dete sa većom vrednošću i vrši se razmena**. Na taj način se veći element podiže, a manji spušta za jedan nivo naniže. Nakon razmene, uslov maks-hipa je ponovo zadovoljen za čvor koji se pomerio naviše, dok se eventualno narušavanje može pojaviti samo niže u stablu.

Ovaj postupak se ponavlja sve dok element ne stigne do pozicije na kojoj je veći ili jednak oba svoja deteta, ili dok ne postane list. Važno je uočiti da se, kao i kod umetanja, narušavanje uslova hipa lokalizuje duž jedne putanje*, ovoga puta od korena ka listovima, dok ostatak stabla u svakom trenutku ostaje ispravan hip.



Slika 3.5: Ilustracija brisanja iz hipa i pomeranja elementa naniže.

Sama operacija izbacivanja realizovana je funkcijom:

```
int hip_izbaci_vrh(Hip* h, int* vrednost) {
    if (!h || h->broj == 0) return 0;

    if (vrednost) *vrednost = h->a[0];

    h->broj--;
    if (h->broj > 0) {
        h->a[0] = h->a[h->broj];
        pomeri_nanize(h, 0);
    }
    return 1;
}
```

Funkcija najpre, po potrebi, čuva vrednost elementa koji se uklanja. Zatim se broj elemenata umanjuje, a poslednji element u nizu premešta na poziciju korena. Ako hip nakon toga nije prazan, poziva se funkcija `pomeri_nanize`, koja obavlja popravku strukture.

Pomeranje naniže realizuje se iterativno:

```
static void pomeri_nanize(Hip* h, size_t i) {
    for (;;) {
```

```
        size_t l = levo_dete(i);
        size_t d = desno_dete(i);
        size_t najveci = i;

        if (l < h->broj && h->a[l] > h->a[najveci])
            najveci = l;
        if (d < h->broj && h->a[d] > h->a[najveci])
            najveci = d;

        if (najveci == i) break;

        zameni(&h->a[i], &h->a[najveci]);
        i = najveci;
    }
}
```

U svakom koraku bira se dete sa najvećom vrednošću, čime se obezbeđuje da se veći element podigne ka vrhu, dok se manji element pomera naniže. Pošto se u jednoj iteraciji prelazi tačno jedan nivo u stablu, ukupan broj iteracija ograničen je visinom hipa. Kako je hip potpuno binarno stablo, njegova visina iznosi $O(\log n)$, pa je i složenost izbacivanja elementa iz maks-hipa jednaka $O(\log n)$.

3.7.3 Hipsort

Sortiranje brojeva može biti realizovano direktnom implementacijom algoritma *heap sort*, u kome se red sa prioritetom realizuje pomoću ručno implementiranog hipa. Osnovna ideja algoritma je da se **od polaznog niza formira maks-hip**, a zatim da se **elementi jedan po jedan izbacuju iz hipa i smeštaju na kraj niza**, čime se dobija sortiran niz.

Važna osobina heap sort-a jeste to što se isti niz (ili vektor) može koristiti istovremeno za: polazne podatke, predstavljanje hipa i smeštanje sortiranog rezultata. Konkretno, elementi koji čine hip smeštaju se u početni deo niza, dok se sortirani deo niza postepeno gradi u njegovom krajnjem delu. Na ovaj način se postiže sortiranje *u mestu* (eng. in-place), bez korišćenja dodatne memorije.

Ovakva realizacija zahteva blago prilagođavanje funkcija za rad sa hipom. Naime, tokom izvršavanja algoritma, stvarni broj elemenata koji pripadaju hipu se smanjuje, jer se sortirani elementi izdvajaju na kraj niza. Zbog toga funkcija za pomeranje naniže, pored samog niza, mora kao parametar da primi i broj elemenata koji trenutno čine hip. Na osnovu tog broja jednostavno se određuje koji elementi niza pripadaju hipu, a koji pripadaju već sortiranom delu. Shodno tome, poređenje sa stvarnom dimenzijom niza zamenjuje se poređenjem sa ovim parametrom.

U nastavku će biti prikazane dve varijante algoritma heap sort. One se razlikuju isključivo po načinu formiranja početnog hipa. Prva varijanta gradi hip postepenim umetanjem elemenata i ima veću vremensku složenost, dok je druga varijanta efikasnija . Obe varijante su korektne i biće detaljno ilustrovane, ali će se pokazati da je druga varijanta u praksi i teoriji povoljnija.

Formiranje hipa naniže (Vilijamsov metod)

Jedan način da se od niza formira hip (u istom memorijskom prostoru) jeste da se krene od praznog hipa i da se elementi ubacuju jedan po jedan. Nakon ubacivanja svakog novog elementa, on se pomera naviše dok se ne uspostavi uslov maks-hipa. Ovaj pristup se naziva *formiranje hipa naniže* ili *Vilijamsov metod*.

```
static void napravi_hip_sporo(int a[], int n)
{
    for (int i = 1; i < n; i++)
        pomeri_navise(a, i);
}

static void pomeri_navise(int a[], int i)
{
    while (i > 0) {
        int roditelj = (i - 1) / 2;
        if (a[i] <= a[roditelj])
            break;
        swap(&a[i], &a[roditelj]);
        i = roditelj;
    }
}
```

Invarijanta spoljne petlje (tj. induktivna hipoteza) jeste da elementi na pozicijama $[0, i)$ čine ispravan maks-hip. U svakoj iteraciji se element na poziciji i umeće u već postojeći hip primenom operacije pomeranja naviše. Pošto je poznato da ova operacija korektno umeće novi element u maks-hip, invarijanta ostaje očuvana.

Na kraju petlje važi $i = n$, što znači da svi elementi niza, odnosno elementi na pozicijama $[0, n)$, čine maks-hip. Dodatno, tokom izvođenja algoritma vrše se isključivo razmene elemenata, pa se multiskup elemenata u hipu ne menja i ostaje jednak multiskupu elemenata polaznog niza.

Složenost najgoreg slučaja formiranja hipa ovim metodom iznosi $\Theta(n \log n)$. Naime, složenost operacije umetanja, odnosno pomeranja elementa naviše u hipu koji ima k elemenata, jednaka je $O(\log k)$. Ukupna složenost formiranja hipa zato odgovara zbiru $\log 1 + \log 2 + \ldots + \log n$, što je, kako je od ranije poznato, asimptotski jednako $\Theta(n \log n)$.

Formiranje hipa naviše (Flojdov metod)

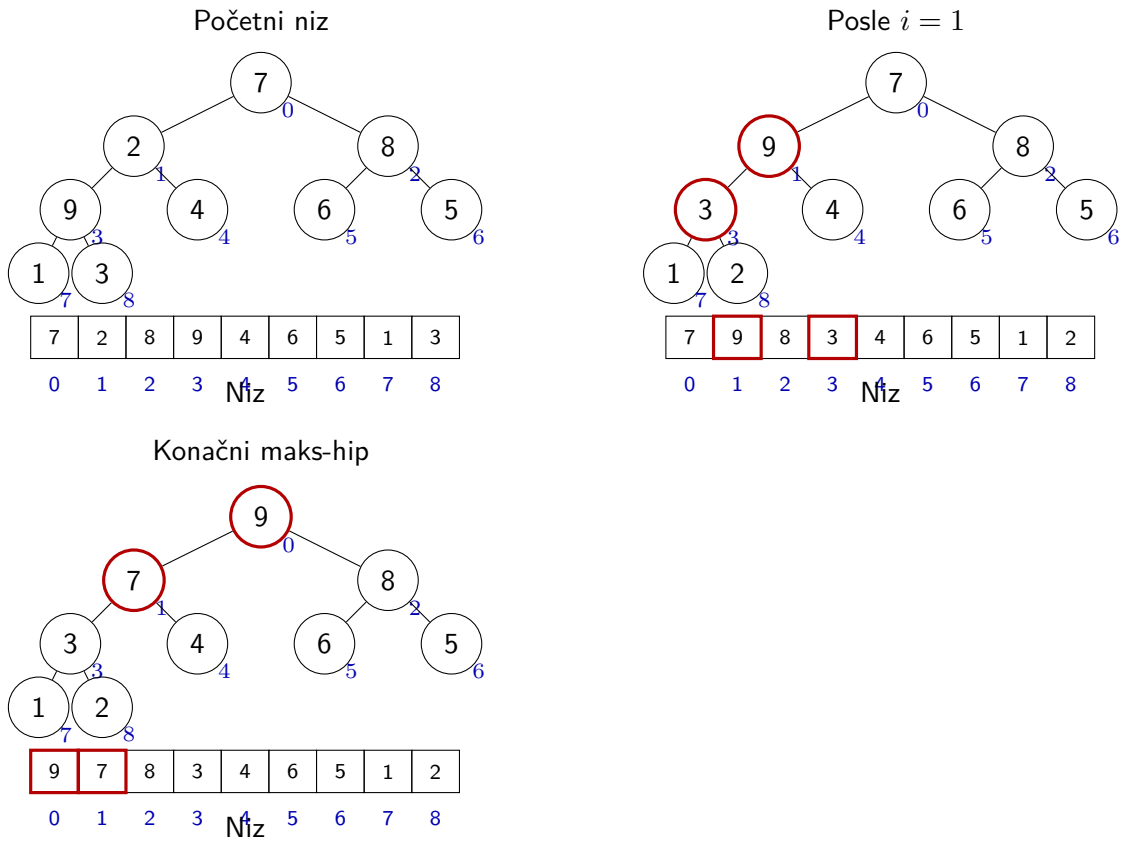
Drugi, efikasniji način formiranja hipa naziva se *formiranje hipa naviše* ili *Flojdov metod*. Osnovna ideja je da se elementi originalnog niza obilaze od pozadi i da se svaki element „ubaci“ u hip čiji je on koren, tako što se po potrebi spušta naniže kroz hip.

Induktivna hipoteza u ovom pristupu glasi: svi elementi iz intervala (i, n) predstavljaju korene ispravnih maks-hipova.

Na primer, ako je dat niz `7 2 8 9 4 6 5 1 3` svi elementi osim prva dva čine korene ispravnih hipova. Proces formiranja hipa za ovaj primer ilustrovan je donjom slikom. Elementi `4, 6, 5, 1` i `3` nemaju potomke i trivijalno predstavljaju ispravne jednočlane hipove. Uopšteno, svi elementi u drugoj polovini niza predstavljaju listove i za njih je induktivna hipoteza trivijalno ispunjena. Dalje, element `9` je veći od oba svoja potomka, pa je koren ispravnog maks-hipa. Isto važi i za element `8`.

Postavlja se pitanje kako proširiti induktivnu hipotezu. Element na poziciji i ne mora biti veći od svojih potomaka, ali znamo da su oba njegova deteta već koreni ispravnih hipova. Zbog toga je dovoljno element na poziciji i spustiti naniže dok se ne uspostavi uslov maks-hipa. Upravo ovu operaciju smo ranije razmatrali prilikom brisanja elementa iz hipa.

Dakle, svaki element, počev od pozicije $\lfloor \frac{n}{2} \rfloor - 1$ pa unazad do pozicije `0`, spušta se naniže koliko je potrebno, tako da hip čiji je on koren postane ispravan.



Slika 3.6: Ilustracija kreiranja hipa naviše od početnog niza.

U navedenom primeru, element 2 se najpre razmenjuje sa svojim većim sinom 9, a zatim se po potrebi spušta dalje niz stablo.

Operacija pomeranja naniže implementirana je na sledeći način:

```
/* pomeranje naniže (sift down) */
static void pomeri_nanize(int a[], int i, int n)
{
    while (1) {
        int najveci = i;
        int levi  = 2 * i + 1;
        int desni = 2 * i + 2;

        if (levi < n && a[levi] > a[najveci])
            najveci = levi;
        if (desni < n && a[desni] > a[najveci])
            najveci = desni;

        if (najveci == i)
            break;

        swap(&a[i], &a[najveci]);
        i = najveci;
```

```
    }
}
```

Na osnovu ove operacije, hip se formira sledećom funkcijom:

```
static void napravi_hip_brzo(int a[], int n)
{
    for (int i = n / 2 - 1; i >= 0; i--)
        pomeri_nanize(a, i, n);
}
```

Na prvi pogled, pošto operacija pomeranja naniže ima složenost $O(\log n)$, moglo bi se pomisliti da je ukupna složenost formiranja hipa $\Theta(n \log n)$. Međutim, to nije slučaj.

Ključna razlika u odnosu na Vilijamsov metod leži u tome što se ne spuštaju svi elementi istom visinom. Ako posmatramo potpun binarni hip visine i, postoji:

- jedan čvor koji se može spustiti visinom i,
- dva čvora koja se mogu spustiti visinom $i-1$,
- četiri čvora koja se mogu spustiti visinom $i-2$,
- osam čvorova se mogu spustiti visinom $i-3$
- i tako dalje.

Ukupan broj koraka odgovara zbiru visina svih čvorova u stablu. Neka je $H(i)$ zbir visina svih čvorova potpunog binarnog stabla visine i. Tada važi rekurentna relacija

$$H(i) = 2H(i-1) + i, H(0) = 0.$$

Razmotavanjem ove relacije dobija se

$$H(i) = 2^{i+1} - i - 2.$$

Pošto je za potpuno binarno stablo broj elemenata $n = 2^{i-1}$, sledi da je

$$H(i) = \Theta(n).$$

Prema tome, ukupna složenost formiranja hipa naviše iznosi $\Theta(n)$, što je asimptotski bolje od Vilijamsovog metoda.

Intuitivno objašnjenje ove činjenice je sledeće: kod formiranja hipa naniže čak za polovinu elemenata (svi listovi) se može desiti da se moraju penjati skoro celom visinom stabla. Nasuprot tome, kod formiranja hipa naviše samo se mali broj elemenata pomera za veći broj nivoa, dok se većina elemenata pomera veoma malo ili uopšte ne.

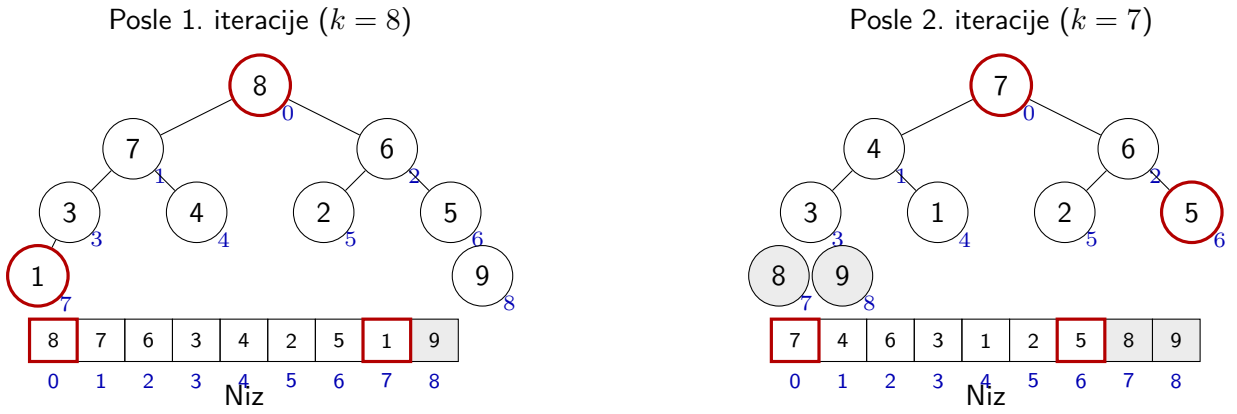
Završna faza: sortiranje pomoću hipa

Završna faza heap sort algoritma sprovodi se vađenjem elemenata iz hipa, nakon što je hip prethodno formiran. Tokom ove faze održava se sledeća invarijanta:

- elementi na pozicijama $[0, i]$ čine ispravno formiran maks-hip,
- elementi u intervalu (i, n) su sortirani i svi su veći ili jednaki od elemenata koji se trenutno nalaze u hipu.

Na početku važi $i = n-1$, pa je invarijanta trivijalno zadovoljena: elementi na pozicijama $[0, n-1]$ čine ispravan hip, dok je interval $(n-1, n)$ prazan.

U svakom koraku algoritma najveći element hipa (element na poziciji 0) razmenjuje se sa elementom na poziciji i, čime se taj najveći element izdvaja iz hipa i dodaje na početak sortiranog dela niza. Element koji je razmenom dospeo na vrh hipa može narušiti uslov hipa, pa se on pomera naniže sve dok se uslov ponovo ne uspostavi. Nakon toga se vrednost i smanjuje za jedan, čime se granica između hipa i sortiranog dela niza pravilno pomera: hip postaje kraći za jedan element, a sortirani deo niza duži za jedan element.



Slika 3.7: Ilustracija 2 koraka sortiranja nakon kreiranja ispravnog hipa. Na obe slike, deo niza koji je sortiran i više se ne razmatra, označen je sivo. Na prvoj slici, 1 i 8 će biti zamenjeni. Nakon toga 1 se spušta na odgovarajuću poziciju radi očuvanja svojstva hipa. Na drugoj slici, 7 i 5 će zameniti mesta.

Kada se petlja završi, važi $i = 0$. Iz invarijante tada neposredno sledi da je niz ispravno sortiran: svi elementi u intervalu $(0, n)$ su sortirani i veći ili jednaki od elementa na poziciji 0. Pošto se tokom algoritma vrše isključivo razmene elemenata, multiskup elemenata niza ostaje nepromenjen.

Implementacija završne faze heap sort algoritma, zajedno sa formiranjem hipa, data je sledećim kodom:

```
void heap_sort(int a[], int n)
{
    napravi_hip_brzo(a, n);

    for (int k = n - 1; k > 0; k--) {
        swap(&a[0], &a[k]);
        pomeri_nanize(a, 0, k);
    }
```

```
}
```

Ukoliko se hip formira Vilijamsovim metodom (postepenim umetanjem elemenata), ukupna složenost heap sort algoritma iznosi $\Theta(n \log n)$, jer i formiranje hipa i završna faza imaju složenost $\Theta(n \log n)$. Sa druge strane, upotrebom Flojdovog metoda za formiranje hipa, složenost formiranja hipa smanjuje se na $\Theta(n)$, dok završna faza i dalje ima složenost $\Theta(n \log n)$. Zbog toga ukupna složenost heap sort algoritma ostaje $\Theta(n \log n)$, ali sa boljom konstantom i efikasnijom početnom fazom, što Flojdov metod čini standardnim izborom u praksi.

Glava 4

Asocijativne strukture podataka

4.1 Binarna stabla

Stablo (ili **drvo**) je struktura podataka koja prirodno opisuje hijerarhijske odnose između objekata. Tipični primeri su porodična stabla, logički i aritmetički izrazi, organizacija fajlova u operativnim sistemima i slični hijerarhijski modeli.

Stablo se sastoji od **čvorova** i **usmerenih grana** između njih. Svaka grana povezuje jedan čvor, koji se u tom kontekstu naziva **roditelj**, sa njegovim **detetom** (neposrednim potomkom). Čvor koji nema roditelja naziva se **koren stabla**. Svaki drugi čvor ima tačno jednog roditelja.

Čvor koji nema potomaka naziva se **list**. Za čvorove (A) i (B) kažemo da je (A) *predak* čvora (B) ako je (A) roditelj čvora (B), ili ako je (A) predak roditelja čvora (B). Koren stabla je predak svim čvorovima stabla (osim samom sebi). Zbog toga je do svakog čvora stabla moguće doći iz korena, i to jedinstveno određenim putem.

Jedna od važnih osobina stabala jeste da se novi čvorovi jednostavno dodaju kao deca postojećih čvorova, čime se struktura prirodno proširuje.

Stablo u kome svaki čvor ima **najviše dva deteta** naziva se **binarno stablo** (ili **binarno drvo**). Ta dva deteta se uobičajeno razlikuju kao **levo dete** i **desno dete**. U nastavku će se razmatrati isključivo binarna stabla.

Binarna stabla se najčešće implementiraju korišćenjem **dinamičke alokacije memorije** i struktura koje sadrže **pokazivače**, analogno povezanim listama. Svaki čvor sadrži podatke, kao i pokazivače ka levom i desnom podstablu.

Podaci u čvorovima stabla mogu biti proizvoljnog tipa i često ih ima više. Radi jednostavnosti, u narednim primerima smatraćemo da čvor stabla sadrži samo jedan podatak tipa `int`. Struktura čvora može se deklarisati na sledeći način:

```
typedef struct cvor_stabla {
    int podatak;
    struct cvor_stabla* levo;
    struct cvor_stabla* desno;
} cvor_stabla;
```

Elementi stabla kreiraju se pojedinačno, a zatim se povezuju sa postojećim stablom. Kreiranje novog čvora stabla analogno je kreiranju novog elementa povezane liste.

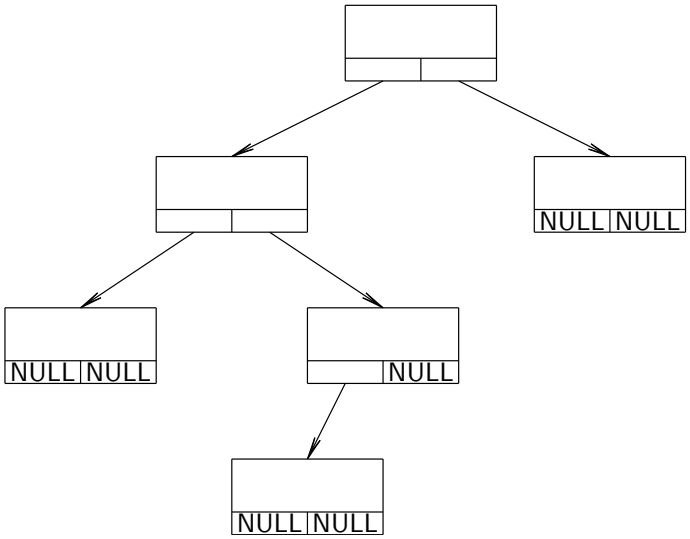
Funkcija za kreiranje novog čvora može imati sledeći oblik:

```
cvor_stabla* novi_cvor_stabla(int podatak)
{
    cvor_stabla* novi = malloc(sizeof(cvor_stabla));
    if (novi == NULL)
        return NULL;

    novi->podatak = podatak;
    novi->levo = NULL;
    novi->desno = NULL;

    return novi;
}
```

Ova funkcija alocira memoriju za novi čvor, inicijalizuje njegov podatak i postavlja pokazivače na levog i desnog potomka na NULL, čime se dobija čvor koji predstavlja list stabla.



Slika 4.1: Binarno stablo

4.1.1 Uređeno binarno stablo

U **uređenom binarnom stablu**[1] podaci su organizovani u odnosu na unapred izabranu **relaciju poretka**. Na primer, za numeričke podatke to može biti uobičajeno rastuće ili opadajuće uređenje. U nastavku će se termini *manje* i *veće* uvek odnositi na tu fiksiranu relaciju poretka, definisanu za konkretno binarno stablo.

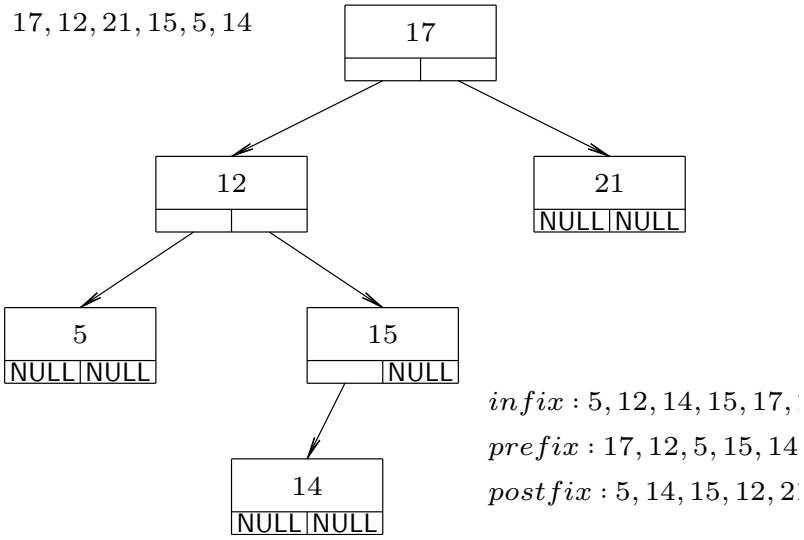
Za svaki čvor n uređenog binarnog stabla važi sledeće svojstvo:

- vrednosti podataka svih čvorova u njegovom **levom podstablu** manje su ili jednake vrednosti podatka u čvoru n;

[1] Uređeno binarno stablo naziva se i **binarno pretraživačko stablo** ili **stablo binarne pretrage**, a u engleskoj literaturi *binary search tree (BST)*.

- vrednosti podataka svih čvorova u njegovom **desnom podstablu** veće su od vrednosti podatka u čvoru n.

Ovo svojstvo uređenosti ilustrovano je na slici *uređeno binarno stablo*.



Slika 4.2: Uređeno binarno stablo

Organizacija podataka u uređenom binarnom stablu omogućava efikasne operacije pretrage, po istom principu kao i **binarna pretraga** nad nizom. Međutim, ova efikasnost se gubi ukoliko stablo postane **degenerisano**, na primer u slučaju kada svaki čvor ima samo desnog neposrednog potomka. U takvim situacijama, kao i kod stabala koja su *skoro degenerisana*, složenost osnovnih operacija — dodavanja i pronalaženja elemenata — postaje linearna u odnosu na broj čvorova stabla.

Postoje klase **balansiranih binarnih stabala** kod kojih se posebnom, netrivijalnom logikom održava ograničenje nad razlikom visina levog i desnog podstabla u svakom čvoru. Tipično se zahteva da se te visine razlikuju najviše za jedan. Primeri takvih struktura su **AVL stabla** i **crveno-crna stabla**. U ovim strukturama složenost operacija dodavanja, brisanja i pretrage elemenata je logaritamska u odnosu na broj elemenata stabla.

Uređeno binarno stablo ima veliki broj primena, od kojih će neke biti opisane u nastavku.

Dodavanje elementa u uređeno binarno stablo

Uređeno binarno stablo se konstruiše postepenim dodavanjem ulaznih vrednosti, jednu po jednu. Od prve ulazne vrednosti formira se *koren stabla*, kojem se ta vrednost pridružuje. Svaka naredna vrednost v umeće se u stablo na jedinstveno određenu ispravnu poziciju, u skladu sa pravilom uređenosti.

Rekurzivni postupak umetanja može se opisati na sledeći način:

- ako je stablo prazno, rezultat je stablo koje se sastoji od jednog čvora;
- ako je vrednost koja se umeće manja od ili jednaka vrednosti u korenu, umetanje se vrši rekurzivno u levo podstablo;
- u suprotnom slučaju, umetanje se vrši rekurzivno u desno podstablo.

Ovakav postupak omogućava višestruka pojavljivanja istih vrednosti u stablu. Ako je vrednost koja se umeće jednaka vrednosti u čvoru, ona se smešta u levo podstablo, čime se u stablu čuva sortirani *multiskup* elemenata.

Ukoliko je cilj implementacija *skupa* (bez ponavljanja elemenata), postupak se može jednostavno prilagoditi: pri pokušaju umetanja vrednosti jednake vrednosti u čvoru, dalji postupak se prekida i stablo ostaje neizmenjeno.

Rekurzivna implementacija

Sledeća rekurzivna funkcija kreira novi čvor sa zadatim podatkom `podatak` i umeće ga u uređeno binarno stablo. Funkcija prima pokazivač na koren stabla, jer se koren može promeniti u slučaju da je stablo inicijalno prazno.

Povratna vrednost funkcije je status koji označava uspeh ili neuspeh operacije.

```
typedef enum {
    OK,
    NEDOVOLJNO_MEMORIJE,
    NEMA_ELEMENATA
} status;

/* ... */

status dodaj_u_stablo(cvor_stabla** pokazivac_na_koren,
                      int podatak)
{
    if (*pokazivac_na_koren == NULL) {
        cvor_stabla* novi = novi_cvor_stabla(podatak);
        if (novi == NULL)
            return NEDOVOLJNO_MEMORIJE;

        novi->levo = NULL;
        novi->desno = NULL;
        *pokazivac_na_koren = novi;
        return OK;
    }

    if (podatak <= (*pokazivac_na_koren)->podatak)
        return dodaj_u_stablo(&((*pokazivac_na_koren)->levo), podatak);
    else
        return dodaj_u_stablo(&((*pokazivac_na_koren)->desno), podatak);
}
```

Ova rekurzija je repna i može se relativno lako eliminisati. Međutim, u slučaju kada se elementi umeću u već uređenom redosledu, svi novi čvorovi se smeštaju na jednu stranu stabla, što može dovesti do *degeneracije stabla u listu*. U tom slučaju, složenost operacija u najgorem slučaju postaje linearna u odnosu na broj elemenata stabla.

Zbog toga postoji i realna opasnost od *prekoračenja programskog steka* pri dubokoj rekurziji. Kod samobalansirajućih stabala visina stabla je logaritamska u odnosu na broj čvorova, pa je ta opasnost

zanemarljiva i rekurzivne implementacije su uobičajene.

Iterativna implementacija

Iterativni postupak umetanja može se opisati sledećim koracima:

- uporedi vrednost v sa vrednošću u korenu; ako je v manja ili jednaka, nastavi ulevo, u suprotnom udesno;
- nastavi poređenje dok se ne naiđe na nepostojeće dete; na toj poziciji dodaj novi čvor.

Jedna moguća iterativna implementacija data je u nastavku:

```
status dodaj_u_stablo(cvor_stabla** pokazivac_na_koren,
                      int podatak)
{
    cvor_stabla* tmp;
    cvor_stabla* novi = novi_cvor_stabla(podatak);
    if (novi == NULL)
        return NEDOVOLJNO_MEMORIJE;

    novi->levo = NULL;
    novi->desno = NULL;

    if (*pokazivac_na_koren == NULL) {
        *pokazivac_na_koren = novi;
        return OK;
    }

    tmp = *pokazivac_na_koren;
    while (1) {
        if (podatak <= tmp->podatak) {
            if (tmp->levo == NULL) {
                tmp->levo = novi;
                return OK;
            }
            tmp = tmp->levo;
        } else {
            if (tmp->desno == NULL) {
                tmp->desno = novi;
                return OK;
            }
            tmp = tmp->desno;
        }
    }
}
```

Iterativni pristup izbegava probleme sa dubokom rekurzijom i prekoračenjem steka, ali zadržava iste osobine složenosti kao i rekurzivna verzija. U nebalansiranom stablu, složenost umetanja u najgorem slučaju ostaje linearna.

Pronalaženje elementa u uređenom binarnom stablu

Pretraživanje uređenog binarnog stabla, odnosno pronalaženje čvora koji sadrži vrednost v, konceptualno je vrlo slično postupku umetanja elementa u stablo. U slučaju kada važi da su elementi u levom podstablu manji od ili jednaki vrednosti u korenu, a elementi u desnom podstablu veći od nje (analogno važi i za suprotno uređenje), pretraživanje se može opisati sledećim koracima:

1. Ako je tekuće stablo prazno, zaključuje se da traženi element ne postoji u stablu.
2. Pretraga započinje od korena stabla.
3. Ako je vrednost v jednaka vrednosti u tekućem čvoru, kao rezultat se vraća taj čvor.
4. Ako je vrednost v manja od vrednosti u tekućem čvoru, pretraga se nastavlja u levom podstablu.
5. U suprotnom slučaju, pretraga se nastavlja u desnom podstablu.

Sledeća rekurzivna funkcija implementira opisani algoritam za pronalaženje elementa u uređenom binarnom stablu:

```
cvor_stabla* pronadji_cvor(cvor_stabla* koren, int podatak)
{
    if (koren == NULL)
        return NULL;

    if (podatak < koren->podatak)
        return pronadji_cvor(koren->levo, podatak);
    else if (podatak > koren->podatak)
        return pronadji_cvor(koren->desno, podatak);
    else
        return koren;
}
```

Ukoliko se u stablu dopuštaju duplikati (odnosno ako stablo predstavlja multiskup elemenata), funkcija vraća pokazivač na *jedan* od čvorova koji sadrže traženu vrednost. Tačan čvor koji će biti vraćen zavisi od konkretne strukture stabla i redosleda umetanja elemenata.

Rekurziju u ovoj funkciji nije teško eliminisati, čime se dobija iterativna verzija algoritma sa istim osobinama složenosti.

Obilazak binarnog stabla

Elementi binarnog stabla (ne nužno uređenog) mogu se obrađivati na sistematičan način. Obrada svih čvorova stabla po nekom pravilu naziva se **obilazak stabla**.

Postoje dve osnovne i suštinski različite metode obilaska stabla (kao, uostalom, i proizvoljnog grafa):

- **obilazak u dubinu** (*depth-first traversal*),
- **obilazak u širinu** (*breadth-first traversal*).

Obilazak binarnog stabla u dubinu

Obilazak u dubinu je obilazak u kojem se polazi od korena stabla i nastavlja obrada čvorova duž jedne grane sve dok se ne naiđe na list. Nakon toga se vrši povratak unazad i obrada preostalih čvorova.

Postoji nekoliko varijanti obilaska u dubinu, koje se razlikuju po redosledu obrade korena i podstabala:

- **infiksni obilazak** (L–K–D): najpre se obilazi levo podstablo, zatim sam čvor (koren), pa desno podstablo;
- **prefiksni obilazak** (K–L–D);
- **postfiksni obilazak** (L–D–K).

Svaki od ovih obilazaka može se realizovati i u obrnutom smeru (zdesna nalevo), mada je to ređe u praksi. Na primer, infiksni obilazak može biti i oblika D–K–L.

Obilazak stabla u dubinu se često implementira rekurzivno. Kod povezanih listi rekurzivni obilazak obično nije prihvatljiv jer zahteva onoliko stek okvira koliko ima elemenata liste. Kod stabala je situacija drugačija: maksimalan broj stek okvira jednak je **visini stabla**.

Ako je stablo balansirano, njegova visina je logaritamska u odnosu na broj čvorova, pa je rizik od prekoračenja programskog steka mali. Alternativno, obilazak u dubinu može se realizovati i iterativno, korišćenjem eksplicitnog steka.

Ispis elemenata binarnog stabla u obilasku u dubinu može se rekurzivno implementirati na tri standardna načina, u zavisnosti od toga da li se tekući čvor obrađuje **pre**, **između** ili **posle** obilaska njegovih podstabala.

Infiksni obilazak (L–K–D):

```
void ispisi_infiksno(cvor_stabla* p)
{
    if (p != NULL) {
        ispisi_infiksno(p->levo);
        printf("%d\n", p->podatak);
        ispisi_infiksno(p->desno);
    }
}
```

Prefiksni obilazak (K–L–D):

```
void ispisi_prefiksno(cvor_stabla* p)
{
    if (p != NULL) {
        printf("%d\n", p->podatak);
        ispisi_prefiksno(p->levo);
        ispisi_prefiksno(p->desno);
    }
}
```

Postfiksni obilazak (L–D–K):

```
void ispisi_postfiksno(cvor_stabla* p)
{
    if (p != NULL) {
        ispisi_postfiksno(p->levo);
        ispisi_postfiksno(p->desno);
        printf("%d\n", p->podatak);
    }
}
```

Brisanje stabla

Brisanje stabla se prirodno realizuje **postfiksnim obilaskom**, jer je pre uklanjanja samog čvora potrebno ukloniti sve njegove potomke:

```
void obrisi_stablo(cvor_stabla* p)
{
    if (p != NULL) {
        obrisi_stablo(p->levo);
        obrisi_stablo(p->desno);
        free(p);
    }
}
```

Obilazak binarnog stabla u širinu

U obilasku stabla u širinu čvorovi se obrađuju **nivo po nivo**, počevši od korena. Ovaj obilazak se prirodno implementira korišćenjem strukture **reda**, čiji su elementi pokazivači na čvorove stabla.

Postupak obilaska u širinu može se opisati na sledeći način:

- u red se najpre ubacuje koren stabla;
- zatim se redom uklanja prvi element iz reda i obrađuje;
- potomci tog čvora se dodaju na kraj reda;
- postupak se ponavlja dok red ne postane prazan.

Na osnovu ove ideje, obilazak binarnog stabla u širinu (uz jednostavan ispis čvorova) može se implementirati na sledeći način:

```
int ispisi_stablo_po_sirini(cvor_stabla* koren)
{
    Red red;
    inicijalizuj_red(&red);

    if (enqueue(&red, koren) != OK)
        return -1;

    void* p;
```

```
    while (dequeue(&red, &p) == OK) {
        cvor_stabla* c = (cvor_stabla*)p;
        printf("%d ", c->podatak);

        if (c->levo != NULL)
            if (enqueue(&red, c->levo) != OK)
                return -1;

        if (c->desno != NULL)
            if (enqueue(&red, c->desno) != OK)
                return -1;
    }

    return 0;
}
```

Sledeća funkcija `main` ilustruje upotrebu obilaska stabla u širinu:

```
int main()
{
    cvor_stabla* koren = NULL;

    if (dodaj_u_stablo(&koren, 3) != OK ||
        dodaj_u_stablo(&koren, 5) != OK ||
        dodaj_u_stablo(&koren, 4) != OK ||
        dodaj_u_stablo(&koren, 2) != OK ||
        dodaj_u_stablo(&koren, 1) != OK) {
        obrisi_stablo(koren);
        return -1;
    }

    ispisi_stablo_po_sirini(koren);
    obrisi_stablo(koren);
    return 0;
}
```

Izlaz programa je:

```
3 2 5 1 4
```

Generičko uređeno binarno stablo

Slično kao što je u poglavljima o generičkom steku i generičkom redu prikazano kako se strukture podataka mogu učiniti generičkim, i binarno stablo se može implementirati tako da (uz trivijalne izmene) radi nad različitim tipovima podataka.

Ideja je da se tip podatka koji se čuva u čvorovima izdvoji u poseban modul, zajedno sa osnovnim operacijama nad tim tipom (poređenje, dodela, ispis). Na taj način, implementacija stabla ne zavisi od konkretnog tipa podataka.

Datoteka `stablo_podatak.h`

U datoteci `stablo_podatak.h` uvodi se tip `Data` (u sledećem jednostavnom primeru: `int`) i deklarišu osnovne funkcije za baratanje podacima tog tipa:

```
#ifndef __STABLO_PODATAK_H
#define __STABLO_PODATAK_H

typedef int Data;

int poredi(const Data *d1, const Data *d2);
int dodeli(Data *d1, const Data *d2);
void ispisi_podatak(const Data *d);

#endif
```

Te funkcije mogu biti definisane u datoteci `stablo_podatak.c`:

```
#include <stdio.h>
#include "stablo_podatak.h"

int poredi(const Data *d1, const Data *d2)
{
    if (*d1 < *d2)
        return -1;
    if (*d1 > *d2)
        return 1;
    return 0;
}

int dodeli(Data *d1, const Data *d2)
{
    *d1 = *d2;
    return 0;
}

void ispisi_podatak(const Data *d)
{
    printf("%d ", *d);
}
```

Datoteka `stablo_genericko.h`

Naredna implementacija generičkog uređenog binarnog stabla koristi tip `Data`, definiše tip `Stablo` i skup osnovnih funkcija. Sadržaj datoteke `stablo_genericko.h` može biti sledeći:

```
#ifndef __STABLO_GENERICKO_H__
#define __STABLO_GENERICKO_H__

#include <stdio.h>
```

```
#include <stdlib.h>
#include "stablo_podatak.h"

typedef enum {
    OK,
    NEDOVOLJNO_MEMORIJE,
    NEMA_ELEMENATA
} status;

typedef struct cvor_stabla {
    Data podatak;
    struct cvor_stabla* levo;
    struct cvor_stabla* desno;
} cvor_stabla;

typedef cvor_stabla* Stablo;

status inicijalizuj_stablo(Stablo* s);

status dodaj_u_stablo(Stablo* s, const Data* podatak,
                      int (*poredi)(const Data*, const Data*));

status pronadji(Stablo s, const Data* trazeno, Data** nadjeno,
                int (*poredi)(const Data*, const Data*));

int  broj_elemenata(Stablo s);
void ispisi_stablo(Stablo s);
void obrisi_stablo(Stablo* s);

#endif
```

Datoteka `stablo_genericko.c`

Sadržaj datoteke `stablo_genericko.c` može biti ovakav:

```
#include <stdio.h>
#include <stdlib.h>
#include "stablo_genericko.h"

cvor_stabla* novi_cvor_stabla(const Data* podatak)
{
    cvor_stabla* novi = malloc(sizeof(cvor_stabla));
    if (novi == NULL)
        return NULL;

    dodeli(&(novi->podatak), podatak);
    novi->levo = NULL;
    novi->desno = NULL;
    return novi;
```

```
}

status inicijalizuj_stablo(Stablo* s)
{
    *s = NULL;
    return OK;
}

status dodaj_u_stablo(Stablo* s, const Data* podatak,
                      int (*poredi)(const Data*, const Data*))
{
    cvor_stabla* tmp;
    cvor_stabla* novi = novi_cvor_stabla(podatak);
    if (novi == NULL)
        return NEDOVOLJNO_MEMORIJE;

    novi->levo = NULL;
    novi->desno = NULL;

    if (*s == NULL) {
        *s = novi;
        return OK;
    }

    tmp = *s;
    while (1) {
        if (poredi(podatak, &(tmp->podatak)) <= 0) {
            if (tmp->levo == NULL) {
                tmp->levo = novi;
                return OK;
            } else {
                tmp = tmp->levo;
            }
        } else {
            if (tmp->desno == NULL) {
                tmp->desno = novi;
                return OK;
            } else {
                tmp = tmp->desno;
            }
        }
    }

    return OK;
}

status pronadji(Stablo s, const Data* trazeno, Data** nadjeno,
                int (*poredi)(const Data*, const Data*))
```

```
{
    if (s == NULL)
        return NEMA_ELEMENATA;

    if (poredi(trazeno, &(s->podatak)) == 0) {
        *nadjeno = &(s->podatak);
        return OK;
    }

    if (poredi(trazeno, &(s->podatak)) < 0)
        return pronadji(s->levo, trazeno, nadjeno, poredi);

    return pronadji(s->desno, trazeno, nadjeno, poredi);
}

int broj_elemenata(Stablo s)
{
    return s == NULL ?
        0 :
        broj_elemenata(s->levo) + 1 + broj_elemenata(s->desno);
}

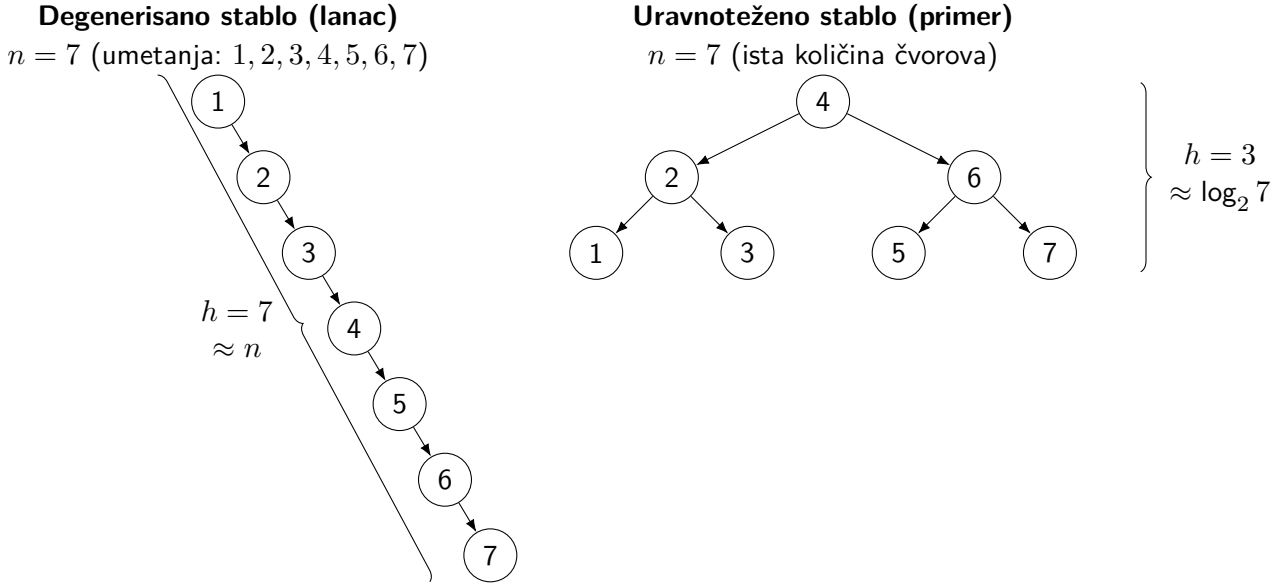
void ispisi_stablo(Stablo s)
{
    if (s != NULL) {
        ispisi_stablo(s->levo);
        ispisi_podatak(&(s->podatak));
        ispisi_stablo(s->desno);
    }
}

void obrisi_stablo(Stablo* s)
{
    if (*s != NULL) {
        obrisi_stablo(&((*s)->levo));
        obrisi_stablo(&((*s)->desno));
        free(*s);
        *s = NULL;
    }
}
```

4.2 Balansirana binarna stabla

U prethodnim odeljcima videli smo da uređeno binarno stablo pruža elegantan način da se elementi organizuju tako da se pretraga, umetanje i brisanje svode na kretanje niz jednu granu stabla. Međutim, efikasnost ovih operacija ne zavisi samo od broja elemenata, već presudno od **oblika stabla**, odnosno njegove **visine**.

U povoljnim slučajevima, kada se stablo razvija približno uravnoteženo, visina ostaje mala i operacije su brze. Nasuprot tome, nepovoljan redosled umetanja (na primer ključevi u strogo rastućem ili strogo opadajućem poretku) može dovesti do degeneracije strukture u oblik sličan linearnoj listi. Tada visina postaje reda n, pa i pretraga, umetanje i brisanje u najgorem slučaju zahtevaju linearno vreme.



Slika 4.3: Ilustracija heš tabele.

Motivacija za **balansirana** uređena binarna stabla je upravo u tome da se obezbedi garancija da visina ostaje $O(\log n)$, nezavisno od redosleda operacija. Ovakva garancija neposredno povlači i složenosti $O(\log n)$ za osnovne operacije u najgorem slučaju, što je od ključne važnosti kada se stablo koristi kao osnova za mape, skupove i druge asocijativne strukture.

Da bi se visina održavala logaritamskom, nakon operacija koje menjaju strukturu (umetanja i brisanja) primenjuju se dodatne lokalne transformacije — **balansiranja**. Pri tome je neophodno da se balansiranjem ne naruši uslov uređenosti, tj. da rezultat i dalje bude uređeno binarno stablo.

Razvijeno je više familija balansiranih uređenih binarnih stabala, koje se razlikuju po invarijantama koje održavaju i po strategiji balansiranja. U ovom poglavlju biće obrađena **AVL stabla** (Adelson–Velski i Landis), koja održavaju strogu ravnotežu podstabala i obezbeđuju logaritamsku visinu. Pored njih, značajnu ulogu u praksi imaju i druge strukture, kao što su **crveno-crna stabla**, **2–3 i 2–3–4 stabla**, kao i **B-stabla** i **B+ stabla**; neke od ovih naprednijih varijanti biće razmotrene u kasnijim poglavljima.

AVL stablo je klasičan predstavnik balansiranih uređenih binarnih stabala. Naziv potiče od inicijala autora (G. Adelson-Velski i E. M. Landis). Osnovna ideja je jednostavna: uređeno binarno stablo se nakon svake operacije koja menja strukturu (umetanje ili brisanje) *po potrebi* lokalno koriguje tako da mu visina ostane logaritamska. Posledica toga je da su operacije *pretrage*, *umetanja* i *brisanja* u najgorem slučaju vremenske složenosti $O(\log n)$, gde je n broj elemenata u stablu.

Pri tome se balansiranost drveta definiše tako da se može lako održavati. Neformalno, ideja AVL drveta je da za svaki čvor važi da se su njegovo levo i desno približnih visina.

Visina stabla. Visina stabla je maksimalna dužina puta od korena do lista. U ovoj knjizi usvajamo konvenciju da se visina meri **brojem grana** na najdužem putu:

- stablo sa jednim čvorom ima visinu (0),
- stablo sa dva čvora (koren + jedno dete) ima visinu (1),
- prazno stablo ima visinu (-1).

Faktor ravnoteže čvora. Za čvor u definiše se faktor ravnoteže kao razlika visina njegovih podstabala:

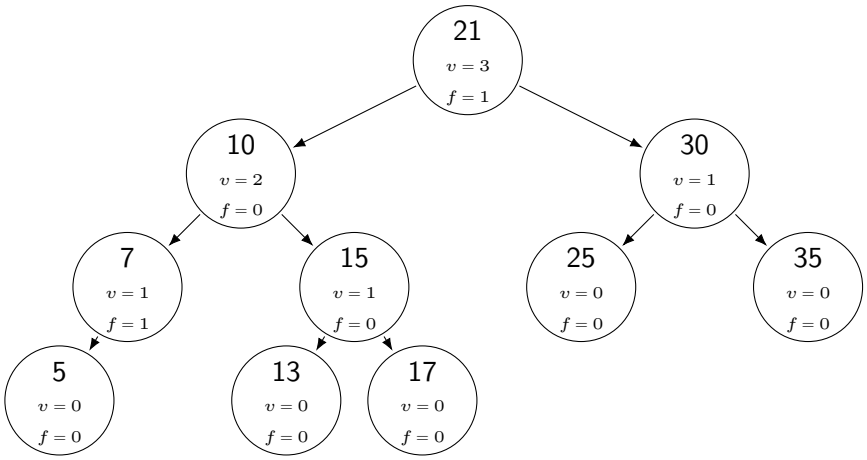
$$f(u) = h(u_{\text{levo}}) - h(u_{\text{desno}}),$$

pri čemu je visina nepostojećeg (praznog) podstabla (-1).

Definicija (AVL stablo). AVL stablo je uređeno binarno stablo u kome za svaki čvor u važi:

$$|f(u)| \leq 1.$$

Drugim rečima, visine levog i desnog podstabla svakog čvora mogu se razlikovati najviše za jedan.



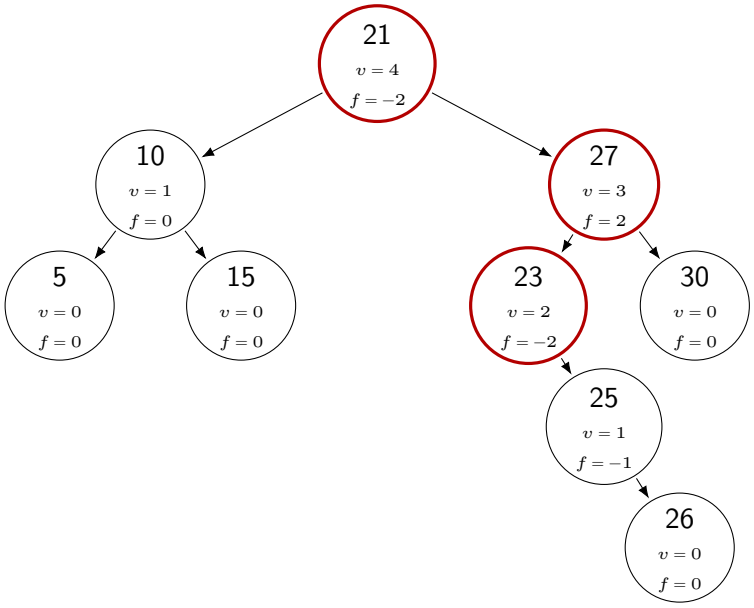
Slika 4.4: Primer AVL stabla. Uz svaki čvor prikazane su visina v podstabla i faktor ravnoteže f čvora.

Na slici 4.4 dat je primer AVL stabla. Uz svaki čvor prikazane su visina podstabla sa korenom u tom čvoru i faktor ravnoteže. Uočava se da su faktori ravnoteže svih čvorova u skupu $\{-1, 0, 1\}$. Na primer, faktor ravnoteže čvora sa ključem 7 jednak je 1, jer je visina levog podstabla 0, a desnog -1. Slično, faktor ravnoteže čvora sa ključem 21 jednak je 1, jer su visine levog i desnog podstabla 2 i 1.

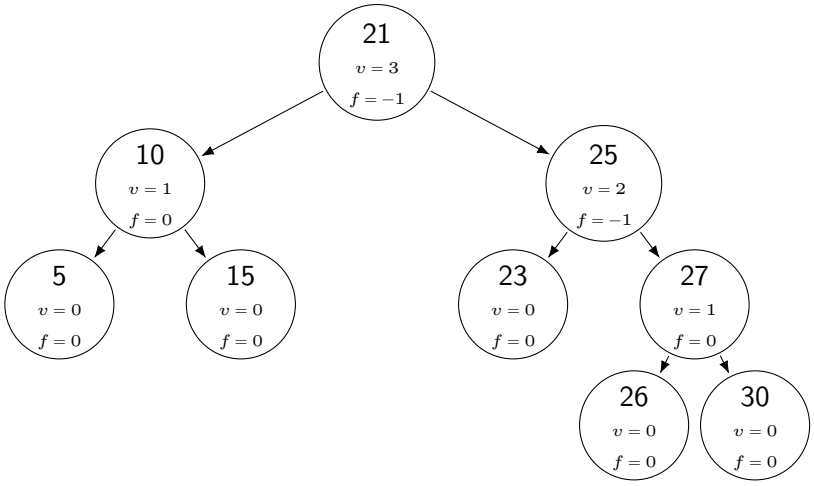
Na slici 4.5 prikazano je uređeno binarno stablo koje *nije* AVL stablo: postoje čvorovi (u prikazanom primeru oni sa ključevima 21, 27 i 23) čiji faktor ravnoteže nije u skupu $\{-1, 0, 1\}$. Ipak, lokalnim korekcijama strukture (operacijama balansiranja) moguće je transformisati takvo stablo u AVL stablo, kao na slici 4.6.

Iz definicije neposredno sledi da uslov $|f(u)| \leq 1$ mora važiti u **svakom** čvoru, pa je svako podstablo AVL stabla takođe AVL stablo.

Preostaje ključno pitanje: da li je uslov $|f(u)| \leq 1$ za svaki čvor dovoljno jak da garantuje da je visina AVL stabla sa n čvorova zaista $O(\log n)$? To dokazujemo narednom teoremom.



Slika 4.5: Primer uređenog binarnog stabla koje nije AVL stablo.



Slika 4.6: Transformacija prethodno ilustrovanog ne-AVL stabla u AVL stablo operacijom balansiranja.

Teorema 4.2.1. *Neka je h visina AVL stabla sa n čvorova. Tada važi*

$$h \leq 2 \log_2 n.$$

Dokaz. Dokažimo ovo tvrđenje indukcijom po visini drveta h. Prvo, primetimo da visina drveta ne određuje jednoznačno broj čvorova u drvetu. Zato posmatramo ekstreman slučaj — AVL stabla *sa najmanjim mogućim brojem čvorova* za datu visinu. Ako i ta stabla zadovoljavaju traženu nejednakost, tada je ona automatski tačna i za sva ostala AVL stabla iste visine (jer je logaritamska funkcija monotono rastuća).

Neka je T_h AVL stablo visine $h \geq 0$ sa najmanjim mogućim brojem čvorova, i neka je n_h broj čvorova stabla T_h. Drvo T_0 sadrži samo koren, a drvo T_1 koren i jedno dete, recimo desno.

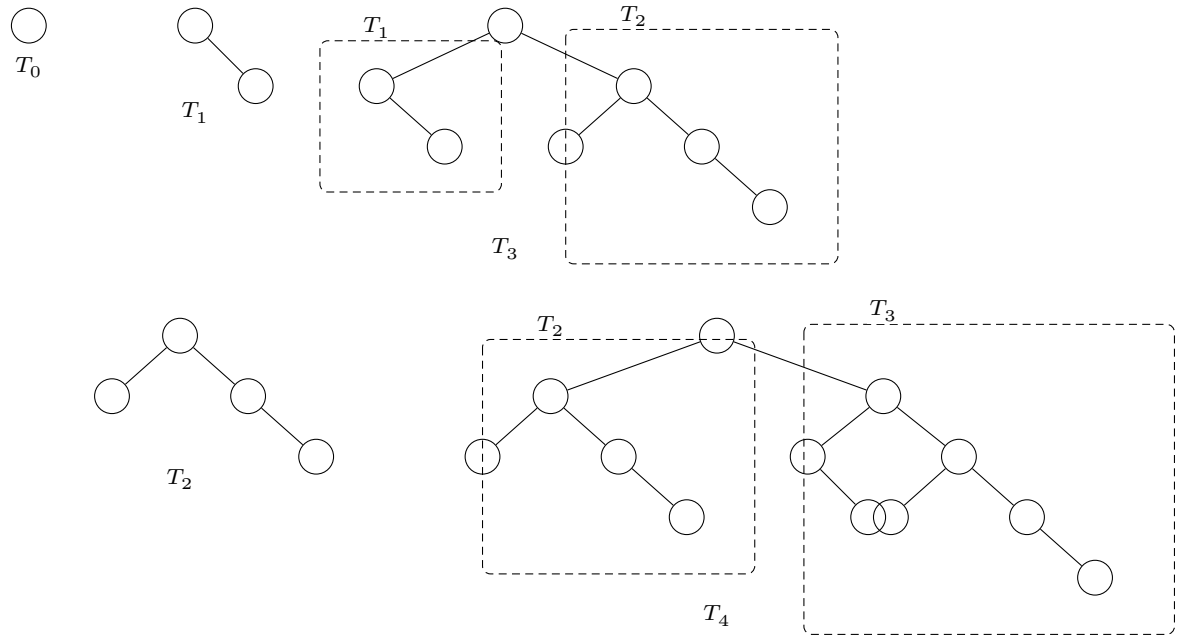
Po dogovorenoj konvenciji visina praznog stabla je -1, a visina stabla sa jednim čvorom je 0, pa imamo početne vrednosti

$$n_0 = 1, \qquad n_1 = 2.$$

Razmotrimo sada $h \geq 2$. U AVL stablu visine h bar jedno od podstabala korena mora imati visinu $h-1$ (inače bi visina celog stabla bila $\leq h-1$). Zbog uslova $|f| \leq 1$, drugo podstablo tada može imati visinu najviše $h-2$. Pritom, da bi dato AVL drvo imalo najmanji mogući broj čvorova, onda i njegovo poddrvo manje visine $h-2$ takođe treba da bude AVL drvo sa minimalnim brojem čvorova, dakle T_{h-2}; slično, njegovo drugo poddrvo treba da bude T_{h-1}.; otuda dobijamo rekurenciju

$$n_h = 1 + n_{h-1} + n_{h-2}, \qquad h \geq 2.$$

Drveta opisana ovakvom rekurentnom jednačinom nazivaju se *Fibonačijeva drveta*, jer broj čvorova prati Fibonačijev obrazac.



Slika 4.7: Fibonačijeva AVL stabla: minimalna AVL stabla za datu visinu.

Označimo sa n_h minimalni broj čvorova AVL drveta visine h, tj. broj čvorova drveta T_h. Važi $n_0 = 1$, $n_1 = 2$, $n_2 = 4, \ldots$ Prema definiciji Fibonačijevih drveta važi $n_h = n_{h-1} + n_{h-2} + 1$, za $h \geq 2$. S obzirom na to da je $n_h > n_{h-1}$ za svako $h \geq 1$ važi:

$$n_h = 1 + n_{h-1} + n_{h-2} \geq 1 + 2n_{h-2} \geq 2n_{h-2}. \tag{4.1}$$

Sada dokazujemo indukcijom po h da važi

$$n_h \geq 2^{h/2}.$$

Za $h = 0$ i $h = 1$ tvrdnja važi jer je $n_0 = 1 = 2^0$ i $n_1 = 2 \geq 2^{1/2}$.

Pretpostavimo da tvrdnja važi za $h-2$, tj. $n_{h-2} \geq 2^{(h-2)/2}$. Tada iz 4.1 dobijamo

$$n_h \geq 2n_{h-2} \geq 2 \cdot 2^{(h-2)/2} = 2^{h/2}.$$

Dakle, nejednakost $n_h \geq 2^{h/2}$ važi za sve $h \geq 0$.

Nakon logaritmovanja obe strane nejednakosti dobijamo

$$\frac{h}{2} \leq \log_2 n_h \quad \Longrightarrow \quad h \leq 2 \log_2 n_h.$$

Pošto svako AVL stablo visine h ima bar n_h čvorova, za dato stablo sa n čvorova važi $n \geq n_h$, pa je

$$h \leq 2 \log_2 n_h \leq 2 \log_2 n.$$

□

Pošto je za AVL stablo visine h pretraga proporcionalna visini (u najgorem slučaju prelazi se put od korena do lista), prethodna teorema neposredno daje da je *pretraga* u AVL stablu složenosti $O(\log n)$. Preostaje da pokažemo kako se realizuju *umetanje* i *brisanje* tako da se nakon svake izmene strukture ponovo uspostavi uslov $|f| \leq 1$. To vodi u operacije balansiranja (rotacija), koje obrađujemo u nastavku.

Umetanje elementa u AVL stablo

Umetanje novog elementa u AVL stablo sprovodi se u dve faze.

1. **BST faza (bez balansiranja).** Element se umeće kao uobičajeno u uređeno binarno stablo: prateći pravilo uređenosti (ulevo za manje/"jednako", udesno za veće) pronalazi se mesto i dodaje se novi čvor kao **list**.

2. **AVL faza (po potrebi balansiranje).** Pošto umetanje lista može da poveća visinu jednog podstabla na putu ka korenu, potrebno je proveriti da li je narušen AVL uslov $|f| \leq 1$, gde je

$$f(u) = h(u_{\text{levo}}) - h(u_{\text{desno}}).$$

Pre umetanja za svaki čvor važi $|f| \leq 1$. Nakon umetanja, pošto se visina može povećati samo duž jedne grane, razlika visina kod nekih predaka može porasti **najviše do 2** (nikada više od toga pri jednoj operaciji umetanja). Dakle, jedina moguća povreda AVL svojstva je pojava $|f| = 2$ u nekom čvoru.

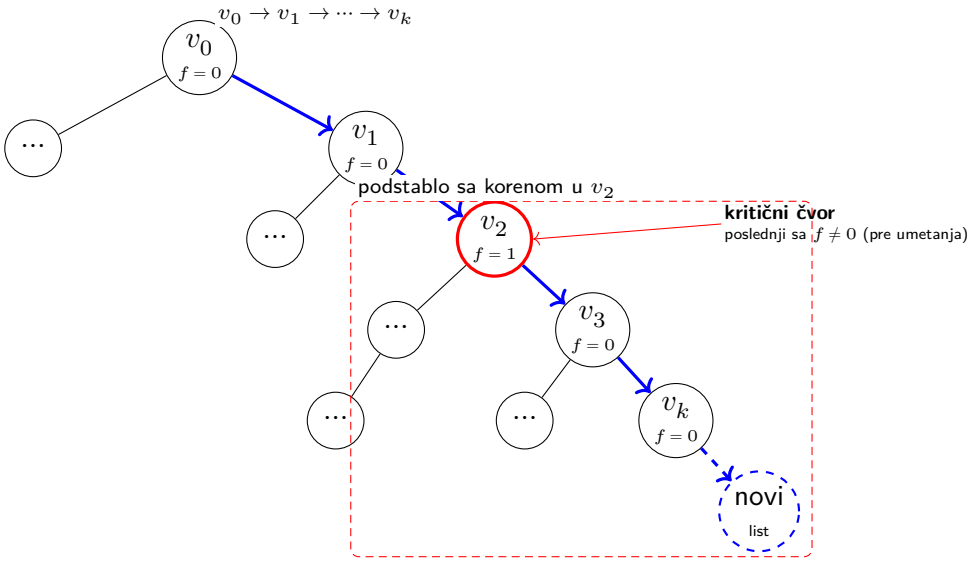
Neka je put od korena do mesta umetanja:

$$v_0 \to v_1 \to \cdots \to v_k,$$

gde je v_k roditelj novog lista.

Za umetanje je naročito važan poslednji čvor na tom putu čiji faktor ravnoteže *nije nula* pre umetanja. Taj čvor u literaturi često zovu **kritični čvor**, u smislu "poslednji čvor sa $f \neq 0$ na putu pre umetanja" (Slika 4.8). Intuicija je sledeća:

- Čvorovi **iznad** kritičnog čvora imaju faktor ravnoteže 0 (na putu pre umetanja), pa se kod njih visine levog i desnog podstabla pre umetanja podudaraju.
- Umetanje može promeniti visine samo u podstablu kroz koje smo prošli. Zato je "kritična tačka" upravo najniži čvor na putu gde je stablo već bilo blago nagnuto ($f = \pm 1$); tu se nakon umetanja najlakše dobija $|f| = 2$.

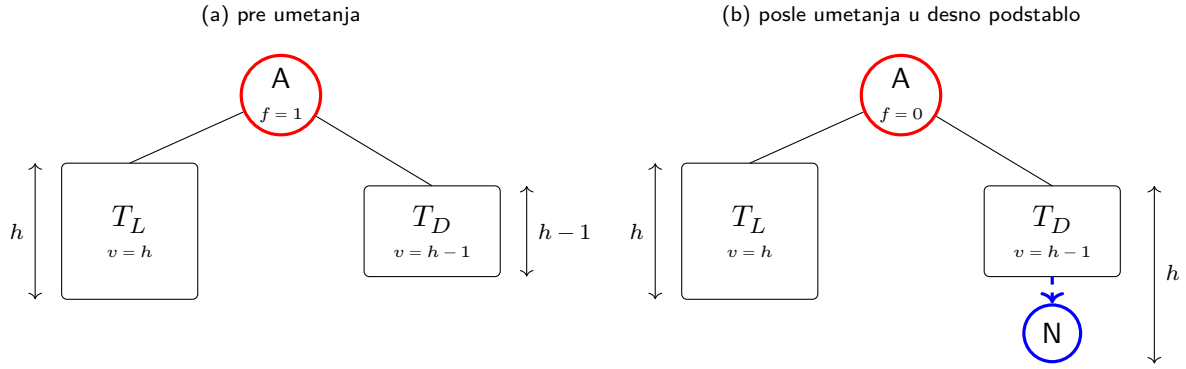


Slika 4.8: Ilustracija umetanja i detekcija kritičnog čvora.

Ispostavlja se da je kod umetanja **dovoljno izbalansirati podstablo sa korenom u tom kritičnom čvoru**: jednom kada se ono popravi odgovarajućom rotacijom, visina tog podstabla postaje ista kao pre umetanja, pa iznad njega ne dolazi do novih povreda AVL uslova.

Kada umetanje ne zahteva rotacije

Pretpostavimo da je A kritični čvor i da je pre umetanja $f(A) = 1$, tj. levo podstablo ima visinu za 1 veću od desnog. Novi čvor N može završiti:



Slika 4.9: Ilustracija umetanja koja ne zahteva rotacije.

- u **desnom** podstablu čvora A (Slika 4.9): tada se visina desnog podstabla može povećati za 1, pa $f(A)$ postaje 0. Podstablo sa korenom u A ostaje AVL, i korekcije nisu potrebne.

- u **levom** podstablu čvora A: tada se visina levog podstabla povećava za 1, pa $f(A)$ postaje 2 i AVL uslov je narušen. Tada je potrebna rotacija.

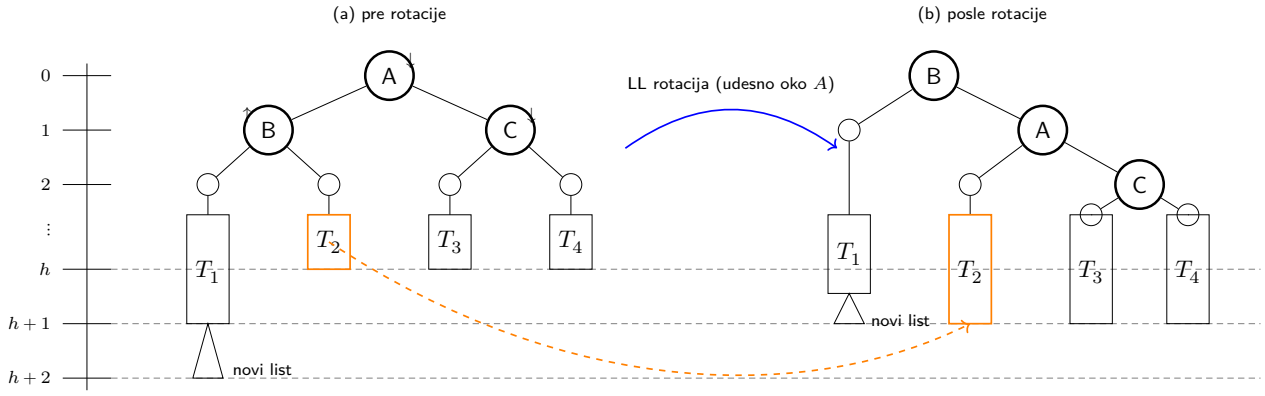
Potpuno simetrično važi za slučaj $f(A) = -1$.

LL i LR slučaj: za $f(A) = +2$

Neka je A kritični čvor i nakon umetanja je levo-težak, $f(A) = +2$. Označimo sa B levo dete čvora A. Pošto je novi čvor umetnut u levo podstablo, postoje dve mogućnosti:

1. **LL slučaj** (umetanje u levo podstablo levog deteta, Slika 4.10).

   U ovom slučaju, nakon umetanja novi čvor pripada levom podstablu levog deteta kritičnog čvora A. Neuravnoteženost je oblika $f(A) = +2$, pri čemu je levo dete B čvora A takođe "levo-orijentisano" u smislu da je umetanje izvršeno u njegovom levom podstablu. Korekcija se sprovodi **jednostrukom rotacijom udesno** oko čvora A (LL rotacija).



Slika 4.10: Ilustracija LL rotacije.

Standardni raspored podstabala može se opisati na sledeći način: neka su T_1 i T_2 redom levo i desno podstablo čvora B, a neka je T_3 desno podstablo čvora A. Tada je:

- **pre rotacije:** A je koren posmatranog podstabla, B je njegovo levo dete, T_2 je desno podstablo čvora B, a T_3 je desno podstablo čvora A;
- **posle rotacije:** čvor B postaje novi koren posmatranog podstabla, čvor A postaje desno dete čvora B, dok podstablo T_2 prelazi na poziciju levog podstabla čvora A (podstabla T_1 i T_3 zadržavaju odgovarajuće strane).

Ovakva reorganizacija **ne narušava uslov uređenosti**. Naime, zbog uređenosti izvornog stabla važi

$$\forall x \in T_1 : \ x \leq B, \qquad \forall x \in T_2 : \ B < x \leq A, \qquad \forall x \in T_3 : \ A < x,$$

pa je premeštanje podstabla T_2 na mesto levog podstabla čvora A konzistentno sa poretkom ključeva.

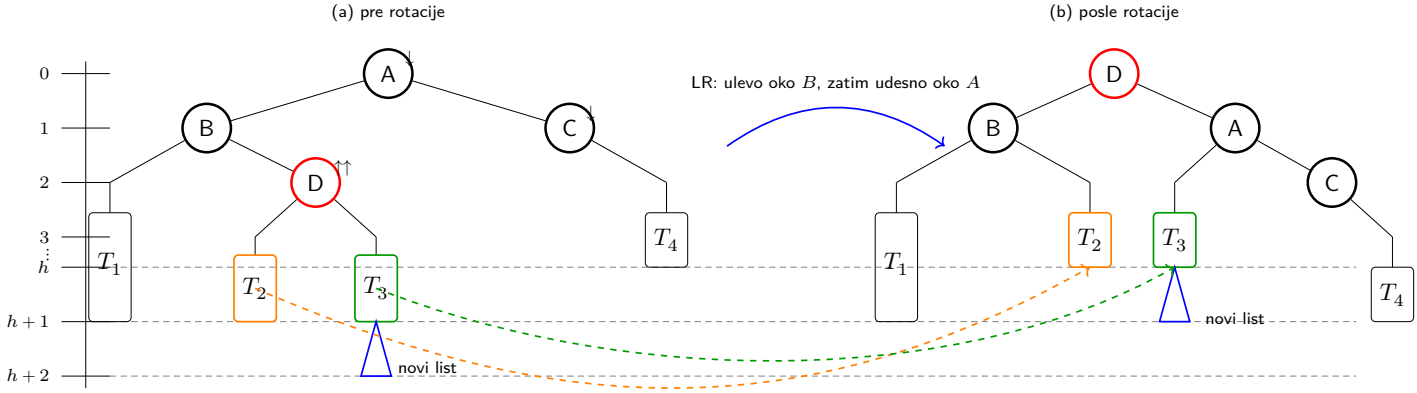
Pošto je A *kritični čvor*, sledi da su svi čvorovi na putu od A ka mestu umetanja imali faktor ravnoteže jednak 0. Posebno, za njegovo levo dete B važi $f(B) = 0$, pa podstabla T_1 i T_2 imaju istu visinu. Posle izvršene LL rotacije, novi koren posmatranog podstabla postaje čvor B, pri čemu se ponovo dobija $f(B) = 0$ (a analogno i čvor A postaje balansiran), pa korigovano podstablo ponovo zadovoljava AVL uslov.

2. **LR slučaj** (umetanje u desno podstablo levog deteta, Slika 4.11).

   U ovom slučaju, nakon umetanja novi čvor pripada **desnom podstablu** levog deteta B kritičnog čvora A. Tada se neuravnoteženost u čvoru A ispoljava kao $f(A) = +2$, dok je levo dete B "desno-orijentisano" u smislu da je povećanje visine nastalo u njegovom desnom podstablu. Jednostruka rotacija više nije dovoljna da obnovi uslov $|f| \leq 1$, pa se primenjuje **dvostruka rotacija** (LR rotacija), i to u sledećem redosledu:

- najpre se vrši rotacija **ulevo** oko čvora B,
- zatim se vrši rotacija **udesno** oko čvora A.

Dvostruka rotacija se može posmatrati kao lokalna transformacija koja "promoviše" srednji čvor (desno dete čvora B) u koren posmatranog podstabla, pri čemu se veze između podstabala preuređuju tako da se **očuva poredak**, odnosno uslov uređenosti stabla.



Slika 4.11: Ilustracija LR rotacije.

Napomena (ažuriranje faktora ravnoteže). Nakon LR (i analogno RL) rotacije, tačne vrednosti faktora ravnoteže čvorova koji učestvuju u rotaciji zavise od položaja umetnutog lista u odnosu na "srednji" čvor (tj. od toga da li je umetanje izvršeno u njegovo levo ili desno podstablo). U implementaciji postoje dva standardna pristupa:

1. čuvati **visinu** u svakom čvoru i nakon svake rotacije lokalno ponovo izračunati visine (pa iz njih izvesti faktor ravnoteže), ili
2. čuvati **faktor ravnoteže** i ažurirati ga prema kratkoj tabeli podslučajeva.

U ovoj knjizi ćemo se osloniti se na prvi pristup, jer minimizuje mogućnost nekonzistentnog stanja i čini korektnost rotacija transparentnom.

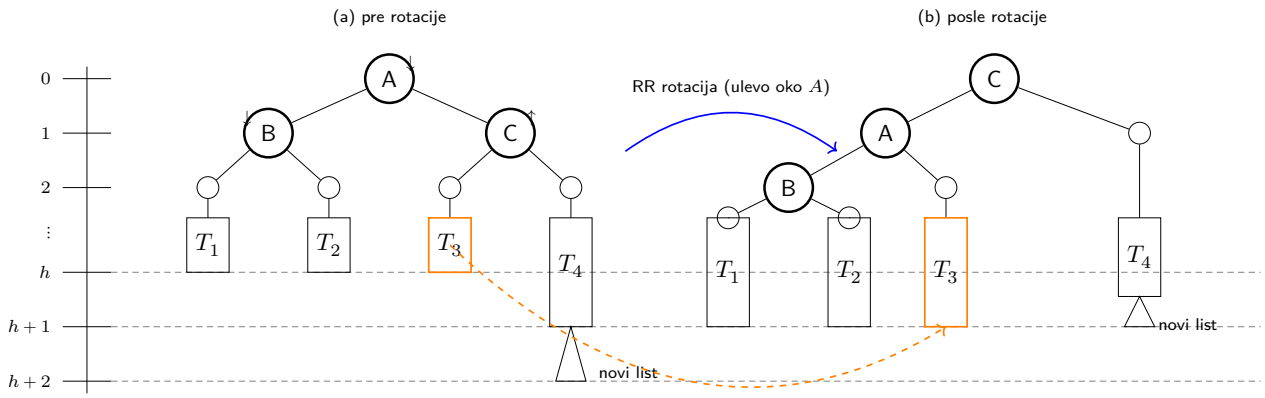
Slučajevi **RR** (Slika 4.12) i **RL** (Slika 4.13) dobijaju se simetrijom (zamenom "levo" i "desno").

U svim slučajevima umetanja (LL/LR/RR/RL) važi važna činjenica:

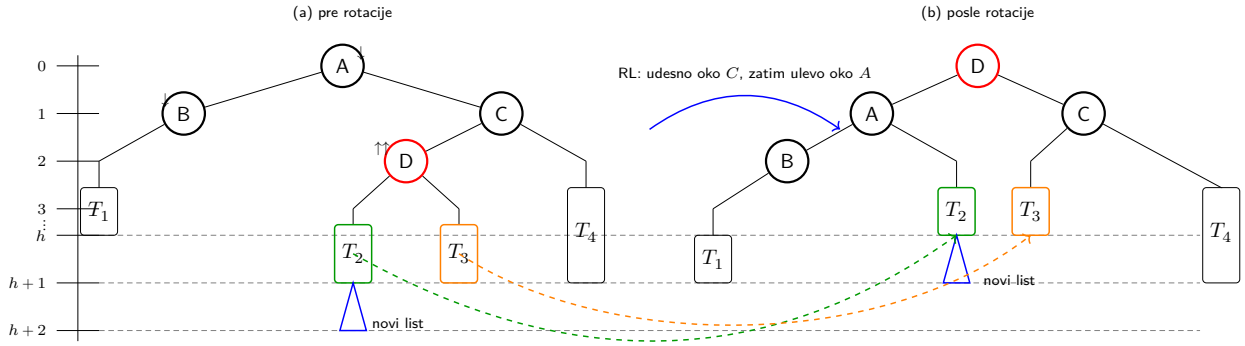
> **Visina podstabla sa korenom u kritičnom čvoru nakon balansiranja ostaje ista kao pre umetanja.**

Zato iznad kritičnog čvora nema potrebe za dodatnim korekcijama: kod predaka se visina relevantnog deteta "vrati" na staru vrednost, pa se njihovi faktori ravnoteže ne mogu pogoršati.

Ovo je ujedno razlog zašto se u iterativnim implementacijama često radi sledeće:



Slika 4.12: Ilustracija RR rotacije.



Slika 4.13: Ilustracija RL rotacije.

- tokom spuštanja ka mestu umetanja pamti se kritični čvor (poslednji sa $f \neq 0$),
- nakon umetanja se ponovo prolazi put od kritičnog čvora ka umetnutom listu i ažuriraju visine/faktor ravnoteže,
- zatim se uradi odgovarajuća rotacija u kritičnom čvoru i postupak se završava.

U rekurzivnoj implementaciji (koju ćemo koristiti u C kodu ispod), kritični čvor se "prirodno" pojavi kao **prvi čvor na povratku iz rekurzije** kod koga se detektuje $|f| = 2$.

C implementacija: rotacije + umetanje (AVL)

U nastavku je minimalan kod: čuva se samo **visina**, a faktor ravnoteže se računa kao razlika visina. Tako da se čvor može definisati na sledeći način:

```
typedef struct Cvor {
    int kljuc;
    struct Cvor* levo;
    struct Cvor* desno;
    int visina;              /* visina podstabla: prazno=-1, list=0 */
} Cvor;
```

```c
static int max2(int a, int b) { return (a > b) ? a : b; }

static int visina(const Cvor* p) { return p ? p->visina : -1; }

static void osvezi_visinu(Cvor* p)
{
    p->visina = 1 + max2(visina(p->levo), visina(p->desno));
}

static int faktor(const Cvor* p)
{
    return visina(p->levo) - visina(p->desno);
}


/* Jednostruka rotacija udesno (LL korekcija kada je A levo-tezak). */
static Cvor* rotiraj_udesno(Cvor* A)
{
    Cvor* B  = A->levo;
    Cvor* T2 = B->desno;

    B->desno = A;
    A->levo  = T2;

    osvezi_visinu(A);
    osvezi_visinu(B);
    return B; /* novi koren podstabla */
}

/* Jednostruka rotacija ulevo (RR korekcija kada je A desno-tezak). */
static Cvor* rotiraj_ulevo(Cvor* A)
{
    Cvor* B  = A->desno;
    Cvor* T2 = B->levo;

    B->levo  = A;
    A->desno = T2;

    osvezi_visinu(A);
    osvezi_visinu(B);
    return B;
}

/* Rebalansira podstablo sa korenom u A (deca pretpostavljamo vec AVL). */
static Cvor* rebalansiraj(Cvor* A)
{
    int fA = faktor(A);
```

```
    /* Levo-tezak: LL ili LR */
    if (fA > 1) {
        if (faktor(A->levo) < 0) {          /* LR: prvo ulevo oko B */
            A->levo = rotiraj_ulevo(A->levo);
        }
        return rotiraj_udesno(A);           /* LL: udesno oko A */
    }

    /* Desno-tezak: RR ili RL */
    if (fA < -1) {
        if (faktor(A->desno) > 0) {         /* RL: prvo udesno oko B */
            A->desno = rotiraj_udesno(A->desno);
        }
        return rotiraj_ulevo(A);            /* RR: ulevo oko A */
    }

    return A; /* vec je AVL */
}

/* Rekurzivno umetanje + balansiranje na povratku. */
static Cvor* ubaci_rec(Cvor* koren, Key k, bool* ok)
{
    if (!koren) {
        Cvor* n = novi_cvor(k);
        if (!n) *ok = false;
        return n;
    }

    /* Konvencija: duplikate saljemo ulevo (k <= koren->kljuc). */
    if (k <= koren->kljuc) {
        Cvor* novo = ubaci_rec(koren->levo, k, ok);
        if (!*ok) return koren;
        koren->levo = novo;
    } else {
        Cvor* novo = ubaci_rec(koren->desno, k, ok);
        if (!*ok) return koren;
        koren->desno = novo;
    }

    osvezi_visinu(koren);
    return rebalansiraj(koren);
}

bool avl_ubaci(Cvor** pkoren, Key k)
{
    bool ok = true;
    Cvor* novo = ubaci_rec(*pkoren, k, &ok);
    if (!ok) return false;
```

```
    *pkoren = novo;
    return true;
}
```

- "Kritični čvor" se u ovoj rekurzivnoj varijanti manifestuje kao **prvi čvor na povratku** gde `rebalansiraj()` detektuje $|f| = 2$.
- Rotacije su lokalne: menjaju mali broj pokazivača i posle `osvezi_visinu()` stanje (visine) postaje konzistentno.
- Pošto se visina rebalansiranog podstabla vraća na prethodnu vrednost, iznad toga se više ne javljaju nove rotacije kod umetanja.

Brisanje elementa iz AVL drveta

Operacija brisanja u AVL drvetu zasniva se na standardnoj proceduri brisanja u uređenom binarnom drvetu, uz dodatni uslov da se nakon svake strukturalne promene mora obnoviti AVL invarijanta

$$|f(u)| \leq 1 \quad \text{za svaki čvor } u$$

gde je $f(u) = h(u_{\text{levo}}) - h(u_{\text{desno}}$, a visina praznog drveta iznosi -1. Brisanje je postupak koji se izvodi u 2 koraka:

1. **BST faza (uklanjanje čvora).** Pronalazi se čvor sa zadatim ključem i uklanja se na uobičajen način:
   - ako čvor nema potomke, uklanja se list;
   - ako ima tačno jedno dete, čvor se zamenjuje svojim detetom;
   - ako ima dva deteta, čvor se zamenjuje svojim in-order sledbenikom (npr. **minimumom desnog poddrveta**), pri čemu se taj minimum uklanja iz desnog poddrveta.

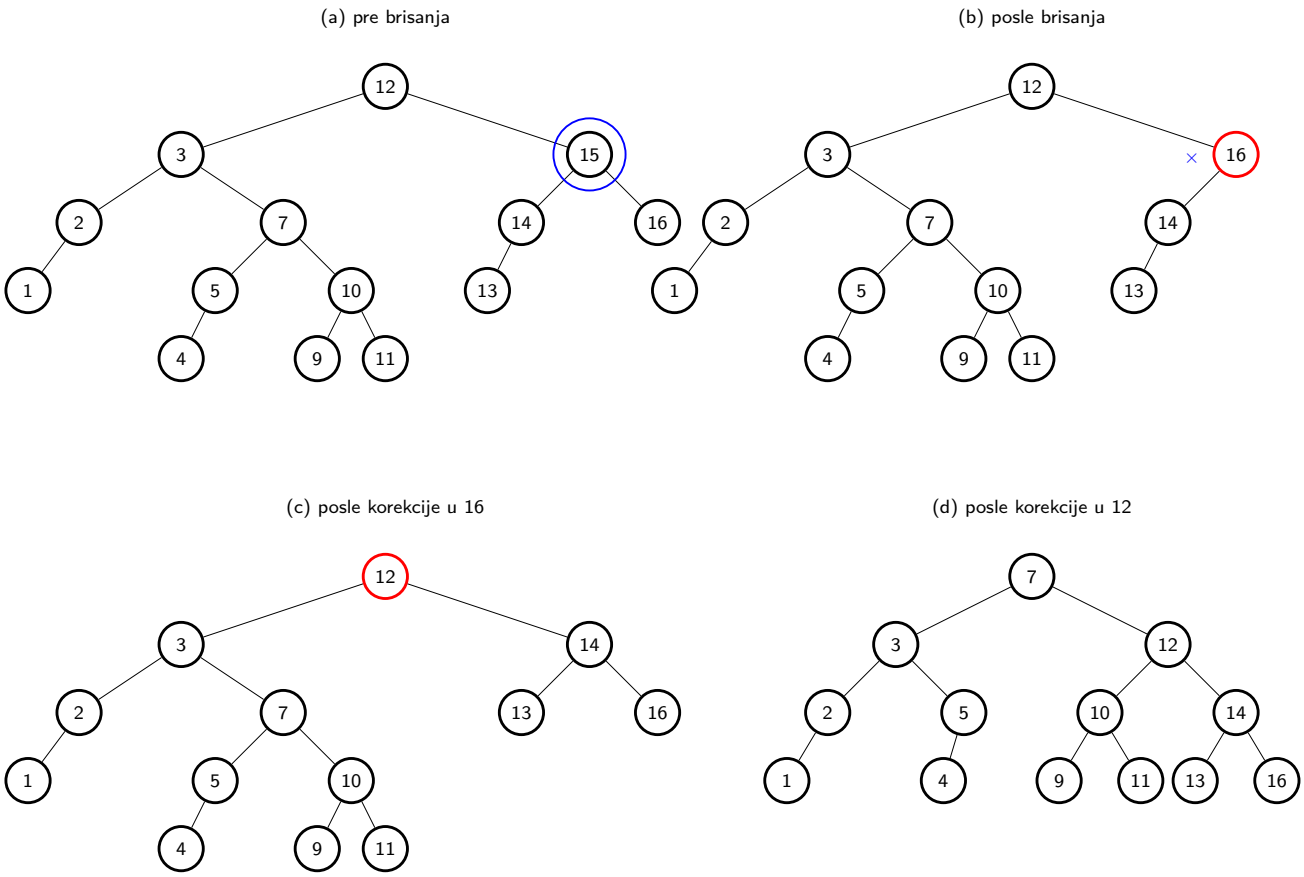
2. **AVL faza (obnavljanje ravnoteže).** Nakon uklanjanja čvora, visina nekog poddrveta može da se *smanji*. Zbog toga je potrebno, duž puta od mesta brisanja ka korenu, ažurirati visine i, po potrebi, sprovoditi rotacije koje vraćaju faktor ravnoteže u skup $\{-1, 0, 1\}$.

Ključna razlika u odnosu na umetanje je sledeća: kod umetanja se visina poddrveta može samo povećati, i nakon korekcije u kritičnom čvoru (prvom neuravnoteženom pretku) visina se tipično "vrati" na prethodnu, pa se korekcija ne propagira dalje. Kod brisanja, međutim, visina poddrveta se može smanjiti, i taj pad se može preneti na pretke, tako da se neuravnoteženost može pojavljivati u više čvorova na putu do korena. Otuda u opštem slučaju balansiranje ne mora stati posle jedne (ili dve) rotacije (Slika 4.14).

U ekstremnim primerima, kao što su Fibonačijeva AVL drveta, brisanje nepovoljno izabranog čvora (npr. lista na najvećoj visini) može zahtevati $h-2$ rotacija, gde je h visina drveta, odnosno $\Theta(\log n)$ rotacija za drvo sa n čvorova. Ipak, pošto je svaka rotacija konstantne složenosti, ukupno vreme izvršavanja brisanja, uključujući i obnavljanje AVL svojstva, ostaje $O(\log n)$.

Nakon brisanja, u svakom čvoru na putu ka korenu mogu nastupiti sledeće situacije:

- visina posmatranog poddrveta ostaje ista, pa se ni faktori ravnoteže iznad ne menjaju značajno;



Slika 4.14: Uravnotežavanje drveta posle brisanja čvora, pomoću dve rotacije.

- visina se smanjuje za 1, pa faktor ravnoteže roditelja može preći iz $\{-1, 0, 1\}$ u $\{-2, 2\}$, čime se narušava AVL uslov.

Zbog toga algoritam brisanja u AVL drvetu ima oblik: **"ukloni kao u BST, pa na povratku ažuriraj i rebalansiraj u svakom čvoru"**. Rekurzivna implementacija upravo prirodno realizuje ovaj obrazac: kada se rekurzija vraća iz levog/desnog poddrveta, u tekućem čvoru se:

1. osveži visina,
2. proveri faktor ravnoteže,
3. po potrebi primeni odgovarajuća rotacija,
4. i zatim se nastavi povratak ka korenu.

Izbor rotacije pri brisanju

Za obnavljanje AVL svojstva koriste se iste četiri rotacije kao kod umetanja: **LL**, **LR**, **RR** i **RL**. Kriterijum izbora ostaje isti: posmatra se čvor A u kome se pojavilo $|f(A)| = 2$, i zatim "orijentacija" težeg deteta.

- Ako je $f(A) = +2$, problem je u levom poddrveta:
  - **LL** ako je levo dete "levo-teže" ili uravnoteženo,
  - **LR** ako je levo dete "desno-teže".
- Ako je $f(A) = -2$, problem je u desnom poddrveta:

  - **RR** ako je desno dete "desno-teže" ili uravnoteženo,
  - **RL** ako je desno dete "levo-teže".

Kod brisanja se može se desiti da "teže" dete ima faktor ravnoteže 0. Tada će odgovarajuća jednostruka rotacija obnoviti uslov $|f| \leq 1$, ali visina rebalansiranog poddrveta *ne mora* da se smanji. U implementacijama koje eksplicitno prate promenu visine, ova situacija može predstavljati tačku u kojoj se propagacija korekcija zaustavlja. U pristupu koji čuva visine i rebalansira na povratku (kao u našem kodu), nije potrebno uvoditi poseban mehanizam za zaustavljanje: korekcija se i dalje vrši lokalno, a rekurzija se vraća do korena u svakom slučaju, uz ukupno $O(\log n)$ rada.

```
/* Iz poddrveta uklanja minimalni cvor i kroz pmin vraca
   njegov pokazivac.
   Funkcija vraca novi koren poddrveta nakon uklanjanja,
   uz AVL rebalansiranje. */
static Cvor* izvadi_min(Cvor* koren, Cvor** pmin)
{
    if (!koren) { *pmin = NULL; return NULL; }

    if (!koren->levo) {
        *pmin = koren;
        return koren->desno;              /* moze biti NULL */
    }

    koren->levo = izvadi_min(koren->levo, pmin);

    osvezi_visinu(koren);
    return rebalansiraj(koren);
}

static Cvor* obrisi_rec(Cvor* koren, Key k, bool* obrisan)
{
    if (!koren) return NULL;

    if (k < koren->kljuc) {
        koren->levo = obrisi_rec(koren->levo, k, obrisan);
    } else if (k > koren->kljuc) {
        koren->desno = obrisi_rec(koren->desno, k, obrisan);
    } else {
        /* pronadjen cvor koji treba ukloniti */
        *obrisan = true;

        /* 0 ili 1 potomak */
        if (!koren->levo || !koren->desno) {
            Cvor* dete = koren->levo ? koren->levo : koren->desno;
            free(koren);
            return dete;                  /* moze biti NULL */
        }
```

```
        /* 2 potomka: zamena inorder-sledbenikom
           (minimum desnog poddrveta) */
        Cvor* min = NULL;
        koren->desno = izvadi_min(koren->desno, &min);
        koren->kljuc = min->kljuc;
        free(min);
    }

    if (!koren) return NULL;        /* slucaj kada je uklonjen list */

    osvezi_visinu(koren);
    return rebalansiraj(koren);
}

bool avl_obrisi(Cvor** pkoren, Key k)
{
    bool obrisan = false;
    *pkoren = obrisi_rec(*pkoren, k, &obrisan);
    return obrisan;
}
```

Napomena (invarijanta povratka iz rekurzije). U rekurzivnoj implementaciji brisanja održava se sledeća invarijanta: nakon što se iz levog ili desnog poddrveta vrati pokazivač na njegov (eventualno izmenjen) koren, to poddrvo već zadovoljava AVL uslov, a vrednost polja `visina` u njegovim čvorovima je korektno ažurirana. Zatim se u tekućem čvoru osvežava njegova visina i primenjuje `rebalansiraj`, čime se obezbeđuje da i poddrvo sa tekućim čvorom kao korenom ponovo bude AVL. Indukcijom po dubini povratka sledi da je nakon završetka procedure celo stablo uređeno i balansirano (AVL).

Složenost

Pošto AVL stablo sa n čvorova ima visinu $O(\log n)$, procedura brisanja (BST faza) obilazi put dužine $O(\log n)$. Na povratku se u svakom posećenom čvoru vrši konstantan broj operacija: osvežavanje visine, izračunavanje faktora ravnoteže i eventualno jedna rotacija (jednostruka ili dvostruka). Zato je ukupno vreme brisanja sa balansiranjem

$$O(\log n).$$

Primene AVL drveta

AVL drveta su samobalansirajuća uređena binarna stabla kod kojih je visina u najgorem slučaju logaritamska u funkciji broja čvorova. Zbog toga su osnovne operacije — *pretraga*, *umetanje* i *brisanje* — u najgorem slučaju složenosti $O(\log n)$.

Istorijski, AVL drveta su među prvim formalno uvedenim samobalansirajućim strukturama ove vrste i razvijena su upravo iz potrebe da se u sistemima sa velikom količinom podataka obezbedi pouzdana (najgora-slučaj) efikasnost operacija nad indeksima. U praksi su naročito pogodna u scenarijima u kojima se **pretrage obavljaju često**, dok su **umetanja i brisanja relativno ređa**: strožiji uslov balansiranosti u odnosu na neke druge varijante balansiranih stabala tipično vodi nešto većem trošku ažuriranja, ali stabilnijoj visini i predvidljivijem vremenu pretrage.

AVL drveta (odnosno, šire, balansirana uređena binarna stabla) koriste se u nizu sistemskih i aplikativnih domena koji zahtevaju efikasno pretraživanje i održavanje poretka: pri implementaciji internih kolekcija poput uređenih skupova i uređenih mapa, pri organizaciji indeksa u pretraživačima, pri održavanju struktura u fajl-sistemima, kao i u tabelama simbola u kompajlerima i interpreterima (gde se uz identifikator čuvaju i pridružene informacije, kao što su tip, doseg i slično). U sistemima sa realno-vremenskim ograničenjima dodatna prednost je upravo **garantovana** složenost u najgorem slučaju.

U standardnoj biblioteci C++ uređene asocijativne strukture podataka realizuju se kroz kontejnere:

- `std::set` i `std::multiset` (uređeni skupovi),
- `std::map` i `std::multimap` (uređene mape, tj. parovi *ključ–vrednost*).

Ovi kontejneri održavaju elemente *sortirane* po ključu i garantuju da su osnovne operacije *pretrage*, *umetanja* i *brisanja* logaritamske složenosti u veličini kolekcije. Pored toga, operacije za rad sa opsezima i „granicama“ u sortiranom poretku, kao što su `lower_bound`, `upper_bound` i `equal_range`, takođe su logaritamske.

Važno je naglasiti da C++ standard *ne propisuje* kojom se tačno strukturom ovi kontejneri implementiraju; zahtevi se formulišu kroz semantiku i složenosti. U tipičnim implementacijama standardnih biblioteka, `std::map` i `std::set` se realizuju najčešće pomoću crveno-crnih stabala (eng. red–black trees). AVL stabla su konceptualno veoma bliska: i ona obezbeđuju isti tip logaritamskih garancija, ali uz strožiji kriterijum balansiranosti, što se u praksi često prevodi u drugačiji kompromis između cene ažuriranja i cene pretrage.

U tabeli su navedene najtipičnije operacije nad `std::set`/`std::map` i njihova očekivana složenost (za n elemenata):

Operacija	Značenje	Složenost
`find(k)`	pronalazak elementa po ključu	$O(\log n)$
`insert(x)` / `emplace(...)`	umetanje elementa	$O(\log n)$
`erase(k)` / `erase(it)`	brisanje po ključu ili iteratoru	$O(\log n)$
`lower_bound(k)` / `upper_bound(k)`	granice opsega u sortiranom poretku	$O(\log n)$
iteracija (`begin()` … `end()`)	obilazak elemenata u rastućem poretku ključeva	linearno u broju obilazaka; poredak je sortiran

Ova veza sa C++ je korisna i iz praktičnog ugla: kada je potreban *sortirani poredak* i operacije tipa *„nađi prvi* $\geq k$*“*, *„svi elementi u intervalu* $[a, b]$*“* i slično, tada se u C++ prirodno bira `std::set`/`std::map` (balansirano stablo). Kada poredak nije bitan, a prioritet je prosečno brzo pristupanje, tada se tipično koriste heš varijante `std::unordered_set` i `std::unordered_map` (heš tabele), o čemu je bilo reči u poglavlju o heširanju.

4.3 Heširanje i heš-tabela

Heširanje (engl. *hashing*), koje se često naziva i **direktna organizacija podataka**, jedan je od osnovnih načina za implementaciju **neuređenih skupova** i **neuređenih mapa**. Ove strukture podataka

služe za čuvanje zapisa (slogova) koji se identifikuju pomoću nekog ključnog atributa, tj. **ključa**. Kod skupova se čuvaju samo ključevi, dok se kod mapa uz svaki ključ čuva i odgovarajuća vrednost.

Na primer, u mapi (rečniku) ključevi mogu biti JMBG brojevi osoba, dok slogovi uz ključ sadrže i ime, prezime i dodatne informacije o osobama.

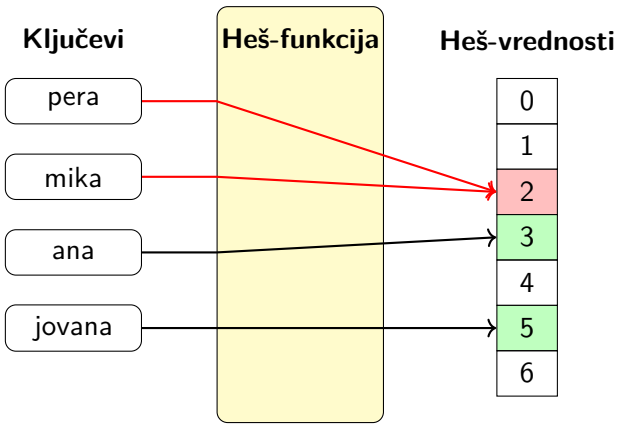
Osnovni cilj heširanja je da se najčešće operacije nad podacima — *pretraga*, *ubacivanje* i *brisanje* — izvršavaju u očekivanom konstantnom vremenu, tj. u vremenskoj složenosti $\mathbf{O(1)}$. Ovo predstavlja značajnu prednost u odnosu na strukture podataka kod kojih su iste operacije složenosti $O(n)$ ili $O(\log n)$.

Heširanje omogućava efikasno:

- dodavanje novih elemenata u skup ili mapu,
- pretragu na osnovu ključa (proveru postojanja ključa ili pronalaženje njemu pridružene vrednosti),
- brisanje slogova sa zadatim ključem,
- izmenu vrednosti pridružene datom ključu.

Sa druge strane, za razliku od uređenih struktura podataka (npr. uređenih stabala), heš-tabele *ne omogućavaju ispis elemenata u sortiranom redosledu*, *niti efikasno pronalaženje ključa koji je najbliži zadatom*.

Osnovna ideja heširanja je da se na osnovu vrednosti ključa direktno odredi pozicija u tabeli na kojoj se odgovarajući podatak nalazi. Ova ideja se formalizuje uvođenjem **heš-funkcije**, koja svakom ključu pridružuje indeks u tabeli.



Slika 4.15: Ilustracija heš tabele.

Heš-tabela je struktura podataka koja koristi heš-funkciju oblika

$$h : U \to 0, 1, \ldots, m - 1,$$

gde je U skup svih mogućih ključeva (tzv. univerzum), a m veličina heš-tabele. Zadatak heš-funkcije je da ključeve iz potencijalno veoma velikog prostora preslika u relativno mali skup indeksa tabele.

Međutim, pošto je broj mogućih ključeva u realnim problemima gotovo uvek znatno veći od broja pozicija u tabeli, **kolizije** su neizbežne: različiti ključevi mogu biti preslikani na isti indeks. Zbog toga heš-tabela nije određena samo heš-funkcijom, već i **strategijom rešavanja kolizija**. Drugim rečima,

heš-tabela = heširanje + rešavanje kolizija.

4.3.1 Heš-funkcije

Uloga heš-funkcije i osnovni pojmovi

Heširanje podrazumeva korišćenje tzv. **heš-funkcije**, koja svakom ključu pridružuje ceo broj, koji se naziva **heš-vrednost** ili **heš-kod** ključa. Na osnovu ove vrednosti određuje se pozicija u **heš-tabeli** na kojoj se čuva odgovarajući podatak.

Heš-tabela je najčešće implementirana kao niz. Svaka pozicija niza može čuvati jedan slog, ili — u zavisnosti od izabrane strategije — više slogova. Pozicije u tabeli često se nazivaju **slotovi** (engl. *slots*) ili **kofe** (engl. *buckets*).

Ako je dimenzija heš-tabele n, heš-funkcija preslikava ključeve iz proizvoljno velikog skupa u skup indeksa

$$[0, n).$$

Najjednostavniji slučaj heširanja javlja se kada je moguće uspostaviti bijekciju između skupa ključeva i dozvoljenih pozicija u tabeli. Na primer, ako svakom malom slovu engleske abecede želimo da dodelimo jedinstvenu poziciju u nizu od 26 elemenata, heš-funkcija može biti

$$h(c) = c - \text{'a'},$$

čime se slovo a preslikava u indeks 0, b u 1 itd. U ovom slučaju kolizije ne postoje, jer je preslikavanje injektivno.

Međutim, u realnim primenama broj potencijalnih ključeva gotovo je uvek znatno veći od dimenzije heš-tabele. Zbog toga heš-funkcija ne može biti injektivna i **kolizije su neizbežne** — različiti ključevi mogu imati istu heš-vrednost i biti preslikani na istu poziciju u tabeli.

Prilikom primene heširanja neophodno je rešiti dva osnovna problema:

- izbor heš-funkcije kojom se smanjuje verovatnoća nastanka kolizija,
- rešavanje kolizija kada one nastanu.

Na broj kolizija utiče i veličina heš-tabele, pa se u praksi heš-tabele često **dinamički proširuju** kada se tabela u velikoj meri popuni i kada broj kolizija postane prevelik.

Kolizije kao neizbežna pojava

Kolizija je situacija u kojoj dva ili više različitih ključeva imaju istu heš-vrednost, odnosno bivaju preslikani na istu poziciju u heš-tabeli. Drugim rečima, različiti ključevi se „sudaraju“ u istoj kofi (slotu) tabele.

Na broj kolizija utiče i **veličina heš-tabele**. Ako je tabela mala u odnosu na broj elemenata koji se u nju smeštaju, kolizije će se javljati često, a performanse će opadati. Zbog toga se heš-tabele u praksi često **dinamički proširuju**, odnosno vrši se ponovno heširanje (*rehash*) kada popunjenost tabele pređe određenu granicu.

Važno je naglasiti da se kolizije ne javljaju tek kada se tabela „popuni“. Čak i kada je tabela gotovo prazna, verovatnoća njihovog nastanka može biti značajna. Ova činjenica može se kvantitativno objasniti sledećom analizom.

Teorema 4.3.1. *Pretpostavimo model **ravnomernog heširanja**, u kome se svaki ključ, nezavisno od ostalih, preslikava ravnomerno na neku od n pozicija u tabeli. Razmotrimo situaciju u kojoj se u heš-tabelu veličine n ubacuje d elemenata, pri čemu je $d \leq n$. Za $d \gtrsim 1.18\sqrt{n}$ verovatnoća kolizije je veća od 0.5.*

Dokaz. Najpre posmatramo verovatnoću da **ne dođe do kolizije**, tj. da svi ključevi budu smešteni na različite pozicije.

- Prvi ključ može u bilo koji slot: verovatnoća 1.
- Drugi mora u neki od preostalih $n-1$: $\frac{n-1}{n}$.
- Treći mora u preostalih $n-2$: $\frac{n-2}{n}$.
- ...
- d-ti mora u preostalih $n-d+1$: $\frac{n-d+1}{n}$.

Zato je

$$\text{Pr(nema kolizije)} = \left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{d-1}{n}\right).$$

Otuda je verovatnoća da se **pojavi bar jedna kolizija**:

$$\text{Pr(kolizija)} = 1-\left(1-\frac{1}{n}\right)\left(1-\frac{2}{n}\right)\cdots\left(1-\frac{d-1}{n}\right).$$

Za velike vrednosti n i relativno male odnose $\frac{i}{n}$, ovaj izraz se može aproksimirati korišćenjem standardne aproksimacije $\ln(1-x) \approx -x$, tj. $1-x \approx e^{-x}$ čime se dobija

$$\text{Pr(kolizija)} \approx 1-e^{-\frac{1}{n}}e^{-\frac{2}{n}}\cdots e^{-\frac{d-1}{n}} = 1-e^{-\frac{1}{n}(1+2+\cdots+(d-1))}.$$

Kako je $1+2+\cdots+(d-1) = \frac{d(d-1)}{2}$, dobija se

$$\text{Pr(kolizija)} \approx 1-e^{-\frac{d(d-1)}{2n}} \approx 1-e^{-\frac{d^2}{2n}}.$$

Posmatrajmo kada je ova verovatnoća veća od 0.5. To važi kada $1-e^{-\frac{d^2}{2n}} \geq 0.5$, tj. $e^{-\frac{d^2}{2n}} \leq 0.5$. Kako je $\ln 0.5 = -\ln 2$, sledi da uslov važi kada

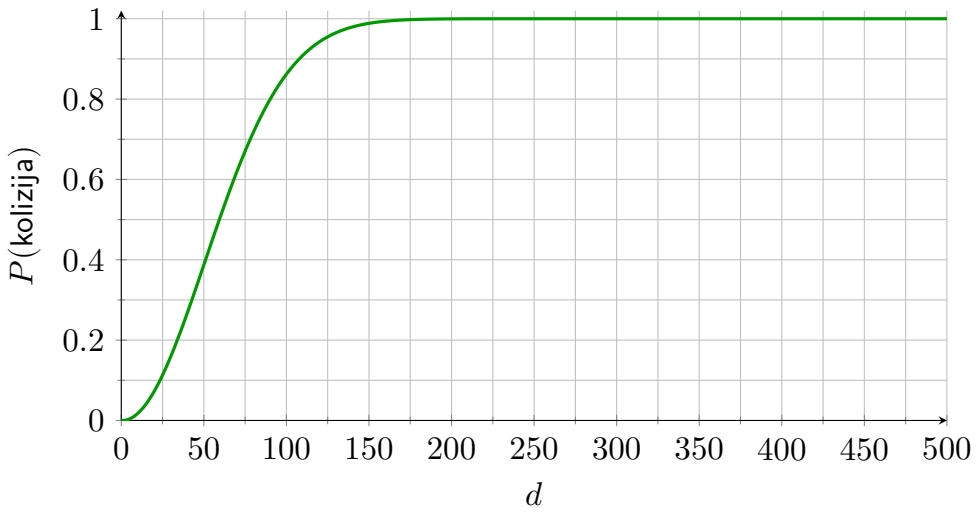
$$-\frac{d^2}{2n} \leq \ln 0{,}5 \quad \Longleftrightarrow \quad d^2 \geq 2n \ln 2.$$

Dakle, verovatnoća pojave kolizije veća je od 0.5 već kada važi

$$d \geq \sqrt{2n \ln 2} \approx 1{,}18\sqrt{n}.$$

□

Drugim rečima, već nakon ubacivanja reda veličine $\sqrt{n}$ elemenata postoji velika verovatnoća da se bar jedna kolizija pojavi, iako je tabela tada još uvek skoro prazna. Ovaj rezultat je u tesnoj vezi sa tzv. **rođendanskim paradoksom**, prema kome je u grupi od samo 23 osobe verovatnije da dve osobe imaju isti rođendan nego da svi imaju različite datume rođenja.



Slika 4.16: Verovatnoća da u tabeli sa 2500 elemenata postoji kolizija nakon ubacivanja d elemenata.

Na primer, ako heš-tabela ima milion pozicija $n = 10^6$, tada je $d \approx 1.18\sqrt{10^6} \approx 1.18 \cdot 1000 \approx 1180$. Dakle, već posle oko 1180 ubacivanja postoji preko 50% šanse da se desila bar jedna kolizija — iako je tabela popunjena tek oko 0.118%.

> Iz ove analize sledi da samo povećavanje veličine heš-tabele može smanjiti učestalost kolizija, ali ih ne može ukloniti, pa je izbor odgovarajuće strategije rešavanja kolizija ključan za efikasnost heš-tabele.

Iz svega navedenog sledi da kolizije ne treba posmatrati kao izuzetak ili grešku, već kao **normalnu i stalno prisutnu pojavu** kod heširanja. Cilj dobre implementacije nije da kolizije eliminiše, već da njihov broj zadrži malim i da obezbedi da njihovo rešavanje ne naruši očekivanu konstantnu složenost osnovnih operacija.

Izbor heš-funkcije

Kao što je pokazano u prethodnom odeljku, kolizije su neizbežna pojava kod heširanja. Zbog toga je izbor heš-funkcije od ključnog značaja za efikasnost heš-tabele. Dobra heš-funkcija treba da zadovolji dva osnovna zahteva:

- da se **izračunava brzo**, poželjno u vremenu O(1) sa malim konstantnim faktorom;
- da **ravnomerno raspoređuje ključeve** po pozicijama heš-tabele, čime se smanjuje broj kolizija.

Brzina izračunavanja heš-funkcije je posebno važna jer se ona koristi pri svakoj osnovnoj operaciji nad heš-tabelom: ubacivanje, pretraga, brisanje i izmena elemenata. Zbog toga je neprihvatljivo koristiti heš-funkcije koje se sporo izračunavaju, čak i ako bi one imale dobre teorijske osobine.

Drugi, podjednako važan zahtev je **ravnomernost raspodele**. Ravnomernost raspodele je svojstvo heš-funkcije da ključeve preslikava na pozicije u heš-tabeli tako da su sve pozicije približno jednako opterećene.

Drugim rečima, dobra heš-funkcija ne bi trebalo da proizvodi oblasti tabele u kojima se nagomilava veliki broj elemenata, dok su druge oblasti gotovo neiskorišćene. Ravnomerna raspodela smanjuje

verovatnoću nastanka kolizija i sprečava pojavu loših performansi usled sistematskog klasterisanja ključeva.

Važno je naglasiti da ravnomernost ne znači nasumičnost u strogom smislu, već odsustvo pravilnosti koje bi bile povezane sa strukturom ključeva (npr. zajednički delitelji, isti krajnji bitovi, slični prefiksi).

Mnoge naizgled jednostavne heš-funkcije ne zadovoljavaju ovaj zahtev. Na primer, ako bi se za smeštanje JMBG brojeva koristila heš-tabela od 1000 elemenata, a heš-vrednost bi se određivala samo na osnovu tri cifre koje predstavljaju godinu rođenja, došlo bi do izrazito neravnomerne raspodele. Veliki broj ključeva bio bi preslikan na mali broj pozicija koje odgovaraju određenim godinama, dok bi ostatak tabele ostao gotovo neiskorišćen. Ovakva heš-funkcija bi proizvodila veliki broj kolizija i loše performanse, iako je njeno izračunavanje veoma brzo.

U praksi se heširanje često izvodi u dva koraka. U prvom koraku ključ se preslikava u neki ceo broj (često širine 32 ili 64 bita), a u drugom koraku se ta vrednost preslikava u opseg indeksa heš-tabele, najčešće određivanjem ostatka pri deljenju sa veličinom tabele n.

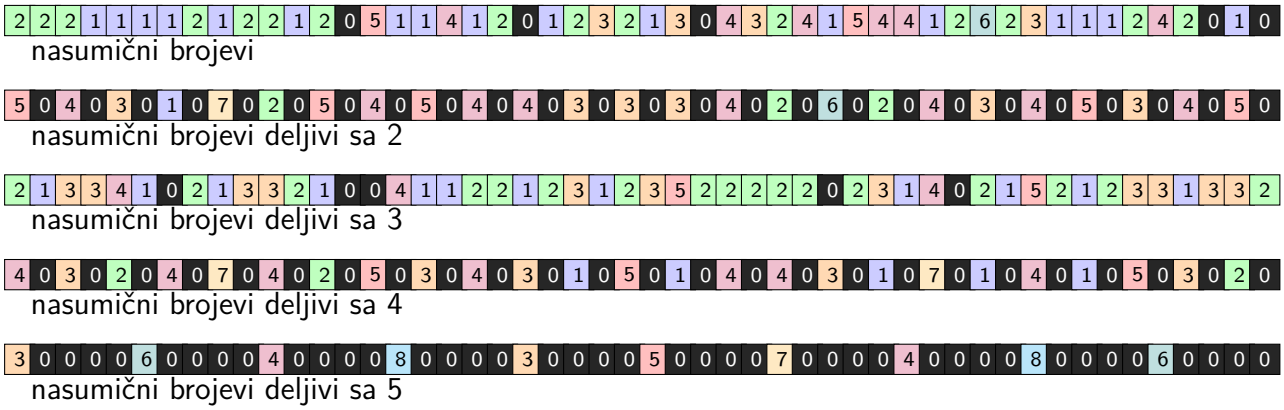
Ako je ključ celobrojan i ako je veličina tabele prost broj, često se koristi funkcija oblika

$$h(x) = x \bmod n.$$

Ovakav pristup je jednostavan i brz, ali njegov kvalitet u velikoj meri zavisi od odnosa između vrednosti ključeva i broja n. Ako n nije pažljivo izabran, mogu se pojaviti izraženi obrasci u raspodeli ključeva.

Posebno problematičan slučaj nastaje kada je veličina tabele stepen broja 2. Tada se operacija x mod n svodi na izdvajanje određenog broja krajnjih bitova u binarnom zapisu ključa, što je veoma efikasno sa stanovišta brzine. Međutim, u tom slučaju heš-vrednost zavisi isključivo od krajnjih bitova ključa, dok se informacije sadržane u višim bitovima potpuno zanemaruju.

Veličina tabele 50, koristi se heš-funkcija x mod 50:



Slika 4.17: Ako je veličina tabele 50 i ako se heširaju brojevi koji su svi deljivi sa 2, 3, 4 ili 5, raspodela elemenata po tabeli može biti izrazito neravnomerna, što dovodi do velikog broja kolizija i neiskorišćenosti velikog dela tabele. Ovakvi primeri ilustruju da ni naizgled dobra heš-funkcija ne mora dati dobre rezultate za sve skupove ključeva..

Da bi se izbegli problemi vezani za zavisnost od krajnjih bitova, u praksi se često primenjuje dodatno *menjanje* vrednosti ključa pre operacije preslikavanja u opseg indeksa tabele. Čest izbor su funkcije oblika

$$h(x) = (a \cdot x + b) \bmod p,$$

gde je p prost broj veći od veličine tabele, a konstante a i b biraju se tako da se razbiju pravilnosti u raspodeli ključeva. Ovakav pristup koristi informacije iz svih bitova ključa i značajno smanjuje mogućnost sistematskog nagomilavanja elemenata.

Važno je naglasiti da za svaku heš-funkciju može postojati neka familija ključeva koja dovodi do velikog broja kolizija i loših performansi. Zbog toga se u nekim slučajevima primenjuje **univerzalno heširanje**. Univerzalno heširanje je pristup u kome se heš-funkcija ne fiksira unapred, već se nasumično bira iz unapred definisanog skupa dobrih heš-funkcija.

Osnovna ideja univerzalnog heširanja je da se, iako za svaku pojedinačnu heš-funkciju može postojati nepovoljna familija ključeva koja izaziva veliki broj kolizija, verovatnoća lošeg ponašanja značajno smanjuje ako se konkretna funkcija bira nasumično.

Na taj način se garantuje da je matematičko očekivanje broja kolizija malo, nezavisno od skupa ključeva koji se pojavljuje na ulazu. Univerzalno heširanje se često poredi sa nasumičnim izborom pivota u brzom sortiranju, gde se sličnim mehanizmom izbegavaju patološki slučajevi.

Množenje i šiftovanje (engl. *multiply–shift*)

Kod celobrojnih ključeva, veoma čest obrazac u praksi je da kod ključa najpre razbiju pravilnosti, a zatim da se iz dobijene vrednosti izvuče indeks u tabeli. Jedna od najbržih i najpopularnijih varijanti je **množenje i šiftovanje**.

Neka je dužina mašinske reči w (tipično $w = 32$ ili $w = 64$). Biraju se dva parametra:

- a — neparan broj, $0 < a < 2^w$
- b — neki broj (tzv. "salt"), najčešće $0 \leq b < 2^w$

Heš-funkcija se zatim definiše kao:

$$h(x) = ((ax + b) \bmod 2^w) \operatorname{div} 2^{w-k}.$$

Intuitivno: izračuna se $a \cdot x + b$ u w-bitnoj aritmetici (prekoračenje se zanemari), a zatim se uzme k *najviših bitova* dobijene vrednosti.

Parametar a se bira kao neparan broj, jer množenje $a \cdot x$ uvek ima *najmanje značajan bit jednak nuli* (a često i nekoliko najnižih bitova), pa se u startu uvodi sistematska pravilnost u heš-vrednostima, što može dovesti do neravnomerne raspodele i većeg broja kolizija.

Pošto se a i b biraju (pseudo)slučajno, ovo je primer *univerzalnog heširanja*: ne oslanja se na jednu fiksnu funkciju koja može imati "lošu familiju ključeva", već parametre bira tako da je verovatnoća lošeg ponašanja mala.

U mnogim savremenim implementacijama heš-tabela veličina tabele se bira kao *stepen dvojke*, $m = 2^k$. Razlog je praktičan: preslikavanje u opseg $[0, m)$ tada može da se uradi veoma brzo, a i strategije proširivanja (dupliranje) su jednostavne. To nije univerzalno pravilo: neke implementacije koriste proste brojeve ili druge politike, ali 2^k je veoma čest izbor u praksi upravo zato što omogućava jeftinu aritmetiku.

U C implementacije se koriste **neoznačeni** tip (`uint32_t`), jer je tada prekoračenje definisano kao računanje modulo 2^{32}. Tada formula iznad postaje:

$$h(x) = (ax + b) \gg (32 - k).$$

```c
#include <stdint.h>
#include <stdlib.h>   /* rand() */

/* Parametri heš-funkcije (biraju se jednom po tabeli) */
typedef struct {
    uint32_t a;   /* neparan */
    uint32_t b;
    unsigned k;   /* tabela ima 2^k elemenata */
} HashMS;

/* Inicijalizacija: biraju se a i b
   (ovde preko rand(), radi jednostavnosti) */
HashMS hashms_init(unsigned k)
{
    HashMS h;
    h.k = k;

    /* izbor parametara */
    srand((unsigned)time(NULL));

    h.a = ((uint32_t)rand() << 16) ^ (uint32_t)rand();
    h.a |= 1u;                  /* obavezno neparan */

    h.b = ((uint32_t)rand() << 16) ^ (uint32_t)rand();
    return h;
}

/* Heš: rezultat je u opsegu [0, 2^k) */
uint32_t hashms(HashMS h, uint32_t x)
{
    uint32_t y = h.a * x + h.b;          /* mod 2^32 (prekoračenje
                                            se zanemaruje) */
    return y >> (32u - h.k);             /* uzma se k najviših bitova */
}
```

Funkcija `hashms_init` se poziva jednom pri kreiranju tabele, a `hashms` pri svakoj operaciji nad ključem.

Heširanje složenih ključeva (stringovi, nizovi)

U dosadašnjim primerima pretpostavljali smo da je ključ celobrojna vrednost. Međutim, u mnogim praktičnim problemima ključ je *složen*: niz vrednosti ili niska karaktera. U tom slučaju heširanje se obično sprovodi u dva koraka.

U **prvom koraku** složen ključ se preslikava u jednu celobrojnu vrednost (najčešće 32-bitnu ili 64-bitnu), tj. vrši se neka vrsta **agregacije** svih elemenata ključa u jedan broj. U **drugom koraku**

dobijena vrednost se preslikava u opseg indeksa heš-tabele (tehnikama opisanim u prethodnom delu).

Najjednostavniji pristup je da se elementi niza kombinuju korišćenjem operacija kao što su sabiranje, množenje konstantom, ili bitovske operacije (npr. ekskluzivna disjunkcija, XOR). Ideja je da se informacija iz svih delova ključa "rasporedi" u dobijenoj celobrojnoj vrednosti.

Važno je naglasiti da trivijalne agregacije (npr. prosto sabiranje karaktera) često daju loše rezultate, jer različite niske mogu imati isti zbir ili mogu stvarati pravilnosti koje dovode do povećanog broja kolizija. Zbog toga se u praksi koriste agregacije koje tokom obrade svakog elementa uvode i neku vrstu "mešanja" (npr. množenje prethodne vrednosti konstantom, pa dodavanje trenutnog elementa).

Jedan klasičan i široko korišćen pristup za heširanje stringova zasniva se na tumačenju elemenata niske kao koeficijenata polinoma. Ako je niska $s = s_0 s_1 \dots s_{k-1}$, može se definisati heš-vrednost oblika

$$h(s) = \left(\sum_{i=0}^{k-1} s_i \cdot p^i \right) \bmod m,$$

gde je p neka fiksna konstanta (najčešće mali prost broj), a m je broj pozicija u tabeli. Intuitivno, ovakva definicija razlikuje položaj karaktera u stringu, pa niske koje imaju iste karaktere u različitom redosledu dobijaju različite vrednosti (za razliku od prostog sabiranja karaktera).

Posebno jednostavna ilustracija ove ideje je sledeća: ako se niska sastoji samo od 26 slova engleskog alfabeta, može se tumačiti kao broj zapisan u osnovi 26. Na primer, za nisku $c_0 c_1 \dots c_{k-1}$ može se posmatrati vrednost

$$c_0 \cdot 26^{k-1} + c_1 \cdot 26^{k-2} + \cdots + c_{k-1} \cdot 26^0,$$

pri čemu se u praksi prekoračenje zanemaruje (radi se u aritmetici ograničene širine).

U implementacijama se polinomsko heširanje obično ne računa direktno preko stepena, već *&iterativno*, tako da se string prolazi sleva nadesno:

$$H_{i+1} = H_i \cdot p + s_i.$$

Ova forma je pogodna za implementaciju, a vrednost se tipično računa u `uint64_t` (ili `uint32_t`), pri čemu se prekoračenje automatski odseca na 2^{64} (ili 2^{32}) i time implicitno dobija modularna aritmetika. Nakon toga se dobijeni broj preslikava u opseg indeksa heš-tabele.

Implementacija heš funkcije nad niskom:

```
#include <stdint.h>
#include <stddef.h>

/* Polinomsko heširanje stringa, pa mapiranje u [0, 2^{h.k}). */
size_t hash_string_poly(const char* s, HashMS64 h)
{
    const uint64_t p = 131;      /* tipična mala konstanta (može i 31, 53, 127, ...)
    uint64_t acc = 0;

    for ( ; *s != '\0'; s++) {
```

```
        acc = acc * p + (unsigned char)(*s);
    }

    return (size_t)hashms64(H, acc);
}
```

Vrednost se računa u `uint64_t`, pri čemu se prekoračenje automatski zanemaruje, što implicitno daje aritmetiku modulo 2^{64}.

Kod ključeva promenljive dužine, kao što su niske karaktera, izračunavanje heš-vrednosti nužno zahteva obradu svih (ili bar većine) karaktera ključa. Zbog toga se složenost izračunavanja heš-funkcije u ovom slučaju ne može posmatrati kao konstantna, već zavisi od dužine niske.

Pri analizi performansi heš-tabela korisno je razlikovati dva aspekta složenosti:

- složenost u odnosu na broj elemenata u tabeli (koja je, uz odgovarajuće pretpostavke, očekivano konstantna),
- složenost koja potiče od obrade samog ključa (koja zavisi od dužine ključa).

Prema tome, za ključ koji je niska s, operacije nad heš-tabelom imaju očekivanu složenost

$$O(|s| + 1) = O(|s|),$$

gde $|s|$ označava dužinu niske. Član $O(1)$ odgovara pristupu tabeli i rešavanju kolizija (u očekivanom konstantnom vremenu), dok član $O(|s|)$ potiče od izračunavanja heš-vrednosti ključa (i, po potrebi, poređenja ključeva).

Zaključak: svojstva dobre heš-funkcije

Izbor heš-funkcije direktno utiče na broj kolizija i, posledično, na efikasnost heš-tabele. Dobra heš-funkcija treba da zadovolji sledeća svojstva:

- **Brzina izračunavanja.** Heš-vrednost se računa pri svakoj operaciji nad tabelom (ubacivanje, pretraga, brisanje, izmena), pa je poželjno da se heš-funkcija izračunava u vremenu O(1), odnosno da ima mali konstantni faktor. Kod ključeva promenljive dužine (npr. stringova) izračunavanje heša nužno zahteva prolazak kroz ključ, pa je prirodno očekivati složenost $O(|s|)$, gde je $|s|$ dužina ključa.
- **Ravnomerna raspodela.** Heš-funkcija treba da preslikava ključeve tako da se elementi raspoređuju približno jednako po svim pozicijama tabele, čime se smanjuje mogućnost nagomilavanja i prevelikog broja kolizija. Ravnomernost raspodele je važnija od "slučajnosti" u intuitivnom smislu: cilj je izbeći sistematske pravilnosti povezane sa strukturom ključeva.
- **Stabilnost.** Za isti ključ heš-funkcija mora uvek vraćati istu vrednost (odnosno isti indeks za datu veličinu tabele i fiksirane parametre), jer se u suprotnom ne može obezbediti korektna pretraga i brisanje već ubačenih elemenata.

Na kraju, napomenimo da je teorijski moguće konstruisati **savršenu heš-funkciju** ukoliko je unapred poznat konačan skup ključeva koji će se javljati u programu. Takva heš-funkcija je injektivna na tom skupu i preslikava svaki ključ u jedinstvenu poziciju u tabeli odgovarajuće veličine, pa kolizije ne

postoje. Međutim, konstrukcija savršenih heš-funkcija je u opštem slučaju algoritamski i implementaciono zahtevna, a dodatno pretpostavlja da se skup ključeva ne menja; zbog toga se u većini praktičnih primena koriste heš-funkcije koje dobro raspoređuju ključeve, uz eksplicitnu strategiju rešavanja kolizija.

4.3.2 Razrešavanje kolizija

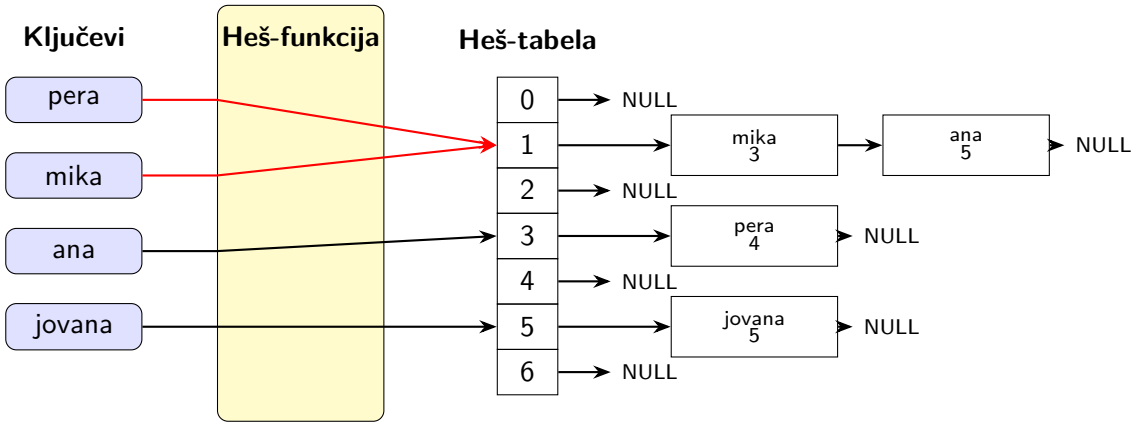
Kolizije su neizbežne, pa je svaka praktična heš-tabela definisana ne samo heš-funkcijom, već i tehnikom razrešavanja kolizija. Postoji više pristupa, ali dva osnovna su:

- **odvojeno ulančavanje** (engl. *separate chaining*),
- **otvoreno adresiranje** (engl. *open addressing*).

U nastavku prvo razmatramo ulančavanje, jer je konceptualno najjednostavnije i prirodno se oslanja na strukture podataka koje su već uvedene (liste).

Odvojeno ulančavanje

Odvojeno ulančavanje (engl. *separate chaining*), koje se ponekad naziva i zatvoreno adresiranje ili otvoreno heširanje, zasniva se na sledećoj ideji: heš-tabela je niz slotova, a na svakoj poziciji niza čuva se **lanac** svih slogova čiji se ključ heš-funkcijom preslikava na tu poziciju. Najčešće se lanac implementira kao **povezana lista**.



Slika 4.18: Ilustracija razrešavanja kolizija – odvojeno ulančavanje.

Ako heš-tabela ima n slotova, a za ključ k važi $i = h(k)$, tada se slog sa ključem k nalazi u lancu koji pripada slotu sa indeksom i.

U narednom delu je dat kod kojim je definisana heš-tabela:

```
typedef enum {
    OK,
    NEDOVOLJNO_MEMORIJE,
    NEMA_ELEMENATA
} status;

/* Slog u lancu: (kljuc, vrednost) */
```

```
typedef struct hm_slog {
    int kljuc;
    int vrednost;
    struct hm_slog* sledeci;
} hm_slog;

/* Heš-mapa sa odvojenim ulančavanjem */
typedef struct {
    hm_slog** slot;        /* slot[i] je sentinel-slog */
    uint32_t  slot_broj;   /* broj slotova (= 2^k) */
    uint32_t  elem_broj;   /* broj elemenata u tabeli */
    unsigned  k;           /* jer je slot_broj = 2^k */
    HashMS    H;           /* heš-parametri */
} HesMapa;
```

Prilikom kreiranja heš-tabele, potrebno je inicijalizovati veličinu tabele, odnosno broj slotova koje tabela ima:

```
status hm_inicijalizuj(HesMapa* m, unsigned k)
{
    if (!m || k == 0 || k > 31) return NEDOVOLJNO_MEMORIJE;

    m->k = k;
    m->H = hashms_init(k);
    m->slot_broj = (uint32_t)(1u << k);
    m->elem_broj = 0;

    m->slot = (hm_slog**)malloc(m->slot_broj * sizeof(hm_slog*));
    if (!m->slot) return NEDOVOLJNO_MEMORIJE;

    for (uint32_t i = 0; i < m->slot_broj; i++) {
        /* kreira se sentinel na svakom slotu */
        m->slot[i] = novi_slog(0, 0, NULL);
    }

    return OK;
}
```

Funkcija `novi_slog` kreira novi cvor na standardan način:

```
static hm_slog* novi_slog(int kljuc, int vrednost, hm_slog* sledeci)
{
    hm_slog* p = (hm_slog*)malloc(sizeof(*p));
    if (!p) return NULL;
    p->kljuc = kljuc;
    p->vrednost = vrednost;
    p->sledeci = sledeci;
    return p;
```

```
}
```

Osnovne operacije su:

- **Pretraga.** Najpre se izračuna indeks $i = h(k)$, a zatim se vrši linearna pretraga lanca u slotu i dok se ne pronađe slog sa ključem k ili dok se lanac ne iscrpi.

```
status hm_nadji(const HesMapa* m, int kljuc, int* pvrednost)
{
      uint32_t idx = hashms(m->H, kljuc);
      hm_slog* s = m->slot[idx];

      for (hm_slog* p = s->sledeci; p != NULL; p = p->sledeci) {
          if (p->kljuc == kljuc) {
              *pvrednost = p->vrednost;
              return OK;
          }
      }
      return NEMA_ELEMENATA;
 }
```

- **Brisanje.** Brisanje se svodi na pronalaženje elementa u odgovarajućem lancu i njegovo uklanjanje iz liste.

```
status hm_obrisi(HesMapa* m, int kljuc)
{

  uint32_t idx = hashms(m->H, (uint32_t)kljuc);
  hm_slog* s = m->slot[idx];

  hm_slog* prev = s;
  hm_slog* cur  = s->sledeci;

  while (cur) {
      if (cur->kljuc == kljuc) {
          prev->sledeci = cur->sledeci;
          free(cur);
          m->elem_broj--;
          return OK;
      }
      prev = cur;
      cur = cur->sledeci;
  }
  return NEMA_ELEMENATA;
}
```

- **Umetanje.** Novi slog se smešta u odgovarajući lanac (najčešće na početak liste, jer je to najbrže). Ako duplikati nisu dozvoljeni, pre ubacivanja se proverava da li ključ već postoji u lancu.

```
status hm_ubaci(HesMapa* m, int kljuc, int vrednost)
{
    uint32_t idx = hashms(m->H, (uint32_t)kljuc);
    hm_slog* s = m->slot[idx];

    /* ako postoji, ažuriraj */
    for (hm_slog* p = s->sledeci; p != NULL; p = p->sledeci) {
        if (p->kljuc == kljuc) {
            p->vrednost = vrednost;
            return OK;
        }
    }

    /* inače ubaci na početak lanca */
    hm_slog* novi = novi_slog(kljuc, vrednost, s->sledeci);
    if (!novi) return NEDOVOLJNO_MEMORIJE;

    s->sledeci = novi;
    m->elem_broj++;
    return OK;
}
```

U praksi se često dešava da je lanac prazan ili sadrži samo jedan element. Zbog toga se u nekim implementacijama, radi smanjenja nivoa indirekcije i ubrzanja pristupa, u samom slotu čuva **slog** (ne pokazivač), uz dodatnu informaciju koja omogućava razlikovanje prazne i zauzetog slota. Tek kada dođe do kolizije, prelazi se na strukturu koja može da čuva više elemenata (npr. lista).

Ova optimizacija je korisna, ali nije neophodna za razumevanje osnovne ideje ulančavanja.

Kako lanci postaju duži, linearna pretraga u njima zahteva sve više vremena. U najgorem slučaju (što najčešće ukazuje na veoma loš izbor heš-funkcije), svi elementi mogu završiti u istoj slotu, pa operacije nad heš-tabelom postaju linearne složenosti $O(m)$, gde je m broj elemenata u tabeli.

U realnim implementacijama, heš-tabela se povremeno **proširuje**: povećava se broj slotova, a svi elementi se ponovo raspoređuju (što zahteva ponovno izračunavanje indeksa, tj. rehash). Proširivanje se obično sprovodi geometrijskom strategijom (npr. dupliranjem dimenzije), čime se obezbeđuje da amortizovana složenost operacija ostane povoljna.

Proširivanje se obično pokreće na osnovu **faktora opterećenja** (engl. *load factor*), definisanog kao

$$\alpha = \frac{m}{n},$$

gde je m broj elemenata upisanih u tabelu, a n broj slotova. Kada α pređe određenu granicu (na primer oko 0.75), povećanje tabele smanjuje očekivane dužine lanaca. Naime, prosečna dužina lanca kod otvorenog adresiranja je približno jednaka α. Alternativno (ili dodatno), kao kriterijum se može koristiti i pojava predugačkih lanaca: kada se u praksi uoči da se formiraju lanci značajne dužine, tabela se proširuje.

Treba naglasiti da proširivanje tabele ne može kompenzovati lošu heš-funkciju: ako heš-funkcija ne raspoređuje ključeve ravnomerno, povećanje dimenzije samo rasipa memoriju, dok problem nagomilavanja ostaje.

U implemetaciji se može dodati funkcija za izračunavanje faktora opterećenja:

```
static double faktor_opterecenja(const HesMapa* m)
{
    return (double)m->elem_broj / (double)m->slot_broj;
}
```

Zatim, na početku funkcije za unos elemenata `hm_ubaci`, potrebno je dodati narednu proveru:

```
    /* proširivanje pre umetanja (da  ostane pod kontrolom) */
    if (faktor_opterecenja(m) > HM_MAX_LOAD) {
        status st = prosiri(m);
        if (st != OK) return st;
    }
```

pri čemu je ranije izabrana vrednost za `HM_MAX_LOAD`, na primer:

```
#define HM_MAX_LOAD 0.75
```

Funkcija za proširivanje heš-tabele može se implementirati na sledeći način:

```
/* Rehash u tabelu sa 2^(m->k+1) slotova. */
static status prosiri(HesMapa* m)
{
    unsigned novi_k = m->k + 1;
    uint32_t novi_slot_broj = (uint32_t)(1u << novi_k);

    hm_slog** novi_slot = (hm_slog**)malloc(novi_slot_broj * sizeof(hm_slog*));
    if (!novi_slot) return NEDOVOLJNO_MEMORIJE;

    /* novi sentinel za svaki slot */
    for (uint32_t i = 0; i < novi_slot_broj; i++) {
        novi_slot[i] = novi_slog(0, 0, NULL);
        if (!novi_slot[i]) {
            for (uint32_t j = 0; j < i; j++) free(novi_slot[j]);
            free(novi_slot);
            return NEDOVOLJNO_MEMORIJE;
        }
    }

    /* nova heš-funkcija (novo nasumično a,b), jer se menja k */
    HashMS novi_H = hashms_init(novi_k);

    /* premesti sve slogove iz stare tabele u novu */
    for (uint32_t i = 0; i < m->slot_broj; i++) {
        hm_slog* s = m->slot[i];       /* stari sentinel */
        hm_slog* p = s->sledeci;
```

```
        while (p) {
            hm_slog* sledeci = p->sledeci;

            uint32_t idx = hashms(novi_H, (uint32_t)p->kljuc);
            p->sledeci = novi_slot[idx]->sledeci;
            novi_slot[idx]->sledeci = p;

            p = sledeci;
        }

        free(s); /* oslobodi stari sentinel */
        m->slot[i] = NULL;
    }

    free(m->slot);

    m->slot = novi_slot;
    m->slot_broj = novi_slot_broj;
    m->k = novi_k;
    m->H = novi_H;

    return OK;
}
```

Operacija proširivanja zahteva linearno vreme $O(n+m)$, ali ona se izvodi relativno retko. Ako se dimenzija tabele uvećava geometrijski (npr. udvostručavanjem), tada je ukupan trošak svih proširivanja tokom niza od N umetanja linearan, pa je amortizovana složenost umetanja i dalje $O(1)$ (uz pretpostavku ravnomerne raspodele heš-funkcije).

Iako su povezane liste najčešći izbor za lance, oni se mogu implementirati i drugačije: kao dinamički nizovi, ili čak kao uređena balansirana stabla. Međutim, složenije strukture ne donose nužno bolje performanse, naročito kada su lanci kratki, jer trošak održavanja (npr. balansiranja) može nadmašiti dobit.

Odvojeno ulančavanje: prednosti i mane

Prednosti:

- **Jednostavno razrešavanje kolizija.** Kolizije se prirodno rešavaju ulančavanjem elemenata u listu, bez komplikovanih pravila probanja.
- **Manja osetljivost na faktor opterećenja.** Za razliku od otvorenog adresiranja, ovde nije neophodno da $\alpha < 1$. Tabela može sadržati i više elemenata nego što ima slotova, jer se višak smešta u lance. Kod ravnomerne raspodele, očekivana dužina lanca je približno α.
- **Brisanje je jednostavno.** Brisanje elementa se svodi na uklanjanje iz odgovarajućeg lanca i ne zahteva posebne oznake, niti narušava korektnost pretrage ostalih elemenata.
- **Stabilno ponašanje pri kolizijama.** Čak i kada dođe do većeg broja kolizija, problem ostaje lokalizovan na nekoliko lanaca; ne nastaje globalno "zagušenje" celog niza, što se može desiti kod otvorenog adresiranja.

Mane:

- **Dodatni memorijski trošak.** Svaki slog u lancu mora da sadrži pokazivač, a elementi su tipično alocirani pojedinačno. To povećava potrošnju memorije i uvodi fragmentaciju.
- **Lošija lokalnost u memoriji.** Elementi lanca nisu nužno uzastopni u memoriji, pa pretraga kroz lanac može imati slabije performanse na savremenim procesorima (keš promašaji), naročito kada lanci postanu duži.
- **Najgori slučaj je linearan.** Ako je izbor heš-funkcije loš ili su ključevi nepovoljni, može se desiti da veliki broj elemenata završi u istom slotu; tada složenost operacija postaje $O(m)$, gde je m broj elemenata u tabeli.
- **Proširivanje je skupo.** Proširivanje zahteva ponovni raspored svih elemenata, što je linearna operacija. Ipak, ako se dimenzija povećava geometrijski (npr. udvostručavanjem), ovaj trošak ne narušava amortizovanu složenost osnovnih operacija.

Otvoreno adresiranje

Drugi široko korišćen način rešavanja kolizija je **otvoreno adresiranje** (engl. *open addressing*), koje se u literaturi ponekad naziva i **zatvoreno heširanje** (jer se svi slogovi čuvaju „unutar" same tabele). Za razliku od odvojenog ulančavanja, ovde ne postoje spoljne liste: svi slogovi se smeštaju isključivo u ćelije originalnog niza, tj. heš-tabele.

Osnovna ideja je sledeća: slog čiji je „prirodni" indeks `h(x)` već zauzet ne odlazi u listu, već se smešta u **neku drugu ćeliju tabele**, prema unapred definisanom pravilu koje određuje redosled obilaska ćelija. Taj postupak naziva se **popunjavanje** (engl. *probing*). Formalno, za ključ `x` definiše se niz kandidatskih pozicija

$$p_i(x) = f(h(x), i), \qquad i = 0, 1, 2, \ldots$$

i zatim se redom ispituju ćelije $p_0(x), p_1(x), \ldots$ sve dok se ne pronađe slobodna ćelija (pri umetanju), odnosno dok se ne pronađe traženi ključ ili dok se ne naiđe na praznu ćeliju (pri pretrazi).

U praksi se najčešće koriste sledeće šeme popunjavanja:

- **linearno popunjavanje** (eng. *linear probing*)
- **kvadratno popunjavanje** (eng. *quadratic probing*)
- **dvostruko heširanje** (eng. *double hashing*)

Linearno popunjavanje

Kod **linearnog popunjavanja**, ako je „prirodna" pozicija `h(x)` zauzeta, slog sa ključem `x` pokušava da se smesti u narednu ćeliju tabele, zatim u sledeću itd, uz kružno vraćanje na početak tabele. Drugim rečima, ako je neka pozicija `p` zauzeta, sledeća kandidatska pozicija je

$$(p + 1) \bmod m.$$

Opštija varijanta koristi konstantan korak k:

$$p_i(x) = (h(x) + i \cdot k) \bmod m, \qquad i = 0, 1, 2, \ldots$$

Element se umeće na **prvu slobodnu** poziciju u ovom nizu. Kada je `k=1`, ispituju se uzastopne ćelije tabele, što daje vrlo dobre performanse u odnosu na keš-memoriju savremenog hardvera, jer se pristupa susednim lokacijama u nizu. Ako se koristi `k \neq 1`, poželjno je da važi

$$\gcd(k, m) = 1,$$

kako bi niz kandidatskih pozicija mogao da obiđe sve ćelije tabele; u suprotnom, popunjavanje može da se „zaglavi" u podskupu pozicija iako u tabeli postoje slobodna mesta.

Iako se i ovde formiraju **„lanci" elemenata** koji imaju istu početnu heš-vrednost, oni nisu smešteni u odvojenim listama (kao kod ulančavanja), već nastaju unutar same tabele. Dodatno, linearno popunjavanje izaziva i **sekundarne kolizije**: element koji je zbog kolizije pomeren na kasniju poziciju može zauzeti „prirodno" mesto nekog drugog ključa, pa i taj ključ mora da bude pomeren dalje u nizu.

U nastavku koristimo tabelu kapaciteta $m = 2^k$, što omogućava efikasno računanje modula pomoću maske $m - 1$. Svaka ćelija ima stanje (*prazno*, *zauzeto* ili *obrisano*), a heš-vrednost se računa multiply–shift metodom (`HashMS`):

```
typedef struct {
    int key;
    int value;
    uint8_t state; /* 0=PRAZNO, 1=ZAUZETO, 2=OBRISANO */
} oa_slot;

typedef struct {
    oa_slot* a;
    uint32_t cap;   /* uvek 2^k */
    uint32_t size;  /* broj zauzetih */
    uint32_t tomb;  /* broj obrisanih */
    HashMS   h;     /* parametri multiply-shift heša */
} HesMapaOA;
```

Linearno popunjavanje se u kodu svodi na generisanje sledeće pozicije

$$p_i = (h + i) \bmod m,$$

gde je `h = hashms(m->h, kljuc)`, a modulo se računa maskiranjem:

```
uint32_t mask = m->cap - 1u;
uint32_t h    = hashms(m->h, (uint32_t)key);

for (uint32_t i = 0; i < m->cap; i++) {
    uint32_t idx = (h + i) & mask;   /* (h+i) mod 2^k */
    /* ... provera stanja i eventualno umetanje / pronalazenje ... */
}
```

Pretraga koristi isti niz pozicija. Ona se završava uspešno kada se pronađe ćelija sa traženim ključem, a neuspešno kada se naiđe na **praznu** ćeliju (jer to znači da se traženi ključ nije mogao „preskočiti" tokom umetanja):

```
oa_status hm_oa_get(const HesMapaOA* m, int key, int* out_value)
{
    if (!m || !m->a) return OA_NEMA;

    uint32_t mask = m->cap - 1u;
    uint32_t h = hashms(m->h, (uint32_t)key);

    for (uint32_t i = 0; i < m->cap; i++) {
        uint32_t idx = (h + i) & mask;
        const oa_slot* s = &m->a[idx];

        if (s->state == OA_EMPTY) return OA_NEMA;
        if (s->state == OA_FULL && s->key == key) {
            if (out_value) *out_value = s->value;
            return OA_OK;
        }
    }
    return OA_NEMA;
}
```

Brisanje u heš-tabelama sa otvorenim adresiranjem je osetljivije nego kod odvojenog ulančavanja. Razlog je što se pretraga oslanja na činjenicu da se niz kandidatskih pozicija ispituje redom sve dok se ne naiđe na **praznu** ćeliju. Ta prazna ćelija predstavlja „dokaz" da traženi ključ ne postoji u tabeli: da je postojao, morao bi biti smešten pre prve prazne ćelije u svom nizu popunjavanja.

Ako bismo pri brisanju jednostavno obrisali slog i njegovu ćeliju označili kao praznu, mogli bismo da „presečemo" niz popunjavanja i time pokvarimo ispravnost pretrage.

Primer (linearno popunjavanje, korak 1). Neka tri ključa sa istom početnom pozicijom `h(x)` budu umetnuta redom, pa završe na pozicijama `h(x)`, `h(x)+1`, `h(x)+2`. Ako se zatim obriše slog sa pozicije `h(x)+1` i to mesto proglasi praznim, pretraga za ključem koji je na `h(x)+2` će stati već na `h(x)+1` (jer je ćelija prazna) i pogrešno zaključiti da ključ ne postoji.

Zbog toga se u praksi koristi jedan od sledeća dva pristupa.

1. **Tombstonovi (obrisane ćelije).** Umesto da se ćelija proglasi praznom, ona se označi kao *obrisana*. Pretraga takvu ćeliju ne sme tretirati kao kraj (mora da nastavi dalje), dok umetanje može da je iskoristi kao kandidat za smeštanje novog sloga.

   Nedostatak je što veći broj obrisanih ćelija produžava lance popunjavanja; zato se povremeno radi **reheš** (ponovno umetanje postojećih elemenata u novu ili „očišćenu" tabelu), čime se tombstonovi uklanjaju.

2. **Pomeranje unazad (backward-shift deletion).** Nakon brisanja, neki od naknadno smeštenih slogova se pomeraju unazad tako da se očuva uslov ispravne pretrage.

   Ovaj pristup je koncepcijski zahtevniji i može u najgorem slučaju uzrokovati veći broj pomeranja.

U implementaciji koja sledi koristi se prva varijanta: ćelije imaju eksplicitno stanje (*prazno*, *zauzeto* ili *obrisano*), a brisanje samo postavlja stanje na *obrisano*. Pretraga se i dalje završava neuspešno tek kada naiđe na zaista praznu ćeliju.

```
oa_status hm_oa_erase(HesMapaOA* m, int key)
{
    if (!m || !m->a) return OA_NEMA;

    uint32_t mask = m->cap - 1u;
    uint32_t h = hashms(m->h, (uint32_t)key);

    for (uint32_t i = 0; i < m->cap; i++) {
        uint32_t idx = probe_linear(h, i, mask);
        oa_slot* s = &m->a[idx];

        /* Prazna ćelija prekida pretragu: ključ ne postoji. */
        if (s->state == OA_EMPTY) return OA_NEMA;

        if (s->state == OA_FULL && s->key == key) {
            /* Umesto prazno: označavamo kao obrisano (tombstone). */
            s->state = OA_TOMB;
            m->size--;
            m->tomb++;

            /* Po potrebi „čistimo" tombstonove re-heshiranjem. */
            if (need_clean_tombs(m)) (void)rehash(m, m->cap);
            return OA_OK;
        }
    }
    return OA_NEMA;
}
```

Faktor opterećenja i reorganizacija

Kod otvorenog adresiranja je **presudno** da faktor opterećenja

$$\alpha = \frac{n}{m}$$

(odnos broja zauzetih slogova n i kapaciteta tabele m) ostane **značajno ispod 1**. Intuicija je sledeća: pri otvorenom adresiranju se kolizije rešavaju obilaskom niza kandidatskih pozicija, pa sa porastom α raste verovatnoća da će svaka naredna pozicija biti zauzeta. Kada $\alpha \to 1$, očekivani broj koraka pri pretrazi i umetanju naglo raste, a umetanje može postati praktično neizvodljivo bez reorganizacije.

Zbog toga se u praktičnim implementacijama uvodi prag, npr. $\alpha \leq 0.7$ ili $\alpha \leq 0.8$. Kada bi novo umetanje prešlo prag, tabela se *proširuje* (najčešće geometrijski, npr. udvostručavanjem kapaciteta), a zatim se svi postojeći slogovi ponovo umeću u novu tabelu. Ovaj postupak se naziva **ponovno heširanje** (eng. *rehash*). Ista ideja se koristi i kada broj obrisanih ćelija (*tombstone*-ova) postane velik: iako one formalno nisu zauzete, one produžavaju lance popunjavanja i pogoršavaju performanse, pa se reorganizacijom „čiste".

U nastavku je prikazana tipična funkcija `reheš`. Ona:

1. alocira novi niz kapaciteta `new_cap` i inicijalno ga postavlja na prazno,

2. resetuje brojače `size` i `tomb`,
3. (opciono) bira nove parametre heš-funkcije `HashMS` za novi broj bitova `k`,
4. prolazi kroz stari niz i ponovo umeće samo zauzete slogove u novu tabelu,
5. oslobađa stari niz.

```
static oa_status reheš(HesMapaOA* m, uint32_t new_cap)
{
    oa_slot* old     = m->a;
    uint32_t old_cap = m->cap;

    oa_slot* a = (oa_slot*)malloc((size_t)new_cap * sizeof(oa_slot));
    if (!a) return OA_NEDOVOLJNO_MEMORIJE;

    for (uint32_t i = 0; i < new_cap; i++) a[i].state = OA_EMPTY;

    m->a    = a;
    m->cap  = new_cap;
    m->size = 0;
    m->tomb = 0;

    /* za new_cap = 2^k biramo odgovarajući k i (po želji) nove parametre heša */
    m->h = hashms_init(cap_to_k(new_cap));

    uint32_t mask = new_cap - 1u;

    for (uint32_t i = 0; i < old_cap; i++) {
        if (old[i].state != OA_FULL) continue;

        int key = old[i].key;
        int val = old[i].value;

        uint32_t h = hashms(m->h, (uint32_t)key);

        /* ponovni upis u novu tabelu (linearno popunjavanje) */
        for (uint32_t j = 0; j < new_cap; j++) {
            uint32_t idx = probe_linear(h, j, mask);
            if (m->a[idx].state == OA_EMPTY) {
                m->a[idx].key   = key;
                m->a[idx].value = val;
                m->a[idx].state = OA_FULL;
                m->size++;
                break;
            }
        }
    }

    free(old);
    return OA_OK;
```

```
}
```

Reorganizacija je skupa operacija (zahteva obilazak svih slogova), ali se izvodi retko. Ako se kapacitet povećava geometrijski (npr. udvostručavanjem), ukupni trošak ovih povremenih re-heshiranja „razliva" se kroz veliki broj narednih umetanja, pa se u praksi dobija dobra amortizovana efikasnost.

Kvadratno popunjavanje i dvostruko heširanje

Linearno popunjavanje je jednostavno i veoma efikasno u pogledu keš-memorije, ali može dovesti do izraženog nagomilavanja slogova. Zbog toga se u praksi često koriste i druge šeme popunjavanja koje imaju za cilj da smanje verovatnoću nastanka dugih uzastopnih blokova zauzetih ćelija.

- **Kvadratno popunjavanje** (eng. *quadratic probing*). Kandidatske pozicije dobijaju se formulom

$$p_i(x) = (h(x) + c_1 i + c_2 i^2) \bmod m.$$

Ova šema umanjuje *primarno grupisanje*, ali i dalje može doći do *sekundarnog grupisanja*, jer ključevi sa istom početnom vrednošću `h(x)` obilaze iste pozicije. Takođe, za neke izbore parametara `m, c_1, c_2` niz kandidatskih pozicija ne mora obići sve ćelije tabele, pa se parametri u praksi moraju pažljivo birati.

- **Dvostruko heširanje** (eng. *double hashing*). Koriste se dve nezavisne heš-funkcije `h1` i `h2`, pa je niz kandidatskih pozicija dat izrazom

$$p_i(x) = (h_1(x) + i \cdot h_2(x)) \bmod m.$$

Druga heš-funkcija određuje „korak" obilaska, pa različiti ključevi tipično imaju različite nizove popunjavanja. Zbog toga se u velikoj meri izbegava i primarno i sekundarno grupisanje (osim u retkom slučaju da dva ključa imaju i istu vrednost h_1 i istu vrednost h_2). Neophodno je da $h_2(x)$ nikada ne bude nula i da bude uzajamno prosta sa m. Na primer, ako je $m = 2^k$, dovoljno je da $h_2(x)$ bude neparan broj.

Zaključak

Otvoreno adresiranje predstavlja elegantan pristup u kome se svi slogovi čuvaju unutar same tabele, bez pomoćnih listi i dodatnih alokacija po elementu. Posebno je privlačno **linearno popunjavanje**, jer obilazi uzastopne ćelije niza i zato se u praksi često ponaša veoma dobro u odnosu na keš-memoriju. Međutim, taj dobitak dolazi uz važna ograničenja: brisanje zahteva posebnu pažnju, a performanse su snažno vezane za **faktor opterećenja** i redovno **rehešovanje**.

Ključna slabost linearnog popunjavanja je pojava **nagomilavanja (primarnog grupisanja)**. Kada se formira duži neprekidan blok zauzetih ćelija, svaki naredni ključ koji „upadne" u taj blok ima povećanu verovatnoću da ga dodatno produži. Time nastaje pozitivna povratna sprega: blokovi rastu, prosečan broj koraka pretrage i umetanja raste sve brže, a degradacija postaje naročito izražena pri većem faktoru opterećenja. Zbog toga se u ozbiljnim implementacijama otvorenog adresiranja pažljivo bira prag opterećenja i, po potrebi, koriste alternative poput kvadratnog popunjavanja ili dvostrukog heširanja, koje ublažavaju efekat grupisanja.

4.3.3 Veza sa C++ standardnom bibliotekom: `unordered_set` i `unordered_map`

Koncepti uvedeni u ovom poglavlju neposredno se pojavljuju i u standardnoj biblioteci C++ kroz tzv. **neuređene asocijativne kontejnere** (*unordered associative containers*). To su gotove implementacije struktura tipa **skup** i **mapa** koje su zasnovane na **heš-tabelama**, pa se njihovo ponašanje i performanse mogu razumeti upravo kroz pojmove: heš-funkcija, kolizije, faktor opterećenja i rehash (ponovno raspoređivanje).

- `std::unordered_set<Key>` predstavlja **neuređeni skup**: čuvaju se samo ključevi, bez pridružene vrednosti.
- `std::unordered_map<Key, T>` predstavlja **neuređenu mapu**: čuvaju se parovi oblika *(ključ i vrednost)*.

Naziv „neuređeni“ ovde ima precizno značenje: elementi **nisu održavani u sortiranom poretku** po ključu, niti kontejner obezbeđuje poredak obilaska koji je nezavisan od unutrašnje organizacije. Redosled obilaska zavisi od raspodele po internim slotovima i može se promeniti nakon proširivanja heš tabele. Ovo je direktna posledica činjenice da je primarni cilj ovih struktura **efikasna pretraga i ažuriranje**, a ne očuvanje uređenosti.

U tipičnoj implementaciji (i uobičajenoj interpretaciji standarda), kontejner je organizovan kao niz slotova, a elementi koji se heširanjem mapiraju na isti slot čine *kolizione grupe*. Kolizije se u praksi najčešće rešavaju **odvojenim ulančavanjem** (svaki slot sadrži lanac elemenata), mada standard ne propisuje tačan mehanizam; propisuje samo semantiku operacija i garantovane klase složenosti.

Podržane operacije i složenosti

Sa stanovišta apstraktnog tipa podataka, `unordered_set` i `unordered_map` podržavaju standardne operacije nad skupovima i mapama: proveru prisutnosti, umetanje, brisanje i pretragu. Njihova glavna prednost je što su ove operacije **u proseku konstantne složenosti**, pod uobičajenim pretpostavkama o kvalitetu heš-funkcije i razumnom faktoru opterećenja.

- **Pretraga (članstvo / pronalaženje)**: `find`, `count` (i u novijim standardima `contains`). Prosečna složenost: **O(1)**, najgora složenost: **O(n)**. Najgori slučaj nastaje kada veliki broj ključeva završi u istom slotu (loša heš-funkcija ili ekstremno nepovoljni ulazi), pa pretraga postaje linearna po veličini kontejnera.
- **Umetanje**: `insert`, `emplace` Prosečna složenost: **O(1)**, uz napomenu da je to često **amortizovano O(1)**. Razlog je što se povremeno mora izvršiti *proširivanje* — operacija u kojoj se broj slotova uveća i svi postojeći elementi ponovo rasporede. Takva operacija košta $O(n)$, ali se dešava relativno retko, pa je prosečni trošak po umetanjima konstantan.
- **Brisanje**: `erase` Prosečna složenost: **O(1)**, najgora složenost: **O(n)** iz istog razloga kao i kod pretrage (kolizione grupe mogu postati velike).

Za `unordered_map` postoje dve posebno važne operacije pristupa vrednosti:

- `at(key)` konceptualno odgovara „dohvatanju vrednosti za postojeći ključ“; složenost je kao kod pretrage.
- `operator[](key)` predstavlja „dohvatanje ili kreiranje“: ako ključ ne postoji, on se ubacuje sa podrazumevanom vrednošću. Ova operacija je zato semantički jača (može izazvati umetanje i proširivanje tabele), ali je i dalje prosečno **O(1)** (amortizovano).

Faktor opterećenja i proširivanje u C++

C++ neuređeni kontejneri eksplicitno izlažu pojam **faktora opterećenja** (*load factor*), koji je analogan odnosu između broja elemenata i broja slotova u heš-tabeli. U interfejsu se pojavljuju:

- `load_factor()` — trenutni faktor opterećenja,
- `max_load_factor()` — prag nakon kog kontejner obično pokreće proširivanje,
- `rehash(b)` — eksplicitno podešavanje minimalnog broja bucket-a,
- `reserve(n)` — zahtev da kontejner unapred obezbedi kapacitet za približno `n` elemenata bez čestih rehash-ova.

Ove funkcije su praktična veza sa teorijom: one omogućavaju da se kontrolom faktora opterećenja smanji verovatnoća kolizija i broj skupih rehash operacija, što je naročito važno kada se unapred zna približan broj elemenata.

Glava 5

Napredne strukture podataka

5.1 Skip-lista

5.2 B-stablo