VIŠJA STROKOVNA ŠOLA

PROGRAM

INFORMATIKA

PREDMET

PROGRAMIRANJE (1.letnik)

GRADIVO ZA INTERNO UPORABO

Srečko Zorman

IME PREDMETA: **PROGRAMIRANJE I (PRO I)**

ŠTEVILO UR PREDMETA PO LETNIKIH IN OBLIKAH IZOBRAŽEVALNEGA DELA

Št.	Ime modula/predmeta/druge sestavine	Obvezno / izbirno	Let.	Št. kontaktnih ur				Kreditne točke
				PR	SV	LV	Skupaj	
M3	**Kodiranje programov**	**obvezno**	**1.**	**72**	**-**	**96**	**168**	**18 KT**
P7	**Programiranje I**	**obvezno**	**1.**	**36**	**-**	**48**	**84**	**7 KT**
P8	Zbirke podatkov I	obvezno	1.	36	-	48	84	7 KT
D3	Praktično izobraževanje – Kodiranje programov	obvezno	1.					4 KT
M7	**Izdelava programskih aplikacij**	**izbirno**	**2.**	**60**	**-**	**84**	**144**	**14 KT**
P14	**Programiranje II**	**izbirno**	**2.**	**36**	**-**	**48**	**84**	**6 KT**
P15	Razvoj programskih aplikacij	izbirno	2.	24	-	36	60	4 KT
D7	Praktično izobraževanje – Izdelava programskih aplikacij	izbirno	2.					4 KT

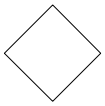
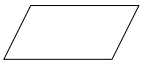
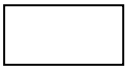
OKVIRNI SEZNAM LITERATURE IN DRUGIH VIROV:

- *Matjaž Prtenjak*: C++ za velike in male, DESK, 1995
- *V. Žumer in N. Korbar:* Programiranje v jeziku C++, Univerza v Mariboru, Maribor 1997
- *Ž. Turk:* Uvod v objektno usmerjeno programiranje in C++, Ljubljana: Državna založba Slovenije, 1991
- *Krista Rizman:* C, C++ in C za okolje oken s primeri, Zavod republike Slovenije za šolstvo, Ljubljana 1995
- *Dragan Urošević*: Algoritmi v programskem jeziku C, Mikro knjiga Beograd, 1996
- *Jernej Kozak*: Podatkovne strukture in algoritmi, Društvo matematikov, fizikov in astronomov SRS, Ljubljana
- WWW.CPLUSPLUS.COM
- ostali internetni viri

Diagrami poteka

Algoritem pomeni splošen postopek (metodo) za reševanje problemov. Računalnik zna izvrševati le postopke, ki so mu predpisani z njemu razumljivimi ukazi. V računalništvu algoritem pomeni postopek (metodo), ki jo lahko izvaja računalnik, da pride do rešitve problema. Algoritem lahko zapišemo na več različnih načinov: z naravnim jezikom (npr. kuharski recept), **diagrami poteka** (grafičen prikaz strukture programa), **struktogrami** (grafičen prikaz strukture programa), **psevdokodo**... **Program je v programskem jeziku zapisan algoritem**. Program mora biti zapisan natančno po pravilih zapisovanja programa (sintaksa programskega jezika), saj računalnik lahko "razume" le omejen nabor ukazov.

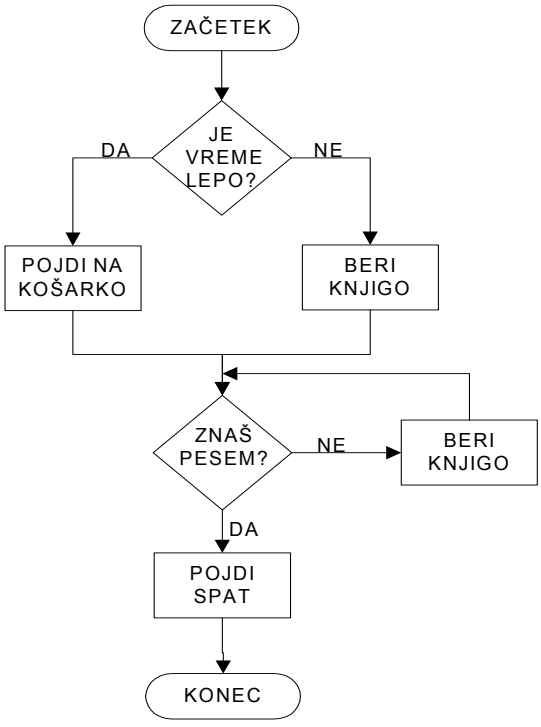
Diagrami poteka uporabljajo grafične simbole za prikaz strukture algoritma. Uporabljamo naslednje simbole:

Grafični simbol	Pomen
	Začetek oz. konec diagrama poteka
	Odločitev
	Branje (vnos) in izpis podatkov
	Izračun, prireditev, opravilo...
	Pot (izvajanja)
	Magnetni disk

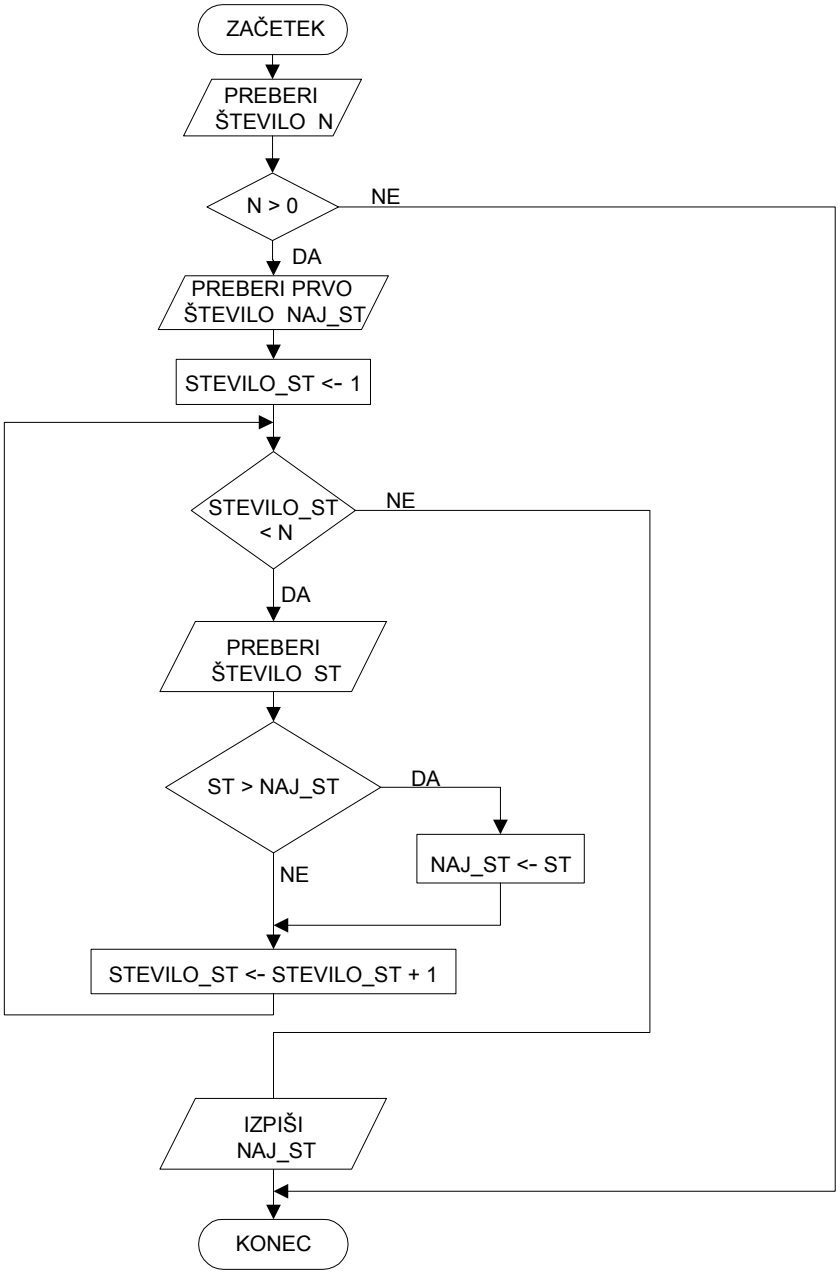
Primeri diagramov poteka:

DP1: Popoldanske aktivnosti

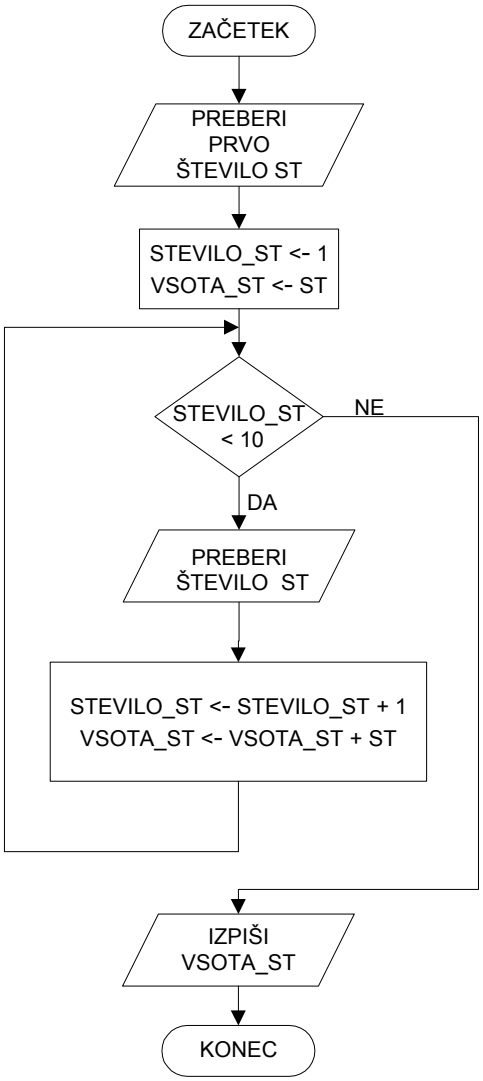
Če bo popoldan lepo vreme, pojdi na košarko v nasprotnem preberi knjigo. Kasneje preveri, če že znaš pesem, ki ste se jo začeli učiti v šoli. Če je ne znaš, beri tako dolgo, da jo boš znal povedati brez besedila. Zvečer pojdi spat.



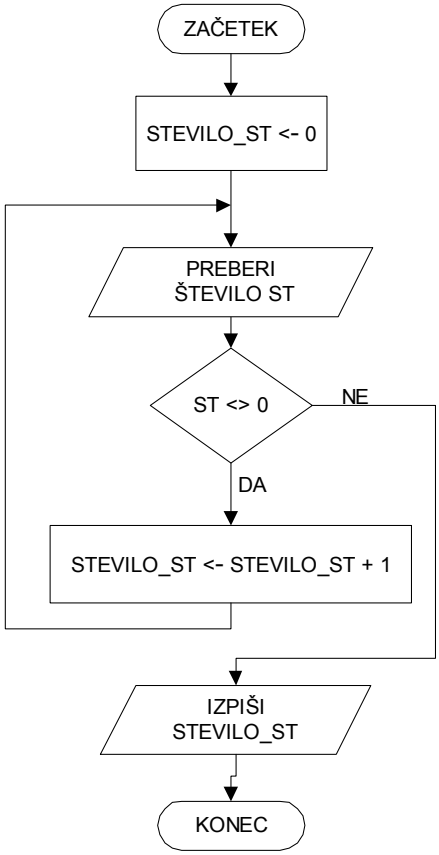
DP2: Največja vrednost N prebranih števil



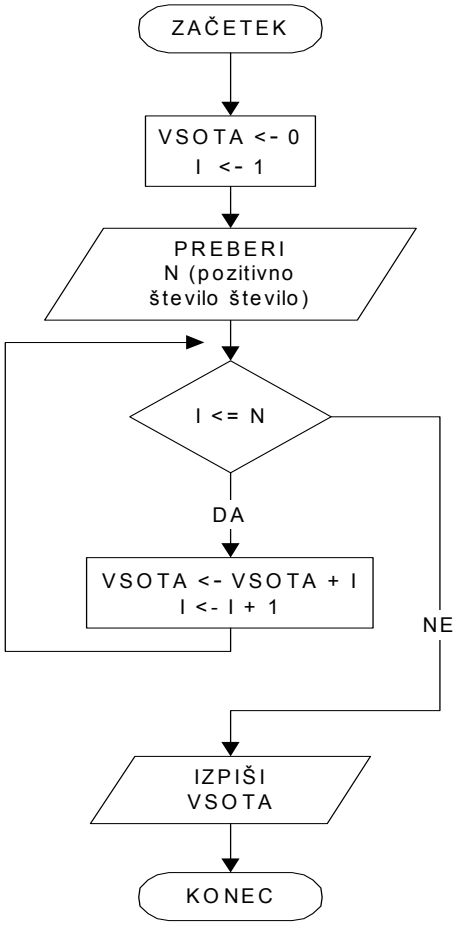
DP3: Vstavimo 10 števil in nam izračuna vsoto vstavljenih števil



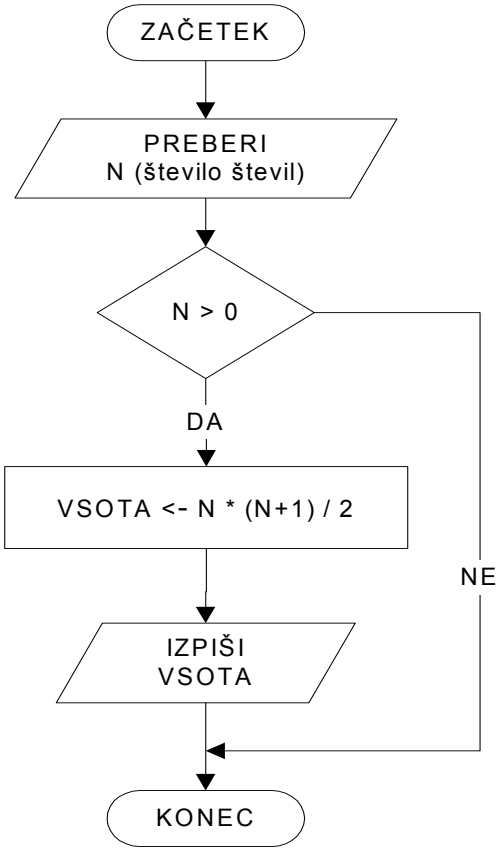
DP4: Vstavljamo števila dokler ne vstavimo 0 in nam izpiše število vstavljenih števil



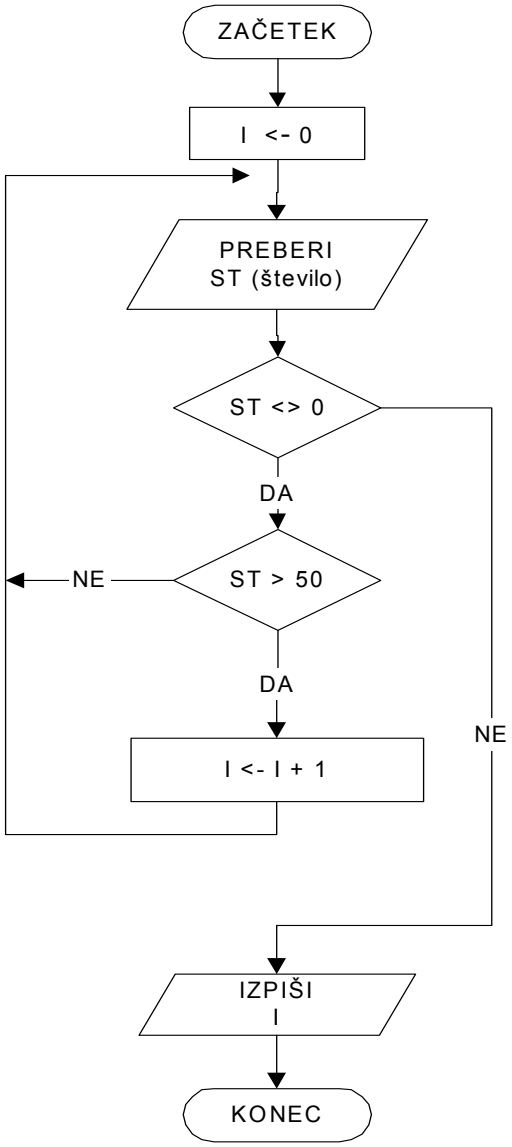
DP5: Izračun vsote števil od 1 do N; N vstavimo.



DP6: Izračun vsote prvih N števil z uporabo Gausovega algoritma (V = n * (n+1) / 2)

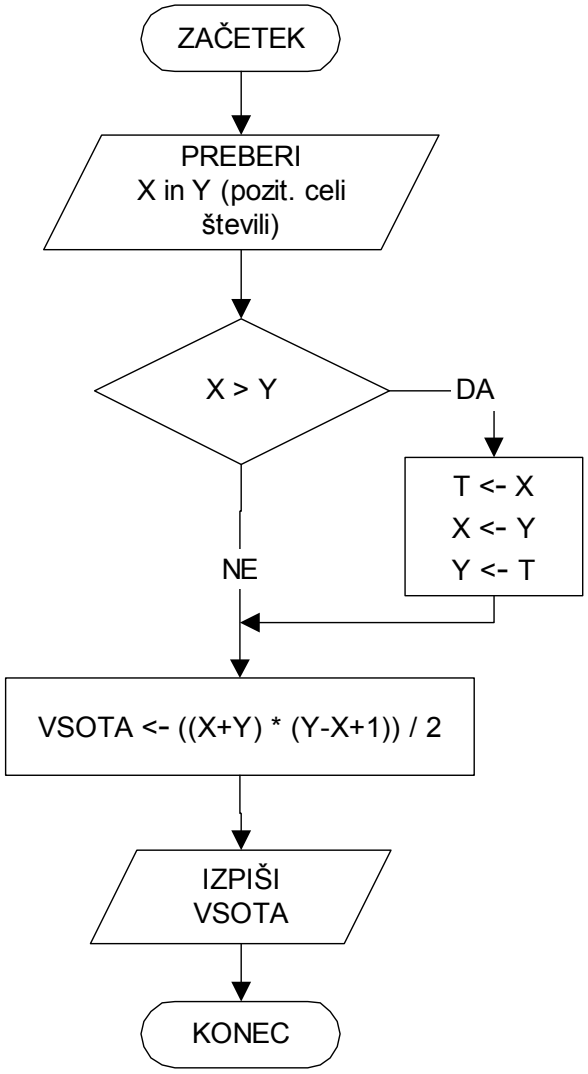


DP7: Prešteje število vnešenih števil, ki so večja od 50. Števila vnašamo dokler ne vnesemo števila 0.



DP8: Izračun vsote naravnih števil od X do Y.

Vrednosti X in Y preberemo (navadno in s pomočjo Gaußovega algoritma 2*Vsota = (x+y) * (y-x+1))

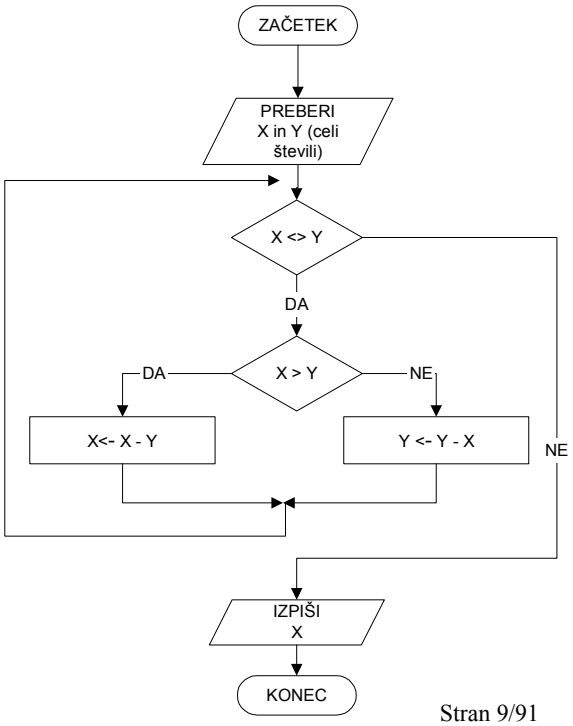


DP9: Izračun geometrijskega zaporedja 1, 2, 4, 8, 16...

Ugotovite kateri člen povzroči, da je vsota zaporedja večja od 450

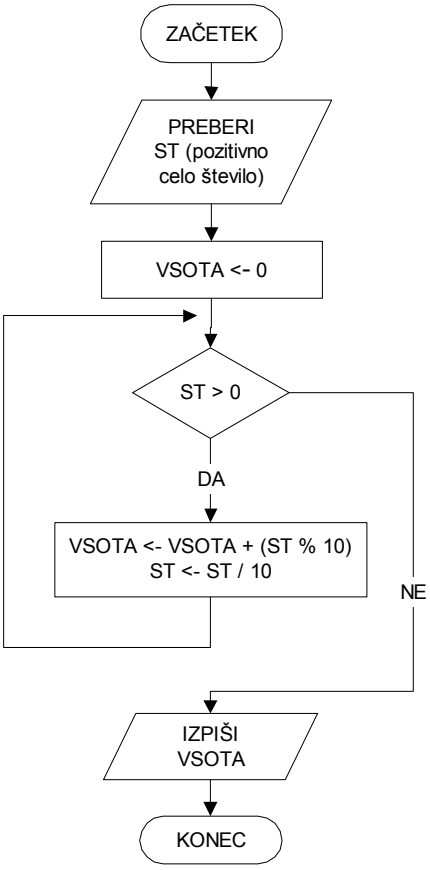
DP10: Izračun največjega skupnega delitelja z Evklidovim algoritmom.

NSD(x,x) = x
NSD(x,y) = NSD(y,x)
če je x > y => NSD(x,y) = NSD(x-y,y)



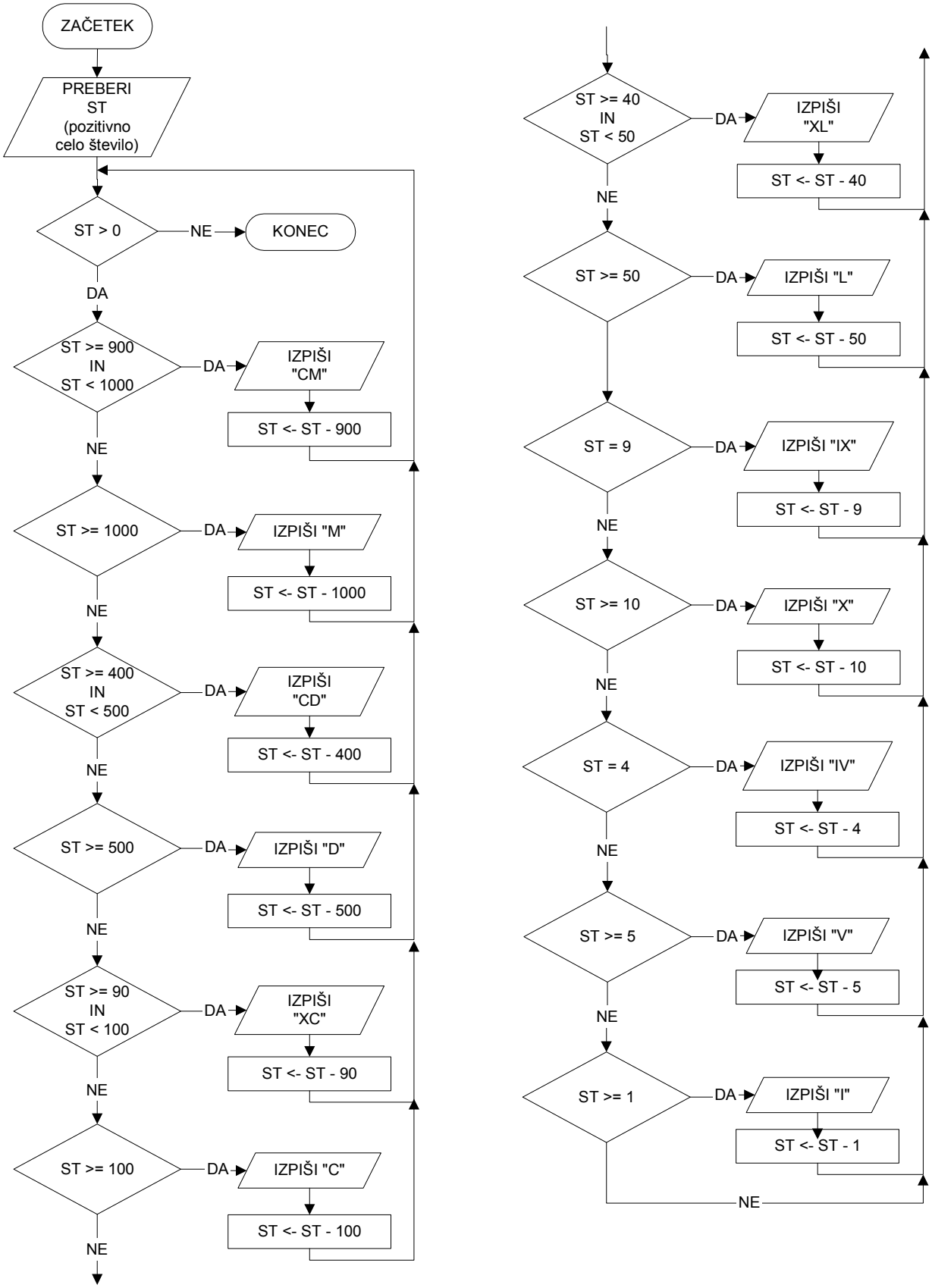
DP 11: Izpis vsote števk celega števila, ki ga vstavimo.

8 % 3 = 2 ostanek po deljenju
8 / 2 = 4 celoštevilčno deljenje



DP 12: Algoritem za pretvorbo arabskih števil v rimske

(1- I, 2 - II, 3 - III, 4 - IV, 5 - V, 6 - VI, 7 -VII, 8 - VIII, 9 - IX, 10 - X,
20 - XX, 30 - XXX, 40 - XL, 50 - L, 60 - LX, 90 - XC,
100 - C... 400 - CD, 500 - D, 900 - CM, 1000 - M)


ZAČETEK
PREBERI ST (pozitivno celo število)
ST > 0
NE
KONEC
DA
ST >= 900 IN ST < 1000
DA
IZPIŠI "CM"
ST <- ST - 900
NE
ST >= 1000
DA
IZPIŠI "M"
ST <- ST - 1000
NE
ST >= 400 IN ST < 500
DA
IZPIŠI "CD"
ST <- ST - 400
NE
ST >= 500
DA
IZPIŠI "D"
ST <- ST - 500
NE
ST >= 90 IN ST < 100
DA
IZPIŠI "XC"
ST <- ST - 90
NE
ST >= 100
DA
IZPIŠI "C"
ST <- ST - 100
NE
ST >= 40 IN ST < 50
DA
IZPIŠI "XL"
ST <- ST - 40
NE
ST >= 50
DA
IZPIŠI "L"
ST <- ST - 50
ST = 9
DA
IZPIŠI "IX"
ST <- ST - 9
NE
ST >= 10
DA
IZPIŠI "X"
ST <- ST - 10
NE
ST = 4
DA
IZPIŠI "IV"
ST <- ST - 4
NE
ST >= 5
DA
IZPIŠI "V"
ST <- ST - 5
NE
ST >= 1
DA
IZPIŠI "I"
ST <- ST - 1
NE

DP 13: Algoritem za izračun ploščine trikotnika, če so znane vse tri stranice trikotnika

(Heronov obrazec Pl=Sqrt(s*(s-a)*(s-b)*(s-c)); s=(a+b+c)/2;
pogoj za obstoj trikotnika: vsota katerikolih dveh stranic >= tretji)

DP 14: Gaußov algoritem (Y > X)

Vsota števil od števila X do števila Y s pomočjo Gaußa:

Vsota = x + (x+1) + (x+2) + (x+3) + ... + (y-2) + (y-1) + y
Vsota = y + (y-1) + (y-2) + (y-3) + ... + (x+2) + (x+1) + x seštejemo +

2*Vsota = (x+y) + (x+y) + (x+y) +... + (x+y) + (x+y) +(x+y)
I_____I
(y-x + 1) elementov

2*Vsota = (x+y) * (y-x+1)
Vsota = (x+y) * (y-x+1) / 2

Če je x = 1:
2*Vsota = (1+y) * (y)

Programski jezik C++

Uvod

Programski jezik C je nastal konec 60-ih let. Iz njega je kasneje v 70-ih razvil C++. Najbolj značilne odlike jezika so:

- prenosljivost (na voljo je na praktično vseh platformah),
- strukturiranost,
- učinkovitost (glede izvajanja),
- prijaznost: omogoča "lepo" pisanje programov, nas pa v to ne sili.

Program, ki ga napišemo, moramo pred uporabo prevesti. Program se lahko nahaja v eni ali več izvornih datotekah. V odvisnosti od operacijskega sistema se prevajanje razlikuje. Vedno moramo opraviti dve fazi: **prevajanje** (compile) in **povezovanje** (linking). Prevajanje zapiše našo izvorno kodo v strojno kodo (ukaze, ki jih razume procesor). Povezovanje združi različne dele programa v zaključeno celoto. Posamezni sistemi nam omogočajo, da to opravimo z enim samim ukazom. Prevajanje nam odkrije sintaktične napake, ki jih program vsebuje.

Prevajanje in povezovanje lahko izvedemo tudi v ukazni vrstici. Sintaksa ukaza za prevajanje z Borlandovim prevajalnikom je: **BCC | BCC32 | BCC32i [opcija [opcija...]] ime_datoteke [ime_datoteke...].** Če ni napak, potem program še povežemo (link) in računalnik nam izdela izvršljivo datoteko za izvajanje programa.

Komentar

Komentar je del kode, ki jo prevajalnik izpusti in je namenjen programerju za opombe in opise (dokumentiranje) programske kode.
V C++ lahko komentar zapišemo na dva načina:

- `// komentar v vrstici (vse kar je desno od // se smatra kot komentar)`
- `/* blok s komentarjem */`

Struktura programa

Standardne funkcije so vključene v standardno knjižnico. Deklaracije standardnih funkcij so zbrane v deklaracijskih datotekah (zaglavje - datoteke s končnico .h). Deklaracijske datoteke vključimo v program s predprocesorskim ukazom #include.

Primer izpisa pozdrava izgleda takole:

```
// prvi program - izpis pozdrava - to je komentar
#include <iostream.h>                 // vključitev deklaracij funkcij za delo z vhodno - izhodnim tokom
int main ()                           // glavni program - funkcija main
{                                     // telo funkcije main
  cout << "Pozdrav iz C-ja!";        // izpis niza na zaslon
  return 0;                           // funkcija vrne 0
}
```

Na začetku smo v program vključili deklaracije funkcij za delo z vhodni izhodnim tokom ("iostream.h"). Te funkcije uporabljamo za vnos podatkov v program oz. za izpis podatkov. Program ima v kodi funkcijo *main()*. Funkcije so sestavni del programa. Main predstavlja glavni program, kjer se prične izvajanje programa. Oklepaji povedo, da je main funkcija. Zaviti oklepaji predstavljajo telo funkcije. Komentarji so zapisani med znaki /* in */ (večvrstični) ali za znakom // (enovrstični). Večvrstični lahko segajo tudi čez več vrstic. Vsak stavek se konča s podpičjem. S standardno funkcijo cout izpisujemo podatke v izhodni tok (zaslon).

Zgled:

```
#include <iostream.h>
// deklaracija (prototipa) funkcije
int vsota (int a, int b);
// glavna funkcija
main ()
{
    int i, j;  //deklaracija celoštevilčnih spremenljivk
    cout << "Vstavi dve celi stevili: "; // izpis na zaslon
    cin >> i >> j;  //branje vrednosti spremenljivk
    int k = vsota(i, j); //deklaracija in inicializacija
    cout << i << " + " << j << " = " << k << endl; //izpis in pomik v novo vrstico
    return 0;
}
// definicija funkcije
int vsota (int a, int b)
{
    int c = a + b;
    return c;
}
```

Podatki

Program obdeluje podatke, ki jih shranjuje v pomnilniku in se med izvajanjem programa lahko spreminjajo ali so konstantni. Podatek je shranjen v pomnilniku na enoličnem naslovu – pomnilniški lokaciji in zasede toliko prostora, kot ga določimo s podatkovnim tipom (deklaracija spremenljivke).

Spremenljivke in konstante

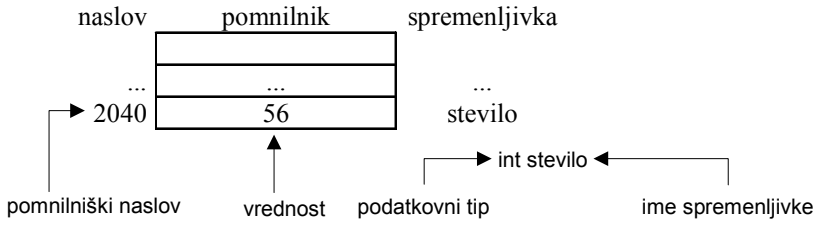
Ime spremenljivke ali konstante naj bo smiselno in naj pove čimveč o vlogi spremenljivke oz. konstante. Ime je lahko poljubna kombinacija črk (angleška abeceda) in števk ter znaka podčrtaj ('_'). Prvi znak mora biti obvezno črka ali znak '_'. **Prevajalnik razlikuje velike in majhne črke**. St in st sta dve različni spremenljivki. Za ime ne smemo uporabiti imena rezervirank: asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t... and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_...
Vse rezervirane besede pišemo z **malimi črkami**. Imena konstant pa po dogovoru pišemo z velikimi črkami. Spremenljivki lahko vrednost priredimo večkrat, medtem ko konstanti samo enkrat na začetku programa.
Vrednost konstante določimo običajno z define, npr.

```
#define IME "Tine"
#define PI 3.1428
```

Pozor: Define ni stavek, zato za njim ni podpičja. Preden prevajalnik pride do programa predprocesor zamenja vse vrednosti uporabljenih imen konstant z vrednostjo, ki določa vrednost konstante.

Lastnosti spremenljivk:

- ime spremenljivke,
- podatkovni tip,
- naslov pomnilniške lokacije, kjer je spremenljivka shranjena,
- vrednost spremenljivke.



Podatkovni tipi

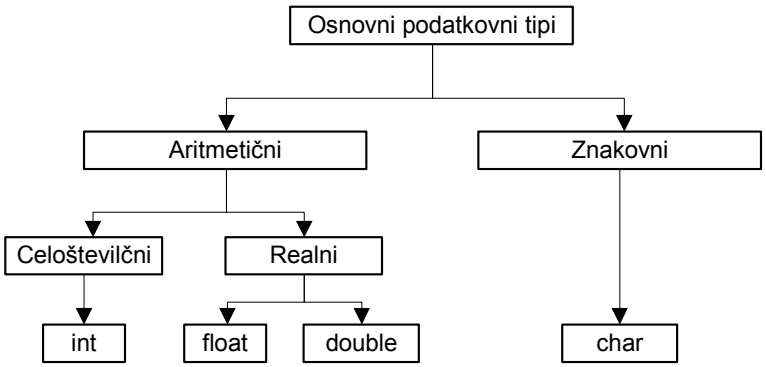
Spremenljivke (podatki) so shranjeni v računalniku v enem ali več zlogih (byte). Različnih možnih znakov je toliko, da lahko shranimo vsak znak v enem zlogu (8 bitov). V dveh ali štirih zlogih hranimo cela števila in v štirih ali osmih zlogih realna števila. Da prevajalniku povemo, koliko prostora naj rezervira za posamezno spremenljivko, moramo pred uporabo spremenljivke deklarirati. Sintaksa:

```
<podatkovni_tip> <ime_spremenljivke>;  // ali
<podatkovni_tip> <ime_spremenljivke> = <izraz>;  //z inicializacijo
```

Najprej zapišemo podatkovni tip spremenljivke in nato ime spremenljivke ali več spremenljivk istega tipa (če jih je več, jih med seboj ločimo z vejico). Deklariramo lahko tudi vsako spremenljivko posebej. Deklaracijo končamo s podpičjem (;).

Osnovni podatkovni tipi so:

- celoštevilčni podatkovni tip: **int** (integer), **long**, **short**, **unsigned**,
- realni podatkovni tip: **float**: realno število enojne natančnosti (3.4 * (10**-38) to 3.4 * (10**+38)), **double**: realno število dvojne natančnosti,
- znakovni podatkovni tip: **char** (character); en znak (črka, cifra ali drugi znak),
- logični podatkovni tip: **bool**.



Primer deklaracij:

```
{
        int Stevec;             //deklaracija celoštevilčne spremenljivke Stevec
        char Znak, Zn;          // deklaracija znakovnih spremenljivk Znak in Zn
        Stevec = 5;             // prireditev vrednosti spremenljivki
        Znak='D';
}
```

Kot dodatek k tem osnovnim tipom imamo kvalifikatorje, ki jih lahko dodamo deklaraciji celega števila. Ti so **short, long, unsigned**. Long in short se nanašata na velikost celega števila, unsigned določa nepredznačeno celo število. Če **unsigned** izpustimo, se privzeto uporabi **signed** (predznačeno). Spremenljivka lahko spreminja svoje vrednosti v predpisanem območju glede na to, kakšnega podatkovnega tipa je.

Podatkovni tip	Število zlogov	Območje oz. obseg
short	2 zloga (16 bitov)	-32768 .. 32767 (predznačena) 0 .. 65535 (nepredznačena)

int	2 zloga (16 bitov) ali 4 zlogi (32 bitov)	*
long	4 zlogi (32 bitov)	2147483648 .. 2147483647 (predznačena) 0 .. 4294967295 (nepredznačena)
char	1 zlog	-128 .. +127 (predznačena) 0 .. 255 (nepredznačena)
float	4 zlogi	3.4e + / - 38 (7 mest)
double	8 zlogov	1.7e + / - 308 (15 mest)
long double	10 zlogov	1.2e + / - 4932 (19 mest)
bool	1	true ali false

Primeri deklaracij:

```
int stev;
short int i;
unsigned int pozit_st;
float realno;
double veliko_realno;
char zn;
```

Celoštevilske konstante so zaporedja števk (0..9), pred katerimi lahko stoji predznak (+ oz. -). Med tisočicami ni pike, in ne smemo uporabiti decimalne vejice. Prav tako za njih ne smemo uporabiti znanstvenega zapisa. Konstante, ki se začno s števko **0**, se interpretirajo kot **osmiška števila**, in konstante, ki se začno z **0x** ali **0X** se interpretirajo kot **šestnajstiška števila**. Pogosto uporabimo konstante za inicializacijo (dodeljevanje začetnih vrednosti) spremenljivkam. Npr.:

```
int i = 45; char zn = 'i';
```

Zgled:

```
// deklaracija in uporaba spremenljivk
#include <iostream.h>

int main ()
{
        // deklaracija
        int a, b;
        int rezultat;
        // obdelava
        a = 5;   b = 2;
        a = a + 1;
        rezultat = a - b;
        // izpis
        cout << rezultat;
        // izpiše 4
        return 0;
}
```

Vaja:
Napišite program, ki bo izračunal porabo goriva avtomobila.

Vhodni podatki:

- število prevoženih kilometrov – st_prevoz_km
- število litrov, ki smo jih natočili – st_litrov

Izhodni podatki:

- poraba na 100 km - poraba

Izračun:

st_prevoz_km ... st_litrov
100 ... poraba
poraba = st_litrov * 100 / st_prevoz_km

Algoritem:
vnos vhodnih podatkov
izračun porabe
izpis porabe

Znaki

Ta podatkovni tip v bistvu določa nepredznačeno celo število v obsegu 0..255. To število je interpretirano kot znak v ASCII tabeli znakov. V tej tabeli ima črka 'a' vrednost 97, črka 'A' pa vrednost 65 in znak '0' vrednost 48.

Znakovna konstanta je črka med enojnima narekovajema. Če narekovaje izpustimo se interpretira kot spremenljivka. Med enojnima narekovajema ne sme biti več znakov (npr. ‘avto’). Znake, ki jih ni na tipkovnici, lahko določimo s pomočjo ubežnih sekvenc.
Primeri ubežnih sekvenc:

- \n - nova vrstica,
- \t - tabulator,
- \b - pomik za znak v levo,
- \r - pomik na začetek vrstice,
- \f - pomik na novo stran,
- \\ - poševna črta nazaj,
- \’ - enojni narekovaj,
- \” - dvojni narekovaj.

Realna števila

Uporabljamo jih predvsem pri matematičnih operacijah. Omogočajo zapis večjih števil in decimalnega dela. Zavzamejo več pomnilniškega prostora in izvajanje je počasnejše. Realne konstante so zaporedja števk z možnim predznakom, decimalnim delom in eksponentnim delom. Eksponentni del označimo s črko e (E) in celoštevilsko konstanto. Ta določa, da moramo število pred decimalno vejico premakniti za toliko mest v desno, kolikor je velikost konstante za znakom e. V realni konstanti ne sme biti presledkov. Pozor: uporabljamo decimalno piko in ne decimalne vejice. Izpustimo lahko pozitivne predznake in vse ničle pred decimalno piko.
Npr.: `56.345, .056, .8e-5, 1002., 23.5.`

Inicializacija spremenljivk

Poznamo dva načina inicializacije spremenljivk, ki sta med seboj enakovredna:

```
podatkovni_tip spremenljivka = zacetna_vrednost ;      // ali
podatkovni_tip spremenljivka ( zacetna_vrednost );
```

Zgled:

```
int a = 5, b = 2;
//ali
int a(5), b(2);
```

Vaja:
Napišite program, ki izpiše znesek denarja, ki ga mora vrniti prodajalka, če vstavimo znesek računa in vplačan znesek kupca.

Vhodni podatki:
- znesek računa – zn_racuna
- vplačan znesek – zn_placan

Izhodni podatki:
- vračilo – zn_vracilo

Izračun:
zn_vracilo = zn_racuna - zn_placan

Algoritem:

vnos vhodnih podatkov
izračun vračila
izpis vračila

Doseg spremenljivk

Življenska doba spremenljivke (»lifetime«) je čas med rezervacijo in sprostitvijo spremenljivke. Doseg spremenljivke (»scope«) določa, v katerem delu programa lahko spremenljivko uporabljamo. Povezljivost (»linkage«) določa katere spremenljivke lahko uporabljamo v različnih prevajalnih enotah.
Spremenljivke so lahko **lokalne** ali **globalne**.
Lokalne spremenljivke so deklarirane znotraj funkcije (v kateremkoli bloku). Dostopne so le znotraj funkcije oz. bloka v katerem so deklarirane. Obstajajo le v času izvajanja funkcije oz. bloka. Lokalna spremenljivka ima doseg in življensko dobo od mesta definicije do konca bloka, v katerem je definirana.
Globalne spremenljivke so deklarirane izven funkcij. Običajno jih deklariramo na začetku programa, preden definiramo prvo funkcijo. Spremenljivka, deklarirana na ta način, je dostopna znotraj vseh funkcij programa in njeno vrednost lahko spremenimo kjerkoli v programu. Zato njihova uporaba ni priporočljiva, saj program postaja nepregleden in se zelo poveča možnost različnih napak, ki jih težko odkrijemo. Globalne spremenljivke obstajajo ves čas dokler se program izvaja. Če na začetku deklaracije uporabimo extern pomeni, da lahko to globalno spremenljivko uporabljamo na nivoju celotnega programa.

Določila za življensko dobo, doseg in povezljivost

auto Ob začetku izvajanja bloka, v katerem je ta vrsta deklaracije, se prostor dodeli (alocira), ko pa izstopimo iz bloka, se prostor sprosti (to je privzet način delovanja).

```
auto int a,b,c;
```

register Ob začetku izvajanja bloka, v katerem je ta vrsta deklaracije, se prostor rezervira v CPE (spremenljivka se shrani v register, hitrejši dostop do spremenljivke, hitrejše izvajanje programa).

```
for (register int i=0; …
```

extern Spremenljivko naredimo vidno v vseh prevajalnih enotah. Če uporabimo to določilo, ni potrebno uporabiti direktive #include.

```
extern int stevec;
extern double izr(double x);
```

static Lokalne spremenljivke z določilom static ohranijo vrednost tudi ko se funkcija konča.
Globalne spremenljivke ali funkcije, ki so definirane s static, so vidne samo znotraj te datoteke (modula). Na ta način skrijemo takšne spremenljivke ali funkcije, pred uporabo od zunaj.

const Če dodamo const, to pomeni, da ima spremenljivka konstantno vrednost in je ne moremo spreminjati.

Različne spremenljivke imajo različen doseg in življensko dobo. Spremenljivke so lahko vidne v: celotni datoteki, razredu, funkciji ali bloku.

Skrivanje spremenljivk

Če pri deklaraciji spremenljivk na različnih mestih uporabimo enako ime spremenljivke, potem pride do skrivanja spremenljivke. Spremenljivka v najbolj notranjem bloku skrije enako imenovane spremenljivke v zunanjih blokih. Z uporabo operatorja **::** lahko v najbolj notranjem bloku dostopamo do skritih globalnih spremenljivk, ne moremo pa dostopati do skritih lokalnih spremenljivk.

Zgled:

```
// skrivanje spremenljivk
#include <iostream.h>
int a = 2; // globalna spremenljivka
int main ()
{
          int a, b;   // lokalni spremenljivki
          // obdelava
          a = 5;    b = 2;
          a = a + 1;
          // izpis
          cout << "Vrednost lokalne spremenljivke a: " << a <<endl;
          cout << "Vrednost lokalne spremenljivke b: " << b <<endl;
          cout << "Vrednost globalne spremenljivke a: " << ::a <<endl;
          return 0;
}

//izpiše
Vrednost lokalne spremenljivke a: 6
Vrednost lokalne spremenljivke b: 2
Vrednost globalne spremenljivke a: 2
```

V/I funkcije

Na standardni izhod (zaslon) izpisujemo z uporabo standardne funkcije **cout**, ki se nahaja v <iostream>. Omogoča kontrolo izpisa v izhodni tok, ki je povezan s standardnim izhodom v <cstdio>.
Za oblikovanje izpisa lahko uporabimo možnosti, ki se nahajajo v zaglavju <iomanip.h>.

showpos	Izpiše predznak + v primeru izpisa pozitivnega števila.
showbase	Izpiše osnovo številskega sestava (bazo).
uppercase	Male črke pretvori v velike.
showpoint	Izpis decimalne pike pri izpisu realnih števil.
left	Vstavi zaporedje zapolnitvenih znakov na desno stran.
right	Vstavi zaporedje zapolnitvenih znakov na levo stran.
dec	Pretvori v desetiško število.
hex	Pretvori v šestnajstiško število.
oct	Pretvori v osmiško število.
fixed	Izpiše realno število v fiksni obliki.
scientific	Izpiše realno število v znanstveni obliki.
setfill(char c)	Določi zapolnitveni znak.
setprecision(int n)	Določi natančnost (določeno število decimalnih mest).
setw(int n)	Določi širino (za izpis) v številu znakov.
endl	Vstavi znak za prehod v novo vrstico v izhodni tok.
ends	Vstavi znak null v izhodni tok.
flush	Počisti izhodni tok.

Privzete vrednosti so: precision() = 6, width() = 0, fill() = 'znak_presledek', flags()=skipws | dec.s

Zgled:

```
// V/I funkcije

# include <iostream.h>
# include <iomanip.h>
```

```
void main ( )          // void -> funkcija ne vrne nobene vrednosti

{
          int i;
          float f;

          // branje celega in realnega števila
          cin >> i >> f;
          // izpis celoštevilčne spremenljivke in zaključek vrstice
          cout << i << endl;
          // izpis realne spremenljivke in zaključek vrstice
          cout << f << endl;
          // izpis spremenljivke i v šestnajstiškem sistemu
          cout << hex << i << endl;
          // izpis spremenljivke i osmiškem in nato desetiškem sistemu
           cout << oct << i << dec << i << endl;
          // izpis spremenljivke i s predzakom

          cout << showpos << i << endl;
          // izpis spremenljivke i v šestnajstiškem sistemu
          cout << setbase(16) << i << endl;
         // izpis sprem. i v desetiškem sistemu na levi strani v širini 20-ih znakov, kjer se na desni doda znak '@'
          cout << setfill('@') << setw(20) << left << dec << i;
          cout << endl;
          // isto še na drug način
          cout.fill('@');
          cout.width(20);
          cout.setf(ios_base::left, ios_base::adjustfield);
          cout.setf(ios_base::dec, ios_base::basefield);
          cout << i << endl;
          // izpis sprem. f v znanstvenem formatu, natančnost na  10 mest
          cout << scientific << setprecision(10) << f << endl;
}

//za vnos vrednosti
25  25.45
// dobimo izpis
25
25.45
19
3125
+25
19
+25@@@@@@@@@@@@@@@@
+25@@@@@@@@@@@@@@@@
+2.5450000763e+01
```

S standardnega vhoda (zaslon) beremo z uporabo standardne funkcije **cin**, ki se nahaja v <iostream>. Omogoča kontrolo branja iz vhodnega toka, ki je povezan s standardnim vhodom v <cstdio>.

Zgled:

```
#include <iostream>
int main ( )
{
        using namespace std;
        char p[50];
        // odstrani vse začetne presledke oz. prehode v novo vrstico
        cin >> ws;  //white space
        // prebere 50 znakov
```

```
        cin.getline(p,50); // branje vrstice
        cout << p;
        return 0;
}
```

Operatorji, izrazi, stavki

Operandi so podatki, nad katerimi izvajamo operacije. Operatorji določajo vrsto operacije, ki se izvede nad enim ali več operandi. Poznamo več vrst operatorjev in sicer: **aritmetični**, **relacijski**, **logični**.

Aritmetični operatorji za cela in realna števila

Operator	Opis
*	množenje
/	deljenje
%	ostanek celoštevilčnega deljenja
+	seštevanje
-	odštevanje
++(sprem.)	povečevanje (za ena) oz. inkrementiranje
--(sprem.)	zmanjševanje (za ena) oz. dekrementiranje

Za operatorja povečevanja in zmanjševanja veljajo enaka pravila glede na to ali stoji operator pred ali za spremenljivko. **Če stoji operator povečevanja pred spremenljivko, potem dobi celoten izraz vrednost povečane spremenljivke. Če stoji za spremenljivko, pa vrednost spremenljivke pred povečanjem.**

Operatorje, ki imajo dva operanda imenujemo **binarni**, operatorje z enim operandom pa **unarni**.

Operator predznaka (-) služi za določanje predznaka.

Operatorji prirejanja

Operator prirejanja = združuje z desne proti levi, kar pomeni, da se najprej ovrednosti desna stran in nato leva.
Veliko stavkov v jeziku C++ je oblike **sprem_x = sprem_x + c**;.
Namesto tega lahko krajše zapišemo **sprem_x +=** c;
Podobno velja še za:

Krajše	Daljše
a += b;	a = a + b;
a -= b;	a = a - b;
a *= b;	a = a * b;
a /= b;	a = a / b;
a %= b;	a = a % b;

Relacijski operatorji

Uporabljamo jih za zapis predikatov v odločitvah.

Operator	Opis
= =	enakost (je enako)
!=	neenakost (ni enako)
<	manjše
>	večje
<=	manjše ali enako
>=	večje ali enako

Zgled:

```
int  izraz()
{
        int a = 5, b = 6;
        if  (a == b)
           a+=1;
        else
           b+=1;
        return a;
}
```

Izraz	*Vrednost*
(7 == 4)	*false*
(34 > 6)	*true*
(23 != 32)	*true*
(1 >= 1)	*true*
(12 < 10)	*false*

Izraz	*Vrednost*
(a < b)	*true*
(a==b)	*false*
(a != 5)	*false*

Logični operatorji (!, &&, ||)

Operator ! je logična negacija (NOT). Nahaja se levo od operanda in vrne logično nasprotno vrednost. Npr.

```
!(67 == 67) // false (ker je izraz desno od ! true)
!(16 <= 14) // true (ker je (16 <= 14) false)
!true       // false
!false      // true
```

Operatorja **&&** - logični in (and) in || - logični ali uporabljamo z dvema operandoma. Logična tabela za && in || je naslednja:

operand a1	**operand a2**	**logični in - &&**	**logični ali - ||**
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

Izraz

Izraz je sestavljen iz operatorjev in operandov. Operandi so lahko konstante, spremenljivke, klici funkcij ali poljubna kombinacija prejšnjih. Izjema je operator prirejanja, pri katerem mora biti na levi strani spremenljivka. Vsak izraz ima svojo vrednost. Vrednosti pri aritmetičnih operatorjih so znane. Vrednost prireditvenega operatorja je kar vrednost izraza na desni strani znaka =. Vrednosti logičnih operatorjev sta 0 (neresnično - false) ali 1 (resnično - true).

Vrstni red izračunavanja izrazov

Operatorji (z izjemo operatorja prirejanja in unarnih operatorjev) združujejo izraz od leve proti desni. To pravilo velja med enakimi operatorji. Stavek:

```
b = e + 4 - 10 + q;
```

se izračuna kot:

```
b = (((e + 4) - 10) + q);
```

Če v izrazu nastopajo operatorji z različno prioriteto, se najprej izračunajo tisti z najvišjo prioriteto in tako naprej proti najnižji prioriteti.

Bitni operatorji (&, |, ^, ~, <<, >>)

Bitni operatorji ovrednotijo izraz na osnovi posameznih bitov, ki predstavljajo vrednost operanda.

Operator	Pomen	Opis
&	**AND**	logični IN
|	**OR**	logični ALI
^	**XOR**	logični ekskluzivni ALI
~	**NOT**	komplement
<<	**SHL**	pomik v levo
>>	**SHR**	pomik v desno

AND &

x	y	x&y
0	0	0
0	1	0
1	0	0
1	1	1

OR |

x	Y	x|y
0	0	0
0	1	1
1	0	1
1	1	1

XOR ^

x	y	x^y
0	0	0
0	1	1
1	0	1
1	1	0

~ NOT

X	~x
0	1
1	0

Zgled:
Dve spremenljivki x in y podatkovnega tipa char vsebujeta vrednosti **195** (**11000011**) in **87** (**01010111**).

Zgled:
```
char a=195;
char b=87;
char c;
c=a&b; // vrednost 67 (01000011)
```

a = 195	1	1	0	0	0	0	1	1	&
b = 87	0	1	0	1	0	1	1	1	
c = 67	0	1	0	0	0	0	1	1	

V C++ obstajajo standardne pretvorbe, če so operandi različnih (mešanih) podatkovnih tipov. To izvršuje prevajalnik v mešanih izrazih.
Veljajo pravila:

- Podatkovni tipi char, short in enum se pretvorijo v podatkovni tip int. Če pri ovrednotenju postane vrednost izraza prevelika za int, se pretvori v podatkovni tip unsigned int.
- Podatkovna tipa unsigned char in unsigned short se pretvorita v podatkovni tip unsigned int.

Kadar je po prejšnjih korakih izraz še vedno mešan, se operandi pretvorijo po naslednji hierarhiji podatkovnih tipov; manjši podatkovni tipi se pretvorijo v večji podatkovni tip:
int < unsigned int < long < unsigned long < float < double < long double.

Operator za pretvarjanje med podatkovnimi tipi - ()

Operator za pretvarjanje nam omogoča, da izraz določenega podatkovnega tipa pretvorimo v drug podatkovni tip tako, da pred izraz v okrogle oklepaje zapišemo podatkovni tip.

```
int celo_st;
float realno_st = 3.14;
celo_st = (int) realno_st; //realno število smo pretvorili v celo število (vrednost 3)
```

Operator sizeof

Operator **sizeof(**objekt ali podatkovni tip**)** vrne velikost zlogov, ki jih zaseda podatkovni tip ali objekt.

```
// število zlogov, ki jih zasedejo osnovni podatkovni tipi
#include <iostream.h>

int main ()
{
   int a;
   char  znak;
   float realno_st;
   double realno_st_dvojna_nat;
   bool zastavica;
   cout << "Spremenljivka podatkovnega tipa int zasede: " << sizeof(a)
        << "(št. zlogov)" <<endl;
   cout << "Spremenljivka podatkovnega tipa char zasede: " << sizeof(znak)
        << " (št. zlogov)" <<endl;
   cout << "Spremenljivka podatkovnega tipa float zasede: "
        << sizeof(realno_st) << " (št. zlogov)" <<endl;
   cout << "Spremenljivka podatkovnega tipa double zasede: "
        << sizeof(realno_st_dvojna_nat) << " (št. zlogov)" <<endl;
   cout << "Spremenljivka podatkovnega tipa bool zasede: "
        << sizeof(zastavica) << " (št. zlogov)" <<endl;
   return 0;
}
// izpiše
Spremenljivka podatkovnega tipa int zasede: 4 (št. zlogov)
Spremenljivka podatkovnega tipa char zasede: 1 (št. zlogov)
Spremenljivka podatkovnega tipa float zasede: 4 (št. zlogov)
Spremenljivka podatkovnega tipa double zasede: 8 (št. zlogov)
Spremenljivka podatkovnega tipa bool zasede: 1 (št. zlogov)
```

Prioritete operatorjev so naslednje:

Prioriteta	Operator	Opis	Združevanje
1	::	doseg	levo
2	() [ ] -> . sizeof		levo
3	++ --	inkrement / dekrement	desno
	~	komplement	
	!	unarni NOT	
	& *	naslov in vrednost naslova	
	(*podat_tip*)	pretvorba med tipi	
	+ -	predznak	
4	* / %	aritmetični operatorji	levo
5	+ -	aritmetični operatorji	levo
6	<< >>	bitni pomik	levo
7	< <= > >=	relacijski operatorji	levo
8	== !=	relacijski operatorji	levo
9	& ^ |	bitni operatorji	levo
10	&& ||	logični operatorji	levo
11	?:	pogojni operator	desno
12	= += -= *= /= %= >>= <<= &= ^= |=	prireditveni operator	desno
13	,	separator	levo

Združevanje določa pravilo, kateri del se ovrednosti najprej, če imamo več operatorjev na istem nivoju prioritete in niso uporabljeni oklepaji.

```
a = 5 * 7 % 2;          // se lahko ovrednoti kot
a = 5 * (7 % 2);        // v primeru desnega združevanja
a =(5 * 7) % 2;         // v primeru levega združevanja
```

Pri operatorjih ++ in -- upoštevamo še pravili:

- teh operatorjev ne uporabljamo na spremenljivkah, ki so del argumenta funkcije z več argumenti (printf(st * st++)),
- ne uporabljamo ju na spremenljivkah, ki nastopajo na več mestih v istem izrazu (nekaj = st /3 + (2 * 4 + st++)).

Krmilne strukture

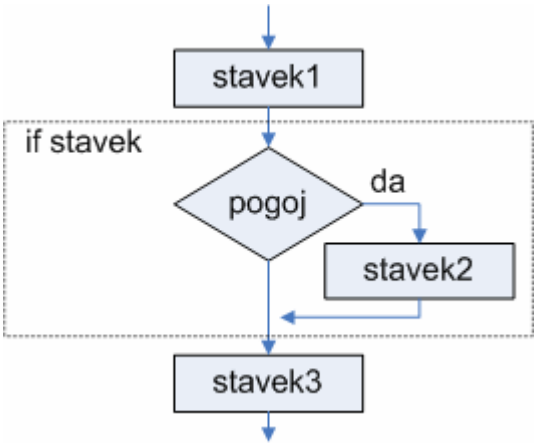
Program je sestavljen iz zaporedja stavkov (sekvenca), izbir stavkov (selekcija) in ponavljanja stavkov (iteracija).
Blok stavkov je več stavkov, ki so med seboj ločeni s podpičjem in se nahajajo med zavitima oklepajema npr.**{** stavek1; stavek2; ... **}**. Če želimo, da je samo en stavek, potem ne uporabimo oklepajev oz. oznake za blok stavkov. V primeru, ko želimo sestavljeni stavek pa uporabimo blok stavkov.

If stavek

Če želimo, da se določen stavek (ali sestavljen stavek) izvede le ob določenem pogoju uporabimo if stavek.

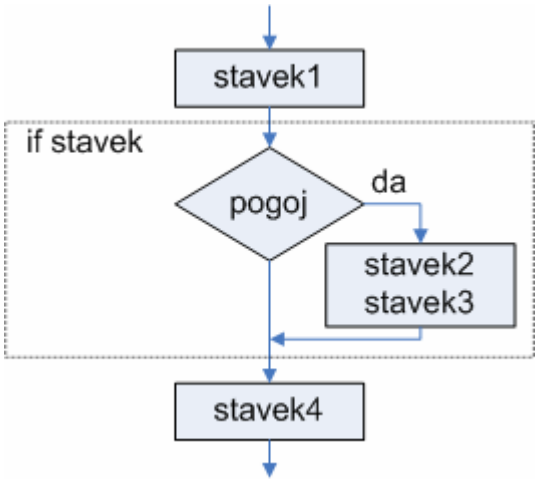
If stavek ima sintakso (oblika 1):

```
stavek1;
if (pogoj)
        stavek2;
stavek3;
```



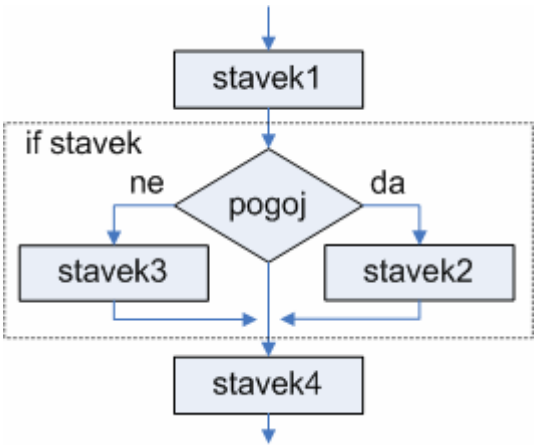
Pogoj je izraz, ki se ovrednosti s true ali false. Le v primeru, če je pogoj izpolnjen (true), se izvede stavek2. Če pogoj ni izpolnjen (false), se stavek2 ne izvede in izvajanje programa se nadaljuje z naslednjim stavkom, ki sledi stavku if (stavek3). Če želimo, da se izvede sestavljen stavek, potem uporabimo blok stavkov.

```
stavek1;
if (pogoj)
{
        stavek2;
        stavek3;
        //...
}
stavek4;
```



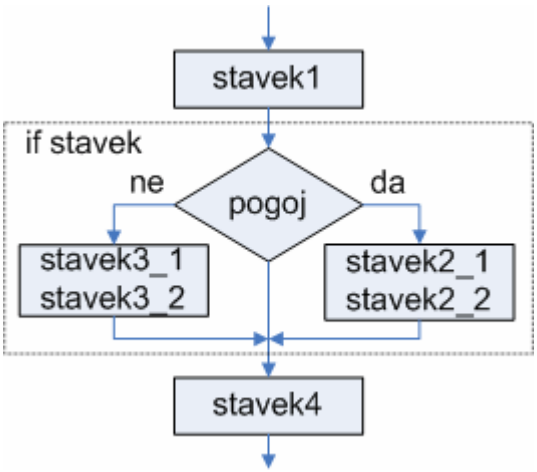
Oblika sintakse z uporabo else (oblika 2):

```
stavek1;
if (pogoj)
        stavek2;
else
        stavek3;
stavek4;
```



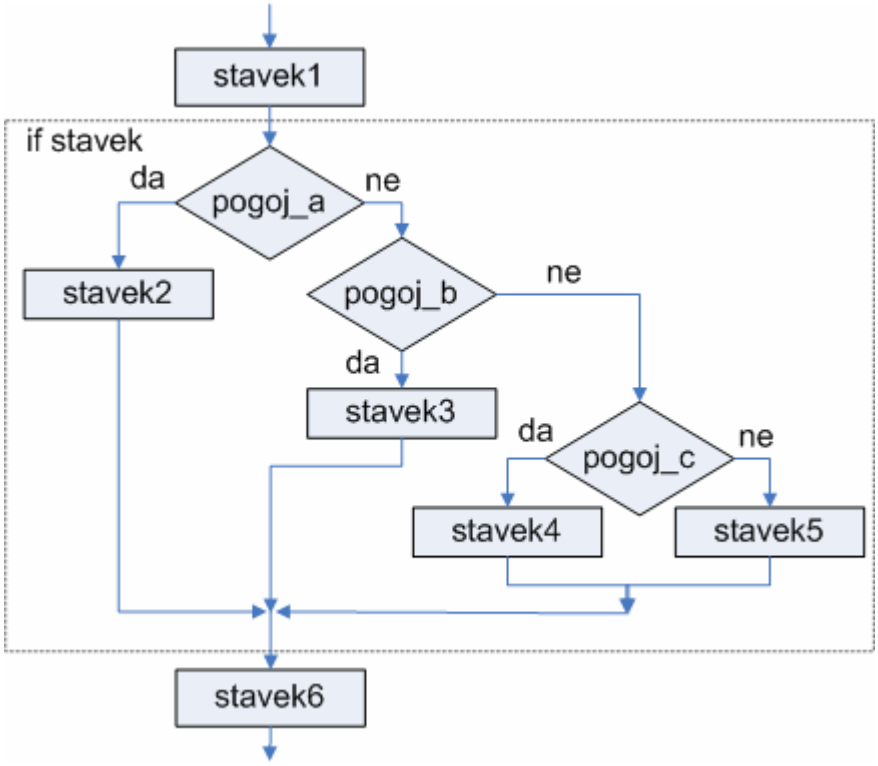

```
// ali
```

```
stavek1;
if (pogoj)
{
        stavek2_1;
        stavek2_2;
        // ...
}
else
{
        stavek3_1;
        stavek3_2;
        // ...
}
stavek4;
```



Stavke if lahko tudi gnezdimo (stavek v if stavku je if stavek).

```
stavek1;

if (pogoj_a)
  stavek2;
else
  if (pogoj_b)
    stavek3;
  else
    if (pogoj_c)
      stavek4;
    else
      stavek_5;

stavek6;
```



Zgled:

```
// preberemo tri števila in izpišemo največje število
#include <iostream.h>
int main ( )
{
        float st1, st2; //deklaracija dveh spremenljivk
        cout <<"Vnesi prvo število: " ;
        cin >> st1; //preberemo prvo število
        cout <<"Vnesi drugo število: " ;
        cin >> st2; //preberemo drugo število
        if  (st2 > st1) // je drugo število večje od prvega
                st1 = st2;   // če je => prvemu številu priredimo vrednost drugega števila
        cout <<"Vnesi tretje število: " ;
        cin >> st2; //preberemo tretje število
        if  (st2 > st1) // je tretje število večje od do sedaj največjega
                st1 = st2; // če je => prvemu številu priredimo vrednost tretjega števila
        cout << "Največje število je: " << st1 <<endl;
        return 0;
}

//izvajanje
Vnesi prvo število: 25
Vnesi drugo število: 36.45
Vnesi tretje število: 22
Največje število je: 36.45
```

Stavek switch

Kadar imamo več vej kot dve (pri stavku if) in se mora izvršiti le posamezna veja, potem uporabimo stavek switch. Glede na vrednost izraza se izvrši običajno le ena veja izmed vseh možnih.

Sintaksa stavka switch:

```
switch (izraz)
{
        case konstanta_1:
                zaporedje_stavkov_1;
                break;
        case konstanta_2:
                zaporedje_stavkov_2;
                break;
          ...
        default:
                privzeto_zaporedje_stavkov;
}
```

Stavek **switch** ovrednoti izraz in preveri, če se ujema z vrednostjo konstanta_1. Če se ujema, izvrši zaporedje_stavkov_1, dokler ne naleti na rezerviranko **break.** Rezerviranka **break** povzroči vejitev (skok) programa na konec stavka switch, kjer se nadaljuje izvajanje programa.
Če izraz ni enak vrednosti konstanta_1, potem program preveri, če je enak vrednosti konstanta_2. Če se ujema, izvrši zaporedje_stavkov_2 dokler ne naleti na rezerviranko **break.**
Če izraz ni enak nobeni izmed vrednosti konstant, potem program izvede stavke, ki so zapisani v **default** delu (če obstaja).

switch

```
switch (x)
{
  case 1:
     cout << "x je 1";
     break;
  case 2:
     cout << "x je 2";
     break;
  default:
     cout << "x vrednost ni ne
     1 in ne 2";
}
```

if

```
if (x == 1)
{
  cout << "x je 1";
}
else if (x == 2)
{
  cout << "x je 2";
 }
    else
    {
      cout << " x vrednost ni ne 1 in ne 2";
     }
```

Pri posamezni veji oz. izbiri ne smemo pozabiti vključiti rezerviranke **break**. Če je ne vključimo, program nadaljuje izvajanje ostalih delov v **switch** stavku.
Namesto konstanta_i ne moremo uporabiti spremenljivk (i+3) ali območij (1..10).

Zgled:

```
// Vnašamo števila iz intervala od 1 do 5 dokler ne vnesemo števila -1.
// Program izpiše frekvenco pojavitve posameznih števil.
// Program naj ne uporablja polj.
#include <iostream.h>

int main ( )
{
        int st, st_1 = 0, st_2 = 0, st_3 = 0, st_4 = 0, st_5 = 0;
       // ponavljenje dokler uporabnik ne vnese -1
        do
        {
                cout <<"Vnesi  število (-1 za konec): " ;
                cin >> st; //preberemo število
                // povečamo ustrezno spremenljivko glede na vrednost
                // vnesenega števila
                switch (st)
                {
                        case 1: ++st_1; break;
                        case 2: ++st_2; break;
```

```
                        case 3: ++st_3; break;
                        case 4: ++st_4; break;
                        case 5: ++st_5; break;
                        default: ; //prazen stavek
                }
        }
        while (st != -1);
        // izpis
        cout << "Število enk je: " << st_1 <<endl;
        cout << "Število dvojk je: " << st_2 <<endl;
        cout << "Število trojk je: " << st_3 <<endl;
        cout << "Število štirk je: " << st_4 <<endl;
        cout << "Število petk je: " << st_5 <<endl;
        return 0;
}
```

```
Vnesi  število (-1 za konec): 3
Vnesi  število (-1 za konec): 2
Vnesi  število (-1 za konec): 3
Vnesi  število (-1 za konec): 2
Vnesi  število (-1 za konec): 3
Vnesi  število (-1 za konec): 2
Vnesi  število (-1 za konec): 5
Vnesi  število (-1 za konec): 1
Vnesi  število (-1 za konec): 4
Vnesi  število (-1 za konec): -1
Število enk je: 1
Število dvojk je: 3
Število trojk je: 3
Število štirk je: 1
Število petk je: 1
```

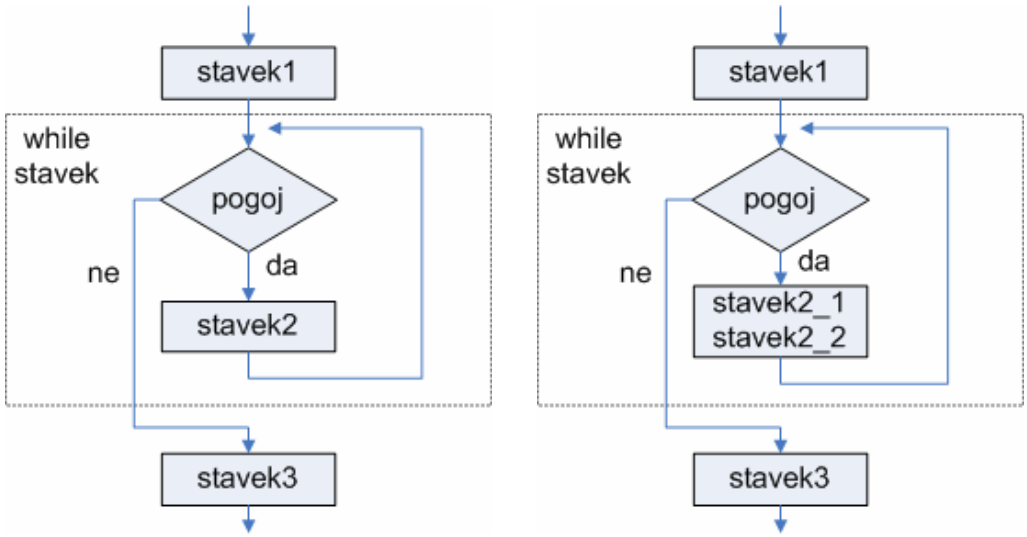
Ponavljanje stavkov (zanke)

Ponavljanje se lahko izvede določeno število krat npr. n – krat ali dokler je izpolnjen pogoj.

While stavek

Sintaksa:

```
stavek1;
while (pogoj)
  stavek2;
stavek3;

// ali

stavek1;
while (pogoj)
{
   stavek2_1;
   stavek2_2;
   ...
}
stavek3;
```



Stavek2 (telo zanke) se izvaja tako dolgo, dokler je izpolnjen pogoj. Če pogoj ob prvem ovrednotenju ni izpolnjen, potem se stavek2 nikoli ne izvede.
Ko prvič ni izpolnjen pogoj, se nadaljuje izvajanje programa v stavku, ki sledi stavku while (v našem primeru je to stavek 3).

Zgled:

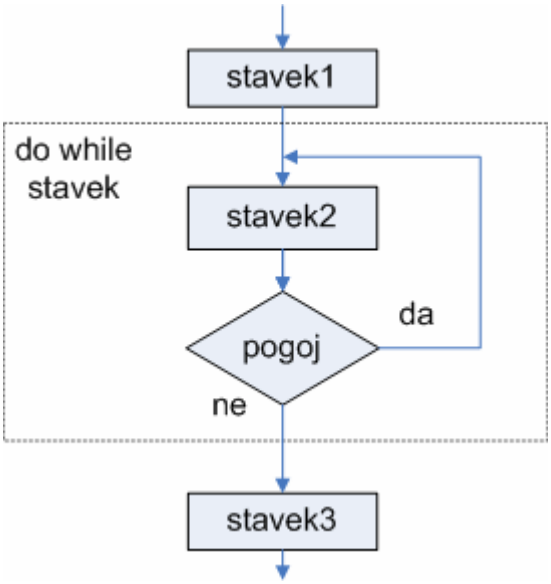
```
// ponavljanje z zmanjševanjem števca
#include <iostream.h>
int main ()
{
          int n;
           cout << "Vpiši celo število > ";
           cin >> n; //preberemo število
           // ponavljamo dokler je število večje od 0
           while (n>0)
           {
                     cout << n << ", ";   // izpišemo število
                     --n;                 // za ena zmanjšamo n
           }
           cout << "Konec!";
           return 0;
}

//izpiše
Vpiši celo število > 5
5, 4, 3, 2, 1, Konec!
```

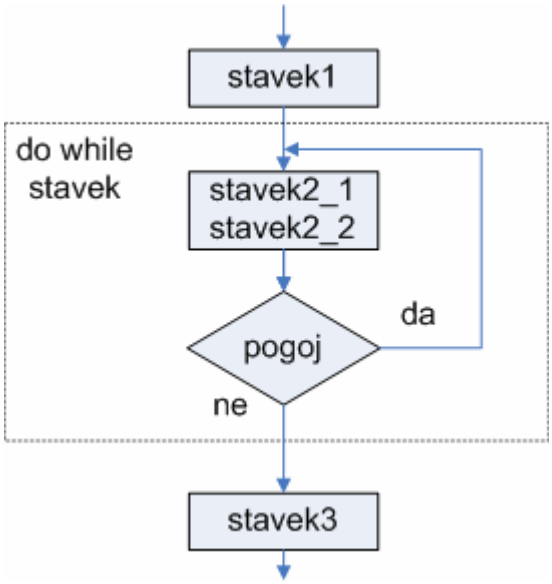
Do - while stavek

Od **while** stavka se razlikuje po tem, da se ne glede na pogoj izvede vsaj enkrat, kar je posledica tega, da se pogoj testira po izvršitvi stavka zanke (stavek2 ali blok stavkov { stavek2_1;stavek2_2}).

Sintaksa:

```
stavek1;
do
   stavek2;
while (pogoj);
stavek3;

// ali

stavek1;
do
{
   stavek2_1;
   stavek2_2;
   ...
} while (pogoj);
stavek3;
```



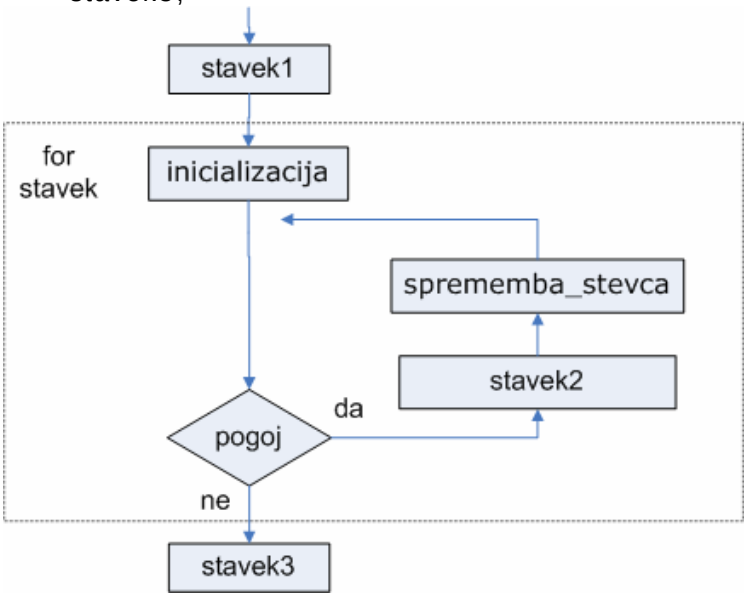


Ko se v preverjanju pogoja ugotovi, da pogoj ni izpolnjen, se nadaljuje izvajanje programa v stavku, ki sledi stavku while (v našem primeru je to stavek 3).

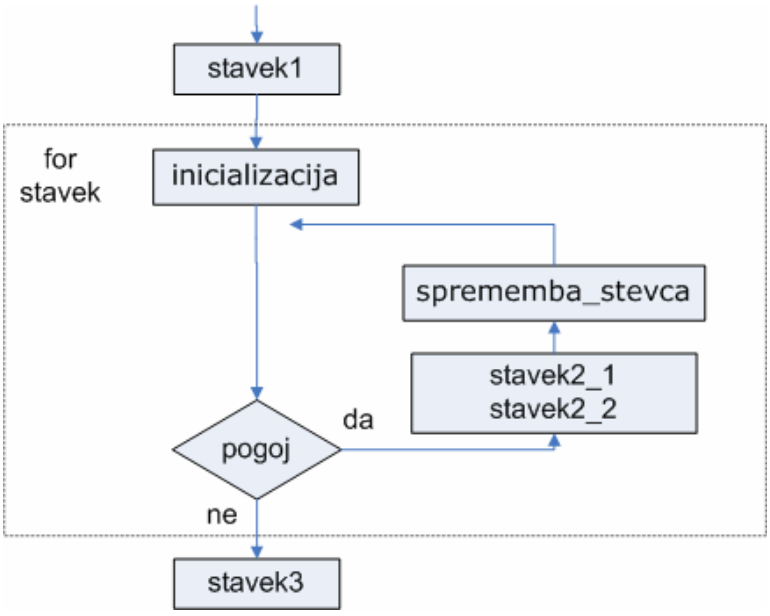
For stavek

Sintaksa:

```
stavek1;
for (inicializacija; pogoj; sprememba_stevca)
        stavek2;
stavek3;
```




```
// ali

stavek1;
for (inicializacija; pogoj; sprememba_stevca)
{
        stavek2_1;
        stavek2_2;
        ...
}
stavek3;
```



Stavek (stavek2) ali blok stavkov v **for** stavku se izvršuje dokler je izpolnjen pogoj. V inicializacijskem delu deklariramo in določimo začetno vrednost spremenljivke, ki jo uporabimo za števec v zanki.

V delu sprememba števca pa določimo spremembo števca (povečujemo oz. zmanjšujemo spremenljivko števca - korak).

Delovanje:
1. Izvrši se inicializacija (**le enkrat**).
2. Preveri se pogoj. Če pogoj ni izpolnjen, se nadaljuje izvajanje programa za **for** stavkom.
3. Izvrši se stavek **for** zanke, ki je lahko posamezen stavek ali blok stavkov ({stavek2_1; stavek2_2... }).
4. Izvrši se sprememba števca in izvajanje se nadaljuje v točki 2.

Zgled:
```
// for zanka z zmanjševanjem števca
#include <iostream.h>
int main ()
{
          for (int n=10; n>0; n--)  //ponavljanje dokler je n > 0
          {
                    cout << n << ", "; //izpis vrednosti števca
          }
          cout << "Konec!";
          return 0;
}

//izpiše
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Konec!
```

Dela inicializacija in sprememba števca sta opcijska (podpičja so obvezna). Npr. zapišemo lahko **`for (;n<10;).`** V posamezen del lahko zapišemo **več stavkov**, ki **jih med seboj ločimo z vejico** (**`,`**). V tem primeru je vejica ukazni separator.
Zgled:
```
          for ( n=0, i=50 ; n!=i ; n++, i-- )
          {
             // obdelava...
          }
```

V inicializaciji priredimo spremenljivki n vrednost 0 in spremenljivki i vrednost 50. Pogoj: spremenljivka n ni enaka spremenljivki i. Ker se n povečuje za ena in i zmanjšuje za ena bo pogoj zanke izpolnjen po 25-ih izvršitvah zanke, ko bosta spremenljivki n in i imeli vrednost 25.

Stavek *break*

Če uporabimo **break** v bloku stavkov zanke, potem se izvajanje zanke **prekine** ne glede na to, da pogoj še ni izpolnjen. Uporabimo ga lahko za izhod iz neskončne zanke (pogoj je vedno izpolnjen).

Stavek continue

Continue povzroči, da program preskoči preostale stavke v zanki v trenutnem ponavljanju zanke in nadaljuje izvajanje, kot da bi zaključil vse stavke v bloku stavkov zanke (izvede se naslednja ponovitev zanke). **Uporabimo ga lahko samo v zankah.**

Stavek *goto*

Stavek ***goto*** povzroči brezpogojno vejitev (skok) na drugo točko (oznako) v programu. Pri tem se ne upoštevajo omejitve gnezdenja ponavljanj. Oznako v programu določimo tako, da za njeno ime postavimo dvopičje. **Stavek ne uporabljamo, ker krši pravila strukturiranega programiranja!**

Zgled:

```
// goto stavek
```

```
#include <iostream.h>
int main ()
{
        int n=10;
oznaka1:
          cout << n << ", ";
          n--;
          if (n>0)
                goto oznaka1;
          cout << "Konec zanke";
          return 0;
}

//izpiše
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Konec zanke
```

Funkcija *exit*

Funkcija **exit** je definirana v knjižnici **cstdlib** (stdlib.h). Namen funkcije je prekinitev izvajanja programa z ustrezno izhodno vrednostjo.
Prototip funkcije:

```
void exit (int izhodna_vrednost);
```

Izhodno vrednost uporabljajo posamezni operacijski sistemi. Izhodna vrednost 0 pomeni, da se program normalno zaključil, vrednost večja od nič pa, da je prišlo do napake med izvrševanjem.

Zgledi:

```
// program izračuna povprečno vrednost N (n vstavimo) celih števil,
// ki jih vstavi uporabnik
#include <iostream.h>
#include <conio.h>

int main ()
{
        int n; //število števil
        int i; // števec
        int vsota = 0, stevilo; // vsota števil in tekoče prebrano število
        float povprecje; // povprečna vrednost N števil
        // počistimo zaslon - conio.h
        clrscr();
        cout << "Vnesi število N: ";
        cin >> n;  //preberemo število števil - n
        for (i=1; i <= n; i++) // ponovimo n - krat
        {
                cout << "Vnesi " << i << ". to število: ";
                cin >> stevilo; // preberemo i - to število
                vsota = vsota + stevilo; // tekoče število prištejemo vsoti vnešenih števil
        }
        if (n > 0) //je vnešeno kakšno število
        {
                povprecje = (float) vsota / n; // izračunamo povprečje
                cout << "Povprečje je " << povprecje <<endl ;//izpišemo povprečje
        }
        return 0;
}

// program izračuna število sodih in število lihih števil
// uporabnik vnaša števila dokler ne vnese števila 0
#include <iostream.h>
#include <conio.h>
```

```
int main ()
{
        int st_sodih = 0, st_lihih = 0; // število sodih in število lihih števil
        int stevilo; // tekoče število
        // počistimo zaslon - conio.h
        clrscr();
        do
        {
                cout << "Vnesi celo število: ";
                cin >> stevilo;  //preberemo tekoče število
                if (stevilo != 0)
                        if (stevilo % 2 == 0) //ostanek po celoštevilčnem deljenju z 2
                                                je enak nič
                                st_sodih++;   //povečamo število sodih števil
                        else
                                st_lihih++;//povečamo število lihih števil
        }
        while (stevilo != 0);
        cout << "Število sodih števil je " << st_sodih <<endl ;
        cout << "Število lihih števil je " << st_lihih <<endl ;
        cout << "Število je  " << stevilo <<endl ;
        return 0;
}
```

```
// program izriše pravokotnik s pomočjo znaka *
// uporabnik vnese stranici pravokotnika

#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

int main ()
{
        int a, b; // stranici pravokotnika
        int i, j; // števca
        // počistimo zaslon -  conio.h
        clrscr();
        cout << "Vnesi stranico a (celo število): ";
        cin >> a;   //preberemo stranico
        cout << "Vnesi stranico b (celo število): ";
        cin >> b;   //preberemo stranico
        cout << endl;   //prehod v novo vrstico

        for (i=1; i <= b; i++) //ponavljanje vrstic
        {
                cout << setw(5) <<" "; //izpis presledka na pet mest
                for (j=1; j <= a; j++) //ponavljanje stolpci
                {
                        cout << "*";   // izpis znaka
                }
                cout << endl;   //prehod v novo vrstico
        }
        cout << endl;   //prehod v novo vrstico
        return 0;
}
// izvajanje
Vnesi stranico a (celo število): 4
Vnesi stranico b (celo število): 7

     ****
     ****
     ****
     ****
     ****
     ****
     ****

--------------------------------------------------------------------------------

// program izriše pravokotnik
// obroba - znak *
// notranjost - znak _
// uporabnik vnese stranici pravokotnika

#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

int main ()
{
        int a, b; // stranici pravokotnika

        // počistimo zaslon -  conio.h
```

```
        clrscr();
        cout << "Vnesi stranico a (celo število): ";
        cin >> a;  //preberemo stranico
        cout << "Vnesi stranico b (celo število): ";
        cin >> b;  //preberemo stranico
        cout << endl;  //prehod v novo vrstico

        for (int i=1; i <= b; i++) //ponavljanje vrstic
        {
                cout << setw(5) <<" "; //izpis presledka na pet mest

                if ((i != 1) && (i !=b))
                {
                        cout << "* ";  // izpis znaka
                        for (int j=2; j <a; j++)
                                cout << "_ ";
                        cout << "* ";  // izpis znaka
                } else
                {
                        for (int j=1; j <=a; j++)
                                cout << "* ";
                }

                cout << endl;  //prehod v novo vrstico

        }
        cout << endl;  //prehod v novo vrstico
        return 0;
}

// izvajanje
Vnesi stranico a (celo število): 4
Vnesi stranico b (celo število): 6
     * * * *
     * _ _ *
     * _ _ *
     * _ _ *
     * _ _ *
     * * * *
```

```
// izpiše lik_
//   *
//   **
//   ***
//   ****
//   *****
//   *___**
//   *___***
//   *___****
//   **********

//     a = 5 (stranica)
//   _____
//   *****         |
//   *___*         |
//   *___*         | b = 5 (stranica)
//   *___*         |
//   ******        |

#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
int main ()
{
        int a, b; // stranici pravokotnika
        // počistimo zaslon -  conio.h
        clrscr();
        cout << "Vnesi stranico a (celo stevilo): ";
        cin >> a;  //preberemo stranico
        cout << "Vnesi stranico b (celo stevilo): ";
        cin >> b;  //preberemo stranico
        cout << endl;  //prehod v novo vrstico
        for (int i=1; i <= a-1; i++) //ponavljanje vrstic
        {
                cout << setw(5) <<" "; //izpis presledka na pet mest
                for (int j=1; j <=i; j++)
                        cout << "* ";
                cout << endl;  //prehod v novo vrstico
        }
        for (int i=1; i <= b; i++) //ponavljanje vrstic
        {
                cout << setw(5) <<" "; //izpis presledka na pet mest

                if ((i == 1) || (i ==b))
                {
                        for (int j=1; j <=a; j++)
                                cout << "* ";
                        if (i == b)
                        {
                                for (int j=1; j <=a; j++)
                                        cout << "* ";
                        }
                }  else
                {
                        cout << "* ";  // izpis znaka
                        for (int j=2; j <a; j++)
                                cout << "_ ";
                        cout << "* ";  // izpis znaka
```

```
                        for (int j=1; j <=i-1; j++)
                                cout << "* ";   // izpis znaka

                }
                cout << endl;   //prehod v novo vrstico

        }
        cout << endl;   //prehod v novo vrstico
        return 0;
}
```

Kazalci

Spremenljivke, ki jih uporabljamo v programih, so shranjene v pomnilniku na določenih pomnilniških lokacijah. Najlažje si pomnilnik predstavljamo v obliki enodimenzionalne tabele, kjer indeks predstavlja pomnilniška lokacija oz. pomnilniški naslov. **Kazalec je spremenljivka, katere vrednost je pomnilniški naslov druge spremenljivke.** To omogoča, da obravnavamo naslov pomnilniške lokacije kot spremenljivko.

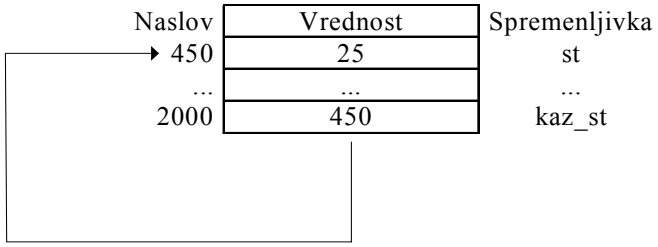
Pomnilniški naslov	Pomnilniška beseda
2000	00010111
2001	10010000
2002	00011000
2003	00111100
2004	11111000
2005	10101010
2006	11001100
2007	10101100
2008	11111110

Ker je vsebina kazalca pomnilniški naslov, na katerem je shranjen določen objekt, lahko dostopimo do tega objekta z uporabo kazalca.

```
int st = 25;
```

Naslov	Vrednost	Spremenljivka
450	25	st

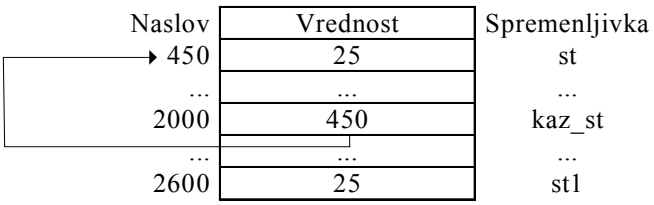
```
int *kaz_st;      // kazalec na pomnilniško lokacijo, kjer je shranjeno celo število

kaz_st = &st;     // prireditev naslova, kjer je shranjena vrednost spremenljivke st  (25)
```




```
int *kaz_st=NULL;      // inicializacija vrednosti kazalca; kazalec ne kaže nikamor
```

Naslov	Vrednost	Spremenljivka
450	25	st
...	...	...
2000	NULL	kaz_st

```
int st1 = *kaz_st;          // dereferenca kazalca – vrednost kamor kaže kazalec
```



Deklarirana naj bo spremenljivka a podatkovnega tipa int in kazalec kazalec_a .

Referenčni operator (&).

Unarni operator & izračuna pomnilniški naslov objekta. Sedaj lahko kazalčni spremenljivki priredimo naslov, na katerem je shranjena spremenljivka a v pomnilniku:

kazalec_a = &a; // sprem. kazalec_a priredimo naslov na katerem je shranjena sprem. a

Pravimo, da kazalec_a "kaže" na a. Operator & lahko uporabimo samo na spremenljivkah in elementih polja. Npr. uporaba je napačna: &(2+a). Ne moremo pa izračunati naslova od registerske spremenljivke.

Dereferenčni operator (*)

Unarni operator * uporabi operand kot naslov ciljnega objekta in vrne vrednost, ki je shranjena na tem naslovu. Torej vrne vrednost z naslova, ki je vrednost operanda.

```
int stevilo, T1;
int *temp;
stevilo = *temp;
T1 = stevilo;
```

Spremenljivka stevilo dobi vrednost 200, ker ima temp vrednost 2200 in na tem naslovu je shranjena vrednost 200.

Naslov	Vrednost
temp	2200
...	...
2200	200
...	...
T1	200
stevilo	200

Ločiti moramo **pomnilniški naslov**, ki je shranjen v spremenljivki temp (2200) od **vrednosti, ki je shranjena na tem naslovu** *temp (200).

stevilo = temp; *// sprem. stevilo priredimo enako vrednost kot sprem. temp (2200)*

stevilo = *temp; *// sprem. stevilo priredimo vrednost na naslovu temp (200)*

Deklaracija kazalcev

Pri deklaraciji kazalcev moramo določiti podatkovni tip objekta, na katerega kaže kazalec. Sintaksa ima obliko:

podatkovni_tip * ime_kazalca;

Zgledi:

```
int * kaz_stevilo;
char * kaz_znak;
float * kaz_realno_st;
```

Vsi trije primeri deklaracije so različnega podatkovnega tipa (objekti na katere kaže kazalec). Kljub temu vsi trije kazalci zavzamejo enako količino pomnilniškega prostora (prostor za pomnilniški naslov, ki je odvisen od operacijskega sistema).
Znak * beremo kot je kazalec in ga ne smemo zamenjati z referenčnim operatorjem.

```
// uporaba kazalca

#include <iostream.h>
int main ()
{
     // deklaracija kazalca na objekt  podatkovnega tipa int
    int * kaz_stevilo;
     // deklaracija in inicializacija spremenljivk
    int vrednost1 = 5, vrednost2 = 15;
     //priredimo pomnilniški naslov spremenljivke vrednost1
    kaz_stevilo = &vrednost1;
     // v pomnilniško lokacijo, ki je določa kaz_stevilo shranimo vrednost 10
    *kaz_stevilo = 10;
     // priredimo pomnilniški naslov spremenljivke vrednost2
    kaz_stevilo = &vrednost2;
     // v pomnilniško lokacijo, ki je določa kaz_stevilo shranimo vrednost 20
    *kaz_stevilo = 20;
    cout << "vrednost1==" << vrednost1 << "/ vrednost2==" << vrednost2;
     // izpiše: vrednost1==10  / vrednost2==20
    return 0;
}
```

Kazalci in funkcijski argumenti

Z uporabo kazalcev lahko prenašamo funkcijske argumente po referenci. Če imamo funkcijo zamenjaj zapisano kot:

```
int zamenjaj(int x, int y)
{
    int temp;
    temp= x;
    x=y;
    y=temp;
    return (0);
}
```

potem ta funkcija ne prenese novih vrednosti v parametra, s katerima kličemo funkcijo (je napačna). S prenosom parametrov po referenci pa lahko popravimo napako. Funkcijo zapišemo kot:

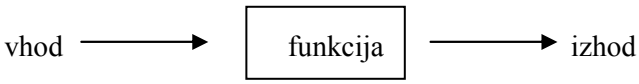
```
int zamenjaj(int &x, int &y)
{   int temp;
    temp = x; x=y; y=temp;
    return (0);
}

// ali s kazalci

int zamenjaj1(int *x, int *y)
{   int temp;
    temp = *x; *x=*y; *y=temp;
    return (0);
}
```

```
void main()
{
    int st1 = 5;  int st2 = 10;
    cout << st1 << "\t" << st2 << endl;
    zamenjaj(st1,st2);
    // ali
    zamenjaj1(&st1,&st2);
    cout << st1 << "\t" << st2 << endl;
}
```

Funkcije

Funkcije nam omogočajo, da program razdelimo na posamezne manjše naloge in jih nato povežemo v zaključeno celoto. Funkcija predstavlja posamezno zaključeno celoto v programu. Predstavljamo si jo kot črno skrinjico, ki ji podamo vhodne podatke, ona pa nam vrne izhodne podatke.
Prednosti uporabe funkcij:

- večji problem razdelimo na več manjših problemov,
- programi so preglednejši in razumljivejši,
- ni ponavljanja enakih zaporedij stavkov na več mestih v programu,
- programe lažje in kvalitetnejše testiramo,
- sorodne funkcije lahko združimo v knjižnico, ki jo lahko nato uporabljamo v svojih programih,
- itd.



Sintaksa:

podatkovni_tip ime_funkcije (argument1, argument2, ...)
stavek

Podatkovni_tip je podatkovni tip podatka, ki ga vrne funkcija (podatkovni tip rezultata).
Ime funkcije je smiselno ime, ki ga uporabimo pri klicu funkcije.
Argumenti nam omogočajo prenose podatkov v in iz funkcije. Vsak argument ima določen podatkovni tip in ime argumenta (identifikator) npr. int stevilo_kock. Argumente ločimo med seboj z vejico.
Stavek predstavlja telo funkcije in je lahko enostaven stavek ali blok stavkov.
Vrednost funkcije določimo s stavkom **return**, ki ima sintakso: **return** izraz.

Zgled:

```
#include <iostream.h>
//funkcija z dvema vhodnima argumentoma - stranici pravokotnika
float  povrsina_prav (float a, float b)
{
        float r;
        r=a * b;              // izračun ploščine
        return (r);           // vračanje rezultata funkcije
}

int main ()                   // glavna funkcija - začetek programa
{
          float z;
          z = povrsina_prav (5,8);          //klic funkcije z dvema parametroma
          cout << "Površina je " << z;      // izpis rezultata
          return 0;                         // zaključek programa
}

//izpiše
Površina je 40
```

```
float  povrsina_prav (float a, float b)

z = povrsina_prav  (  5,     8   );
```

Še enkrat se spomnimo, da se izvajanje programa vedno začne s funkcijo main, ki predstavlja glavni program. Po deklaraciji spremenljivke z, ji priredimo vrednost, ki ga vrne klic funkcije »povrsina_prav«. Klic funkcije vsebuje dva parametra, ki se preneseta v argumenta (spremenljivki) a in b v funkciji povrsina_prav.

V glavnem programu se ob klicu funkcije »povrsina_prav« posredujeta dve vrednosti in sicer 5 in 8 v ustrezna argumenta a in b v funkciji »povrsina_prav«.
Ob klicu funkcije se kontrola izvajana prenese v funkcijo »povrsina_prav«. **Vrednosti** obeh **parametrov** se **kopirata** v **lokalni spremenljivki** a in b **znotraj funkcije**.

Funkcija ima deklarirano novo spremenlivko r in s stavkom r = a * b; se ji priredi vrednost produkta spremenljivk a in b in sicer 40. Stavek return (r); zaključi funkcijo in vrne kontrolo izvajanja nazaj v glavni program na mesto, kjer je bil klic funkcije. Torej se spremenljivki z priredi vrednost 40 (vrednost, ki jo je vrnila funkcija »povrsina_prav«).

Doseg spremenljivk

Spremenljivke, ki so deklarirane znotraj funkcije oz. znotraj bloka stavkov lahko uporabimo samo znotraj tega dela. Pravimo, da so vidne le v delu, v katerem so deklarirane. Torej jih ne moremo uporabiti zunaj tega dela. Imenujemo jih **lokalne** spremenljivke.
Vsekakor pa lahko deklariramo spremenljivke tudi na začetku programa in so vidne iz kateregakoli dela programske kode (znotraj in zunaj funkcij). Takšnim spremenljivkam pravimo **globalne** spremenljivke. Uporaba globalnih spremenljivk **ni priporočljiva**, saj so programi s tem nepreglednejši, težje razumljivi, več je težav pri testiranju posameznih funkcij, večja je sklopljenost med funkcijami itd.
Če znotraj funkcije deklariramo enako spremenljivko, ki je deklarirana tudi med globalnimi spremenljivkami, potem v tem delu ni vidna globalna spremenljivka.

```
#include <iostream.h>

// globalne spremenljivke
int celo_st;
char znak;
unsigned int st_udelezencev;

int main()

{
// lokalne spremenljivke
        int i,j;
        char zn;
        float  stevilo;

// stavki oz. ukazi
        cout << "Vnesi število udeležencev";
        cin  >> stevilo;
        ...
```

Funkcije, ki ne vračajo vrednosti, so podatkovnega tipa **void**.

```
// void funkcija

#include <iostream.h>

void izpis (void)  // podatkovni tip void
{
        cout << "Izpis iz funkcije!";
}


int main ()
{
        izpis ();
        return 0;
}
// izpiše
Izpis iz funkcije!
```

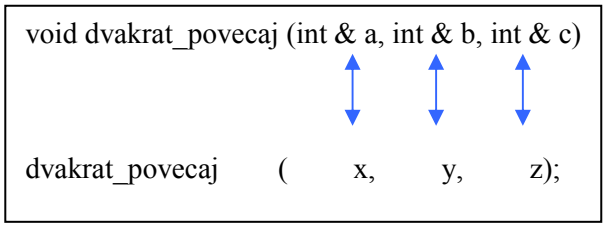
Čeprav ni obvezna uporaba rezerviranke void, če funkcija nima argumentov, pa jo vseeno uporabimo in s tem določimo, da ni argumentov.
Klic funkcije je sestavljen iz imena funkcije (identifikatorja) in med okroglimi oklepaji zapisanim seznamom parametrov.

Prenos parametrov po vrednosti in po referenci

Do sedaj smo prenašali parametre v funkcije po vrednosti (by value). Ob klicu funkcije smo prenesli vrednosti parametrov v argumente funkcije.
Zelo pogosto pa moramo prenesti več vrednosti iz funkcije, ki jo kličemo. Vemo že, da preko vrednosti funkcije lahko prenesemo le eno vrednost. V takšnih primerih uporabimo prenos parametrov po referenci (by reference).

Zgled:

```
// prenos parametrov po referenci
#include <iostream.h>

void dvakrat_povecaj (int & a, int & b, int & c)
{
          a*=2;  // novi vrednosti a-ja priredimo staro vrednost a-ja pomnoženo z 2
          b*=2; c*=2;
}

int main ()
{
        int x=1, y=3, z=7;
        //klic funkcije s prenosom po referenci
        dvakrat_povecaj (x, y, z);
        // izpis vrednosti spremenljivk
        cout << "x=" << x << ", y=" << y << ", z=" << z;
        return 0;
}
// izpiše
x=2, y=6, z=14
```



Argumenti funkcije dvakrat_povecaj so zapisani z uporabo znaka *ampersand* (**&**), ki določa, da je prenos spremenljivke po referenci. Pri prenosu spremenljivke po referenci posredujemo samo spremenljivko in vsaka sprememba spremenljivke parametra znotraj funkcije povzroči tudi spremembo spremenljivke, ki smo jo posredovali po referenci. Določili smo, da imamo povezane argumente (spremenljivke) a,b in c z parametri funkcije x,y in z. Sprememba vrednosti spremenljivke a znotraj funkcije vpliva tudi na spremembo spremenljivke x v glavnem programu. Vsaka sprememba spremenljivke b znotraj funkcije vpliva na spremembo spremenljivke y v glavnem programu, kar velja tudi za c in z.

Privzete vrednosti argumentov

Pri deklaraciji argumentov lahko določimo privzete vrednosti vsakega argumenta. Ta vrednost se uporabi, če ne določimo vrednosti parametra pri klicu funkcije.

Zgled:

```
// privzete vrednosti argumentov
#include <iostream.h>

int produkt (int  a, int b=1)
{
        int rezultat;
        rezultat = a * b;
        return (rezultat);
}

int main ()
{
        int x=1, y=3;
        cout << "Produkt=" <<produkt(x,y); // izpis vrednosti funkcije
        cout <<endl;
        cout << "Produkt=" <<produkt(x);    // izpis vrednosti funkcije
        return 0;
}

//izpiše
```

```
Produkt=3
Produkt=1
```

V glavnem programu imamo dva klica funkcije produkt. Pri drugem klicu smo določili vrednost le enega parametra, čeprav funkcija ima dva argumenta. Pri tem klicu dobi drugi argument privzeto vrednost (1).

Prekrivanje funkcij (overload function)

Dve različni funkciji lahko imata enako ime, če se prototipa funkcij razlikujeta v argumentih. Imata lahko različno število argumentov ali argumente različnih podatkovnih tipov.

Zgled:

```
// prekrivanje funkcij
#include <iostream.h>
int produkt (int  a, int b= 1)
{
        int rezultat;
        rezultat = a * b;
        return (rezultat);
}

float produkt (float  a, float b= 1.0)
{
        int rezultat;
        rezultat = a * b;
        return (rezultat);
}

int main ()
{
        int x=1, y=3;
        cout << "Produkt=" << produkt(x,y);
        cout << endl;
        float i = 2.0, j = 3.0;
        cout << "Produkt=" << produkt(i,j);
        return 0;
}

//izpiše
Produkt=3
Produkt=6
```

V zgledu imamo imamo dve funkciji z enakim imenom, ki se razlikujeta v podatkovnem tipu argumentov in podatkovnem tipu rezultata funkcije.

***inline* funkcije**

Rezerviranko inline postavimo na začetek deklaracije funkcije, če želimo, da se funkcija prevede v kodi na mestu klica funkcije (posledica je nekoliko hitrejše izvajanje programa).

Rekurzija

Rekurzija je oznaka za funkcije, ki v telesu funkcije vsebujejo klic svoje funkcije. Uporabimo jo lahko za probleme, kjer je rešitev problema določena s pomočjo rešitve manjšega problema. Uporabimo jo lahko pri urejanju, izračunu fakultete ipd.
Npr. matematična formula za izračun fakultete se glasi:
$n!$ = n * (n-1)! = ... = n * (n-1) * (n-2) * (n-3) ... * 1
konkretno za izračun fakultete števila 9

```
9! = 9* 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 362880
```

Zgled:

```
// izračun fakultete
#include <iostream.h>

long fakulteta (long a)
{
        if (a > 1)
                return (a * fakulteta (a-1));   // splošna formula
        else
                return (1);   // robni pogoj
}

int main ()
{
        long l;
        cout << "Vpiši celo število: ";
        cin >> l;
        cout << "!" << l << " = " << fakulteta (l);
        return 0;
}
//izvajanje
Vpiši celo število: 6
!6 = 720
```

Rekurzivni klic funkcije se izvede v primerih, če je vrednost argumenta večja od 1. Brez robnega pogoja bi imeli neskončno zanko, ki se bi najverjetneje zaključila s prekoračitvijo sklada (stack overflow).
Zavedati se moramo tudi omejitve podatkovnega tipa, saj bi vrednost argumenta večjega od 12 povzročila prekoračitev območja podatkovnega tipa long.

Prototipi funkcij

Do sedaj smo deklarirali funkcije pred glavno funkcijo main. Če zapišemo deklaracijo funkcije za funkcijo main, potem bi nam prevajalnik javil napako. Pisanju funkcij pred glavno funkcijo se izognemo z uporabo prototipov funkcij. Prototip funkcije je deklaracija glave funkcije, ki jo nato v celoti zapišemo v nadaljevanju programa.
Deklaracija prototipa je enaka deklaraciji glave funkcije.

Sintaksa:

podatkovni_tip ime_funkcije (argument1, argument2, ...);

Prototip funkcije ne vsebuje telesa funkcije in se zaključi s podpičjem.
V deklaracji argumentov je dovolj, da zapišemo podatkovni tip argumentov. Kljub temu (v praksi) zapišemo tudi ime argumenta, saj je s tem program razumljivejši.

Zgled:

```
// uporaba prototipov
#include <iostream.h>

void liho (int a);
void sodo (int a);

int main ()
{
        int i;
        do {
```

```
                cout << "Vpiši število: (0 za izhod) ";
                cin >> i;
                liho (i);
        } while (i!=0);
        return 0;
}

void liho (int a)
{
        if ((a%2)!=0)
                cout << "Število je liho.\n";
        else
                sodo (a);
}

void sodo (int a)
{
        if ((a%2)==0)
                cout << "Število je sodo.\n";
        else
                liho (a);
}

//izvajanje
Vpiši število: (0 za izhod) 25
Število je liho.
Vpiši število: (0 za izhod) 6
Število je sodo.
Vpiši število: (0 za izhod) 3
Število je liho.
Vpiši število: (0 za izhod) 0
Število je sodo.
```

Na začetku programa imamo deklaraciji prototipov funkcij **liho** in **sodo**.

```
void liho (int a);
void sodo (int a);
```

Prototipa dovoljujeta klic teh funkcij pred celotno deklaracijo funkcije. V zgledu kličemo funkcijo liho v glavi funkciji. V našem primeru bi morali vsaj za eno funkcijo zapisati prototip funkcije, saj v nadaljevanju funkcija liho kliče funkcijo sodo, le ta pa funkcijo liho.
Priporočila so, da za vse funkcije zapišemo deklaracijo prototipov, kar nam olajša tudi izdelavo *.h datotek. Deklaracija prototipov je pregleden seznam vseh funkcij, ki obstajajo v programu.

Predloge za funkcije

Predloge (šablone) za funkcije uporabljamo takrat, kadar želimo izvesti nad različnimi podatki identične operacije. Predloge nam omogočajo, da zapišemo implementacijo funkcije samo enkrat, kopije z ustreznimi podatkovnimi tipi pa napravi prevajalnik v času prevajanja.

Zgled:

```
//uporaba predlog za funkcije

#include <iostream.h>

    // deklaracija predloge
    template <class T>
        T najvecje(T vr1, T vr2, T vr3)
```

```
    {
        // vrnemo največjo vrednost
        T najvecje = vr1;
        if (vr2 > najvecje) najvecje = vr2;
        if (vr3 > najvecje) najvecje = vr3;
        return najvecje;
    }


main()
{
    int i1,i2,i3; //tri cela števila
    cout << "Vpisi tri cela stevila: ";
    cin >> i1 >> i2 >> i3;
    float f1,f2,f3; //tri realna števila
    cout << "Vpisi tri realna stevila: ";
    cin >> f1 >> f2 >> f3;
    char c1,c2,c3; // trije znaki
    cout << "Vpisi tri znake: ";
    cin >> c1 >> c2 >> c3;
    cout << endl << "Najvecje celo stevilo: ";
    cout << najvecje(i1,i2,i3) << endl;
    cout << "Najvecje realno stevilo: ";
    cout << najvecje(f1,f2,f3) << endl;
    cout << "Največji znak: ";
    cout << najvecje(c1,c2,c3) << endl;
return 0;
}

// izpis
Vpisi tri cela stevila: 5 15 6
Vpisi tri realna stevila: 15.5 20.4  6.
Vpisi tri znake: A r T
Najvecje celo stevilo: 15
Najvecje realno stevilo: 20.4
Največji znak: r
```

ASCII kodna tabela

Osnovna ASCII tabela določa kode za 128 znakov od 0 do 127. Prvih 32 znakov predstavljajo kontrolne kode, preostalih 96 pa so kode znakov.

*	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Tabela je zapisana v šestnajstiški obliki in sicer se znak **A** nahaja v oznaki vrstice 4 in oznaki stolpca 1 in predstavlja šestnajstiško vrednost **0x41** (desetiško **65**).

Poleg osnovne obstaja tudi razširjena ASCII tabela, ki definira še ostale nestandardne znake in tudi šumnike.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	□	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	□	□
9	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	□	Ÿ
A		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
B	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Knjižnice

Standardne knjižnice vsebujejo funkcije, konstante, spremenljivke, ki so nam na volju v programskem jeziku C++. Po posameznih področjih so shranjenje v različnih zaglavjih: stdio.h, stdlib.h, string.h, time.h, math.h ...

Matematične funkcije (math.h)

Matematične funkcije se nahajajo v math.h. Najpogostejše uporabljene funkcije so:

sqrt(x)	$\sqrt{x}$
exp(x)	e^x
log(x)	$\ln x$
log10(x)	$\log x$
fabs(x)	$\lvert x \rvert$
ceil(x)	zaokroži navzgor
floor(x)	zaokroži navzdol

```
#include <iostream.h>
#include <math.h>

int main ()
{
    cout <<"\nfloor od 2.3 je " << floor (2.3) ;
    cout <<"\nfloor od 3.8 je " << floor (3.8) ;
    cout <<"\nfloor od -2.3  je " << floor (-2.3) ;
    cout <<"\nfloor od -3.8  je " << floor (-3.8) ;
  return 0;
}

/* Izvajanje:
floor od 2.3 je 2
floor od 3.8 je 3
floor od -2.3  je -3
floor od -3.8  je -4
*/
```

pow(x,y)	x^y
fmod(x,y)	ostanek pri deljenju x z y

sin(x), asin(x), sinh(x), cos(x), acos(x), cosh(x), tan(x), atan(x), tanh(x) Pozor: x v radianih

Naključne vrednosti

Knjižnica: stdlib.h

int rand() – vrne naključno vrednost med 0 in RAND_MAX
void srand(unsigned int seed) - inicializacija pseudo generatorja naključnih števil (semena)

Zgled:

```
#include <iostream.h>
// standardna knjižnica (srand in rand)
#include <stdlib.h>
// knjižnica za delo s časom
#include <time.h>

int main ()
{
    int nakljucno_st, nakljucno_st1;
    /* inicializacija začetne vrednosti generatorja - semena */
    srand ( time(NULL) );

    /* naključno število med 0 in 9 */
    nakljucno_st = rand()%10 ;
    /* naključno število med 20 in 29 */
    nakljucno_st1 = rand()%10+20;
    // izpis
    cout << "Nakljucno število med 0 in 9 je: " << nakljucno_st;
    cout << "\nNakljucno število med 20 in 29 je: " << nakljucno_st1;
  return 0;
}
/*
Nakljucno število med 0 in 9 je: 6
Nakljucno število med 20 in 29 je: 27
*/
```

Nizi znakov

Nizi znakov so eden izmed najbolj pogosto uporabljenih sestavljenih tipov. Zapisujemo jih lahko kot polje znakov ali kot kazalec na znake.

Primer zapisa s poljem:

```
char niz[] = "test"
```

Ekvivalenten zapis je:

```
niz[0] = 't';  niz[1] = 'e'; niz[2] = 's';  niz[3] = 't'; niz[4] = '\0';
```

ali:

```
char niz[] = {'t', 'e', 's', 't', '\0'}
```

uporaba pri vhodnem toku:

```
char niz1[20];
cin >> niz1;
```

in izhodnem toku:

```
cout << niz1;
```

Primer izpisa niza znakov s presledki med znaki:

```
for (int i=0; niz[i]!='\0'; i++)
```

```
        cout << niz[i] << ' ';
```

Podobno lahko pri definiciji namesto polja znakov:

char niz[] = "Lep dan"; uporabimo char * niz = "Lep dan"

Obe definiciji rezervirata toliko znakov, kot je število znakov niza in en dodaten ničelni znak. Razlika med definicijama je, da prvi predstavlja identifikator niz naslov prvega elementa polja, v drugi pa je identifikator niz kazalec, ki vsebuje naslov začetka rezerviranega polja znakov.

Primer izpisa niza po posameznih znakih:

```
#include <iostream.h>

void izpisi_posamezne_znake(const char *);

int main()
{
    char niz[] = "izpis posameznih znakov niza.";
    cout << "Niz je:" << endl;
    izpisi_posamezne_znake(niz);
    cout << endl;
    return 0;
}

/* V funkciji izpisi_posamezne_znake je kaz_niz kazalec na nespremenljiv znak.
Znakov se ne da spremeniti preko kaz_niz (kaz_niz je torej kazalec s pravico le do
branja). */
void izpisi_posamezne_znake(const char *kaz_niz)
{
    for ( ; *kaz_niz != '\0'; kaz_niz++)
        cout << *kaz_niz;
}
/*    Program izpiše:
Niz je:
Izpis posameznih znakov niza.
*/
```

V knjižnici "**stdlib**.**h**" so definirane standardne funkcije za pretvorbo. Poglejmo si delovanje nekaterih:

- **double atof** (const char *str) - pretvori niz v število podatkovnega tipa double
  double d = atof("9.80");
- **int atoi** (const char *str) - pretvori niz v število podatkovnega tipa int
  int i = atoi("259");
- **int atol** (const char *str) - pretvori niz v število podatkovnega tipa long (celo število večjega obsega)
  long daljse_celo_st = atol("405000");
- **char *itoa**(int vrednost, char *string, int radix) – pretvori celo število v niz znakov, pri čemer je radix številski sistem in je lahko med 2 in 32.

V knjižnici "**string.h**" so definirane standardne funkcije za delo z nizi. Poglejmo si delovanje nekaterih:

- **void *memcpy** (vpid * dest, const void *src, int c, size_t n) – kopira n zlogov iz src v dest;
  char str1[]="Primer niza";
  char str2[40];
  char str3[40];
  memcpy (str2,str1,strlen(str1)+1);
- **const void *memchr** (const void *s, int c, size_t n) – v prvih n zlogih niza s išče znak c;
  Primer:
  char *niz = "To je niz";
  cout << "Ostanek niza po najdenem znaku 'i' je \""
  << (char *) memchr(niz, 'i', 9) << '\"' << endl;

```
// izpiše "iz"
```

- **int memcmp**(const void *s1, const void *s2, size_t n) – primerja niza med seboj v dolžini n zlogov in vrne: 0 – sta enaka; vrednost > 0 (1) – niz s1 je večji; vrednost < 0 (-1) – niz s2 je večji.
  Primer:

```
…
char n1[] = "ABCDEFG", n2[] = "ABCDXYZ";
cout << "n1 = " << n1 << "\nn2 = " << n2 << endl
        << "\nmemcmp(n1, n2, 4) = " << setw(3) << memcmp(n1, n2, 4)
        << "\nmemcmp(n1, n2, 7) = " << setw(3) << memcmp(n1, n2, 7)
         << "\nmemcmp(n2, n1, 7) = " << setw(3) << memcmp(n2, n1, 7)
        << endl;
…
/*
Izpiše:
n1 = ABCDEFG
n2 = ABCDXYZ

memcmp(n1, n2, 4) =   0  //enaka
memcmp(n1, n2, 7) =  -1 // n2 (drugi) večji
memcmp(n2, n1, 7) =   1 // n2  (prvi) večji
*/
```

- **void *memset**(const void *s, int c, size_t n) – postavi prvih n zlogov niza s na vrednost c.

```
…
char niz1[15] = "BBBBBBBBBBBBB";
cout << "niz1 = " << niz1 << endl;
cout << "niz1 po uporabi memset = "
        << (char *) memset(niz1, 'b', 7) << endl;
…
/* Izpiše;
niz1 = BBBBBBBBBBBBB
niz1 po uporabi memset = bbbbbbbBBBBBB
*/
```

- **char * strcat**(char *dest, const char *src) – doda niz src na konec niza dest;

```
…
char n1[20] = "Srecno ";
char n2[] = "novo leto ";
char n3[40] = "";
cout << "n1 = " << n1 << endl << "n2 = " << n2 << endl;
cout << "strcat(n1, n2) = " << strcat(n1, n2) << endl;
cout << "strncat(n3, n1, 7) = " << strncat(n3, n1, 7) << endl;
cout << "strcat(n3, n1) = " << strcat(n3, n1) << endl;
…
/* Izpis
n1 = Srecno
n2 = novo leto
strcat(n1, n2) = Srecno novo leto
strncat(n3, n1, 7) = Srecno
strcat(n3, n1) = Srecno Srecno novo leto
*/
```

- **char * strchr** (const char *s; int c) – v nizu s išče prvo pojavitev znaka c;

```
…
char *niz = "To je preizkus";
char znak1 = 'a', znak2 = 'z';
if (strchr(niz, znak1) != NULL)
        cout << '\'' << znak1 << "' je bil najden v \""
                << niz << "\"." << endl;
else
        cout << '\'' << znak1 << "' ni bil najden v \""
                << niz << "\"." << endl;
```

```
if (strchr(niz, znak2) != NULL)
        cout << '\'' << znak2 << "' je bil najden v \""
                << niz << "\"." << endl;
else
        cout << '\'' << znak2 << "' ni bil najden v \""
                << niz << "\"." << endl;
…
```

- **int strcmp** (const char *s1, const char *s2) – primerja niza med seboj;

```
…
char *n1 = "Srecno novo leto";
char *n2 = "Srecno novo leto";
char *n3 = "Srecno potovanje";

cout << "n1 = " << n1 << endl << "n2 = " << n2 << endl
        << "n3 = " << n3 << endl << endl << "strcmp(n1, n2) = "
        << setw(2) << strcmp(n1, n2) << endl << "strcmp(n1, n3) = "
        << setw(2) << strcmp(n1, n3) << endl << "strcmp(n3, n1) = "
        << setw(2) << strcmp(n3, n1) << endl << endl;

cout << "strncmp(n1, n3, 7) = " << setw(2) << strncmp(n1, n3, 7)
        << endl << "strncmp(n1, n3, 8) = " << setw(2)
        << strncmp(n1, n3, 8) << endl << "strncmp(n3, n1, 8) = "
        << setw(2) << strncmp(n3, n1, 8) << endl;
/* Izpiše:
n1 = Srecno novo leto
n2 = Srecno novo leto
n3 = Srecno potovanje

strcmp(n1, n2) =  0 // sta enaka
strcmp(n1, n3) = -1 // < 0 – n1 (s1) je manjši od n3 (s2)
strcmp(n3, n1) =  1 // > 0 – n3 (s1) je večji od n1 (s2)

strncmp(n1, n3, 7) =  0
strncmp(n1, n3, 8) = -1
strncmp(n3, n1, 8) =  1
*/
```

- **size_t strlen** (const char *s) – vrne dolžino niza

```
…
char *niz1 = "abcdefghijklmnopqrstuvwxyz123";
cout << "Dolzina niza \"" << niz1
        << "\" je " << strlen(niz1) << endl;
/* Izpiše
Dolzina niza "abcdefghijklmnopqrstuvwxyz" je 29
*/
```

- **char *strlwr** (char *s) – pretvori niz s v male črke;
- **char *strupr** (char *s) – pretvori niz s v velike črke;

Funkcije nad znaki

```
/* Uporaba funkcij isdigit (preveri, ce je stevka), isalpha (preveri, ce je
crka), isalnum (preveri, ce je crka ali  stevka), in isxdigit (preveri, ce  je
sestnajstiska stevka)
*/


#include <iostream.h>
#include <ctype.h>
```

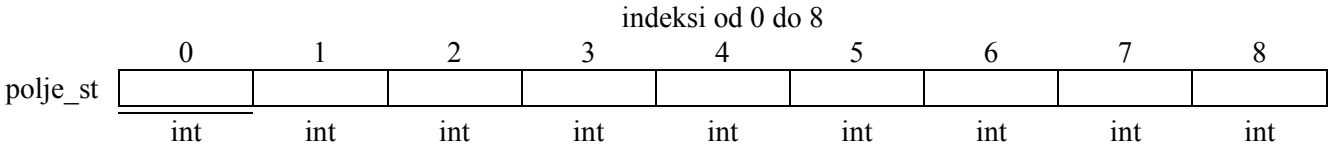
```
int main()
{
    cout << "Glede na isdigit:" << endl
         << (isdigit('8') ? "8 je" : "8 ni") << " stevka" << endl
         << (isdigit('#') ? "# je" : "# ni") << " stevka" << endl;

    cout << endl << "Glede na isalpha:" << endl
         << (isalpha('A') ? "A je" : "A ni") << " crka" << endl
         << (isalpha('b') ? "b je" : "b ni") << " crka" << endl
         << (isalpha('&') ? "& je" : "& ni") << " crka" << endl
         << (isalpha('4') ? "4 je" : "4 ni") << " crka" << endl;

    cout << endl << "Glede na isalnum:" << endl
         << (isalnum('A') ? "A je" : "A ni")
         << " stevka ali crka" << endl
         << (isalnum('8') ? "8 je" : "8 ni")
         << " stevka ali crka" << endl
         << (isalnum('#') ? "# je" : "# ni")
         << " stevka ali crka" << endl;

    cout << endl << "Glede na isxdigit:" << endl
         << (isxdigit('F') ? "F je" : "F ni")
         << " sestnajstiska stevka" << endl
         << (isxdigit('J') ? "J je" : "J ni")
         << " sestnajstiska stevka" << endl
         << (isxdigit('7') ? "7 je" : "7 ni")
         << " sestnajstiska stevka" << endl
         << (isxdigit('$') ? "$ je" : "$ ni")
         << " sestnajstiska stevka" << endl
         << (isxdigit('f') ? "f je" : "f ni")
         << " sestnajstiska stevka" << endl;

    return 0;
}

/*Izpis:
Glede na isdigit:
8 je stevka
# ni stevka

Glede na isalpha:
A je crka
b je crka
& ni crka
4 ni crka

Glede na isalnum:
A je stevka ali crka
8 je stevka ali crka
# ni stevka ali crka

Glede na isxdigit:
F je sestnajstiska stevka
J ni sestnajstiska stevka
7 je sestnajstiska stevka
$ ni sestnajstiska stevka
f je sestnajstiska stevka
*/


/* Uporaba funkcij isspace (preveri, ce je neviden znak), iscntrl (preveri, ce je
krmilni znak), ispunct (preveri, ce je locilo), isprint (preveri, ce je izpisljiv
znak) in isgraph (preveri, ce je izpisljiv znak, ki ni presledek)
*/

#include <iostream.h>
```

```
#include <ctype.h>

int main()
{
    cout << "Glede na isspace:" << endl
         << "Nova vrstica " << (isspace('\n') ? "je" : "ni")
         << " neviden znak" << endl
         << "Vodoraven tabulator "
         << (isspace('\t') ? "je" : "ni")
         << " neviden znak" << endl
         << (isspace('%') ? "% je" : "% ni")
         << " neviden znak" << endl;

    cout << endl << "Glede na iscntrl:" << endl
         << "Nova vrstica " << (iscntrl('\n') ? "je" : "ni")
         << " krmilni znak" << endl
         << (iscntrl('$') ? "$ je" : "$ ni")
         << " krmilni znak" << endl;

    cout << endl << "Glede na ispunct:" << endl
         << (ispunct(';') ? "; je" : "; ni")
         << " locilo" << endl
         << (ispunct('Y') ? "Y je" : "Y ni")
         << " locilo" << endl
         << (ispunct('#') ? "# je" : "# ni")
         << " locilo" << endl;

    cout << endl << "Glede na isprint:" << endl
         << (isprint('$') ? "$ je" : "$ ni")
         << " izpisljiv znak" << endl
         << "Znak za zvonec " << (isprint('\a') ? "je" : "ni")
         << " izpisljiv znak" << endl;

    cout << endl << "Glede na isgraph:" << endl
         << (isgraph('Q') ? "Q je" : "Q ni")
         << " izpisljiv znak, ki ni presledek" << endl
         << "Presledek " << (isgraph(' ') ? "je" : "ni")
         << " izpisljiv znak, ki ni presledek" << endl;
    return 0; }

    /* Izpis:
    Glede na isspace:
    Nova vrstica je neviden znak
    Vodoraven tabulator je neviden znak
    % ni neviden znak

    Glede na iscntrl:
    Nova vrstica je krmilni znak
    $ ni krmilni znak

    Glede na ispunct:
    ; je locilo
    Y ni locilo
    # je locilo

    Glede na isprint:
    $ je izpisljiv znak
    Znak za zvonec ni izpisljiv znak

    Glede na isgraph:
    Q je izpisljiv znak, ki ni presledek
    Presledek ni izpisljiv znak, ki ni presledek
    */
```

```
// Uporaba funkcije strtok (razdeli niz na besede, omejene z omejilnimi znaki)

#include <iostream.h>
#include <string.h>

int main()
{
    char niz[] = "To je stavek s 6 besedami";
    char *kzl_beseda;

    cout << "Niz za razclenitev je:" << endl << niz
         << endl << endl << "Besede so:" << endl;

    kzl_beseda = strtok(niz, " ");

    while (kzl_beseda != NULL)
    {
        cout << kzl_beseda << endl;
        kzl_beseda = strtok(NULL, " ");
    }

    return 0;
}

/*Izpiše:
Niz za razclenitev je:
To je stavek s 6 besedami

Besede so:
To
je
stavek
s
6
besedami
*/
```

Tabele oz. polja (Arrays)

Tabele oz. polja so zaporedja elementov oz. spremenljivk **enakega podatkovnega tipa**, ki so shranjene na sosednjih pomnilniških lokacijah. Do posameznega elementa tabele dostopamo tako, da za ime tabelarične spremenljivke dodamo indeks med oglatima oklepajema.
V deklaraciji spremenljivk lahko rezerviramo npr. devet spremenljivk celoštevilčnega podatkovnega tipa npr. int, ki bodo shranjena na sosednjih pomnilniških lokacijah (zvezno).



Element predstavlja mesto, kjer je shranjena vrednost posameznega elementa tabele (v našem primeru cela števila podatkovnega tipa int). Indeksi elementov tabele so od 0 do 8 (prvi indeks tabele je vedno 0 ne glede na dolžino tabele).
Tako kot pri spremenljivkah, moramo tudi tabelo deklarirati pred njeno uporabo.
Sintaksa deklaracije:

```
podatkovni_tip ime [st_elementov];
```

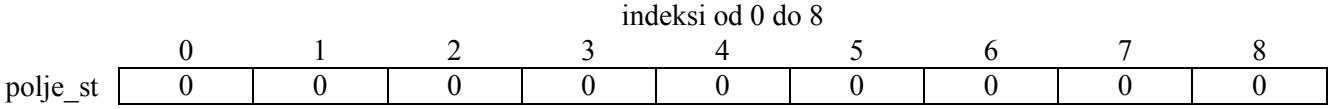
Podatkovni_tip je objekt nekega podatkovnega tipa (int, float...), ime je veljaven identifikator in st_elementov predstavlja število elementov tabele. St_elementov med oglatima oklepajema je konstanta, ki mora imeti določeno vrednost, saj prevajalnik v nasprotnem ne more določiti količino potrebnega pomnilnika za elemente tabele.

```
int  tab_st [12];       // tabela  z dvanajstimi elementi
```

Inicializacija tabele

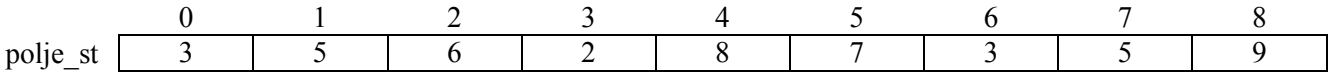
Če deklariramo tabelo kot lokalno znotraj funkcije in ne inicializiramo vrednosti tabele, potem tabela vsebuje nedoločene vrednosti, dokler ne shranimo v njo določenih vrednosti.
Če deklariramo tabelo kot **globalno** (zunaj funkcije) in ne določimo vrednosti elementov, potem se **inicializira** z vrednostmi **0**.
Npr. deklaracija

```
int polje_st[9];// globalna deklaracija => postavi vrednosti elementov na 0:
```



Ob deklaraciji tabele lahko določimo vrednosti elementov tabele (inicializacija) z uporabo zavitih oklepajev, kjer zapišemo seznam vrednosti elementov.
Npr:

```
int polje_st [ 9]  = {3, 5, 6, 2, 8, 7, 3, 5, 9}
```



Število elementov v seznamu vrednosti mora biti enako število elementov tabele (st_elementov).
C++ dopušča, da ob deklaraciji tabele izpustimo število elementov znotraj [ ] in nato se število elementov tabele določi glede na število elementov znotraj { }.
Npr:

```
int polje_st[ ] = {3, 5, 6, 2, 8, 7, 3, 5, 9}
```

Dostop do elementov tabele

Do vrednosti posameznega elementa tabele dostopamo:

```
ime_tabelaricne_spremenljivke[indeks]
```

Zgled:

	polje_st [0]	polje_st [1]	polje_st [2]	polje_st [3]	polje_st [4]	polje_st [5]	polje_st [6]	polje_st [7]	polje_st [8]
polje_st	3	5	6	2	8	7	3	5	9

Če želimo shraniti vrednost 56 v četrti element tabele polje_st zapišemo stavek:

```
polje_st [3] = 56;
```

Vrednost tretjega elementa tabele shranimo v spremenljivko tretji s stavkom:

```
tretji = polje_st [2];
```

```
int polje_st [5];   // deklaracija tabele - šest elementov podatkovnega tipa int
polje_st [2] =  66; // tretjemu elementu tabele določimo vrednost 66
```

Veljavni so tudi naslednji zapisi:

```
polje_st [0] = a;      // v prvi element shranimo vrednost spremenljivke a
polje_st [a] = 15;     // a-temu elementu tabele določimo vrednost 15
b = polje_st  [a+2]; // v spremenljivko b hranimo vrednost a + 2 elementa tabele
polje_st [polje_st [a]] = polje_st [2] + 5;
```

Večdimenzionalne tabele

Večdimenzionalne tabele si lahko predstavljamo kot tabelo, ki ima za elemente drugo tabelo. Za dvodimenzionalno tabelo si predstavljamo indekse tabele v obliki vrstic in stolpcev.

tab_st

vrstice \ stolpci	0	1	2	3	4	5	6	7	8
0									
1									
2									
3									

Tab_st predstavlja dvodimenzionalno tabelo [4][9], ki ima štiri vrstice in devet stolpcev.

tab_st

vrstice \ stolpci	0	1	2	3	4	5	6	7	8
0									
1						■			
2									
3									

Do označenega elementa tabele tab_st dostopamo:

```
izbran = tab_st [1] [5];
```

Večdimenzionalne tabele niso omejene s številom elementov in lahko vsebujejo poljubno število dimenzij (indeksov).
Npr. deklaracija

```
char stoletje [100][365][24][60][60];
```

določa en znak za vsako sekundo v stoletju.

Večdimenzionalne tabele so abstraktna predstavitev. Predstavimo jih lahko z enodimenzionalnimi tabelami. Npr. namesto dekaracije

```
int tab_st [4] [9]; // lahko uporabimo (4 * 9 = 36)
int tab_st [36];
```

V obeh primerih imamo 36 elementov tabele, v katere lahko shranimo vrednosti podatkovnega tipa int.

Prenos tabel preko parametrov funkcije

Večkrat moramo posredovati podprogramu vrednosti elementov tabele. **C++ ne dovoljuje posredovanja vrednosti tabele v parametrih funkcije**. Posredujemo pa lahko pomnilniški naslov (address), na katerem je shranjena tabela. Takšen način je veliko hitrejši in učinkovitejši.

Da omogočimo tabele kot parametre funkcij, moramo deklarirati argument funkcije, ki je tabela z nedoločenim številom elementov ([]).
Zgled:

```
// deklaracija funkcije
void ime_funkcije (int arg[])
…
//deklaracija tabele, ki posredujemo kot parameter
int tabela_stevil [40];
…
// klic funkcije s parametrom tabele
ime_funkcije (tabela_stevil);
```

Celoten zgled:

```
// tabele kot parametri funkcij
#include <iostream.h>

    // funkcija za izpis elementov celoštevlčne tabele
    void izpis_tabele (int tab[], int dolzina_tabele)
    {
        for (int n=0; n<dolzina_tabele; n++)//zanka za vse elemente tabele
            cout << tab[n] << "  ";//izpis posameznega elementa tabele in
presledka
        cout << "\n";                 // prehod v novo vrstico
        // nesmiselno - samo za test prenosa
        tab[0] = 123;       // v prvi element tabele shranimo vrednost 123
    }

    int main ()
    {
        int tabela_ena[] = {5, 10, 15};// deklaracija in inicializacija prve
tabele
        int tabela_dva[] = {2, 4, 6, 8, 10}; // deklaracija in inicializacija
druge tabele
        izpis_tabele (tabela_ena,3); //klic funkcije za izpis prve tabele
        izpis_tabele  (tabela_ena,3);//ponovni klic funkcije za izpis prve
tabele
        izpis_tabele (tabela_dva,5); //klic funkcije za izpis druge tabele
        return 0;
    }

    // izpis programa
    5 10 15
    123 10 15
    2 4 6 8 10
```

Kazalci in tabele

Koncept tabel je zelo povezan s konceptom kazalcev. Tabelarična spremenljivka ima vrednost naslova prvega elementa v tabeli. Kazalec pa ima naslov prvega elementa na katerega kaže. Npr. za deklaracijo

```
int stevila [10];
int * p;
```

je veljaven naslednji stavek:

```
p = stevila;
```

Na tem mestu sta enaka `p` in `stevila` z eno razliko, da lahko priredimo kazalcu p drugo vrednost, medtem ko bo tabelarična spremenljivka stevila vedno kazala na prvi element tabele števil. Rečemo lahko, da je tabelarična spremenljivka konstantni kazalec na začetek tabele števil, ki ji ne moremo prirediti druge vrednosti.

Zgled:

```
// kazalci in tabele
#include <iostream.h>

int main ()
{
      int stevila[8];
      int * kaz_st;
      //kazalec kaže na začetek tabele števil, prvemu elementu tabele dodelimo
vrednost 10
       kaz_st = stevila;  *kaz_st = 10 ;
      //kazalec kaže na drugi element tabele, drugemu elementu tabele dodelimo
vrednost 20
      kaz_st++;  *kaz_st = 20;
      //kazalec kaže na tretji element tabele, tretjemu elementu tabele dodelimo
vrednost 30
       kaz_st = &stevila[2];  *kaz_st = 30;
      //kazalec kaže na četrti element tabele, četrtemu elementu tabele dodelimo
vrednost 40
       kaz_st = stevila + 3;  *kaz_st = 40;
      //kazalec kaže na prvi element tabele, petemu elementu tabele dodelimo
vrednost 50
       kaz_st = stevila;  *(kaz_st+4) = 50;
      // izpišemo prvih pet elementov tabele
       for (int n=0; n<5; n++)
         cout << stevila[n] << ", ";
      // izpiše 10, 20, 30, 40, 50,
       return 0;
}
```

Iz zgleda lahko ugotovimo, da sta naslednja izraza enakovredna:

a[5] = 0;
*(a+5) = 0;

ne glede na to ali je spremenljivka a tabela ali kazalec.

Inicializacija kazalca

Zgled:

```
int stevilo;
int *kaz_st = & stevilo;
```

je enako kot:

```
int stevilo;
int *kaz_st;
kaz_st = & stevilo;
```

Če je v prireditvenem stavku na levi strani kazalec, potem dobi vedno vrednost kamor kaže (ne pa vrednosti, ki je na pomnilniški lokaciji na katero kaže). Znak * v deklaraciji določa le to, da je to kazalec (in ne referenčni operator).
Torej znak * predstavlja dva različna operatorja glede na to, na katerem mestu je zapisan.

Inicializacija kazalca na zaporedje znakov:

```
char * kaz_znake = "znaki";
```

rezervira statični pomnilnik za "znaki" in kazalec kaže na prvi znak tega zaporedja torej 'z'. Če je to zaporedje shranjeno od pomnilniškega naslova 2000 je pomnilniška slika naslednja:

...	2000	2001	2002	2003	2004	2005	...
...	'z'	'n'	'a'	'k'	'i'	'\0'	...

...	kaz_znake	...
...	2000	...

Kazalec kaz_znake kaže na niz znakov in ga lahko uporabimo kot tabelarično spremenljivko.

Npr. tretji znak v nizu lahko spremenimo z

```
kaz_znake [2] = '_'; //ali
*( kaz_znake +2) = '_';
```

...	2000	2001	2002	2003	2004	2005	...
...	'z'	'n'	'_'	'k'	'i'	'\0'	...

...	kaz_znake	...
...	2000	...

Aritmetika s kazalci

Aritmetične operacije s kazalci se razlikujejo od teh operacij nad števili. Nad kazalci sta dovoljeni le operaciji seštevanje in odštevanje. Obe operacijo določita rezultat glede na podatkovni tip objekta na katerega kaže kazalec. Npr. char zavzame 1 zlog, short 2 zloga, long 4 zloge itd.
Deklaracija kazalcev:

```
char *kaz_znak;
short *kaz_st;
long *kaz_vel_st;
```

Predpostavimo, da kažejo na pomnilniške naslove **1000**, **2000** in **3000**.
Po stavkih

```
kaz_znak++;
kaz_st++;
kaz_vel_st++;
```

kaz_znak vsebuje vrednost 1001, kaz_st vsebuje vrednost 2002 in kaz_vel_st vsebuje vrednost 3004. Če prištejemo 1 kazalcu, potem ta kaže na naslednji objekt enakega podatkovnega tipa. Enako velja pri odštevanju.

Paziti moramo prioriteto, saj imata operatorja (++) in (--) višjo prioriteto kot operator (*).
Zgled:

```
*p++; // je enako *(p++) – poveča pomnilniški naslov na katerega
kaže
*p++ = *q++; // stavek je enak zaporedju treh stavkov:
*p = *q;
p++;
q++;
```

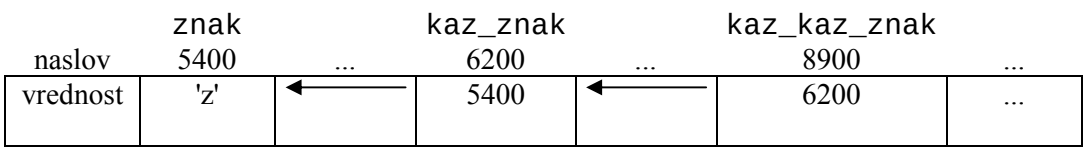
Priporočljiva je uporaba (), da ne pride do zmešnjav.

Kazalci na kazalce

C++ dovoljuje uporabo kazalcev na kazalce, ki nato kažejo na podatke. V tem primeru samo dodamo še en operator * za naslednji nivo kazalca.

```
char znak;
char * kaz_znak;
char ** kaz_kaz_znak;
znak = 'z';
kaz_znak = &znak;
kaz_kaz_znak = & kaz_znak;
```

Če so shranjene deklaracije spremenljivk na naslovih 5400, 6200 in 8900, potem je pomnilnilniška slika naslednja:



Kazalec void

Kazalec podatkovnega tipa void lahko kaže na podatek kateregakoli podatkovnega tipa. Njegova omejitev je v tem, da do podatkov ne moremo dostopati preko referenčnega operatorja *, saj njegova velikost ni določena. Možna uporaba je posredovanje splošnega parametra v funkcijo.

```
#include <iostream.h>

void povecaj (void* data, int type)
{
  switch (type)
  {
        case sizeof(char) : (*((char*)data))++; break;
        case sizeof(short): (*((short*)data))++; break;
        case sizeof(long) : (*((long*)data))++; break;
  }
}


int main ()
{
    char a = 5;
    short b = 9;
    long c = 12;
    povecaj (&a,sizeof(a));
    povecaj (&b,sizeof(b));
    povecaj (&c,sizeof(c));
    cout << (int) a << ", " << b << ", " << c;
    // izpiše 6, 10, 13
    return 0;
}
```

Operator sizeof

Operator sizeof vrne število zlogov, ki jih zaseda parameter. Npr. sizeof(char) je 1, ker je podatkovni tip char dolžine 1 zloga.

Kazalec na funkcijo

C++ dovoljuje uporabo kazalcev na funkcije.
To omogoča posredovanje funkcije kot parametra v drugo funkcijo (pri funkcijah ne moremo uporabiti operatorja &). Za deklaracijo kazalca na funkcijo moramo funkcijo deklarirati kot prototip funkcije, kjer v okroglih oklepajih zapišemo operator * in nato ime funkcije.

Zgled:

```
// kazalec na funkcijo
#include <iostream.h>

int sestej (int a, int b)
{ return (a+b); }

int odstej (int a, int b)
{ return (a-b); }
```

```
int (*minus)(int,int) = odstej;

int operacija (int x, int y, int (*klic_funkcije)(int,int))
{
  int g;
  g = (*klic_funkcije)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  m = operacija (7, 5, sestej);
  n = operacija (20, m, minus);
  cout <<n;
 // izpiše 8
  return 0;
}
```

V zgledu je **minus** kazalec na funkcijo, ki ima dva parametra podatovnega tipa **int.** Kazalecu priredimo, da kaže na funkcijo odstej (int (*minus)(int,int) = odstej;).

Urejanje podatkov

Pogosto se v praksi srečamo z nalogo, da moramo podatke urediti po predpisanem vrstem redu. Podatke lahko uredimo na več možnih načinov.

Urejanje po metodi izbora najmanjšega elementa:

- poiščemo najmanjše število v neurejeni tabeli,
- postavimo ga na prvo mesto neurejenega dela tabele, hkrati pa število, ki je bilo do tedaj na prvem mestu neurejenega dela tabele postavimo na mesto, na katerem smo našli najmanjše število v neurejeni tabeli,
- neurejeni del tabele se s tem zmanjša za eno število,
- to ponavljamo dokler v neurejenem delu tabele ne ostane samo eno število, ki je hkrati največje v urejeni tabeli.

Pomembno je, da si shranimo mesto, na katerem smo našli najmanjše število in tudi začetno mesto neurejene tabele.

Zgled:

Neurejene tabela tab ima 5 elementov:

indeks i->	0	1	2	3	4
tab[i]	23	4	12	34	5

Začetno mesto neurejene tabele ima indeks 0. Poiščemo najmanjši element od mesta neurejene tabele naprej. Ugotovimo, da je to število 4, ki je na indeksu 1. Zamenjamo števili na indeksih 0 in 1.

indeks i->	0	1	2	3	4
tab[i]	4	23	12	34	5

Začetno mesto neurejene tabele ima sedaj indeks 1. Poiščemo najmanjši element od mesta neurejene tabele naprej. To je število 5, ki je na indeksu 4 v tabeli. Zamenjamo števili na začetnem mestu neurejene tabele in mestu najmanjšega v neurejeni tabeli (indeksa 1 in 4).

indeks i->	0	1	2	3	4
tab[i]	4	5	12	34	23

Začetno mesto neurejene tabele ima indeks 2. Poiščemo najmanjši element od mesta neurejene tabele naprej. To je število 12, ki je na indeksu 2 v tabeli. Zamenjava ni potrebna, saj je najmanjše število v neurejeni tabeli že na prvem mestu neurejene tabele.

indeks i->	0	1	2	3	4
tab[i]	4	5	12	34	23

Začetno mesto neurejene tabele ima indeks 3. Poiščemo najmanjši element od mesta neurejene tabele naprej. To je število 23, ki je na indeksu 4 v tabeli. Zamenjamo števili na začetnem mestu neurejene tabele in mestu najmanjšega v neurejeni tabeli (indeksa 3 in 4).

indeks i->	0	1	2	3	4
tab[i]	4	5	12	23	34

V neurejenem delu tabele je ostalo samo eno število in s tem je tabela urejena po velikosti.

Podprogram:

```
// urejanje z izborom najmanjšega elementa
int  urediMin
        (float tab[], //vhod - izhod: enodimenzionalna tabela števil
        int n // vhod: število elementov
        )
{
    int i, j, IndMin;
    float Min;
    /*   i - mesto za prvi element v neurejeni tabeli;
  j - indeks za neurejeni del tabele,
  IndMin - mesto najmanjsega elemenenta (od neurejenega dela tabele) Min v tabeli
*/
    for (i=0; i<=n-2;i++) //se izvrši n-1 krat z indeksi od 0 so n-2
    {
        IndMin=i; // prvi indeks neurejenega dela tabele določimo za minimalni
        // vse vrednosti indeksov neurejene tabele razen najmanjšega
        for (j=i+1; j<=n-1;j++)
        {
            // je vrednost trenutnega manjša od vrednosti do sedaj najmanjšega v
            // neurejenem delu tabele
            if (tab[j] < tab[IndMin])
            //določimo novo vrednost indeksa najmanjšega elementa v neurejenem delu tabele
                IndMin=j;
        }
        if (IndMin != i )
        { //zamenjava elementov
                Min= tab[IndMin];
                tab[IndMin]=tab[i];
                tab[i]=Min;
        }
    }
    return (1);
}
```

Število primerjav, ki jih opravimo je: prvo pregledovanje: n-1, drugo = n-2,...in pri zadnjem 1. Skupaj je primerjav: n * (n-1) / 2.

Urejanje po metodi mehurčkov (Bubble sort)

- začnemo na začetku tabele, primerjamo i-ti in i+1-vi element tabele; če nista v predpisanem zaporedju, ju zamenjamo,
- v enem prehodu skozi tabelo pomaknemo največje število na konec tabele, manjša števila pa pomikamo proti začetku tabele,
- zgornja koraka ponavljamo dokler ni tabela urejena.

Zgled:

indeks i->	0	1	2	3	4
tab[i]	23	4	12	34	5

Primerjamo prvi in drugi element tabele in ju zamenjamo.

indeks i->	0	1	2	3	4
tab[i]	4	23	12	34	5

Primerjamo drugi in tretji element tabele in ju zamenjamo.

indeks i->	0	1	2	3	4
tab[i]	4	12	23	34	5

Primerjamo tretji in četrti element tabele; zamenjava ni potrebna.

indeks i->	0	1	2	3	4
tab[i]	4	12	23	34	5

Primerjamo predzadnji in zadnji element tabele in ju zamenjamo.

indeks i->	0	1	2	3	4
tab[i]	4	12	23	5	34

Končali smo prvi prehod, največje število 34 je na zadnjem mestu v tabeli. Na enak način nadaljujemo še s preostalimi prehodi in opravimo naslednje zamenjave.

indeks i->	0	1	2	3	4
tab[i]	4	12	5	23	34

indeks i->	0	1	2	3	4
tab[i]	4	5	12	23	34

indeks i->	0	1	2	3	4
tab[i]	4	5	12	23	34

Podprogram:

```
// mehurčno urejanje
int  urediMehurcno
        (float tab[], //vhod - izhod: enodimenzionalna tabela števil
        int n // vhod: število elementov
        )
{
    int i=0; //i - stevec prehodov skozi tabelo
    bool Urejena=false; // indikator zamenjave v posameznem prehodu
    float Pom; // pomozna spremenljivka
```

```
        // dokler tabela ni urejena; i - prehodi: 0..n-1
        while ((i < n) && (!Urejena))
        {
            // predpostavimo, da v trenutnem prehodu ni zamenjave
            Urejena= true;
            i++;
            for (int j=0; j<=n-2;j++) // n-1 krat ponovimo
            {
                // primerjava sosednjih elementov tabele
                if (tab[j] > tab[j+1])
                {
                    // zamenjava vrednosti v sosednih elementih
                    Pom= tab[j];
                    tab[j]=tab[j+1];
                    tab[j+1]=Pom;
                    // še ni urejena, saj smo opravili vsaj eno zamenjavo v tekočem prehodu
                    Urejena = false;
                }
            }
        }
        return (1);
    }
```

Program lahko izboljšamo, če upoštevamo, da je smiselno pregledovati tabelo v naslednjem prehodu le še do mesta, kjer smo v prejšnem prehodu napravili zadnjo zamenjavo. Od mesta zadnje zamenjave naprej so podatki že urejeni.

Binarno iskanje

Problem iskanja določenega elementa v urejenem zaporedju elementov.

Pogoj za binarno iskanje je, da so elementi tabele urejeni po velikosti. Urejeno tabelo razdelimo enakomerno na dve polovici. Iskani podatek je lahko v prvi ali v drugi polovici. Ustrezno polovico tabele izberemo s pomočjo primerjave iskanega podatka s sredinskim elementom tabele. Tako izločimo polovico tabele. Iskanje nadaljujemo na isti način v izbrani polovici. V dveh korakih izločimo tri četrtine podatkov. Postopek ponavljamo dokler ne najdemo števila, ki ga iščemo ali dokler ne ugotovimo, da iskanega števila ni v tabeli.

Zgled: Ugotovimo če je število 23 element tabele?

indeks i->	0	1	2	3	4
tab[i]	4	5	12	23	34
	Spodaj				Zgoraj

Določimo sredinski element:

Sred = (Spodaj + Zgoraj) / 2 = (0 + 4) / 2 = 2

Ker je iskano število večje od tab[2], postavimo spodnjo mejo na Sred + 1 = 3 in nadaljujemo iskanje v zgornji polovici tabele.

indeks i->	0	1	2	3	4
tab[i]	4	5	12	23	34
				Spodaj	Zgoraj

Sred = (Spodaj + Zgoraj) / 2 = (3+4) / 2 = 3

Po dveh primerjavah ugotovimo, da je število 23 element tabele.

Če bi iskali število 22, bi prišli do naslednje situacije:
Iskano število 22 je manjše od sredinskega števila 23 in zgornjo mejo postavimo na sredinsko število. Veljalo bi: Spodaj = Zgoraj = 3.
Torej pri neuspešnem iskanju velja Spodaj = Zgoraj = Sred in lahko pogoj Spodaj < Zgoraj uporabimo za ponavljanje iskanja.

Funkcijski podprogram:

```
// binarno iskanje števila v urejeni tabeli
bool iskanjeBinarno
        (float tab[], //vhod - izhod: enodimenzionalna tabela števil
        int n, // vhod: število elementov, indeksi od 0 .. n-1
        float iskano //vhod: število, ki ga iščemo v tabeli
        )
{
    // spodaj - spodnji indeks tabele,   zgoraj - zgornji indeks tabele
    // v katerem  iščemo iskano število
    int spodaj, zgoraj, sred;
    spodaj = 0;
    zgoraj = n-1;
    while (spodaj < zgoraj)
    {
        sred = (spodaj + zgoraj) / 2;
        if (iskano > tab[sred])
            spodaj = sred + 1;
        else
            zgoraj=sred;
    }
    return (iskano == tab[spodaj]) ;
}
```

Urejanje z vstavljanjem

Navadno vstavljanje

Vse elemente od drugega mesta do konca tabele vstavimo na ustrezno mesto v urejeni del tabele. Vedno primerjamo vrednost elementa, ki ga vstavljamo z vsemi predhodnimi elementi, dokler ne naletimo na prvega manjšega v urejeni tabeli. Pri tem vse večje elemente pomikamo za eno mesto v desno, da pridobimo prostor za vstavljanje na ustrezno mesto v urejeni del tabele. Z ustreznim vstavljanjem enega elementa se urejeni del tabele poveča za ena. Običajno si pomagamo s stražarjem (v tabelo pred prvi element vstavimo element, ki ga vstavljamo), da lažje zaključimo vstavljanje.

Zgled:

indeks i->	-1	0	1	2	3	4
tab[i]		23	4	12	34	5

Drugi element postavimo na mesto stražarja (tab[-1]). Primerjamo drugi element s prvim, ugotovimo da je manjši in zato pomaknemo drugi element za eno mesto v desno. Ker smo naleteli na stražarja, postavimo element, ki ga vstavljamo na prvo mesto (indeks 0).

indeks i->	-1	0	1	2	3	4
tab[i]	4	4	23	12	34	5

Vzamemo naslednji element (indeks 2) in ga postavimo na mesto stražarja. Ker je njegova vrednost manjša od predhodnika, pomaknemo predhodnika za eno mesto v desno. Z naslednjo primerjavo ugotovimo, da sodi vstavljani element na drugo mesto (indeks 1).

indeks i->	-1	0	1	2	3	4
tab[i]	12	4	12	23	34	5

Ta postopek nadaljujemo še s preostalimi elementi, ki jih moramo vstaviti na ustrezno mesto.

indeks i->	-1	0	1	2	3	4
tab[i]	34	4	12	23	34	5

indeks i->	-1	0	1	2	3	4
tab[i]	5	4	5	12	23	34

Podprogram:

```
// urejanje z vstavljanjem (brez stražarja)
int  urediZVstavljanjem
        (float tab[], //vhod - izhod: enodimenzionalna tabela števil
        int n // vhod: število elementov
        )
{
    // od drugega  do n-tega elementa
    for (int i = 1; i<=n-1;i++)
    {
        // shranimo vrednost itega elementa
        float itiElement=tab[i];
        int j;
        bool premik=false;
        for (j=i-1;tab[j]>itiElement && j>=0;j--)
        {
            tab[j+1]=tab[j]; // premikamo elemente v desno
            premik=true;
        }
        // če je bil  premik v desno
        if (premik)
        // vstavimo na ustrezno mesto itega
            tab[j+1]=itiElement;
        cout <<"Vrednost i:" <<i  <<endl;
    }
    return (1);
}
```

Binarno vstavljanje

Vse elemente od drugega mesta do konca tabele vstavimo na ustrezno mesto v urejeni del tabele. Elementi, ki se nahajajo v tabeli levo od elementa, ki ga vstavljamo, so urejeni. To nam omogoča uporabo binarnega iskanja mesta, na katerega bomo ta element vstavili. Ko ugotovimo mesto, premaknemo vse elemente urejenega dela tabele od mesta vstavljanja elementa do konca urejenega dela tabele za eno mesto v desno.

Zgled:

indeks i->	0	1	2	3	4
tab[i]	4	23	12	34	5

Za število 12 poiščemo mesto kamor ga moramo vstaviti. L=0 in D=1, Sred=(0+1) / 2 = 0,
12 je večje od tab[0] = 4 => L=Sred +1 = 1, Sred=(1+1) / 2 = 1, ker je 12 < Tab[Sred] je iskano mesto L = 1. To pomeni, da element 23 pomaknemo za eno mesto v desno, na njegovo mesto pa vstavimo element 12.

indeks i->	0	1	2	3	4
tab[i]	4	12	23	34	5

Za število 34 poiščemo mesto kamor ga moramo vstaviti. L=0 in D=2, Sred=(0+2) / 2 = 1,

34 je večje od tab[1] = 12 => L=Sred +1 = 2, Sred=(2+2) / 2 = 2, ker je 34 > Tab[Sred] je iskano mesto L = L + 1 = 3. To pomeni, da element 34 ostane na svojem mestu.

indeks i->	0	1	2	3	4
tab[i]	4	12	23	34	5

Za število 5 poiščemo mesto kamor ga moramo vstaviti. L=0 in D=4, Sred=(0+3) / 2 = 1,
5 je manjše od tab[1] = 12 => D=Sred = 1; novi Sred=(0+1) / 2 = 0, ker je 5 > Tab[Sred] je iskano mesto L = L + 1 = 1. Postopek se konča, ker je 5 < tab[1], ga vstavimo na drugo mesto.

indeks i->	0	1	2	3	4
tab[i]	4	5	12	23	34

Podprogram:

```
// urejanje z binarnim vstavljanjem
int  urediZBinarnimVstavljanjem
        (float tab[], //vhod - izhod: enodimenzionalna tabela števil
        int n // vhod: število elementov
        )
{
    // od drugega  do n-tega elementa
    for (int i = 1; i<=n-1;i++)
    {
        // shranimo vrednost itega elementa
        float itiElement=tab[i];
        // leva in desna meja področja iskanja (v urejenem delu)
        int l1=0,d1=i-1;
        while (l1 <= d1)
        {
            // indeks sredinskega
            int sred= (l1 + d1) / 2;
            // primerjamo itiElement z vrednostjo sredinskega
            if (itiElement < tab[sred])
                // nov desni indeks področja iskanja
                d1=sred - 1;
            else
                // nov levi indeks področja iskanja
                l1 = sred +1 ;
         }
        // prepis v desno
        for (int j=i-1; j >= l1; j--)
            tab[j+1] = tab [j];
        // postavimo itiElement na ustrezno prosto mesto
        tab[l1] = itiElement;
    }
    return (1);
}
```

Dinamični pomnilnik

Z deklaracijo spremenljivk v programu zasežemo statični pomnilnik (za izvajanje programa). Velikokrat želimo učinkovito izrabo pomnilnega prostora, da lahko med izvajanjem programa po potrebi zasegamo (dodeljujemo) in sproščamo pomnilnik. Rešitev je dinamični pomnilnik, katerega količino prilagajamo glede na aktivnosti uporabnika.

Operatorja new in new[]

Operator new uporabimo za dodeljevanje pomnilniškega prostora. Za njim zapišemo podatkovni tip in kot opcijo lahko navedemo tudi število elementov v oglatih oklepajih.
Vrne nam kazalec na začetek novo dodeljenega pomnilniškega prostora.

Sintaksa je:

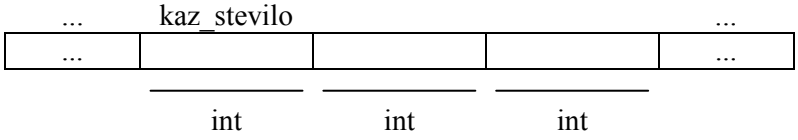
kazalec = new podatkovni_tip // kazalec na en element izbranega podatkovnega tipa

ali

//kazalec na več elementov izbranega podatkovnega tipa
kazalec = new podatkovni_tip [stevilo_elementov]

Zgled:

```
int * kaz_stevilo;              // kazalec na celo število
kaz_stevilo = new int [3];      // kazalec kaže na začetek treh celih števil
```



Z dinamičnim pomnilnikom upravlja operacijski sistem. Če operacijski sistem ne more dodeliti pomnilnika, potem vrne kazalec NULL. Priporočljivo je, da pred uporabo vedno preverimo, če je bil uspešno zasežen oz. dodeljen pomnilniški prostor.
Npr.

```
int * kaz_stevilo; // kazalec na celo število
kaz_stevilo = new int [3];
if (kaz_stevilo == NULL)
{
    // napaka ... sporočilo...
};
```

Operator delete

Ko ne potrebujemo več dinamičnega pomnilnika, ga sprostimo z uporabo operatorja delete.

Sintaksa:

delete kazalec;
ali
delete [] kazalec;

Delete kazalec uporabimo za sprostitev enega elementa. Drugo obliko pa za sproščanje pomnilnika za več elementov. Po izvršitvi sproščanja ima kazalec vrednost NULL.

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

```
int main ()
{
      char input [100];
      int i,n;
      long * l, total = 0;
      cout << "Koliko stevil zeliš vnesti? ";
      cin.getline (input,100); i=atoi (input);
      l= new long[i];
      if (l == NULL) exit (1);
      for (n=0; n<i; n++)
      {
        cout << "Vnesi stevilo: ";
        cin.getline (input,100);
        l[n]=atol (input);
      }
      cout << "Vnesel si: ";
      for (n=0; n<i; n++)
        cout << l[n] << ", ";
      delete [] l;
      return 0;
}
```

Konstanta NULL je že definirana v knjižnicah in ima obliko #define NULL 0.

Dinamični pomnilnik v ANSI-C

V programskem jeziku C obstajajo v knjižnici stdlib.h funkcije za delo z dinamičnim pomnilnikom, ki delujejo tudi v C++.

Funkcija malloc

omogoča dodelitev pomnilniškega prostora kazalcem. Deklaracija prototipa je:

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

kjer je nbytes število zlogov, ki jih želimo dodeliti. Funkcija vrne kazalec na podatkovni tip void (void*), ki ga običajno pretvorimo v kazalec določenega podatkovnega tipa.
Zgled:

```
char * kaz_znak;                          // kazalec ki kaže na znak
kaz_znak = (char *) malloc (12);   // kazalec na pomnilniški blok 12-ih zlogov
```

Ker pogosto vemo za koliko elementov določenega podatkovnega tipa želimo zaseči pomnilnik, pogosto uporabimo funkcijo sizeof. Npr.:

```
char * kaz_znak;                          // kazalec ki kaže na znak
kaz_znak = (char *) malloc (6 * sizeof(int)););   // kazalec na pomnilniški blok 12-ih zlogov
```

Funkcija calloc

Funkcija calloc je po pomenu zelo podobna funkciji malloc. Razlika je v deklaraciji prototipa:

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

Velikost zaseženega pomnilnika je produkt obeh parametrov. Prvi parameter je število elementov, drugi pa velikost enega elementa.
Zgled:

```
int * kaz_stevilo;
kaz_stevilo = (int *) calloc (5, sizeof(int));
```

Calloc se razlikuje od malloc tudi po tem, da inicializira vrednosti elementov na 0.

Funkcija realloc

Funkcija realloc spremeni velikost pomnilnika, ki smo ga že dodelili in na katerega kaže kazalec. Deklaracija prototipa:

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

Prvi parameter je kazalec na dodeljen pomnilniški blok ali kazalec null. Drugi parameter pa določa novo velikost (št. zlogov) dodeljenega pomnilniškega bloka.
Če na danem mestu v pomilniku ni dovolj pomnilniškega prostora, potem se prenese celotna vsebina na novo lokacijo in se s tem zagotovi, da se obstoječi podatki ne izgubijo. Funkcija vrne nov kazalec. Če nova dodelitev pomnilniškega prostora ni možna, potem funkcija vrne kazalec null, pri čemer ostanejo nespremenjene vrednosti podanih parametrov ob klicu funkcije.

Funkcija free

Funkcija free sprosti pomnilniški prostor, ki je bil dodeljen s funkcijo malloc, calloc ali realloc.
Deklaracija prototipa:

```
void free (void * pointer);
```

Sprosti pomnilniški prostor, ki je bil dodeljen z eno izmed zgoraj omenjenih funkcij.

Struktura (structure)

Struktura omogoča združitev podatkov različnega podatkovnega tipa v nov podatkovni tip. S strukturami združimo sorodne podatke (lahko različnega podatkovnega tipa) pod enim imenom in jih nato obravnavamo kot celoto, ki vsebuje posamezne elemente oz. komponente. S strukturami združujemo podatke, ki tvorijo neko smiselno celoto (npr. podatke o osebi, podatke o izdelku…). Podatkom lahko dodamo še metode, ki se izvajajo nad temi podatki. Najprej se osredotočimo samo na združevanje podatkov, ki tvorijo neko smiselno celoto.

Sintaksa:

```
struct naziv_strukture
{
        podatkovni_tip    element1;
        podatkovni_tip    element2;
        podatkovni_tip    element3;
        ...
} naziv_objekta;
```

Definicija strukture se prične z rezerviranko struct, ki ji sledi seznam deklaracij njenih komponent (elementov strukture) med zavitima oklepajema. Struktura ima svoje ime (naziv_strukture), ki ga imenujemo tudi oznaka strukture (structure tag). Za zavitim oklepajem, ki zaključi seznam komponent, lahko sledi seznam spremenljivk (naziv_objekta), ki so podatkovnega tipa te strukture. Do posamezne komponente strukture dostopamo z izrazom naziv_strukture.komponenta. Strukturni operator "." naredi povezavo med imenom strukture in njeno komponento.

Zgled:

```
struct izdelek {
  char naziv [30];
  merska_enota naziv [10];
  float cena;
} ;

izdelek jabolka;
izdelek kivi, banane;
```

ali krajše lahko zgornji primer zapišemo:

```
struct izdelek
{
   char naziv [30];
   merska_enota naziv [10];
   float cena;
} jabolka, kivi, banane;
```

Do posameznih podatkovnih elementov oz. komponent dostopamo z npr. jabolka.naziv, banane.cena...

Zgled 1:

```
// podatki o filmih
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

struct t_film
{
      char naslov [50];
      int leto;
} moji, tvoji;

void printfilm (t_film film);

int main ()
{
      char vmesnik [50];
```

```
      strcpy (moji.naslov, "2001 A Space Odyssey");
      moji.leto = 1968;

      cout << "Naslov: ";
      cin.getline (tvoji.naslov,50);
      cout << "Leto: ";
      cin.getline (vmesnik,50);
      tvoji.leto = atoi (vmesnik);

      cout << "Moj najljubši film je:\n ";
      printfilm (moji);
      cout << "in tvoji:\n ";
      printfilm (tvoji);
      return 0;
}

void printfilm (t_film film)
{
      cout << film.naslov;
      cout << " (" << film.leto << ")\n";
}
```

Zgled 2:

```
struct oseba
{
  char  ime_in_priimek[30];
  char  spol;
  char  naslov[30];
  char  datum_rojstva[30];
  int   postna_stevilka;
};
```

Inicializacije strukture:

```
void main()
{
  oseba  znanec = {"Andrej Osnik",'M',"Pohorska 14", "10.11.1962", 2015};
  oseba sodelavec = znanec;
}
```

Zgled 3:

```
struct oseba
{
  char ime[30];
  char priimek[30];
  char spol;
  naslov  naslov_doma;
  naslov  naslov_v_sluzbi;
  datum   datum_rojstva;
  otroci njegovi_otroci;
};

// funkcija za vnos osebe

void vnesi_osebo (oseba& nova_oseba)

{
    cout << "Vpisi ime in priimek:";
    cin  >> nova_oseba.ime;
    cin  >> nova_oseba.priimek;
    cout << "Vpisi spol:";
    cin >> nova_oseba.spol;
    cout << "Vpisi naslov doma:" << endl;
```

```
    vnesi_naslov (nova_oseba.naslov_doma);
    cout << "Vpisi naslov v sluzbi" << endl;
    vnesi_naslov (nova_oseba.naslov_v_sluzbi);
    cout << "Vpisi datum rojstva" << endl;
    vnesi_datum(nova_oseba.datum_rojstva);
    cout << "Vnesi otroke" << endl;
    vnesi_otroke(nova_oseba.njegovi_otroci);
} // konec vnesi_osebo


// funkcija za izpis osebe

void izpisi_osebo (oseba izpisana_oseba)

{
    cout     <<"Ime in priimek osebe:"
               << izpisana_oseba.ime
              << "  " << izpisana_oseba.priimek
<< endl;
    cout     << "Spol: "
<< ((izpisana_oseba.spol=='m')?"moški" : "ženski") << endl;
    cout     << "Naslov doma: " << endl;
    izpisi_naslov (izpisana_oseba.naslov_doma);
    cout     << "Naslov v sluzbi" << endl;
    izpisi_naslov (izpisana_oseba.naslov_v_sluzbi);
    cout     << "Datum rojstva" << endl;
    izpisi_datum(izpisana_oseba.datum_rojstva);
    if (izpisana_oseba.njegovi_otroci.st_otrok>0)
    {
      cout<<"Otroci:" << endl;
      izpisi_otroke(izpisana_oseba.njegovi_otroci);
    }
    else
        cout << "Oseba nima otrok" << endl;
} // konec izpisi_osebo

// iz strukture oseba in funkcij vnesi_osebo in izpisi_osebo vidimo, da moramo definirati še
// strukturi naslov in datum, ter ustrezne funkcije za vnos in izpis.

// struktura naslov

struct naslov
{
  char  ulica[20];
  char  mesto[20];
  int   postna_stevilka;
};

// funkcija za vnos  naslova
void vnesi_naslov (naslov& novi_naslov)

{
    cout << "Vpisi ulico:";
    cin  >> novi_naslov.ulica;
    cout << "Vpisi mesto:";
    cin >>  novi_naslov.mesto;
    cout << "Vpisi poštno stevilko:";
    cin >>  novi_naslov.postna_stevilka;
}

// funkcija za izpis naslova

void izpisi_naslov (naslov izpisani_naslov)
```

```
{

    cout << izpisani_naslov.ulica << " "
        << izpisani_naslov.postna_stevilka
        << " " << izpisani_naslov.mesto << endl;
}

// struktura za datum:

struct datum
{
  int  dan;
  int  mesec;
  int  leto;
};

// za strukturo datum definiramo funkciji za vnos in izpis

void vnesi_datum (datum& novi_datum)

{
    cout << "Vpisi dan:";
    cin  >> novi_datum.dan;
    cout << "Vpisi mesec:";
    cin >>  novi_datum.mesec;
    cout << "Vpisi leto:";
    cin >>  novi_datum.leto;
}

void izpisi_datum (datum izpisani_datum)

{

    cout << izpisani_datum.dan << "."
        << izpisani_datum.mesec
        << "." << izpisani_datum.leto << endl;

}

// oseba ima definirano strukturo otroci

const int max_otrok = 10; // največje število otrok


struct otroci
{
  int  st_otrok;
  otrok  njegov_otrok[max_otrok];
};


// metode strukture otroci in sicer vnos in izpis

void vnesi_otroke(otroci& vneseni_otroci)
{
    cout << "Vnesi stevilo otrok:";
    cin >> vneseni_otroci.st_otrok;
    for (int i=0; i<vneseni_otroci.st_otrok; i++)
     vnesi_otroka(vneseni_otroci.njegov_otrok[i]);

}
void izpisi_otroke(otroci izpisani_otroci)
{
   cout << "Ime      Starost" << endl;
   for (int i=0; i<izpisani_otroci.st_otrok; i++)
```

```
    izpisi_otroka(izpisani_otroci.njegov_otrok[i]);

}
```

Za posameznega otroka definiramo naslednjo strukturo in funkcije za vnos in izpis:

```
struct otrok
{
  char  ime[20];
  int   starost;
};

void vnesi_otroka(otrok& vneseni_otrok)
{

      cout << "Vpisi ime:";
      cin >> vneseni_otrok.ime;
      cout << "Vpisi starost:";
      cin >> vneseni_otrok.starost;
}

void izpisi_otroka(otrok izpisani_otrok)
{

cout << izpisani_otrok.ime << "   "
     << izpisani_otrok.starost << endl;

}


void main()
{
  oseba jaz, znanec;

  vnesi_osebo(jaz);
  vnesi_osebo(znanec);
  izpisi_osebo(jaz);
  izpisi_osebo(znanec);
}
```

Kazalci na strukture

Kot pri ostalih podatkovnih tipih lahko tudi pri strukturah uporabimo kazalce. Pravila so enaka kot pri enostavnih podatkovnih tipih. Kazalec definiramo kot kazalec na strukturo, ki vsebuje pomnilniški naslov, kjer se objekt te strukture nahaja v pomnilniku.

```
struct t_film
{
  char naslov [50];
  int leto;
};

t_film film;
t_film * kaz_film;
```

Spremenljivka film je objekt strukturnega tipa t_film in kaz_film je kazalec, ki kaže na objekt strukturnega tipa t_film.
Veljaven je izraz kaz_film = &film;

Zgled:

```
// kazalci na strukture
#include <iostream.h>
```

```
#include <stdlib.h>

struct t_film{
    char naslov [70];
    int leto;
};


int main ()
{
    char vmesnik[70];

    t_film film;
    t_film* kaz_film;
    kaz_film = & film;

    cout << "Naslov: ";
    cin.getline (kaz_film->naslov,70);
    cout << "Leto: ";
    cin.getline (vmesnik,70);
    kaz_film->leto = atoi (vmesnik);

    cout << "\nVnesel si:\n";
    cout << kaz_film->naslov;
    cout << " (" << kaz_film->leto << ")\n";

     return 0;
}
```

Operator -> uporabimo za dostop do posamezne komponente strukture, na katero kaže kazalec. Imenujemo ga referenčni operator, ki ga uporabljamo pri kazalcih na strukture. Torej namesto (*kaz_film).naslov lahko uporabimo kaz_film->naslov.

Vgnezdene strukture

Element strukture je lahko tudi struktura.

```
struct t_film{
  char naslov [50];
  int leto;
}

struct t_prijatelj {
  char ime [50];
  char e_naslov [50];
  t_film  najljub_film;
  } bojan, ana;

t_prijatelj * kaz_prijatelj = &ana;
```

Uporabimo lahko naslednje izraze:

```
bojan.ime
bojan.najljub_film.naslov
ana.najljub_film.leto
kaz_prijatelj -> najljub_film.leto
```

Deklaracije lastnih imen podatkovnih tipov

C++ omogoča deklaracijo lastnih podatkovnih tipov na osnovi obstoječih podatkovnih tipov. Sintaksa:

typedef **podatkovni_tip** nov_podatkovni_tip ;

Npr.:

```
typedef char C;
typedef unsigned int WORD;
typedef char * t_niz;
```

Deklarirali smo tri nove podatkovne tipe: C, WORD in t_niz. Te lahko uporabnimo pri deklaraciji spremenljivk:

```
C zn1, znak2, * kaz_znak;
WORD stevilo;
t_niz kaz_znak1;
```

Deklarirana imena podatkovnih tipov lahko uporabljamo v vseh deklaracijah, pretvorbenih operatorjih... skratka povsod, kjer lahko uporabimo podatkovni tip. Pogosto ga uporabimo, da deklariramo krajše nazive podatkovnih tipov.

Unija (unions)

Unija omogoča, da lahko posamezen del pomnilnika uporablja več spremenljivk različnega podatkovnega tipa (ne istočasno). Sintaksa je podobna strukturi, a je njen pomen popolnoma drugačen.

Sintaksa:

union naziv_modela
{
    podatkovni_tip element1;
    podatkovni_tip element2;
    podatkovni_tip element3;
    ...
} naziv_objekta;

Vsi elementi unije zavzamejo isti pomnilniški prostor. Velikost pomnilniškega prostora je enaka velikosti pomnilnika, ki jo potrebujemo za največji element iz sezama elementov unije.

Zgled:

```
union t_unija_stevila
{
    char c;
    int i;
    float f;
} unija_stevila;
```

Definira tri elemente

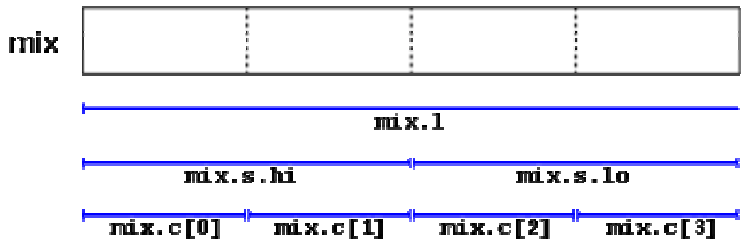
```
unija_stevila.c
unija_stevila.i
unija_stevila.f
```

ki so različnega podatkovnega tipa. Ker uporabljajo vsi elementi isti pomnilnik, pomeni sprememba posameznega elementa dejansko spremembo vseh treh. Spremenljivki unija_stevila lahko priredimo vrednost enega izmed naslednjih podatkovnih tipov: char, int ali float.
Edini dovoljeni operaciji sta dostop do komponente in izračun njenega naslova.

Zgled:

```
union mix_t
{
  long l;
  struct
    {
          short hi;
          short lo;
    } s;
  char c[4];
} mix;
```

Definira tri imena: mix.l, mix.s in mix.c, ki jih lahko uporabimo za dostop do ustreznega dela rezerviranega pomnilnika.



Anonimne unije (anonymous unions)

C++ ima možnost uporabe anonimne unije. Deklariramo jo tako, da izpustimo naziv_objekta za zavitim oklepajem ({ }). V tem primeru dostopamo do elementov unije direktno preko naziva unije.

unija

```
struct
{
    char naslov[50];
    char avtor[50];
    union
    {
        float tolarji;
        int stotini;
    } cena;
} knjiga;
```

anonimna unija

```
struct
{
    char naslov[50];
    char avtor[50];
    union
    {
        float tolarji;
        int stotini;
    };
} knjiga;
```

Edina razlika v kodi je, da smo pri anonimni uniji izpustili naziv_objekta unije (cena). Razlika je tudi pri dostopu do komponent oz. elementov unije.

V prvem primeru uporabimo:
knjiga.cena.tolarji
knjiga.cena.stotini
V drugem pa:
knjiga.tolarji
knjiga.stotini
Ne smemo pozabiti, da ne moremo imeti hkrati vrednosti obeh komponent unije (stotini, tolarji), saj si delita isti pomnilniški prostor.

Naštevni podatkovni tip (enum)

Naštevni podatkovni tip uporabimo, kadar lahko spremenljivka zavzame le eno vrednost izmed konstant iz seznama vrednosti.

Sintaksa:

```
enum naziv_nastevnega_tipa
{
        vrednost1,
        vrednost2,
        vrednost3,
        ...
} naziv objekta;
```

Naštevni podatkovni tip za barve dekariramo kot:

```
enum t_barva {crna, modra, zelena, rdeca, oranzna, rumena, bela};
```

V dekaraciji ne uporabljamo osnovnih podatkovnih tipov. Veljavni so naslednji izrazi:

```
t_barva naj_barva;
naj_barva = modra;
if (naj_barva == zelena)
    naj_barva = rdeca;
```

Naštevni podatkovni tip prevajalnik obravnava kot cela števila in njihove vrednosti so celoštevilčne konstante. Če vrednosti ne določimo, potem elementi dobijo vrednosti 0 – prvi element, 1 – drugi element itd. V našem primeru: crna - 0, modra -1, zelena - 2, rdeca - 3 , oranzna - 4, rumena - 5, bela – 6.

Če določimo celoštevilčno vrednost posameznemu elementu naštevnega podatkovnega tipa, potem se uporabi ta vrednost. Naštevni tip za imena mesecev lahko določimo tako, da številka predstavlja mesec v letu od 1 do 12:

```
enum t_mesec
{
    januar=1, februar, marec, april, maj, junij,
    julij, avgust, september, oktober, november, december
} mesec
```

Dinamične podatkovne strukture

Dinamične podatkovne strukture med izvajanjem programa spreminjajo zasedenost pomnilnika; naraščajo oz. se krčijo (zasegajo in sproščajo pomnilnik). Program uporablja samo toliko pomnilnega prostora, kot ga v resnici potrebuje. Dinamične podatkovne strukture pogosto realiziramo kot množice podatkovnih elementov, ki so med seboj povezane s pomočjo kazalcev. Posamezni elementi se ustvarjajo ali brišejo med izvajanjem programa; spreminjamo pa lahko tudi njihove povezave.
Z uporabo kazalcev lahko tvorimo abstraktne dinamične podatkovne strukture kot so seznami (enosmerni, dvosmerni…), vrste, skladi, drevesa in grafi.

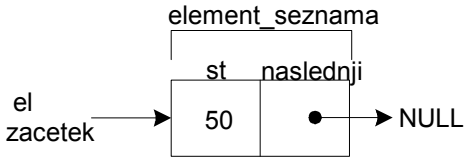
Seznam

Seznam je zaporedje nič ali več elementov: a_1, a_2, a_3, a_4, a_5 … a_n. Pri tem je a_1 prvi element seznama in a_n zadnji element seznama. Elementi seznama niso urejeni po velikosti ali kako drugače. Posamezni elementi seznama se lahko večkrat ponavljajo. Predpisan jim je le vrstni red, kako si sledijo.
Npr. $\mathbf{a_i}$ je i-ti element seznama.
Elementi seznama vsebujejo vsaj dve komponenti:element in kazalec na naslednji element v seznamu.
Definicija strukture za enosmerni seznam celih števil:

```
struct element_seznama
{
    int st; // element
    //kazalec na naslednji element
    element_seznama * naslednji;
};
    // kazalec na začetek seznama
element_seznama * zacetek;
```

Na začetku izvajanja programa elementi seznama števil še ne obstajajo. Kazalec »zacetek« ima nedefinirano vrednost; pomnilniški prostor za elemente seznama pa je potrebno najprej ustvariti. Za to uporabimo operator new:

```
element_seznama * el;

// zaseganje pomnilniškega prostora
el = new(element_seznama);
// določitev vrednosti elementa
el->st = 50;
// naslednji kazalec od elementa
// kaže na začetek
el->naslednji = NULL;
zacetek = el;
```

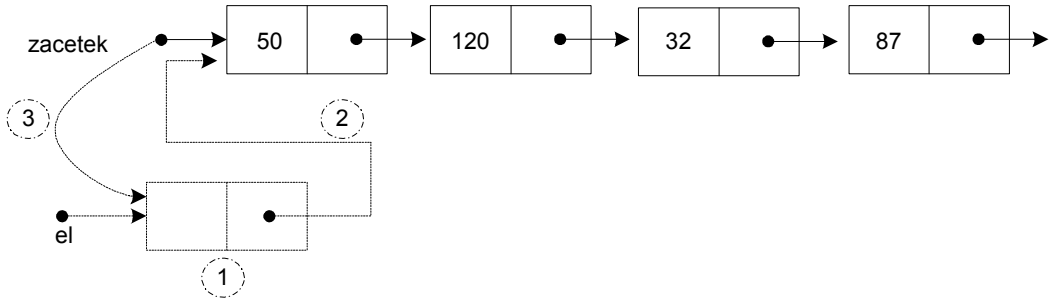


Enosmerni seznam

Enosmerni seznam vsebuje vrednost komponent elementa in kazalec na naslednji element v seznamu.

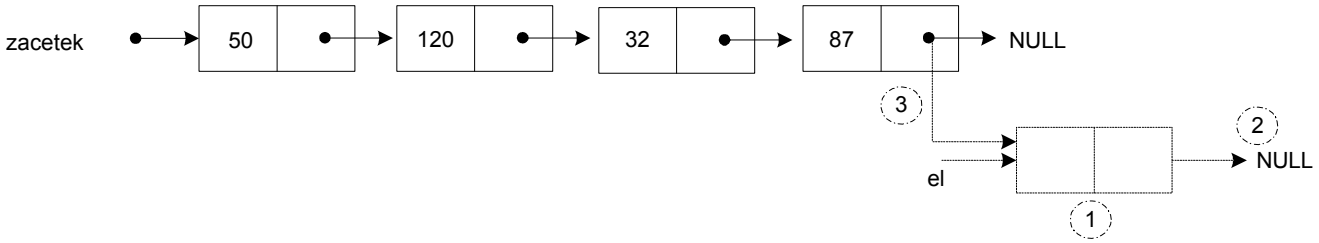
Osnovne operacije nad enosmernim seznamom so:

- dodajanje elementa na začetek seznama (nov element predstavljajo črtkane črte)



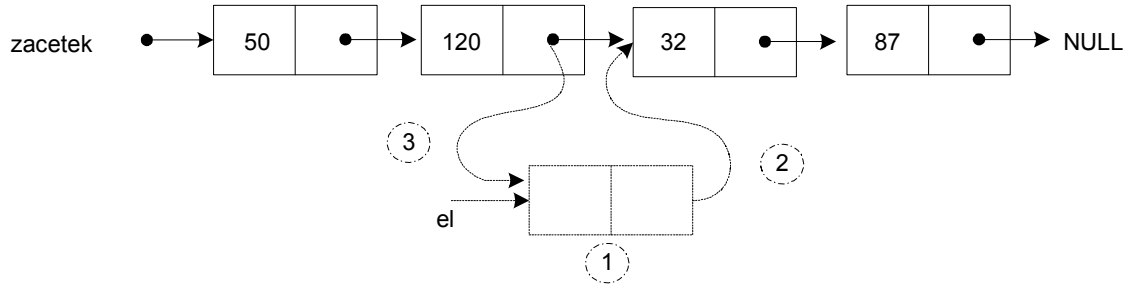

```
int  vstavi_zacetek(int pod)
    {
        element_seznama  * el;
        // zaseganje pomnilniškega prostora
        el = new(element_seznama);
        // določitev vrednosti elementa
        el->st = pod;
        // naslednji kazalec od elementa kaže na začetek
        el->naslednji = zacetek;
        // začetek seznama je nov element
        zacetek = el;
        return (1);
    }
```

- dodajanje novega elementa na konec



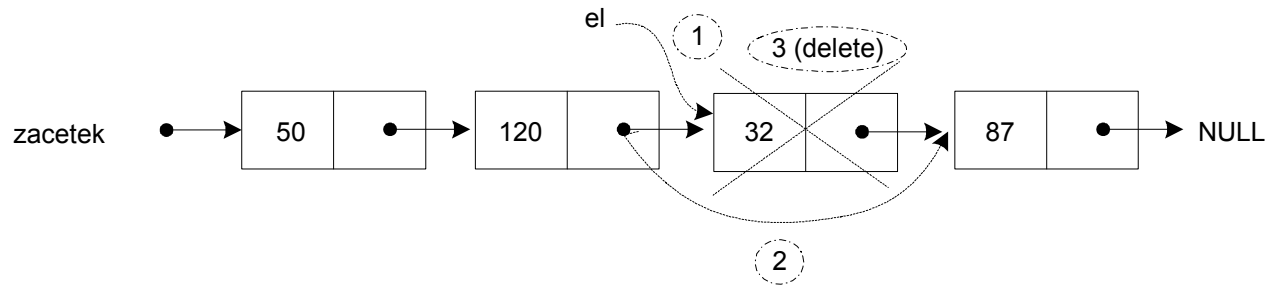

```
int  vstavi_konec (int pod)
    {
        element_seznama  * el;
        element_seznama  * seznam;
        // zaseganje pomnilniškega prostora
        el = new(element_seznama);
        // določitev vrednosti elementa
        el->st = pod;  // element
        el->naslednji = NULL; //nima naslednika
        // seznam prazen
        if (zacetek == NULL)
        {
            // novi element je začetek seznama
            zacetek = el;
        }
        else
        {
            seznam = zacetek;
            // postavimo se na zadnji element v seznamu
            while (seznam->naslednji != NULL)
                seznam = seznam->naslednji;
            // naslednji od zadnjega elementa je nov dodan element
            seznam->naslednji = el;
        }
        return (1);
    }
```

- vstavljanje v seznam pred ali za podanim elementom




```
int  vstaviPredElement (int pod, int elem)
    {
        element_seznama  * el;
        element_seznama  * seznam;
        // zaseganje pomnilniškega prostora
        el = new(element_seznama);
        // določitev vrednosti elementa
        el->st = pod;   // element

        // pred prvega
        if (zacetek->st == elem)
        {
            //naslednji od novega elementa je začetek
            el->naslednji = zacetek;
            // novi element je začetek seznama
            zacetek = el;
        }
        else
        {
            seznam = zacetek;
            // postavimo se na element pred katerega moramo vstaviti nov element
            while (seznam->naslednji != NULL && seznam->naslednji->st != elem)
                seznam = seznam->naslednji;
            if (seznam->naslednji->st == elem)
            {
                // prevezava kazalcev
                el->naslednji = seznam->naslednji;
                seznam->naslednji = el;
            }
        }
        return (1);
    }
```

- brisanje elementa iz seznama



```
int brisiPrvoPojavitev(int pod)
 {
     element_seznama * el_odstrani;
     element_seznama  * seznam;
     seznam = zacetek;

     // ? brisanje prvega elementa
     if (seznam->st == pod)
     {
         el_odstrani = seznam;
         zacetek = seznam->naslednji;
         delete el_odstrani;
     }
     else
     {
         while (seznam->naslednji != NULL)
         {
             if (seznam->naslednji->st == pod)
             {
                 //kazalec na element, ki ga bomo izbrisali
                 el_odstrani = seznam->naslednji;
                 // prevezava kazalca (naslednji)
                 seznam->naslednji=seznam->naslednji->naslednji;
                 // sproščanje pomnilnika
                 delete el_odstrani;
                 return (0);
             }
             // postavimo se na naslednji element
             seznam=seznam->naslednji;
         }
     }
     return (0);
 }
```

Pogosto je naloga obdelati vse elemente v seznamu. Za to je potreben sprehod po seznamu. Obhod elementov:

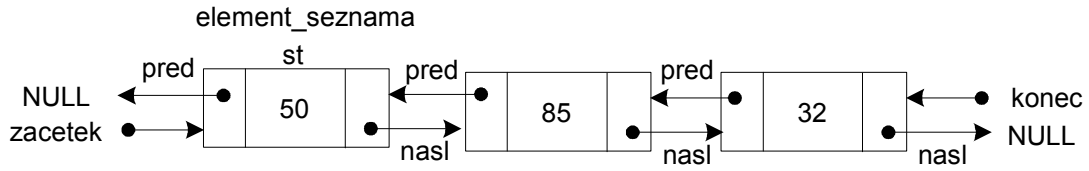
```
int obdelavaSeznama(void)
{
    element_seznama  * seznam;
    // pomozen kazalec na začetek seznama
    seznam = zacetek;
    while (seznam != NULL)
    {
        // obdelava elementa
        seznam = seznam->naslednji; //pomik na naslednji element
    }
    return (1);
}
```

Dvosmerni seznam

Dvosmerni seznam vsebuje vrednost komponent elementa in kazalec na naslednji element ter kazalec na prejšnji element v seznamu.

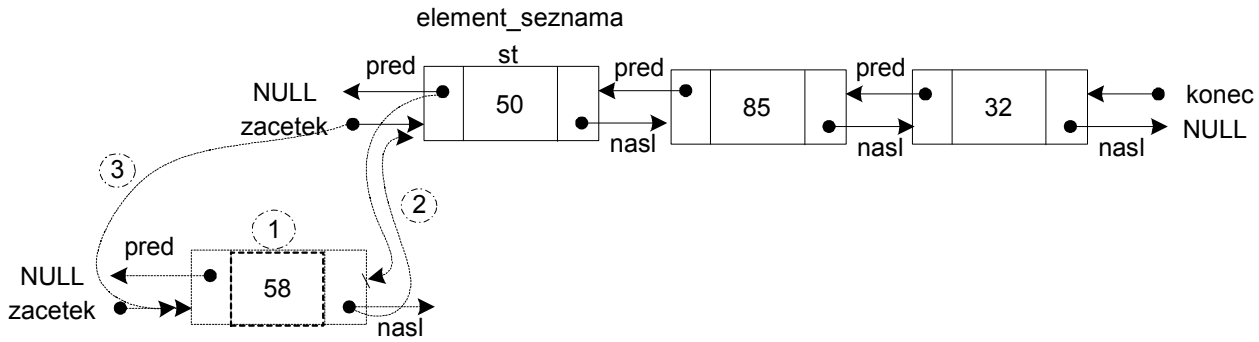
Definicija in deklaracija strukture:

```
struct str_student
{
    float visina; // višina študenta
    char ime[dolzina_imena]; //ime
    char priimek[dolzina_imena]; //priimek
    str_student * nasl;  // naslednik
    str_student * pred;  // predhodnik
};

// kazalec na začetek seznama
str_student * zacetek = NULL;
// kazalec na konec seznama
str_student * konec=NULL;
```



Pri operacijah za delo z dvosmernim seznamom moramo biti pazljivi, saj je večja možnost napak pri prevezavah kazalcev.

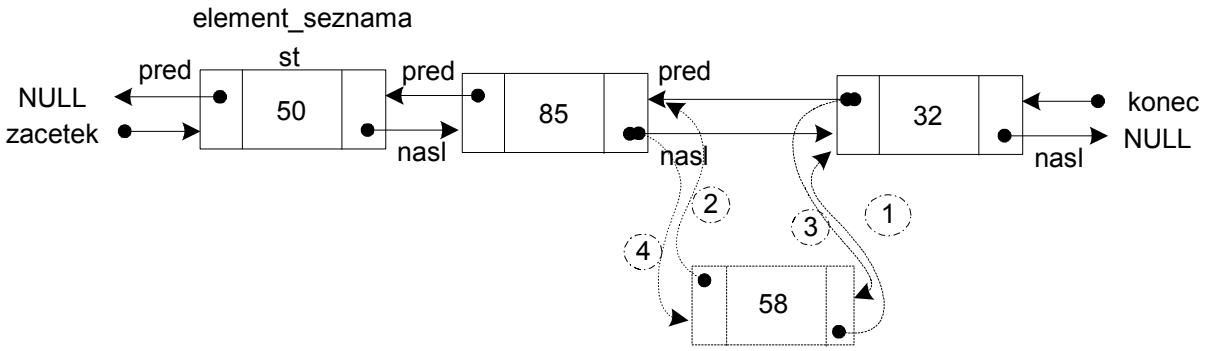
Dodajanje na začetek (seznam ni prazen):




```
int  vstavi_zacetek(void)
    {
        str_student * el;
        el = new(str_student);
        vnos_studenta(el);

        el->pred = NULL;
        // ? prazen seznam
        if (zacetek ==NULL)
        {
            // začetek in konec kažeta na ed element seznama
            konec=el;
            zacetek = el;
```

```
            el->nasl = NULL;
        }
        else
        {
            // prednik od zacetka je novi element
            zacetek->pred=el;
            el->nasl=zacetek;
            // začetek kaže na dodani novi element
            zacetek = el;
        }
        return (1);
    }
```

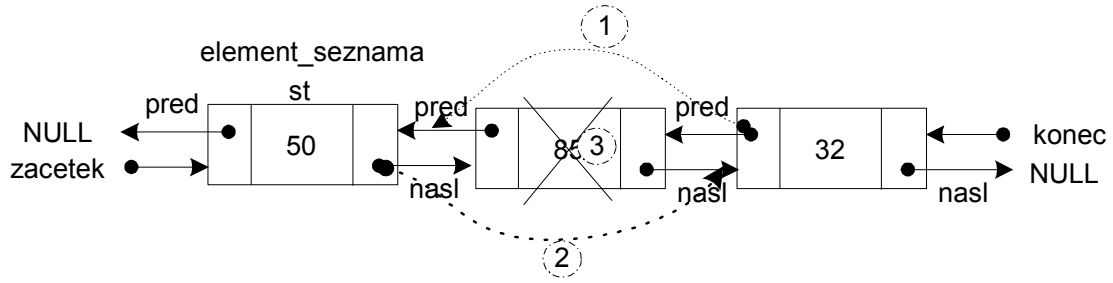
Dodajanje pred element




```
int dodajPredPrvoPojavitev(char * time, char  * tpriimek)
  // argumenta sta kriterij, pred katerega dodamo nov element
{
    bool dodan = false;
    str_student * el_dodaj;
    str_student * seznam;
    seznam = zacetek;
    // ni konec seznama in ni bilo  izvedeno dodajanje
    while (! dodan && seznam != NULL)
    {
        // ? pred trenutnega
        if (strcmp(seznam->ime , time)==0  &&
              strcmp(seznam->priimek,tpriimek)==0)
        {
            dodan = true;
            el_dodaj=new(str_student);
            vnos_studenta(el_dodaj);
            //dodajanje pred edinega elementa v seznamu
            if (seznam == zacetek)
                {
                    el_dodaj->nasl=zacetek;
                    el_dodaj->pred=NULL;
                    zacetek->pred=el_dodaj;
                    zacetek=el_dodaj;
                }
            else
                {
                    //dodajanje vmes
                    el_dodaj->nasl=seznam;
                    el_dodaj->pred=seznam->pred;
                    seznam->pred->nasl=el_dodaj;
                    seznam->pred=el_dodaj;
```

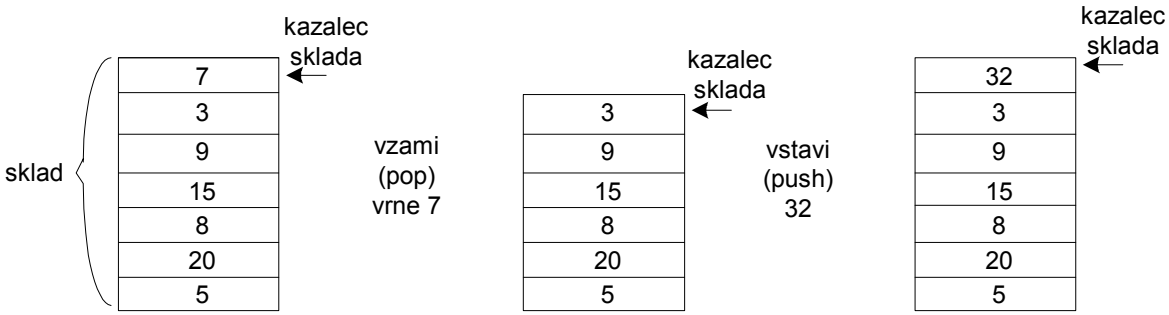
```
                }
            }
            else
                seznam=seznam->nasl;

        }
        return (0);
    }
```

Brisanje iz seznama (vmesni element):



Sklad

Sklad je podatkovna struktura, ki si jo najlažje predstavljamo s skladom knjig. Novo knjigo vedno dodamo na vrh sklada in ko jemljemo knjige s sklada, jih vedno jemljemo z vrha. Na vrh sklada kaže kazalec sklada. Če na skladu ni nobenega elementa, potem ima vrednost NULL (kazalec ne kaže nikamor).




```
//struktura za element sklad
    struct elem
    {
        int st; // element sklada
        elem * naslednik; // kazalec na naslednji element
    };

    elem *vrh_sklada; // kazalec na vrh sklada
```

Operacija vstavi (push)

```
// vstavi novo število na vrh sklada (push) in ustrezno spremeni kazalec na vrh sklada
int  vstavi(int pod)
    {
        elem  * el;
        el = new(elem);
        el->st = pod;
        el->naslednik = vrh_sklada;
        vrh_sklada = el;
        return(1);
    }
```

Operacija vzami (pop)

```
// vzame element z vrha sklada (pop) in ustrezno spremeni kazalec na vrh sklada
int vzami(int &t_st)
{
    elem * el_odstrani;
    el_odstrani = vrh_sklada;
    t_st = el_odstrani-> st;
    vrh_sklada=vrh_sklada->naslednik;
    delete el_odstrani;
    return (0);
}
```