Слободанка Ђенић

Програмски језици C и C++

Висока школа електротехнике и рачунарства струковних студија
Академија техничко-уметничких струковних студија
Београд, 2020.

Аутор: Слободанка Ђенић

Рецензенти: Јелена Митић, Светлана Штрбац Савић

Корице: Јелена Лабус Грујић (MASSVision)

Издавачи: Висока школа електротехнике и рачунарства струковних студија, Београд
Академија техничко-уметничких струковних студија Београд

Штампа: Развојно-истраживачки центар графичког инжењерства ТМФ, Београд

Тираж: 30

Издање: 1.

ISBN 978-86-7982-325-0

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42 (075.8)
004.432.2C++ (075.8)

ЂЕНИЋ, Слободанка, 1965 -
Програмски језици C и C++ / Слободанка Ђенић. - 1. изд. - Београд : Висока школа електротехнике и рачунарства струковних студија : Академија техничко-уметничких струковних студија Београд, 2020 (Београд : Развојно-истраживачки центар графичког инжењерства ТМФ). - 163 стр. : илустр. ; 24 cm
Тираж 30. – Библиографија: стр. 163.
ISBN 978-86-7982-325-0
a) Програмирање b) Програмски језик "C" c) Програмски језик "C++"
COBISS.SR-ID 21983241

Предговор

Циљ наставе из предмета Програмски језици у Високој школи електротехнике и рачунарства струковних студија у Београду, за који је написан овај уџбеник, јесте да на једноставан начин објасни концепте пројектовања програма на језицима *C* и *C++* као и напредне технике програмирања на овим језицима.

Програмски језик C изабран је јер је то језик који је послужио као база за развој актуелних програмских језика. Програмски језик C++ је значајан као хибридни језик који може послужити за израду програма који су структурно али и објектно оријентисано пројектовани и самим тим сматра се добрим избором за прелаз ка објектно оријентисаном програмирању.

Лекције уџбеника обухватају теоријски део наставе из предмета Програмски језици, кроз следеће тематске целине на језицима *C* и *C++*:

- Улаз / излаз података;
- Функције, низови и показивачи;
- Динамичка додела меморије;
- Структуре података;
- Рад са датотекама;

као и тематске целине искључиво на језику *C:*

- Структурно пројектовање програма;

и искључиво на језику *C++:*

- Објектно оријентисано пројектовање програма.

Текст у лекцијама уџбеника написан је у кратким, прегледним и лако савладивим сегментима, са одговарајућим илустрацијама. На почетку сваке лекције постоји кратак преглед њеног садржаја. Сваки нови појам у лекцији дефинисан је у тексту и објашњен у илустрацији. На крају сваке лекције изложена су питања за проверу знања градива лекције, за редовну проверу напредовања у учењу.

На крају уџбеника изложен је индекс кључних појмова.

У Београду, 2020. године Аутор

Садржај

1. Улаз / излаз података у програмима на језику C

Кратак преглед лекције

У сваком програму заступљен је улаз / излаз података. У језику *C* за ово су предвиђене готове функције из стандардних библиотека које користе ниво улазно / излазних токова података. Програм на језику *C* може добити команде и податке при позиву, из командне линије или током извршавања, са тастатуре. Веће количине података обично су сачуване у датотекама и програм их чита из датотека. Резултате свог рада програм може слати на екран, као привремени излаз, или у датотеке, на трајно чување.

1.1. Улазно / излазни токови

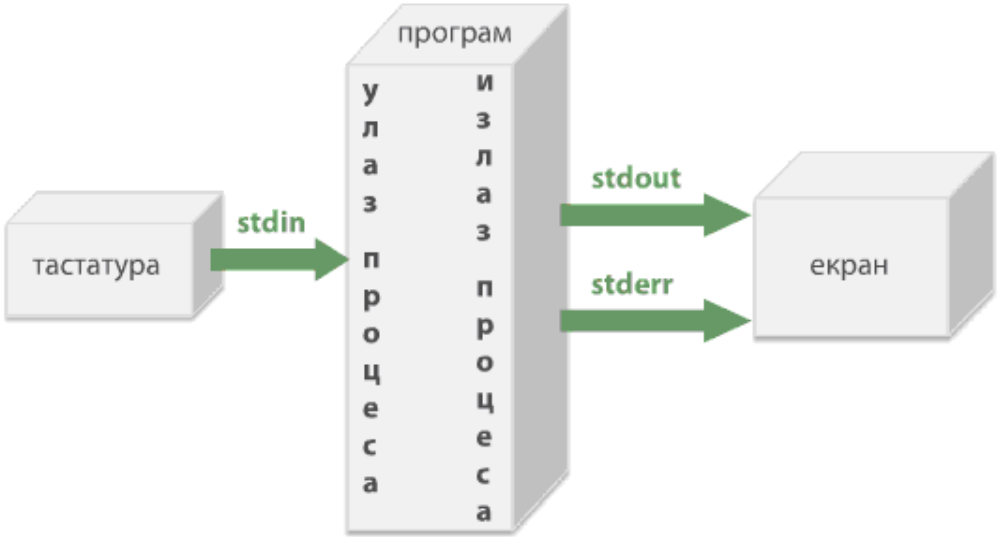
Улаз / излаз података на нивоу улазно / излазних токова стандардизован је у језику *C* јер омогућава: аутоматско баферовање података (њихово прослеђивање у редовима, у једном и другом смеру) и на тај начин ефикасан улаз / излаз, независност од детаља улазно / излазних уређаја и преносивост између разних оперативних система. Због овога се рад *C* функција улаза / излаза *(scanf(), printf(), getchar(), putchar(), gets(), puts()…)* заснива на коришћењу улазно / излазних токова.

Појам улазно / излазног тока

Улазно / излазни ток *(stream)* је секвенца бајтова података. Улазни ток *(input stream)* је секвенца бајтова са одређеног улазног уређаја, а излазни ток *(output stream)* секвенца бајтова ка одређеном излазном уређају. Улазно / излазни ток може се сматрати комуникационим каналом између улазно / излазног уређаја и улаза / излаза процеса програма. Постоје две врсте ових токова. Текстуални ток чине форматирани подаци распоређени у редовима, са ознаком за крај EOF (-1). Бинарни ток чине само бинарне цифре (0 и 1) без распоређивања и ознака.

Стандардни улазно / излазни токови

Од оперативног система зависи дозвољени број тренутно активних улазно / излазних токова. Аутоматски су дозвољени само стандардни токови, слика 1.1. То су: главни стандардни улаз *(stdin)*, главни стандардни излаз *(stdout)* и стандардни излаз за грешке *(stderr)*. Прва два тока (*stdin* / *stdout*) подразумевано повезују процес програма са тастатуром / екраном али се могу преусмерити и на неки други улазно / излазни уређај. Због тога је издвојен стандардни излаз за грешке *(stderr)* који се не може преусмерити са екрана.



Слика 1.1 Стандардни улазно / излазни токови (streamovi)

1.2 Функције улаза / излаза и стандардни улазно / излазни токови

Једној групи функција улаза / излаза припадају функције које користе само стандардне улазно / излазне токове. То су функције *scanf(), printf(), getchar(), putchar(), gets(), puts()*... Другу групу чине функције улаза / излаза којима је потребно дефинисати са којим улазно / излазним током раде (са стандардним или неким другим претходно дефинисаним). Другој групи припадају функције: *fscanf(), fprintf(), fgetc(), fputc(), fgets(), fouts()*... Свака од ових функција у аргументу добија информацију о улазно / излазном току, слика 1.2.

функције које користе стандардне streamove	функције које траже дефинисање streamova	Реализација улаза / излаза
scanf() printf()	fscanf() fprintf()	форматирано
getchar() putchar()	fgetc() fputc()	карактер по карактер
gets() puts()	fgets() fputs()	ред по ред

Префикс f (од file) у називу функције

Слика 1.2 Функције улаза / излаза из C библиотеке <stdio.h>

Функције за форматиран улаз

За улаз нумеричких, или комбинацију нумеричких и текстуалних података, добар избор су функције форматираног улаза: *scanf()* и *fscanf()*, слика 1.3. Функција *scanf()* чита податке са тастатуре, а функција *fscanf()* их чита са тастатуре или из датотеке (ово омогућава додатни аргумент са информацијом о улазном току). Ако у овом аргументу добије ознаку главног улазног тока *stdin*, слика 1.4, функција *fscanf()* ради са тастатуром, а ако добије ознаку улазног тока од одређене датотеке, ради са њом.

```
int scanf(          const char *format, void *ptr, . . .);

int fscanf(FILE *fptr,  const char *format, void *ptr, . . .);
```

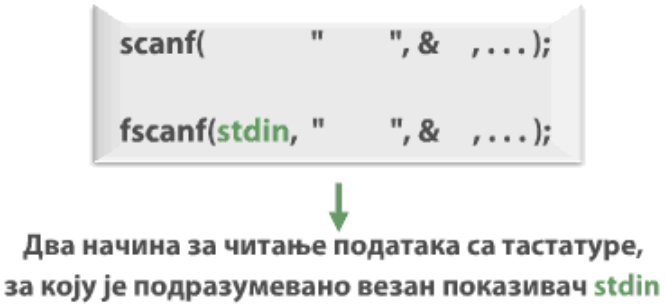
Разлика само у 1. аргументу

Адреса у меморији за учитани податак

Број успешно прочитаних података или EOF(-1)

Показивач на стринг формат

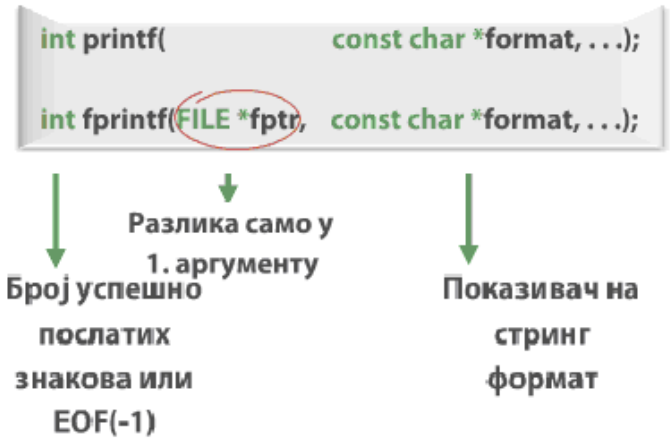
Слика 1.3 Функције за форматиран улаз



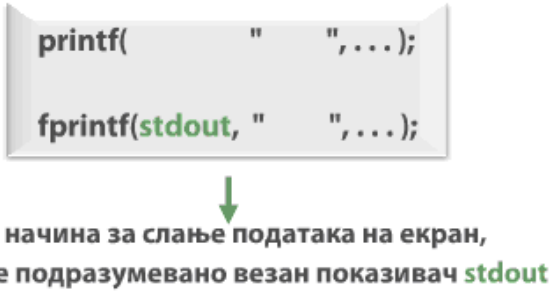
Слика 1.4 Функције за форматиран улаз са тастатуре

Функције за форматиран излаз

За излаз нумеричких и текстуалних података у одређеном формату (у табелама, извештајима, графицима...) користе се функције *printf()* и *fprintf()*, слика 1.5. Функција *printf()* приказује податке на екрану, а функција *fprintf()* их приказује на екрану или уписује у датотеку (ово омогућава додатни аргумент са информацијом о излазном току). Ако у овом аргументу добије ознаку главног излазног тока *stdout*, слика 1.6, функција *fprintf()* ради са екраном а ако добије ознаку излазног тока ка одређеној датотеци, ради са њом.



Слика 1.5 Функције за форматиран излаз



Слика 1.6 Функције за форматиран излаз на екран

Функције за улаз знак по знак

За улаз једног знака користе се функције: *getchar()* и *fgetc()*, слика 1.7. Функција *getchar()* чита знак са тастатуре, а функција *fgetc()* чита знак са тастатуре или из датотеке (ово омогућава додатни аргумент са информацијом о улазном току). Ако у овом аргументу добије ознаку главног улазног тока *stdin*, слика 1.8, функција *fgetc()* ради са тастатуром а ако добије ознаку улазног тока од одређене датотеке, ради са њом. Свака од ових функција очекује један знак са улазног тока, али и знак за прелаз у нови ред као потврду уноса.

Слика 1.7 Функције за улаз карактер по карактер

Слика 1.8 Функције за улаз карактер по карактер са тастатуре

Функције за излаз знак по знак

За излаз једног знака користе се функције: *putchar()* и *fputc()*, слика 1.9. Функција *putchar()* приказује знак на екрану, а функција *fputc()* приказује знак на екрану или га уписује у датотеку (ово омогућава додатни аргумент са информацијом о излазном току). Ако у овом аргументу добије ознаку главног

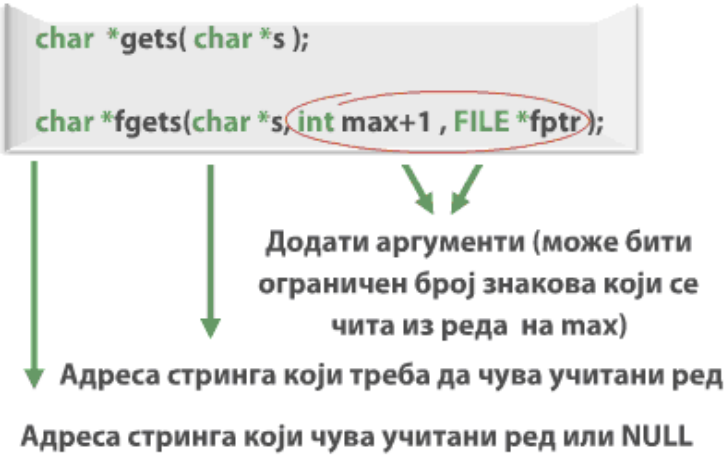
излазног тока *stdout*, слика 1.10, функција *fputc()* ради са екраном а ако добије ознаку излазног тока ка одређеној датотеци, ради са њом. Функције *putchar()* и *fputc()* користе се у комбинацији са функцијама *getchar()* и *fgetc()*.

Слика 1.9 Функције за излаз каратер по карактер

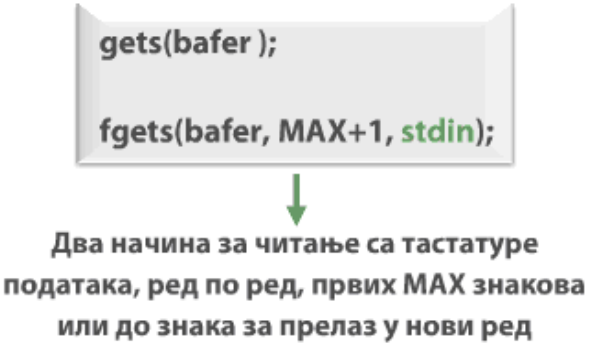
Слика 1.10 Функције за излаз каратер по карактер на екран

Функције за улаз ред по ред

За улаз једног реда знакова користе се функције: *gets()* и *fgets()*, слика 1.11. Функција *gets()* чита ред знакова са тастатуре, а функција *fgets()* чита знакове са тастатуре или из датотеке (ово омогућава додатни аргумент са информацијом о улазном току) и то: или одређени број знакова (још један додатни аргумент) или до краја реда. Ако у аргументу са информацијом о улазном току добије ознаку главног *stdin* функција *fgets()* ради са тастатуром, а ако добије ознаку улазног тока од одређене датотеке ради са њом, слика 1.12.



Слика 1.11 Функције за улаз ред по ред

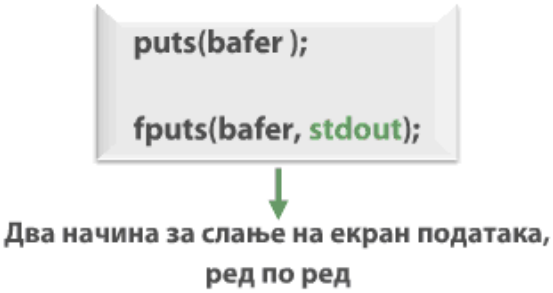


Слика 1.12 Функције за улаз ред по ред са тастатуре

Функције за излаз ред по ред

За излаз једног реда знакова користе се функције излаза: *puts()* и *fputs()*, слика 1.13. Функција *puts()* приказује ред знакова на екрану, а функција *fputs()* приказује ред знакова на екрану или уписује у датотеку (ово омогућава додатни аргумент са информацијом о излазном току). Ако у овом аргументу добије ознаку главног излазног тока *stdout*, слика 1.14, функција *fputs()* ради са екраном, а ако добије ознаку излазног тока ка одређеној датотеци ради са њом. Функције *puts()* и *fputs()* користе се у комбинацији са функцијама *gets()* и *fgets()*.

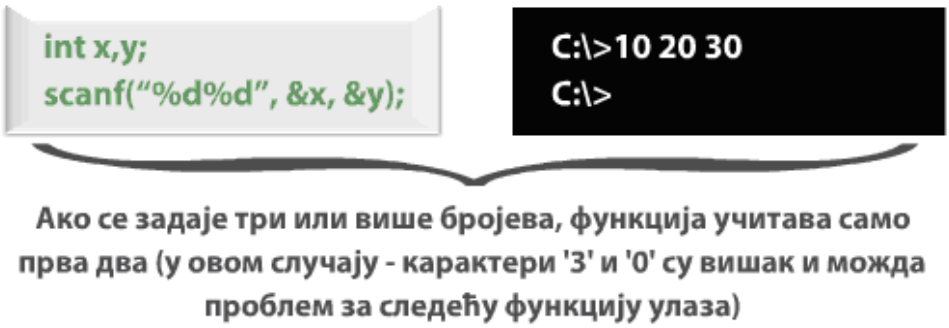
Слика 1.13 Функције за излаз ред по ред



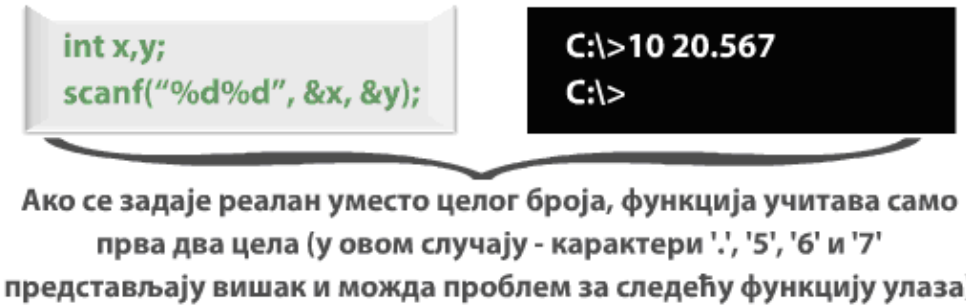
Слика 1.14 Функције за излаз ред по ред на екран

1.3 Решавање вишка карактера у главном улазном току

Функција *scanf()* пројектована је само за пажљиво форматирање података и не прихвата грешке. Ако се подаци уносе у већем броју од предвиђеног, слика 1.15, или ако је вредност податка у већем опсегу од предвиђеног, слика 1.16, на учитавање у улазном току чека тај вишак карактера. Ако се у таквим случајевима функција *scanf()* позива у комбинацији са осталим функцијама улаза непрочитани карактери могу заварати остале функције да су подаци за њих већ задати и условити код њих грешке.



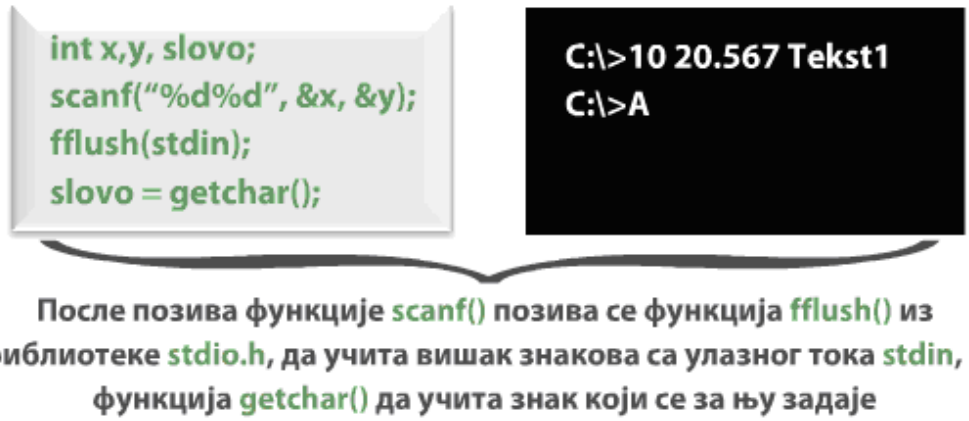
Слика 1.15 Проблем заосталих карактера у улазном току, случај 1.



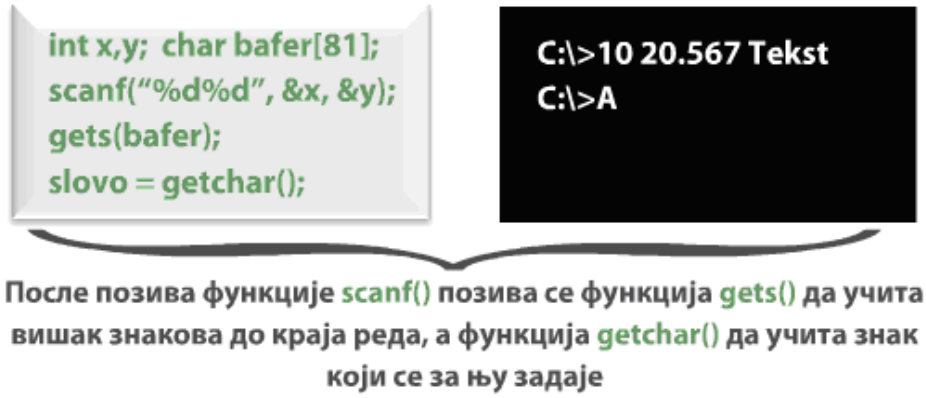
Слика 1.16 Проблем заосталих карактера у улазном току, случај 2.

Решење за проблем непрочитаних карактера

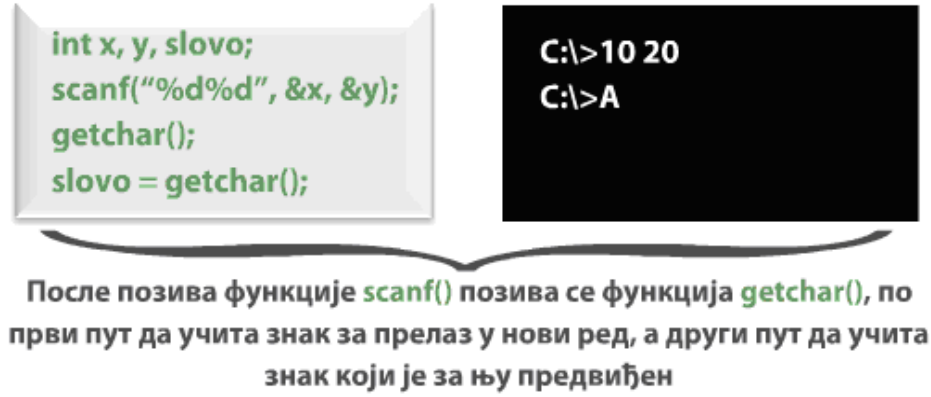
За решавање проблема непрочитаних карактера може бити искоришћено више функција. Једна од њих је функција *fflush()* која брише заостале карактере са главног улазног тока и због тога се препоручује њен позив после сваког комплета позива функције *scanf()*, слика 1.17. Друга је функција за читање реда знакова *gets()* која се у овом случају користи за читање вишка карактера задатих до краја реда, слика 1.18. За читање знака за прелаз у нови ред, који се користи као потврда уноса, користи се позив функције *getchar()*, слика 1.19.



Слика 1.17 Једно могуће решење проблема више заосталих карактера



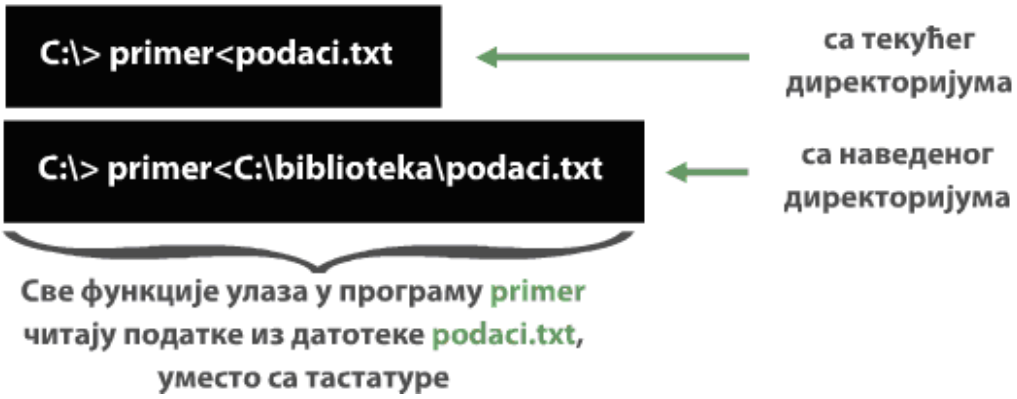
Слика 1.18 Друго могуће решење проблема више заосталих карактера



Слика 1.19 Решење проблема једног заосталог карактера

1.4 Улаз / излаз програма са могућношћу преусмеравања

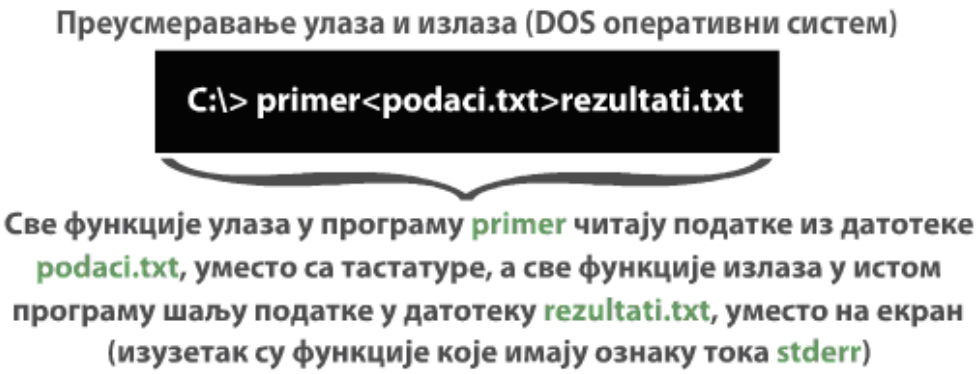
При позиву програма из командне линије могуће је преусмеравање улаза / излаза. Ако се преусмерава улаз процеса онда програм не добија податке са тастатуре, као што је у њему написано, већ из датотеке, слика 1.20. Ако се преусмерава излаз процеса програм не шаље податке на екран, као што је у њему написано, већ у датотеку, слика 1.21, и то од почетка или у продужетку постојећег садржаја, слика 1.22. Могуће је преусмеравање улаза и излаза помоћу датотека, слика 1.23, или без њих, слика 1.24.



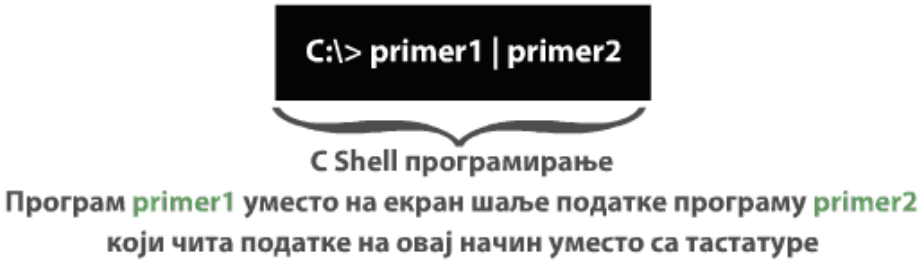
Слика 1.20 Преусмеравање улаза

Слика 1.21 Преусмеравање излаза, начин 1.

Слика 1.22 Преусмеравање излаза, начин 2.



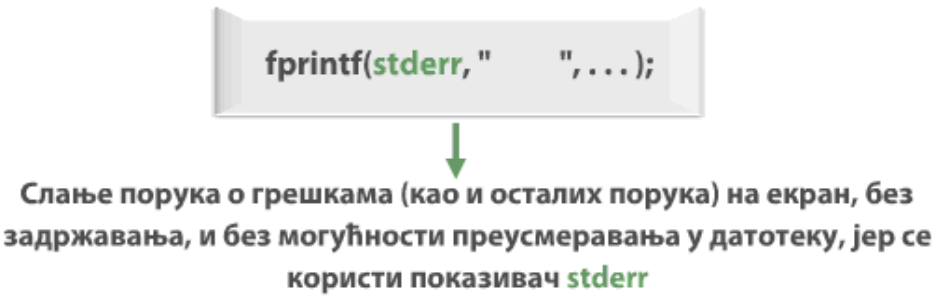
Слика 1.23 Преусмеравање улаза и излаза



Слика 1.24 Повезивање команди (pipeline)

Излаз програма без могућности преусмеравања

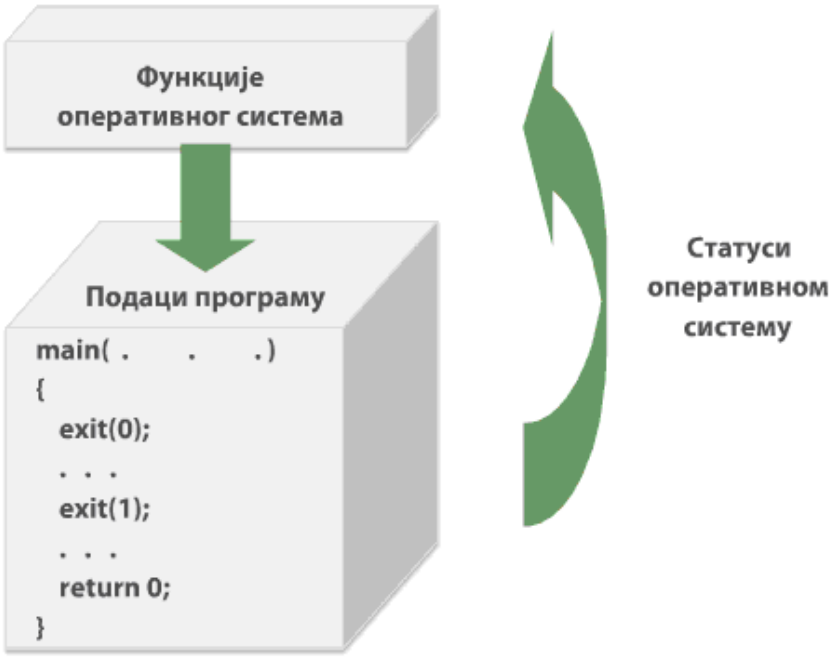
Преусмеравање улаза / излаза програма могуће је при позиву свих функција које подразумевано користе главни стандардни улаз / излаз: *scanf()*, *printf()*, *getchar()*, *putchar()*, *gets()*, *puts()*… Могуће је преусмеравање и код функција које траже информације о улазно / излазном току, *fscanf()*, *fprint()*, *fgetc()*, *fputc()*, *fgets()*, *fputs()*…, ако се напишу ознаке главног стандардног улаза / излаза *stdin* / *stdout*. Преусмеравање излаза није могуће код ових функција само ако се напише ознака излазног тока за грешке *stderr*, слика 1.25.



Слика 1.25 Излаз програма без могућности преусмеравања

1.5 Подаци из командне линије

Један смер комуникације између програма и оперативног система је слање статуса оперативном систему позивом функције *exit()* или последњом наредбом главне функције *return*. Комуникација у смеру од оперативног система ка програму корисна је код решавања административних задатака где је потребан командни интерпретер. Оперативни систем онда предаје податке из командне линије програму, слика 1.26, а главна функција програма их прихвата у својим аргументима, ако је тако написана, слика 1.27.



Слика 1.26 Комуникација са оперативним системом

Слика 1.27 Облик главне функције која прихвата податке из командне линије

Обрада података из командне линије

Главна функција може обрадити податке из командне линије, као низове знакова, помоћу два своја аргумента. Један аргумент је цео број ових података *(int argc)*, при чему се узима у обзир и само име програма. Други аргумент је показивач на низ показивача на све податке редом из команде линије *(char *argv[])*. Први од показивача *(argv[0])* у овом низу је показивач на име програма, а остали су показивачи на све податке задате редом после имена програма *(argv[1], argv[2],...,argv[argc-1])*, слика 1.28.

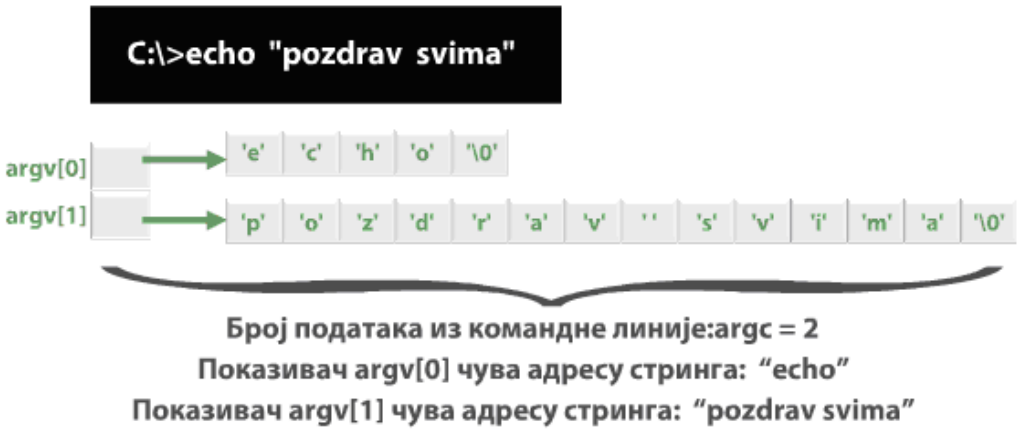
Слика 1.28 Рад са подацима из командне линије

Читање stringova из командне линије

Главна функција третира сваки податак из командне линије као низ знакова, слика 1.29, а може приступити и појединачно сваком знаку тако задатог низа, слика 1.30. Ако се податак у командној линији задаје између знакова навода онда тај податак може садржати и беле знакове, слика 1.31. И у овом случају могућ је приступ појединачним знаковима тако задатог низа, слика 1.32. Главна функција може, из својих аргумената а на веома једноставан начин, да добије информације о подацима из командне линије, слика 1.33.

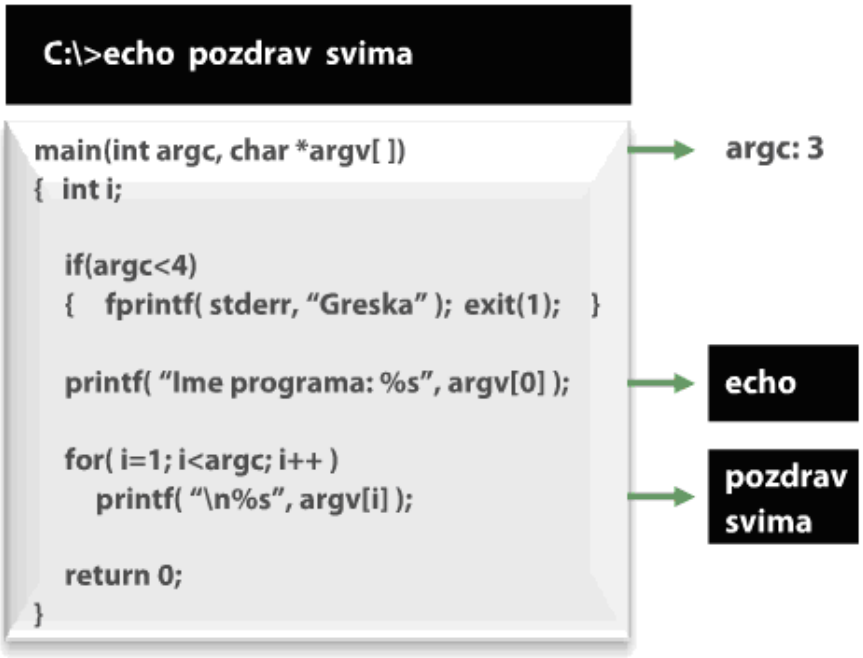
Слика 1.29 Приступ сваком податку из командне линије

Слика 1.30 Приступ сваком знаку сваког податка из командне линије



Слика 1.31 Приступ сваком податку из командне линије, задатом између знакова навода

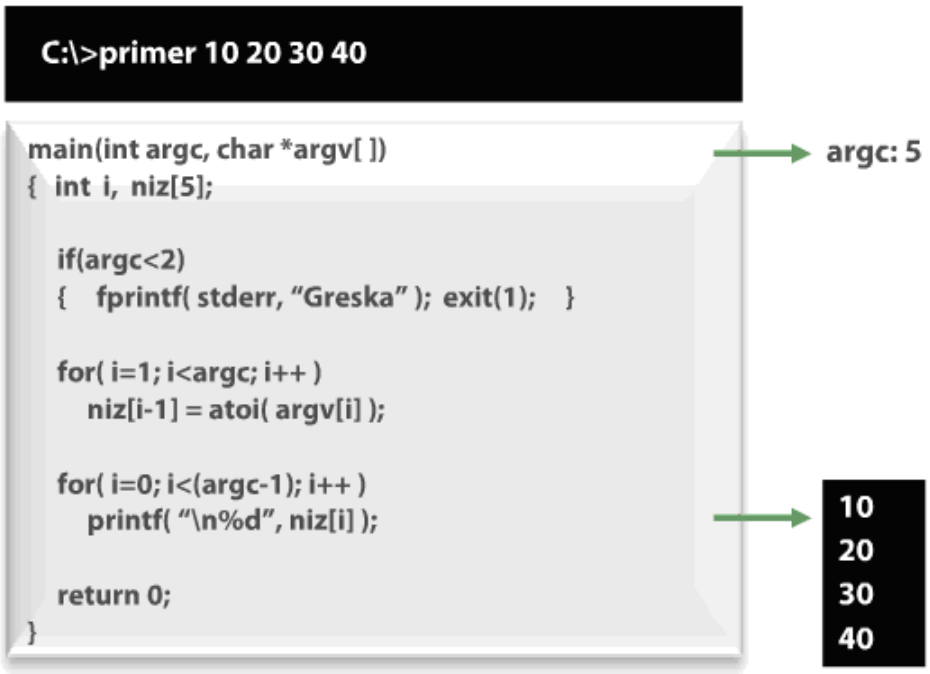
Слика 1.32 Приступ сваком знаку сваког податка из командне линије, задатог између знакова навода



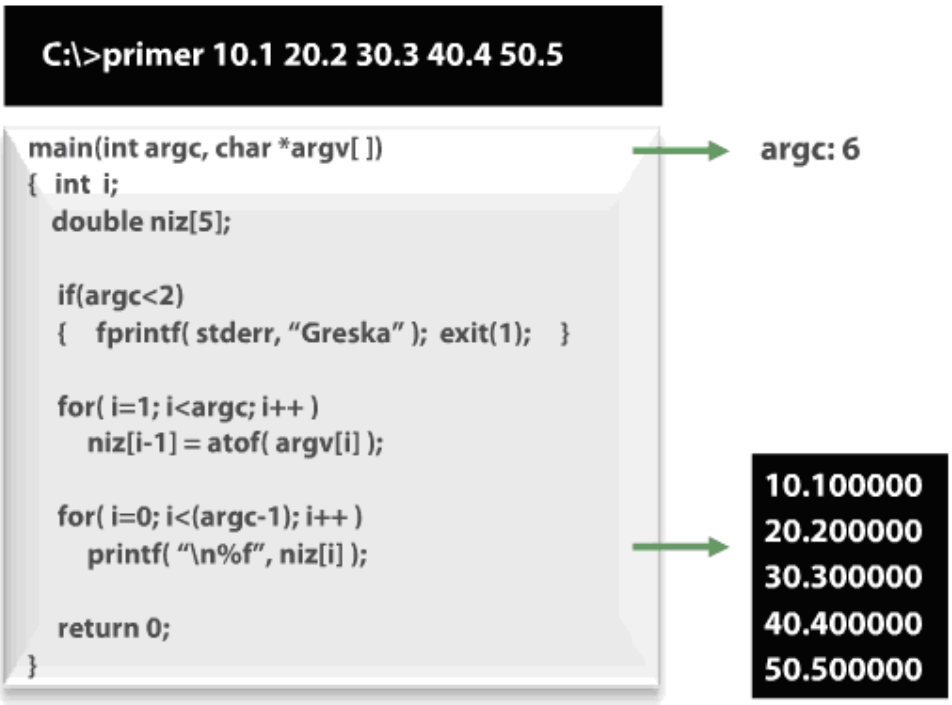
Слика 1.33 Читање *stringova* из командне линије

Читање бројева из командне линије

Иако све податке из командне линије третира као низове знакова, главна функција из ових података може добити и бројеве. На већ поменут начин, из аргумената главне функције добијају се информације о задатим низовима знакова као *stringovima*, а за превођење *stringova* у целе или реалне бројеве користи се функција *atoi()*, слика 1.34 или *atof()*, слика 1.35. На овај начин програму се на почетку поред команди могу задавати и низови бројева. Током извршавања програма онда нема додатних захтева за њихов унос.



Слика 1.34 Читање целих бројева из командне линије



Слика 1.35 Читање реалних бројева из командне линије

1.6. Питања

1. Које се ознаке у *C* програмима користе за главне стандардне улазно / излазне токове?
2. Који се улазо / излазни токови сматрају главним стандардним токовима?
3. Који стандардни излазни ток је баферован а који није и шта то значи?
4. Која је разлика између *C* функција улаза / излаза, са и без префикса f (на пример, између функција fscanf() и scanf())?
5. Како раде функције getchar() и fgetc()?
6. Како раде функције gets() и fgets()?
7. Која је разлика између *C* функција fscanf(), fgetc() и fgets()?
8. Шта значи преусмеравање стандардног улаза / излаза?
9. У којим случајевима је могуће преусмеравање стандардног улаза / излаза?
10. Како је потребно написати заглавље главне функције *C* програма да би програм читао податке из командне линије?
11. Шта представља сваки од аргумената главне функције *C* програма који чита податке из командне линије?
12. Када главна функција *C* програма користи аргументе из командне линије, да ли се име програма рачуна као један од тих података или не?
13. Да ли постоји ограничење у броју података из командне линије?
14. Како *C* програм третира податке из командне линије?
15. Како *C* програм ради са бројевима задатим у командној линији?

2. Функције и показивачи у програмима на језику C

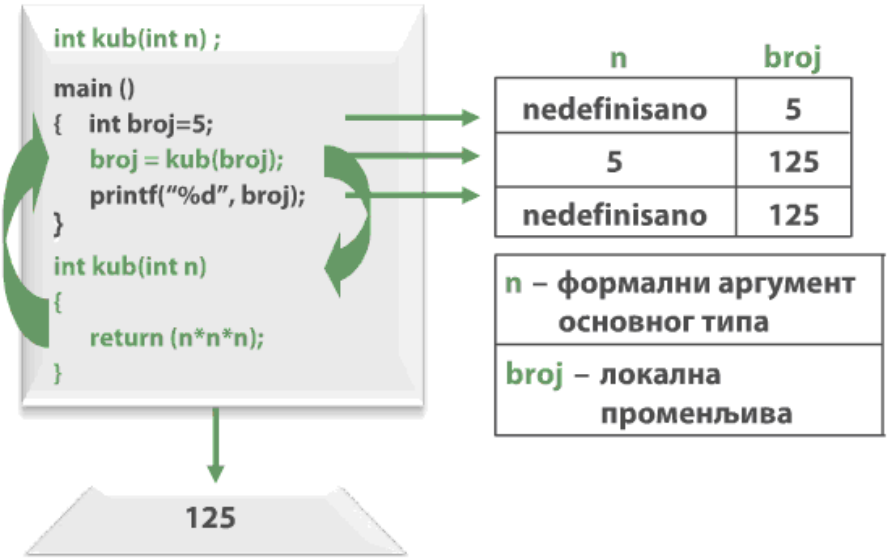
Кратак преглед лекције

Поред функција које међусобно размењују вредности података основних типова, у језику *C* дефинисане су и функције које размењују адресе помоћу показивача. Показивач може бити аргумент функције и повратна вредност од функције. Покретањем извршне верзије функције, њене наредбе добијају простор у меморији од одређене адресе. За издвајање и коришћење ове адресе у програму користи се показивач на функцију. За приступ подацима у функцијама постоје правила локалног и глобалног досега промењљивих.

2.1. Функције које користе и враћају вредности показивача

Прослеђивање аргумената функцији - по вредности

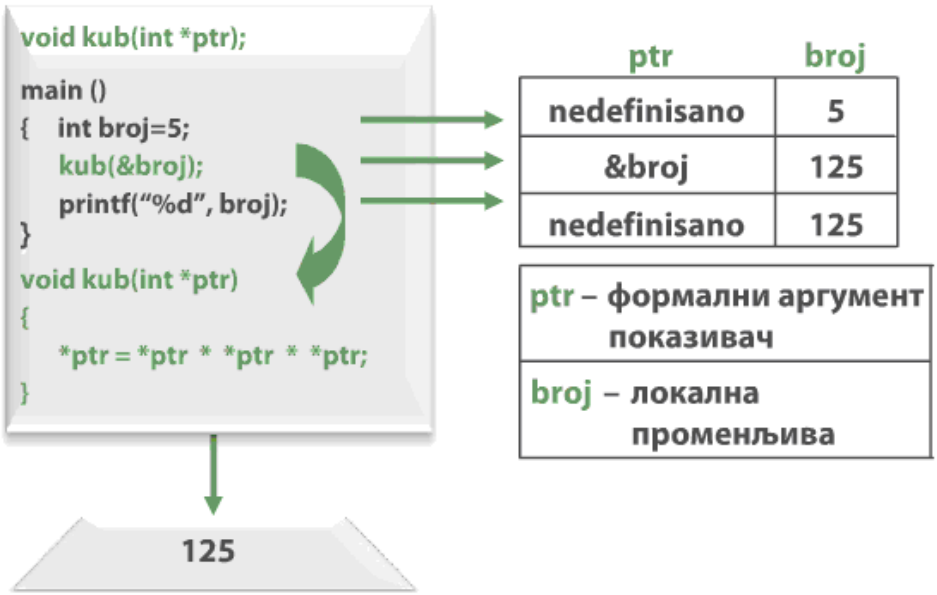
Размена података између појединих функција у програму може се реализовати помоћу аргумената и повратних вредности. Један начин прослеђивања аргумената *C* функцији је прослеђивање аргумената по вредности. У овом случају аргументи функције су вредности основних типова, слика 2.1. Ако добије аргументе по вредности функција ради с копијама садржаја оригиналних меморијских локација и нема могућност да измени њихове садржаје, већ само да их користи. Овај начин прослеђивања аргумената је подразумевани.



Слика 2.1 Прослеђивање аргумената функцији по вредности

Прослеђивање аргумената функцији – по референци

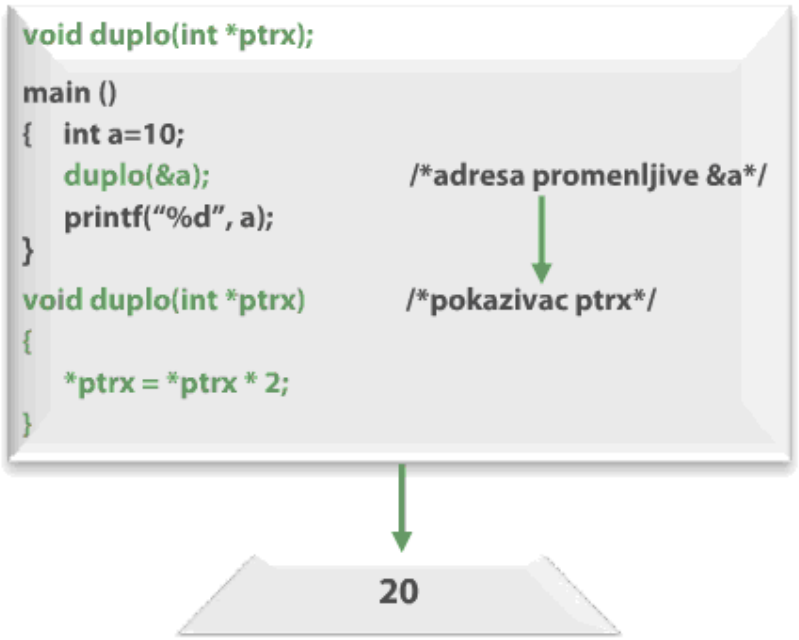
Поред основног начина прослеђивања аргумената *C* функцији, постоји и алтернативан начин, то је прослеђивање аргумената по референци помоћу показивача (адресе), слика 2.2. За аргумент показивач пише се: у декларацији функције обавезно тип на који показује *(tip *ptr)*, при позиву функције стварна адреса која се додељује формалном показивачу *(&ime_prom)* и у дефиницији саме функције приступ променљивој помоћу оператора индиректног приступа и имена формалног показивача *(*ptr)*.



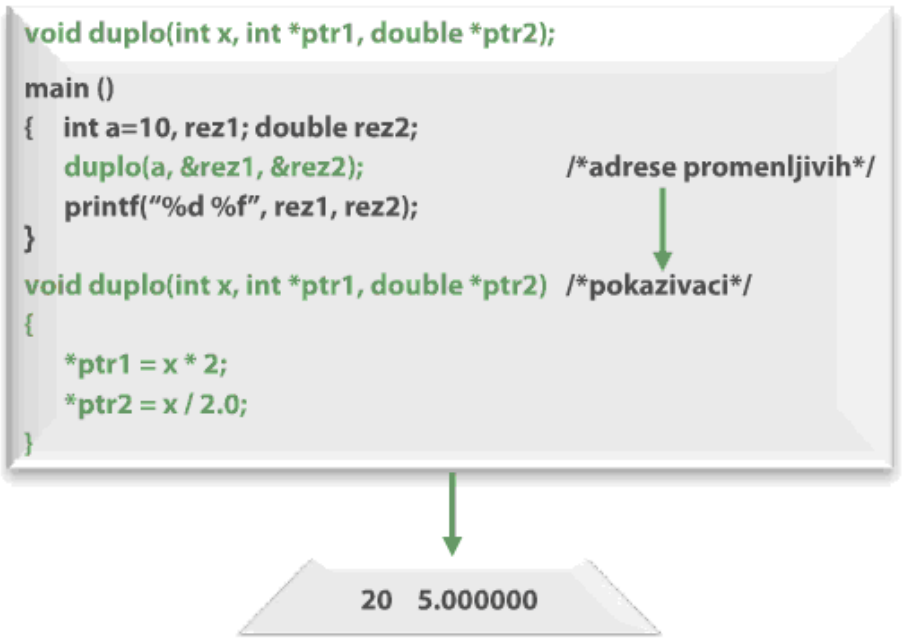
Слика 2.2 Прослеђивање аргумената функцији по референци

Предности прослеђивања аргумената функцији - по референци

Прослеђивање аргумената по референци значи да функција добија адресе оригиналних меморијских локација и има могућност измене њихових садржаја, слика 2.3. На овај начин од функције се може добити произвољан број резултујућих вредности (уписаних од адреса које су функцији прослеђене у аргументима), слика 2.4. Прослеђивање аргумената по референци уједно је и једини начин да једна функција приступи низу који је декларисан у некој другој функцији истог програма.



Слика 2.3 Измена вредности промењиве чија се адреса прослеђује функцији



Слика 2.4 Више резултујућих вредности преко аргумената функције

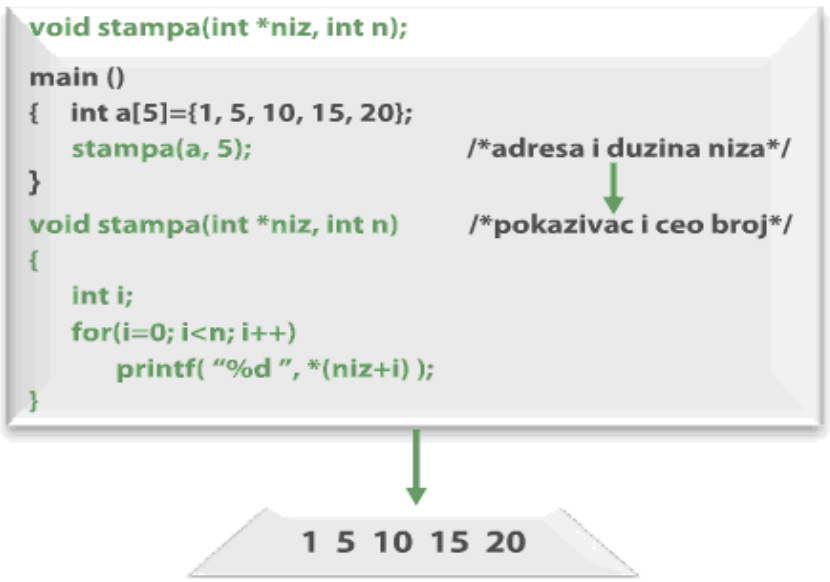
Аргумент функције - показивач на низ

За размену информација о низовима између појединих функција у програму користи се прослеђивање аргумената по референци. Низ обично има велики број елемената, тако да нема смисла да се функцији сви ти елементи прослеђују

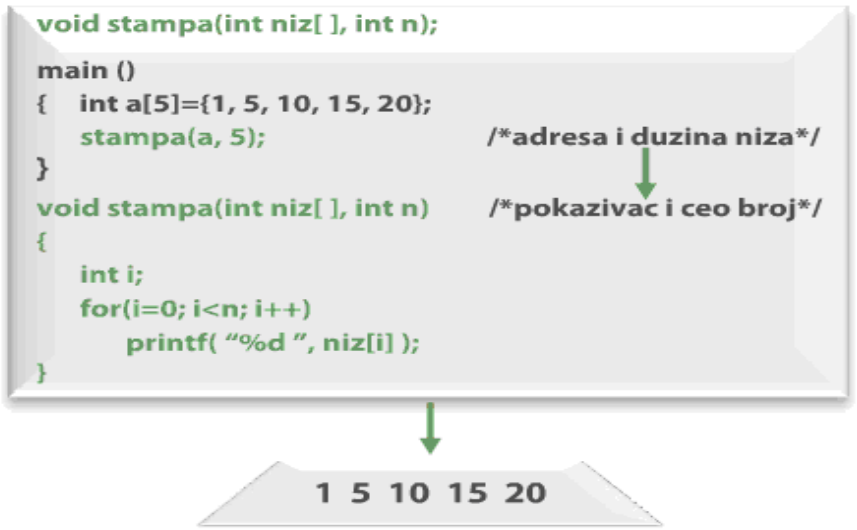
по вредности. Једна функција може имати приступ низу који је декларисан у некој другој функцији ако добије у аргументима: адресу одређеног типа низа и евентуално дужину низа (дужина низа није потребна ако се ради о *stringu* или ако је задато функцији да она на неки начин израчуна дужину низа).

Писање функција чији је аргумент показивач на низ

У језику C постоје два начина за писање функције која има аргумент показивач на низ. Један начин је следећи: у декларацији функције пишу се показивач на тип променљиве и број ових променљивих у низу *(tip *niz, int l)*, при позиву пишу се стварна адреса и дужина низа *(ime_niza, n)* и у дефиницији индиректно се приступа низу *(*(niz+i), i=0..n-1)*, слика 2.5. Други начин писања користи се само за низ: угласте заграде у декларацији *(tip niz[ ],int n)* и за директан приступ низу у *(niz[i], i=0..n-1)*. У позиву је и овде стварно име низа, слика 2.6.



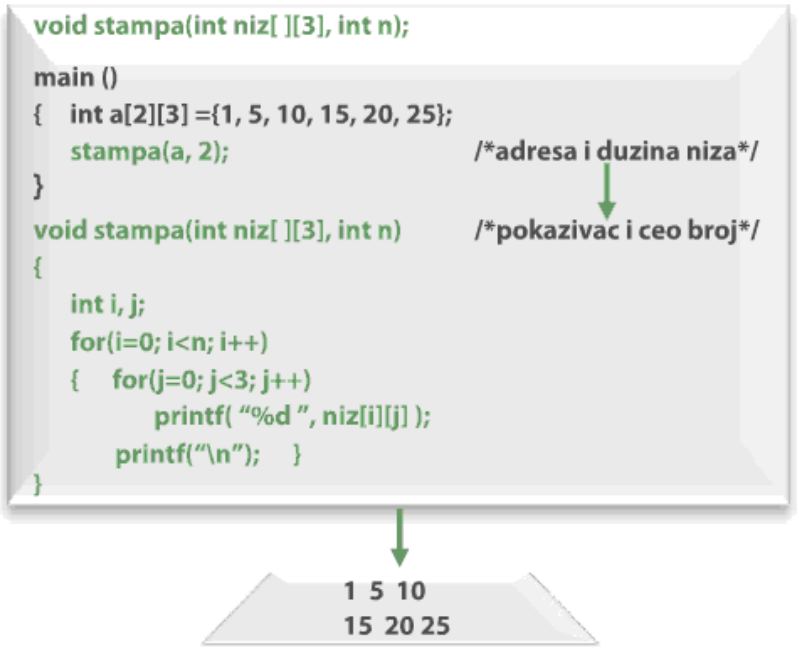
Слика 2.5 Аргумент функције – показивач на низ (1. начин писања)



Слика 2.6 Аргумент функције – показивач на низ (2. начин писања)

Аргумент функције – показивач на вишедимензионалан низ

Код вишедимензионалних низова могуће је прослеђивање функцији показивача на низ, само у овом случају неопходно је да буду познате све димензије изузев једне, да би се знало колико елемената има сваки низ у низу низова. Код показивача на дводимензионалан низ, на пример, декларација са угластим заградама пише се са два пара ових заграда, од којих један пар обавезно има и димензију *(tip niz_2d[ ][m], int l)* а приступ низу у функцији може бити директан *(niz[i][j], i=0..n-1, j=0..m-1).* У позиву је стварно име низа, слика 2.7.



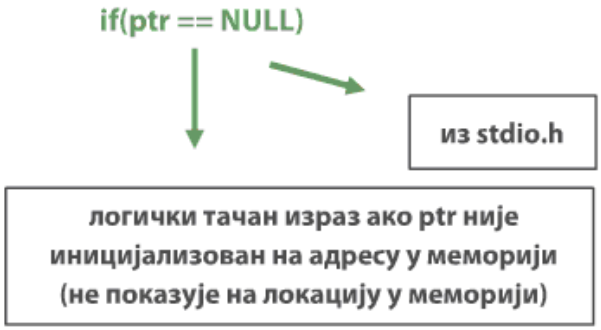
Слика 2.7 Аргумент функције – показивач на вишедимензионалан низ

Повратна вредност од функције – показивач

Поред функција које не враћају вредност или враћају вредност променљиве основног типа, у језику *C* дефинисане су и функције које враћају вредност показивача на било који тип променљиве (адресу променљиве), слика 2.8. Декларација овакве функције, испред имена функције има ознаку: показивач на тип *(tip *ime_funkcije(...);).* Дефиниција функције обавезно има наредбу *return* која враћа адресу на одговарајући тип променљиве. Резултат позива функције је адреса, слика 2.9 или *NULL* показивач у случају грешке, слика 2.10.

Слика 2.8 Показивач – повратна вредност од функције

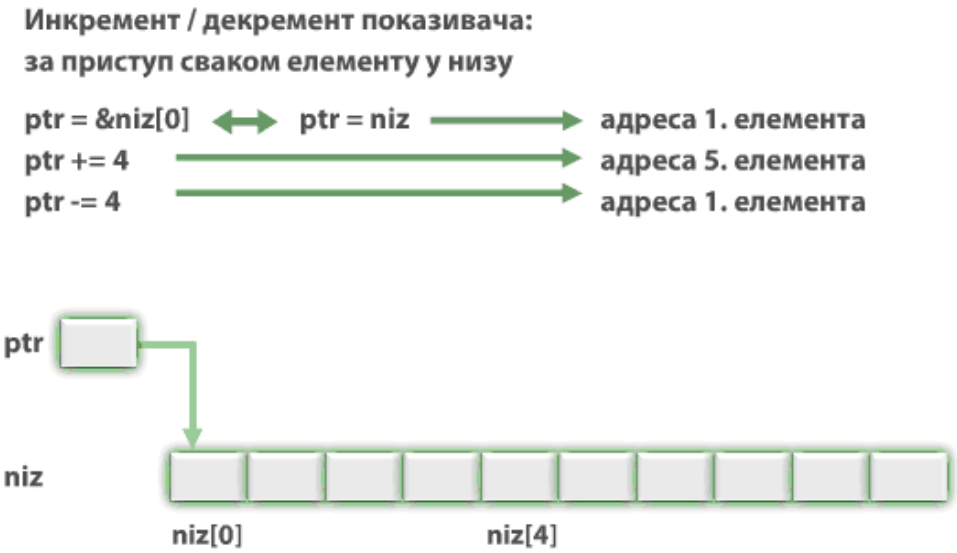
Слика 2.9 Наредба повратка из функције помоћу показивача



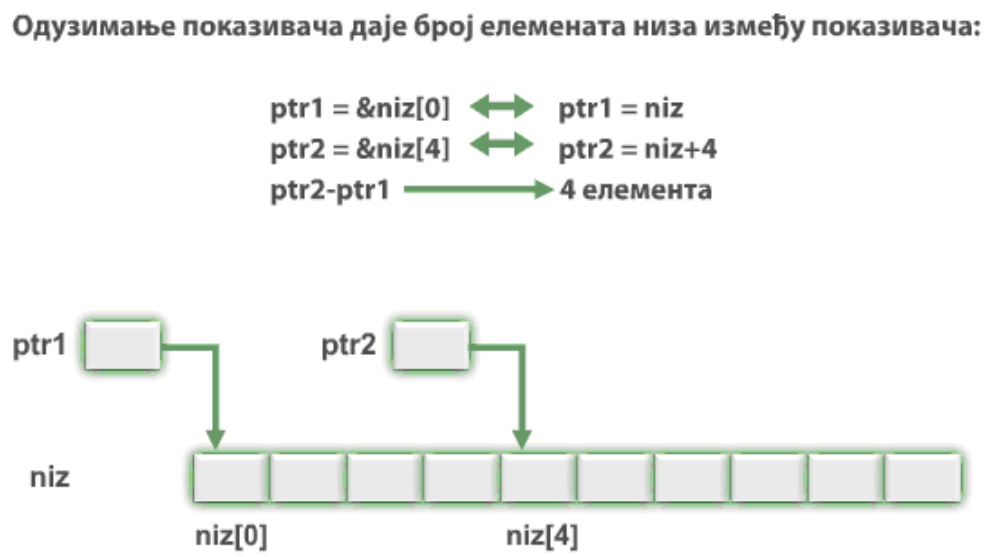
Слика 2.10 Показивач null – повратна вредност од функције

Аритметика показивача у функцијама

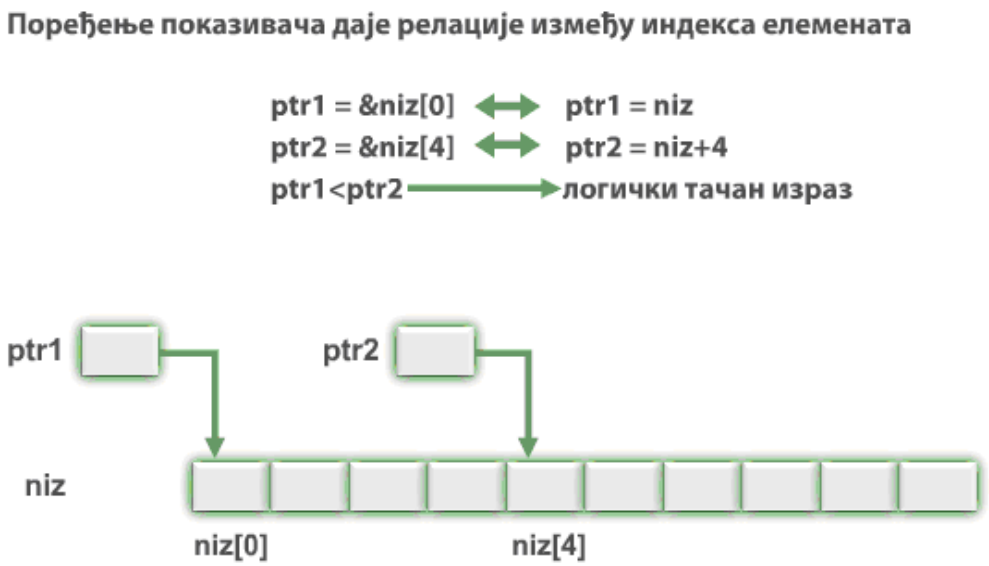
У функцијама које користе показиваче на низове примењује се аритметика показивача. Од свих дозвољених операција над показивачима, највише су заступљене следеће: инкремент / декремент – за индиректан приступ следећем / претходном елементу у низу (инкремент увећава а декремент умањује вредност показивача за величину типа низа у бајтовима), слика 2.11, одузимање – за одређивање броја елемената низа између показивача, слика 2.12, и поређење – за одређивање релација између показивача, слика 2.13.



Слика 2.11 Инкремент и декремент показивача



Слика 2.12 Одузимање показивача



Слика 2.13 Поређење показивача

2.2. Готове *string* функције које раде са показивачима

Функција *strlen*

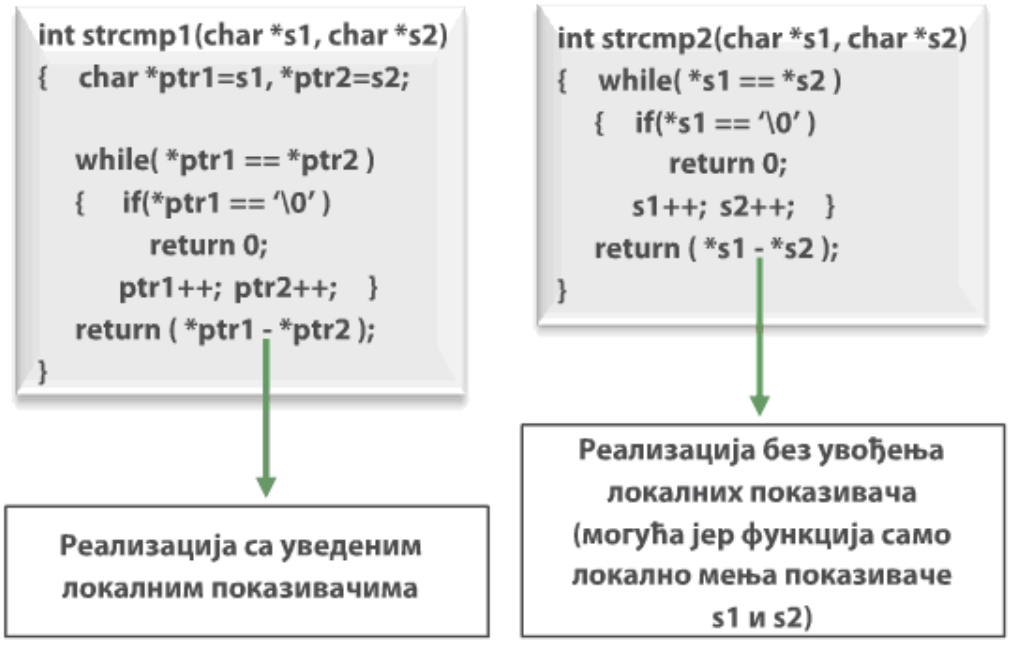
Резултат позива функције *strlen(s);* за име *stringa s* је дужина овог *stringa*. Функција добија у аргументу адресу *stringa* и ради тако што уводи један показивач *ptr на* тип карактер и иницијализује га на почетак *stringa s* (адресу првог карактера). Све док садржај карактера на који показује *ptr* није знак за крај *stringa (‘\0’)* показивач *ptr* се инкрементира на адресу следећег карактера.

Кад стигне до ознаке за крај *stringa*, функција враћа број карактера *stringa* без знака за крај (то је разлика вредности показивача: *ptr-s*), слика 2.14.

Слика 2.14 Реализација функције *strlen*

Функција *strcmp*

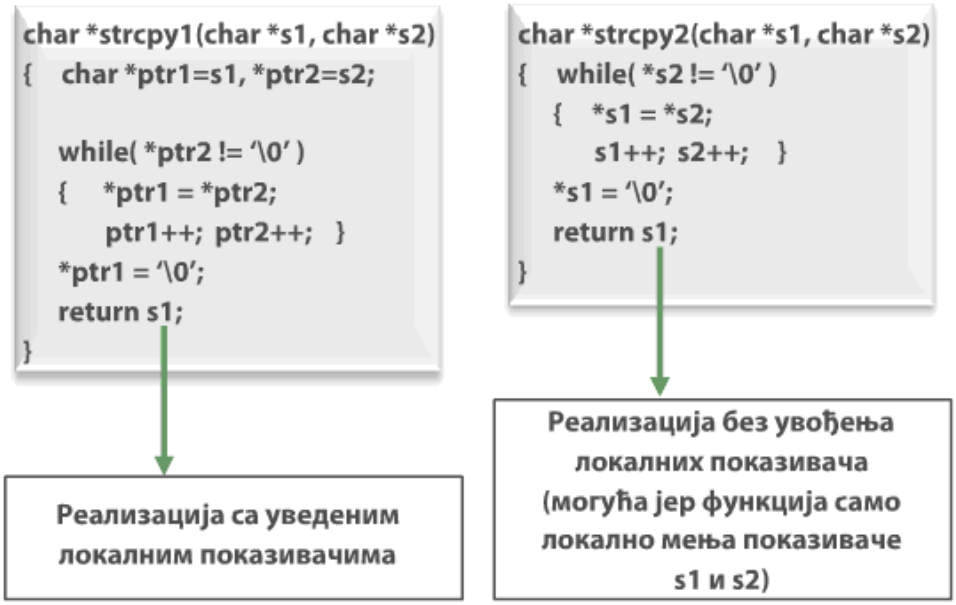
За поређење садржаја *stringова* чија су имена *s1* и *s2* користи се функција *strcmp(s1,s2)*. Функција добија у аргументима адресе *stringова*, уводи показиваче *ptr1* и *ptr2* и иницијализује их на ове адресе. Инкрементира оба показивача и пореди *ASCII* кодове одговарајућих карактера редом у *stringовима s1* и *s2* (на које показују *ptr1* и *ptr2*). Ако препозна разлику вредности у неком карактеру, враћа ту разлику а у супротном (ако су *stringови* међусобно исте дужине и садржаја) враћа вредност 0, слика 2.15.



Слика 2.15 Реализација функције *strcmp*

Функција *strcpy*

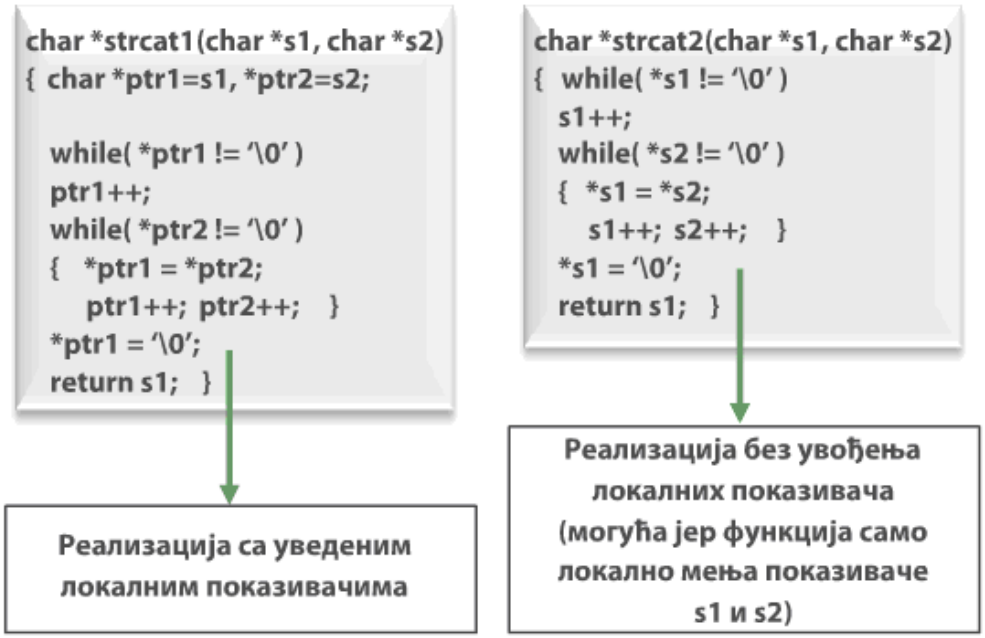
За копирање садржаја *stringa s2* у *string s1* користи се функција *strcpy(s1,s2).* Функција добија у аргументима адресе *stringova*, уводи показиваче *ptr1* и *ptr2* и иницијализује их на ове адресе. Све док не препозна крај у *stringu s2 (*помоћу показивача *ptr2)* садржај једног по једног његовог карактера додељује одговарајућем карактеру *stringa s1* и инкрементира показиваче *ptr1* и *ptr2*. Ако препозна крај у *stringu s2* додаје ознаку за крај *stringu s1* и враћа адресу *stringa s1* јер је то сада резултујући *string*, слика 2.16.



Слика 2.16 Реализација функције *strcpy*

Функција *strcat*

За надовезивање садржаја *stringa s2* на крај *stringa s1* користи се функција *strcat(s1,s2).* Функција добија у аргументима адресе *stringова*, уводи показиваче *ptr1* и *ptr2* и иницијализује их на ове адресе. Све док не препозна крај у *stringu s1* инкрементира показивач *ptr1* и доводи га до адресе знака за крај у овом *stringu.* Ова функција даље ради на потпуно исти начин као и функција *strcpy(s1,s2)*, само што се копирање не реализује од почетка већ од краја *stringa s1*. Стринг *s1* је резултујући и функција враћа његову адресу, слика 2.17.



Слика 2.17 Реализација функције *strcat*

2.3. Показивачи на функције

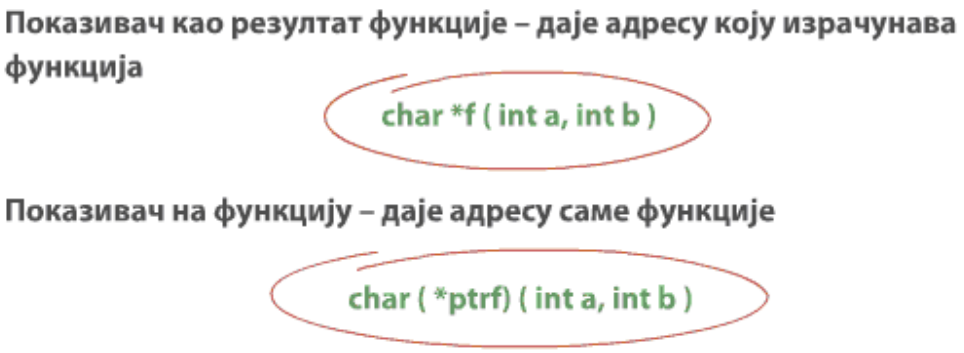
Декларација показивача на функцију

Када се програм покрене његове функције добијају простор у меморији и свака има своју адресу. Може се предвидети да се ове адресе у програму користе. Као и код осталих показивача за коришћење показивача на функцију потребно је написати декларацију и иницијализацију. Декларација показивача на функцију треба да има облик декларације функције на коју показује (исту ознаку типа који враћа и исту листу аргумената), само што уместо имена функције садржи између заграда симбол * и име показивача *(tip (* ptrf) (lista_form_arg))*.

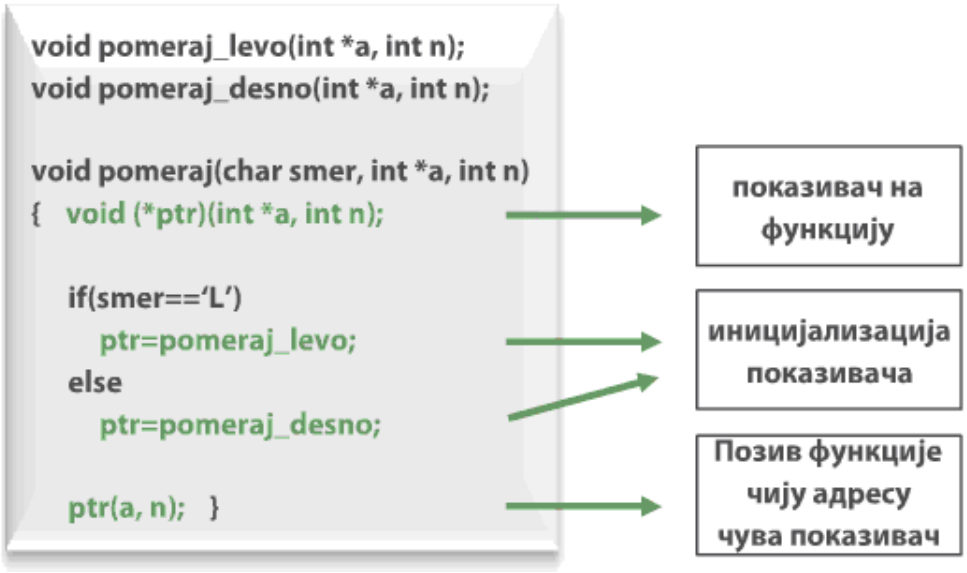
Позив функције помоћу показивача на исту функцију

Разлика између показивача на функцију и показивача од функције види се у декларацији (по употреби или не заграда), слика 2.18. Показивач на функцију користи се код селекције једне од више функција у програму, које су међусобно истог типа повратне вредности и исте листе формалних аргумената. Кад се показивач декларише, дефинишу се услови његове иницијализације на адресу

сваке од ових функција. Помоћу показивача позива се она функција на чију је адресу показивач иницијализован, слика 2.19.



Слика 2.18 Разлика између показивача на функцију и показивача од функције



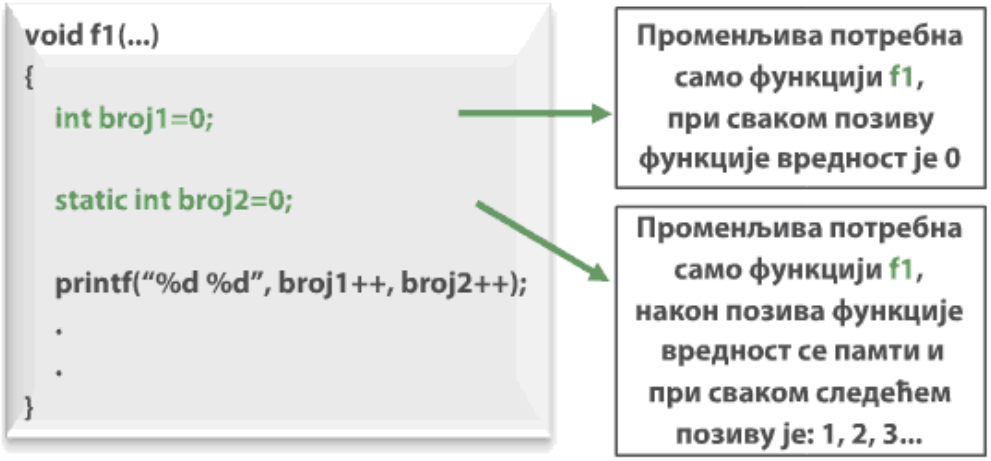
Слика 2.19 Показивач на функцију

2.4. Глобални и локални досег у програму

Локалне променљиве – аутоматске и статичке

Локална променљива декларисана је унутар неке функције програма и видљива је унутар те функције. Декларација унутар функције подразумеваног облика (*tip ime;* ⇔ *auto tip ime;)* резервише простор за аутоматску локалну променљиву и не иницијализује њен садржај. Статичка локална декларација *(static tip ime;)* иницијализује променљиву на 0, ако у програму није наведена нека друга

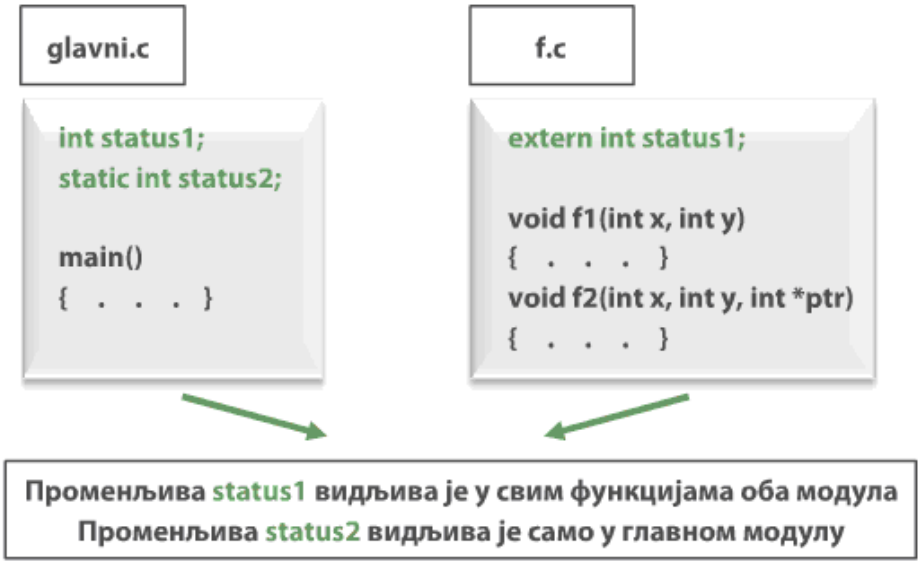
вредност, а између два позива функције чува достигнуту вредност промењиве (што значи да се промењива не иницијализује сваки пут када се функција позове), слика 2.20.



Слика 2.20 Локално декларисане промењиве

Глобалне промењиве – аутоматске, екстерне и статичке

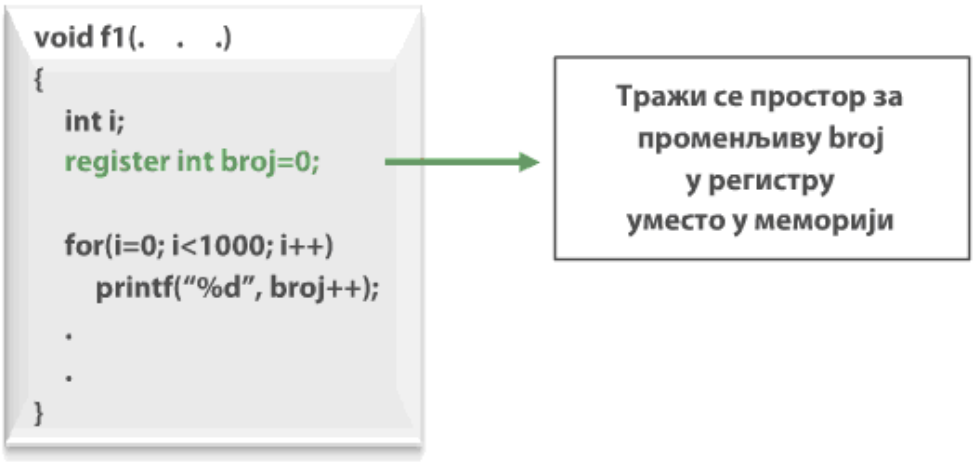
Глобална промењива је декларисана ван сваке функције програма. Аутоматска глобална декларација (*tip ime;* или *auto tip ime;)* је подразумевана, што значи да је промењива видљива у свим функцијама модула где је декларисана. Екстерна декларација исте промењиве у другом модулу програма *(extern tip ime;)* омогућава јој видљивост и у том модулу. Статичка глобална декларација *(static tip ime;)* не дозвољава екстерну декларацију. Глобална декларација обавезно и иницијализује промењиву на 0, слика 2.21.



Слика 2.21 Глобално декларисане променљиве

Регистарске променљиве

Регистарске променљиве су аутоматске променљиве за које се тражи резервисање простора у регистру процесора уместо у меморији. Ако се податак чува у регистру нема трошења времена на размену података између меморије и регистара. Декларација *register tip ime;* представља само захтев, а ако је регистар доступан преводилац одлучује да променљива буде регистарска. Овакав захтев корисно је писати за променљиве којима се често приступа у релативно кратком сегменту програма, слика 2.22.



Слика 2.22 Регистарске променљиве

Препоруке за писање декларација променљивих

Функција обично има своје локалне променљиве, које заузимају простор у меморији само за време њеног извршавања. Глобалне променљиве заузимају простор у меморији током читавог извршавања програма, због тога су оправдане само ако им приступа већи број функција програма и онда је то један начин за размену података између њих. За размену података између мањег броја функција програма, уместо глобалне декларације, може бити изабран механизам прослеђивања аргумената и добијања повратне вредности.

Ако нема посебних захтева променљиве су подразумевано аутоматске. За аутоматску променљиву, којој се често приступа у програму, а значајна је брзина извршавања програма, оправдано је тражити да буде регистарска. Ако је глобална променљива потребна функцијама из више модула декларише се као екстерна. За забрану приступа глобалној променљивој у неком другом модулу програма, потребно је декларисати је као статичку. Локална променљива декларише се као статичка, да би била сачувана њена вредност од једног до другог позива функције.

2.5. Питања

1. Ког типа може бити аргумент функције у *C* програму, да ли аргумент може бити само основни, или и изведени *C* тип?
2. Шта значи прослеђивање аргумената по вредности *C* функцији?
3. Шта значи прослеђивање аргумената по референци *C* функцији?
4. Које могућности има *C* функција којој се прослеђују аргументи по референци?
5. Да ли *C* функција којој се проследи показивач на променљиву, може само користити, само мењати, или користити и мењати вредност те променљиве?
6. Колико се *C* функцији највише адреса може проследити помоћу показивача?
7. Да ли се *C* функцији може проследити вредност показивача на било који *C* тип променљиве ?
8. Да ли се *C* функцији могу проследити информације о низу само помоћу показивача или не?
9. Шта је *C* функцији у аргументима неопходно, када треба да ради са низом бројева који није декларисан у тој функцији?
10. Шта је *C* функцији у аргументима неопходно, када треба да ради са stringom који није декларисан у тој функцији?
11. Да ли и како функција strlen() у свом аргументу користи показиваче?
12. Да ли и како функција strcmp() у својим аргументима користи показиваче?
13. Да ли и како функција strcpy() у својим аргументима и повратној вредности користи показиваче?
14. Да ли и како функција strcat() у својим аргументима и повратној вредности користи показиваче?
15. Које су основне препоруке за писање декларација променљивих, глобалних и локалних?

3. Динамичка додела меморије у програмима на језику C

Кратак преглед лекције

Резервисање меморијског простора за статички декларисане промењиве под контролом је преводиоца. Овако додељен меморијски простор није изменљив током извршавања програма. Да би меморија била искоришћена на што ефикаснији начин, за велике објекте програма постоји могућност динамичке доделе меморије. Динамичка додела (алокација) меморије остварује се током извршавања програма, а помоћу готових функција из стандардне библиотеке *stdlib: malloc(), calloc(), realloc(), free().*

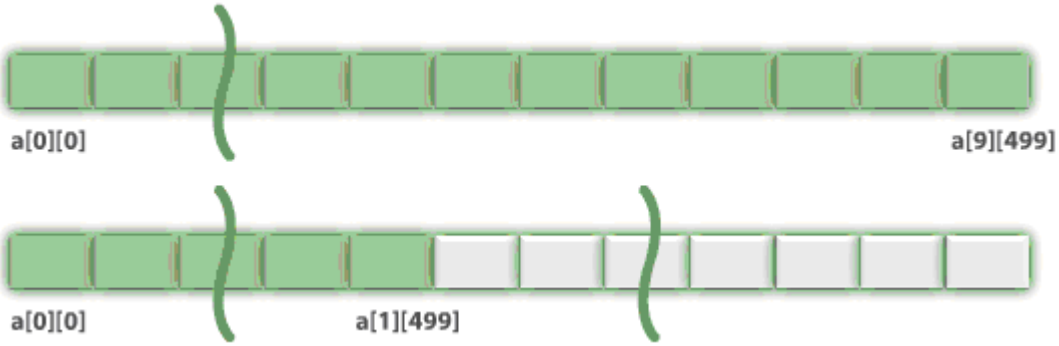
3.1. Статичка и динамичка додела меморије

Ограничења код статичке доделе меморије

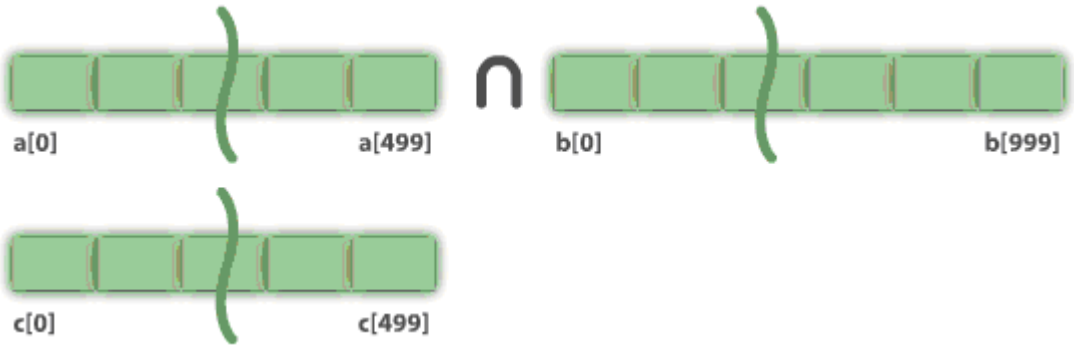
Меморијски простор који је резервисан за статички декларисане промењиве не може се мењати а ни укидати током извршавања програма, слика 3.1. Овај начин резервисања меморије није погодан: ако се користи много мањи број података од дозвољеног, слика 3.2, ако се неки комплет података више не користи од одређене тачке програма, слика 3.3, ако у свим деловима програма нема потребе за међусобно истим комплетима података, на пример уместо два низа промењивих потребан је њихов пресек, слика 3.4, или унија, слика 3.5...

Слика 3.1 Ограничења код статичке доделе меморије

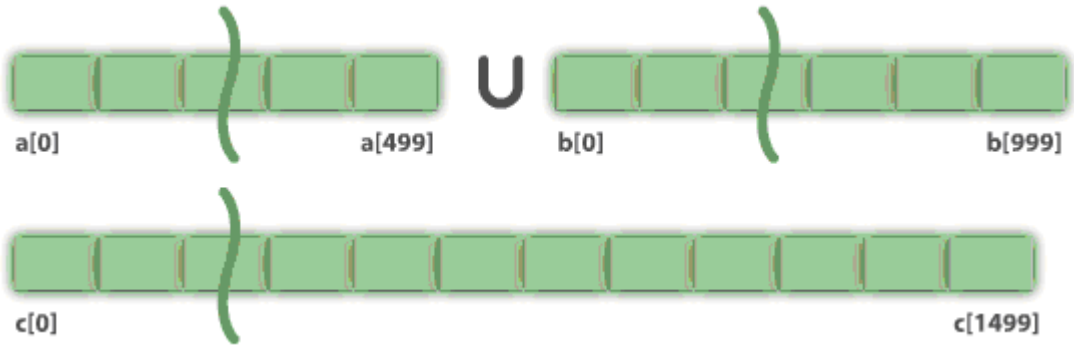
Слика 3.2 Коришћење много мањег броја елемената низа од максималног



Слика 3.3 Коришћење различитих делова низа у различитим деловима програма



Слика 3.4 Коришћење пресека низова



Слика 3.5 Коришћење уније низова

Меморијска *heap* зона

Сваки програм добија у меморији три зоне. Једна је *permanent storage* зона за изворни кôд програма, глобалне и статички декларисане промењљиве. Друга је *stack* зона за локалне промењљиве потребне функцијама програма. Трећа зона је смештена између две поменуте и то је *heap* зона која се динамички додељује, слика 3.6. Меморијска *heap* зона је промењљиве величине, због увођења и укидања простора за локалне промењљиве појединих функција, слика 3.7 и неопходна је провера слободног простора у овој меморијској зони.

Слика 3.6 Резервисање меморијских зона потребних програму

Слика 3.7 Промена величине појединих меморијских зона

Примена показивача код динамичке доделе меморије

Ако има слободног простора у *heap* меморијској зони овај простор може бити динамички додељен било ком објекту програма. За приступ објекту у динамички додељеном меморијском простору потребно је обезбедити показивач који ће чувати адресу тог додељеног меморијског простора. Стандардне *C* функције за динамичку доделу меморије само у комбинацији са показивачима могу омогућити узимање и враћање меморијског простора, по потреби и током извршавања програма.

3.2. Функције за динамичку доделу меморије

Функције за доделу, измену и укидање меморије

У *C* библиотеци *stdlib.h* постоје функције помоћу којих се тражи у меморији: динамичка додела простора *(malloc()),* динамичка додела и иницијализација простора *(calloc())* и измена динамички додељеног простора *(realloc()).* Ако се помоћу неке од ових функција тражи простор који је већи од расположивог простора у *heap* зони функција враћа показивач *NULL.* У супротном, функција одради свој задатак и враћа адресу траженог простора. За ослобађање динамички додељеног простора позива се функција *free(),* слика 3.8.

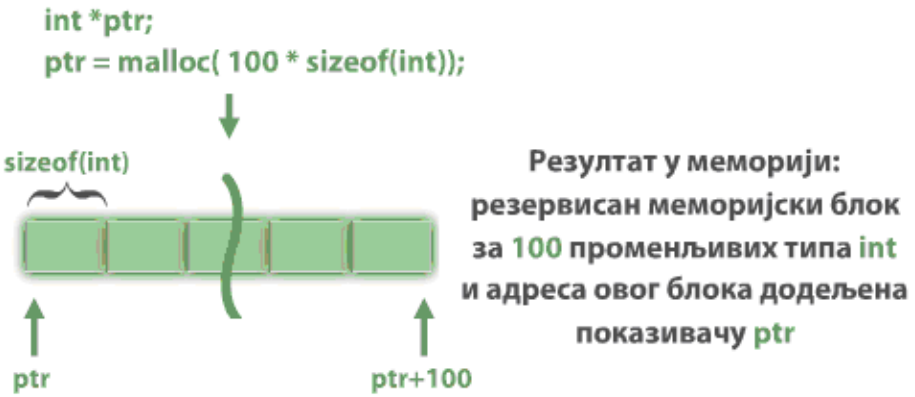
Функција	Задатак који решава
malloc()	Динамичка додела меморије за блок одређеног броја бајтова
calloc()	Динамичка додела меморије за одређени број блокова и иницијализација сваког бајта у сваком блоку на 0
realloc()	Измена величине претходно динамички додељене меморије на одређени број бајтова
free()	Ослобађање претходно динамички додељене меморије

Слика 3.8 Функције за динамичку доделу, промену и ослобађање меморије

Функција *malloc*

Позивом функције *malloc()* током извршавања програма тражи се додела (алокација) континуалног меморијског простора за одређени број бајтова. Аргумент функције је цео број бајтова потребног меморијског простора а резултат је генерички показивач на почетак додељеног простора или *NULL* ако захтев није одобрен, слика 3.9. Уобичајено је да се да се функцији *malloc*() прослеђује број бајтова као резултат оператора *sizeof(tip)* а меморијски простор који се овако додељује није иницијализован, слика 3.10.

Слика 3.9 Декларација функције за динамичку доделу меморије

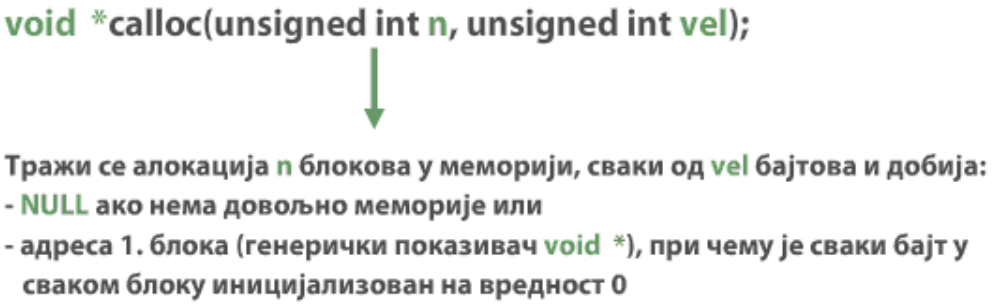


Слика 3.10 Позив функције за динамичку доделу меморије

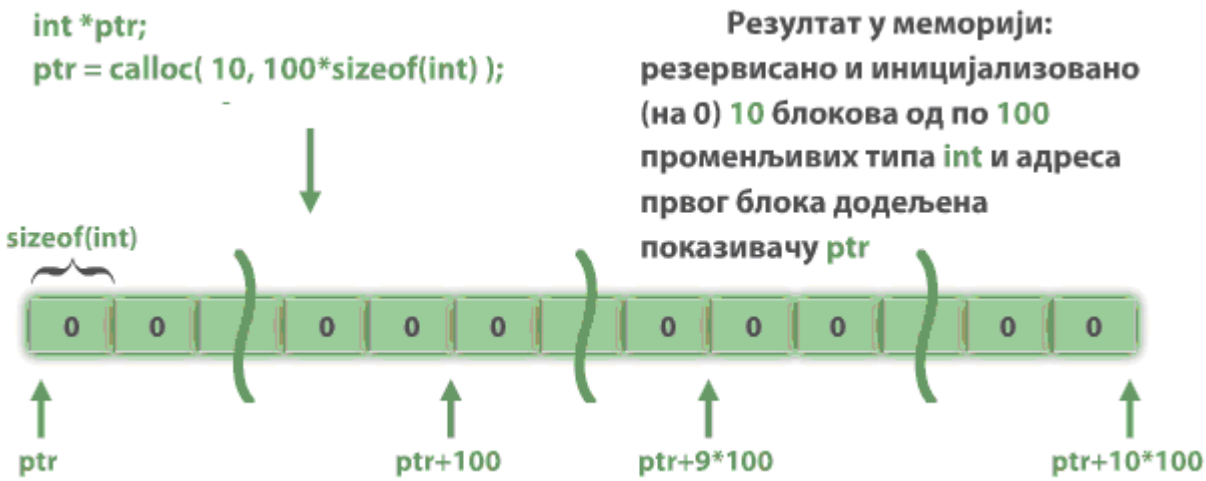
Функција *calloc*

Позивом функције *calloc()* током извршавања програма тражи се додела (алокација) континуалног меморијског простора за одређени број блокова одређене величине. Аргументи функције су: број блокова и број бајтова сваког блока потребног меморијског простора. Резултат је генерички показивач на

почетак додељеног простора или *NULL* ако захтев није одобрен, слика 3.11. И ова функција добија број бајтова као резултат оператора *sizeof(tip)* а додељен меморијски простор је иницијализован на вредност 0, слика 3.12.



Слика 3.11 Декларација функције за динамичку доделу и иницијализацију меморије



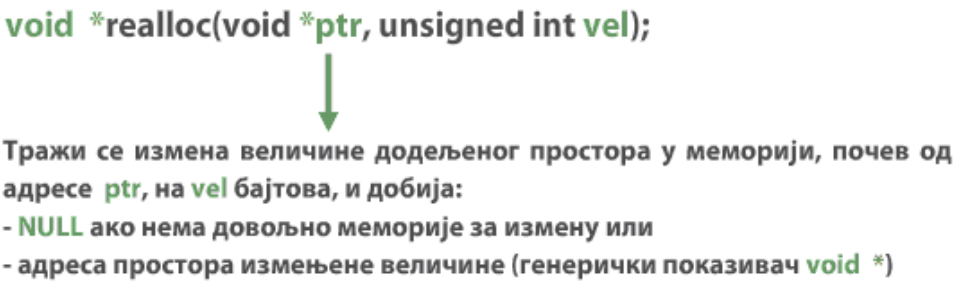
Слика 3.12 Позив функције за динамичку доделу и иницијализацију меморије

3.3. Функције за измену и ослобађање динамички додељене меморије

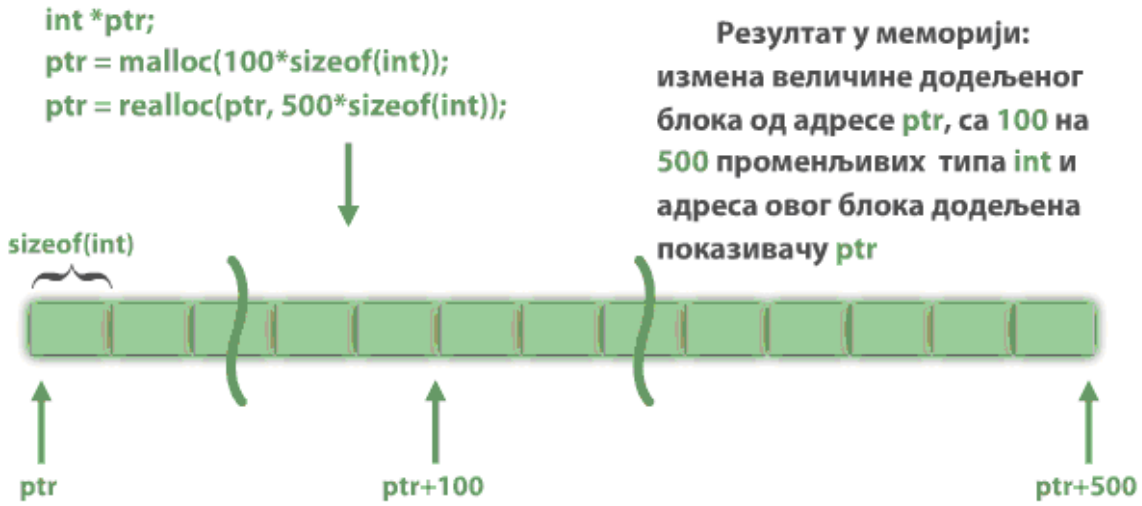
Функција *realloc*

Позивом функције *realloc()* током извршавања програма тражи се измена величине (реалокација) динамички додељеног меморијског простора. Аргументи функције су: адреса меморијског простора и број бајтова на који се тражи измена. Резултат је генерички показивач на почетак измењеног простора или *NULL* ако захтев није одобрен, слика 3.13. Услед повећања и смањења

меморијског простора задржани бајтови не мењају вредности. Код повећања меморијског простора додати бајтови нису иницијализовани, слика 3.14.



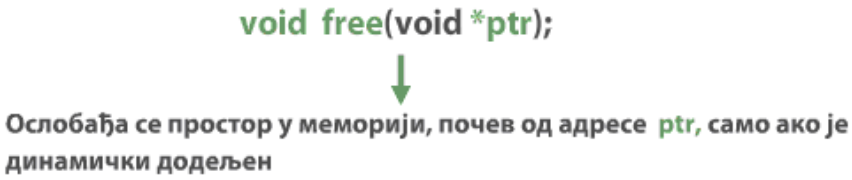
Слика 3.13 Декларација функције за измену величине динамички додељене меморије



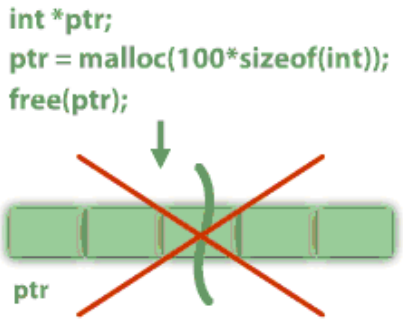
Слика 3.14 Позив функције за измену величине динамички додељене меморије

Функција *free*

Ослобађање меморијског простора могуће је само ако је исти меморијски простор динамички додељен. Један начин је позив функције *realloc()* за промену величине меморијског простора на 0 бајтова. Други, једноставнији и уобичајени начин је позив функције *free()*. Једини аргумент ове функције јесте адреса меморијског простора који треба да се ослободи, слика 3.15. Позив функције *free()* ослобађа комплетан број бајтова који је динамички додељен од тражене адресе, слика 3.16 и не даје повратну вредност.



Слика 3.15 Декларација функције за ослобађање динамички додељене меморије



Слика 3.16 Позив функције за ослобађање динамички додељене меморије

Примена функција за доделу, измену и укидање меморије

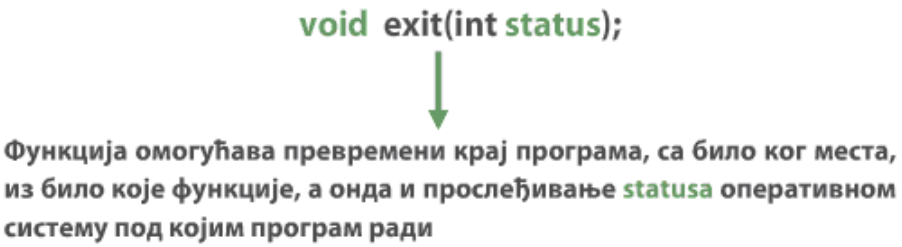
Ако за тражену динамичку доделу, слика 3.17, или измену, слика 3.18, нема расположивог простора у меморији потребно је предвидети: поруку на екрану и прекид даљег извршавања програма помоћу функције *exit()* из библиотеке *stdlib.h*, слика 3.19. Функција *exit()* прослеђује вредност свог аргумента (1 - грешка а 0 – нема грешке) оперативном систему под којим ради, слика 3.20. Ако има довољно простора у меморији за динамичку доделу и измену, програм користи у једном свом делу тај простор и онда га ослобађа, слика 3.21.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
main()
{  char *bafer;
   bafer = malloc( 12 );
   if( bafer == NULL )  { printf("\nGreska alokacije"); exit(1); }
  printf("\nKreiran bafer velicine 12 bajtova");
  strcpy(bafer, "Jedan tekst");
  printf("\nSadrzaj bafera: %s", bafer);
```

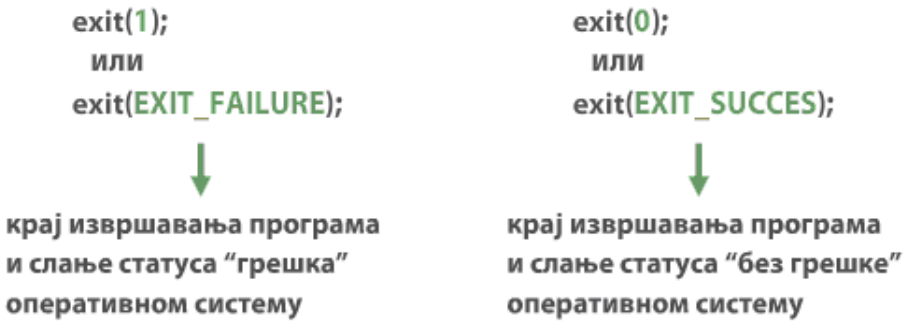
Слика 3.17 Пример алокације меморије (1. део изворног кода)

```
bafer = realloc( bafer, 24 );
if(bafer == NULL) { printf("\nGreska realokacije"); exit(1); }
printf("\nVelicina bafera promenjena na 24 bajtova" );
printf("\nBafer jos uvek sadrzi: %s", bafer);
strcat(bafer, "Drugi tekst");
printf("\nBafer sada sadrzi: %s\n", bafer);
free(bafer);      }
```

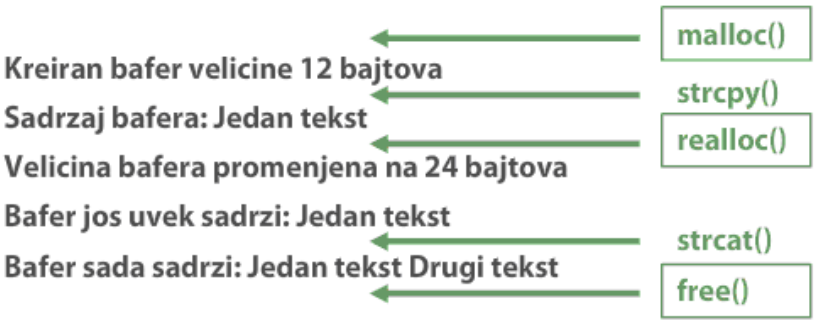
Слика 3.18 Пример реалокације и ослобађања меморије (2. део изворног кода)



Слика 3.19 Декларација функције exit()



Слика 3.20 Могући начини позива функције exit()



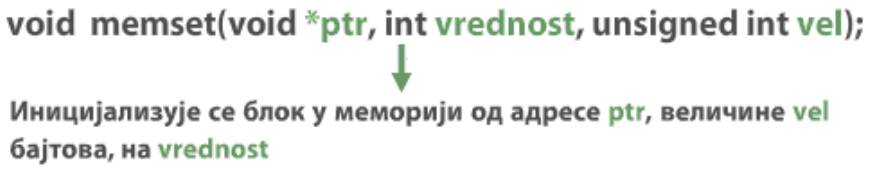
Слика 3.21 Пример коришћења и ослобађања меморије из програма

Функције *memset* и *memcpy*

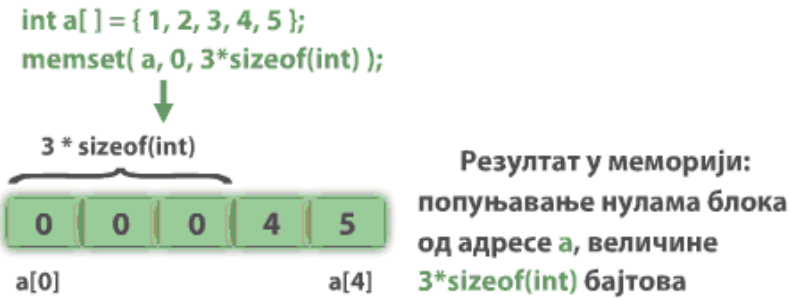
У *C* библиотеци *memory.h* постоје функције *memset()* и *memcpy()* које могу поједноставити и убрзати рад са блоковима података у меморији, слика 3.22. Функција *memset()* добија у аргументима адресу блока, вредност и број бајтова у блоку, слика 3.23 и поставља на задату вредност сваки бајт у задатом меморијском блоку, слика 3.24. Функција *memcpy()* добија у аргументима адресе два блока и број бајтова, слика 3.25 и копира задати број бајтова из блока чија је адреса задата у 2. аргументу у блок чија је адреса задата у 1. аргументу, слика 3.26.

Функција	Задатак који решава
memset()	Иницијализација свих бајтова меморијског блока на одређену вредност
memcpy()	Копирање одређеног броја бајтова једног меморијског блока у други

Слика 3.22 Функције memset() и memcpy()

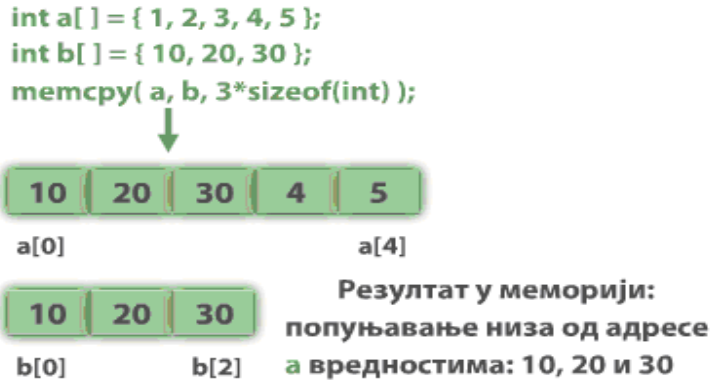


Слика 3.23 Декларација функција memset()



Слика 3.24 Иницијализација меморијског блока на задату вредност

Слика 3.25 Декларација функција memcpy()



Слика 3.26 Копирање садржаја једног меморијског блока у други

3.4. Динамичка додела меморије за низове

Једнодимензионалан низ

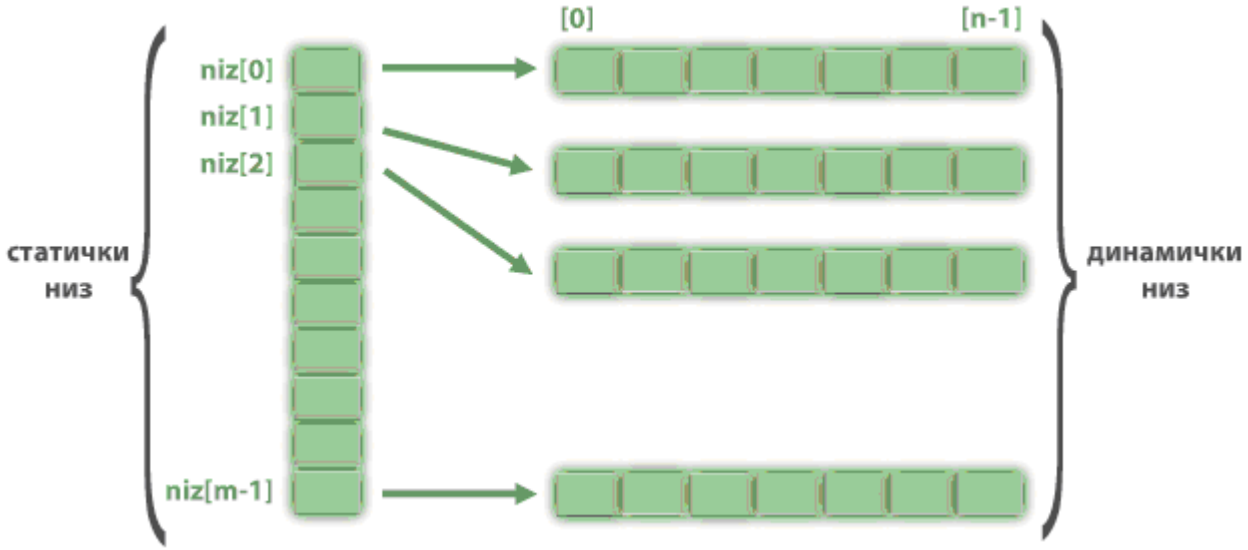
У неким случајевима статички декларисани низ може непотребно заузимати простор у меморији, слика 3.27: на почетку програма (функције) није позната дужина низа већ само његова максимално дозвољена дужина, у различитим деловима програма користе се само поједини делови низа, низ је веома велике дужине а од одређене тачке у програму уопште се не користи... У таквим случајевима статичку декларацију замењује алокација меморије за низ, која се код једнодимензионалног низа остварује на већ описани начин, слика 3.28.

Слика 3.27 Статичка декларација низа

Слика 3.28 Алокација меморије за једнодимензионалан низ

Дводимензионалан низ - помоћу низа показивача

Алокација меморије за дводимензионалан низ димензија *mxn* значи динамичку доделу меморије за *m* низова од по *n* променљивих. Постоје два начина за ово. Један је статичка декларација низа од *m* показивача, алоцирање меморије за *m* низова од по *n* елемената и иницијализација показивача на адресе низова, слика 3.29. Меморија се алоцира сваком од ових низова редом а сваком показивачу у низу *(niz[i], i=1..m-1)* додељује се адреса једног од n низова. Алоцирана меморија ослобађа се за један по један низ редом, слика 3.30.



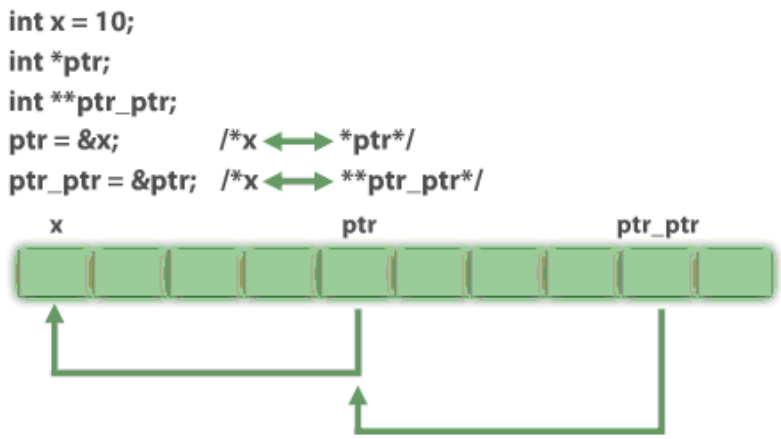
Слика 3.29 Један начин алокације меморије за дводимензионалан низ, помоћу низа показивача

```
int *niz[10];
int m, n;
scanf("%d%d", &m, &n)
for(i=0; i<m; i++)
  { niz[i] = malloc(n * sizeof(int));
    if( niz[i]==NULL ) { printf("\nGreska"); exit(1); }
/*   pristup elementu niza: niz[i][j], i=0...m-1, j=0...n-1  */
for(i=0; i<m; i++)
  free( niz[i] );
```

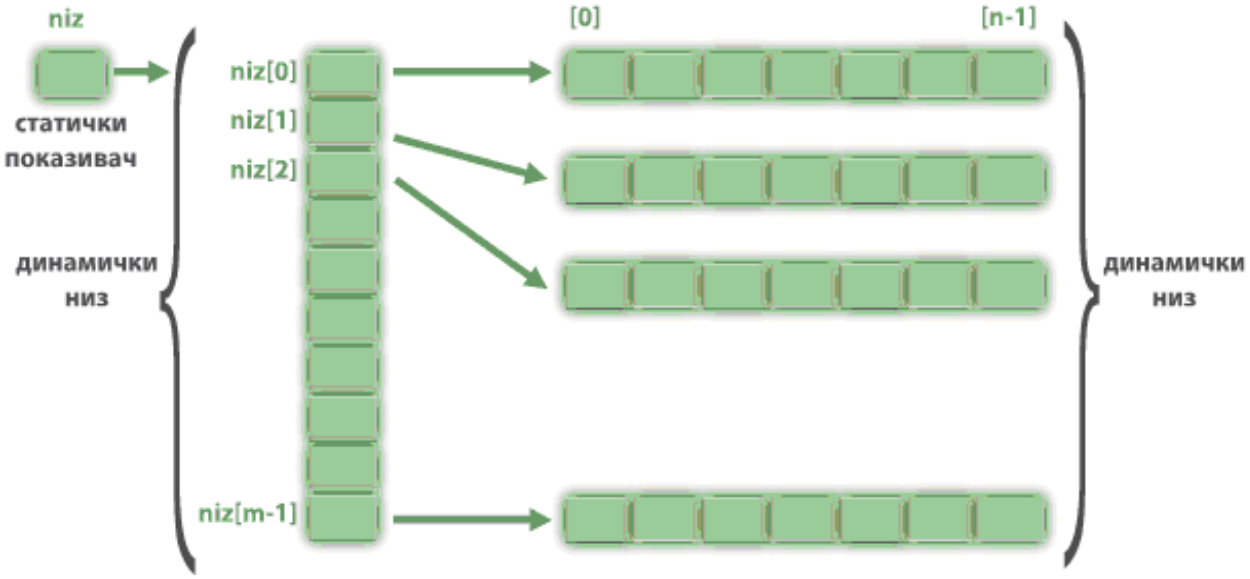
Слика 3.30 Пример алокације меморије за дводимензионалан низ, помоћу низа показивача

Дводимензионалан низ - помоћу показивача на показивач

За други начин алокације меморије уводи се показивач на показивач (променљива која чува адресу другог показивача), слика 3.31. Други начин алокације меморије за дводимензионалан низ димензија *mxn* разликује се од првог по томе што се низ од *m* показивача не декларише статички, већ се на почетку алоцира меморија и за овај низ и његова адреса додељује показивачу (јединому који је у овом случају статички), слика 3.32. Предност другог начина је могућност измене броја низова *m* током извршавања програма, слика 3.33.



Слика 3.31 Показивач на показивач (потребан за други начин алокације меморије за дводимензионалан низ)



Слика 3.32 Други начин алокације меморије за дводимензионалан низ, помоћу показивача на показивач

```
int **niz;
int m, n;  scanf("%d%d", &m, &n);
niz = malloc( m*sizeof( int * ) );
if( niz==NULL ) {  printf("\nGreska");  exit(1); }
for(i=0; i<m; i++)
{  *( niz+i ) = malloc( n * sizeof( int ) );
   if( *( niz+i )==NULL ) {  printf("\nGreska");  exit(1); }
/*   pristup elementu niza: *(*(niz+i)+j), i=0...m-1, j=0...n-1   */
for(i=0; i<m; i++)
  free( *( niz+i ) );
free(niz);
```

Слика 3.33 Пример алокације меморије за дводимензионалан низ, помоћу показивача на показивач

Динамичко израчунавање величине низа

Често током извршавања програма постоји потреба за израчунавањем величине низа у бајтовима. Величина статички декларисаног низа може се израчунати применом оператора *sizeof()* над самим именом низа. Код низа за који је алоцирана меморија име низа је уједно и име показивача на тај низ и примена оператора *sizeof()* над тим именом даје величину показивача. Величина низа

може се израчунати као производ броја елемената и броја бајтова сваког елемента, слика 3.34.

Слика 3.34 Израчунавање дужине низа за потребе динамичке доделе меморије

3.5. Питања

1. Коју *C* библиотеку је потребно укључити, ако су потребне функције за динамичку доделу меморије?
2. Који су разлози динамичке доделе (алокације) меморије из програма?
3. Да ли је коришћење функција за динамичку доделу меморије могуће предвидети из било које функције програма или само из главне функције?
4. Да ли је и на који начин код динамичке доделе меморије потребно користити показиваче?
5. Шта користи од аргумената и како ради функција malloc()?
6. Која је разлика између функција malloc() и calloc()?
7. Да ли функција malloc() аутоматски иницијализује меморијски простор?
8. Шта користи од аргумената и како ради функција realloc()?
9. Под којим условима се може користити функција realloc()?
10. Да ли функција realloc(), под условом да се користи за проширење додељеног меморисјког простора, мења садржај задржаног дела простора?
11. Да ли функција realloc(), под условом да се користи за проширење додељеног меморисјког простора, мења садржај задржаног дела простора?
12. На који начин се може проверити да ли је функција malloc() / realloc() извршила тражену доделу / измену меморијског простора?
13. Шта користи од аргумената и како ради функција free()?
14. Под којим условима се може користити функција free()?
15. Да ли се функције realloc() и free() могу користити и у деловима меморијског простора који је статички додељен из програма?

4. Структуре података у програмима на језику C

Кратак преглед лекције

Сви подаци који имају неку међусобну логичку везу у програму због једноставнијег приступа могу бити груписани у структуру. Структура представља колекцију промењљивих, међусобно истог или различитог типа, које су логички повезане у једну целину. Тип структуре је изведени тип података. Промењљиве једне структуре су чланови структуре, а за приступ овим члановима користе се: име структуре и оператор за приступ члану структуре.

4.1. Појам структуре

Структура је колекција промењљивих које описују један објекат у програму и које су груписане због лакшег приступа. Промењљиве једне структуре могу имати и међусобно различите типове (што код низа није био случај). Структура се сматра у језику *C* изведеним типом података који може бити и веома комплексан. Тип структуре послужио је за развој *C++* апстрактног типа класе који проширује особине структура (а представља основни концепт објектно-оријентисаног пројектовања програма).

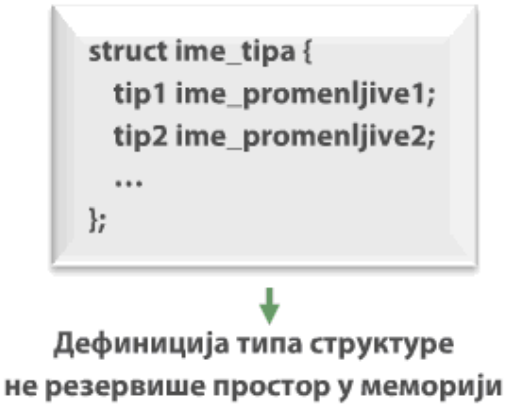
Чланови структуре

Промењљиве које су груписане у један тип структуре представљају чланове те структуре. Чланови структуре могу бити промењљиве најразличитијих типова: промењљиве основног типа, промењљиве показивачи, низови промењљивих, друге структуре... слика 4.1. Свим члановима заједничко је име структуре дефинисаног типа, а сваки члан има и своје јединствено име. Приступ било ком члану структуре могућ је помоћу имена структуре дефинисаног типа, оператора приступа члану и имена члана структуре.

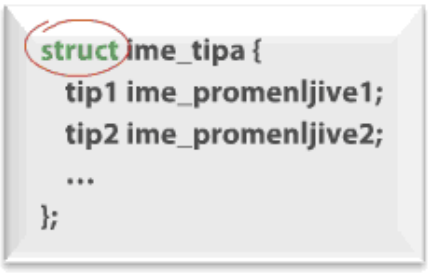
Слика 4.1 Чланови структуре

Дефиниција типа структуре

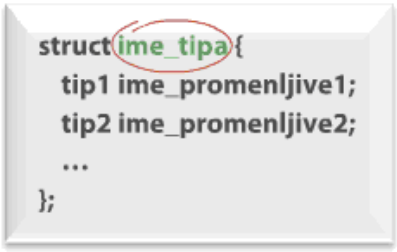
Дефиниција структуре не резервише простор у меморији већ само преводиоцу пружа информације о типу структуре који ће бити коришћен у програму, слика 4.2. Ова дефиниција садржи службену реч *struct*, слика 4.3, онда ознаку типа структуре *(tag)*, слика 4.4, декларацију чланова структуре између витичастих заграда, слика 4.5 и ознаку за крај дефиниције (знак ;), слика 4.6. Ако је више чланова међусобно истог типа њихова декларација може бити написана у једној наредби, слика 4.7.



Слика 4.2 Општи облик дефиниције типа структуре



Слика 4.3 Резервисана *C* реч *struct* имплицира да ће се користити структура



Слика 4.4 Ознака структуре (*tag*) је идентификатор типа структуре

Слика 4.5 Унутар дефиниције типа структуре декларисани су њени чланови

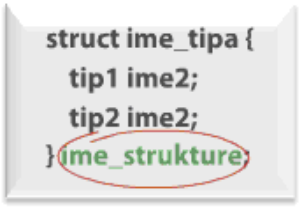
Слика 4.6 Дефиниција типа структуре завршава се знаком ;

Слика 4.7 Декларација чланова структуре може бити сажета

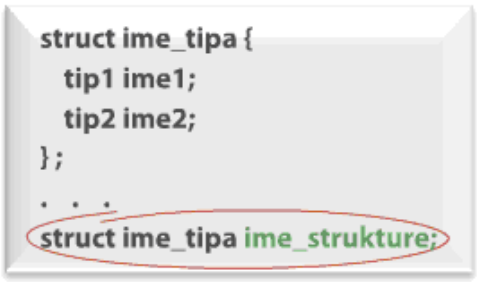
Декларације промењиве дефинисаног типа структуре

За декларацију промењиве дефинисаног типа структуре постоје два начина. Један начин је унутар дефиниције, слика 4.8. Други начин који се чешће користи, слика 4.9., на једном месту (најчешће глобално) дефинише тип

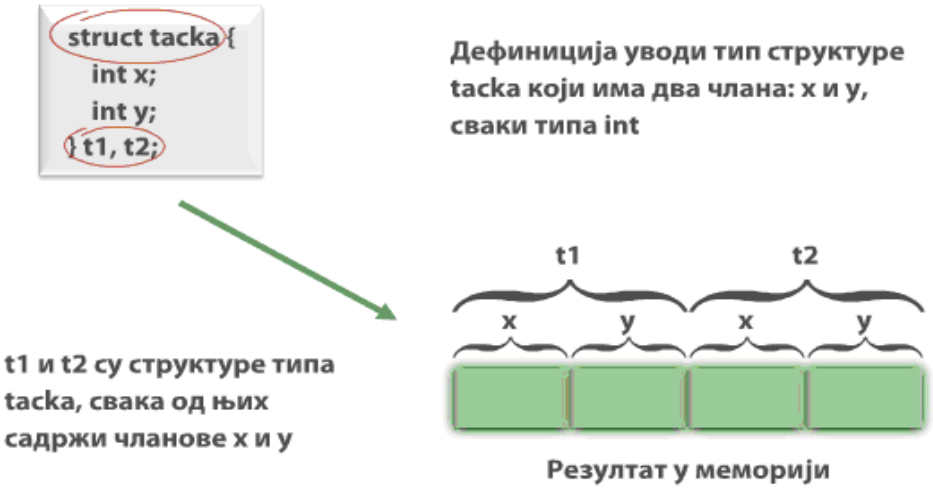
структуре а на произвољном броју места уводи промењљиве тог типа. За сваки тип структуре може бити уведен произвољан број промењљивих на било који од наведених начина, слика 4.10 и слика 4.11. Даље у тексту, за промењљиву дефинисаног типа структуре биће коришћен термин структура.



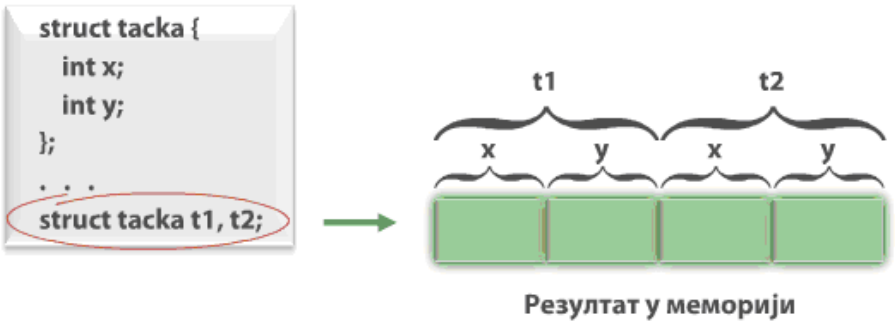
Слика 4.8 Декларација структуре унутар дефиниције типа структуре



Слика 4.9 Декларација структуре ван дефиниције типа структуре



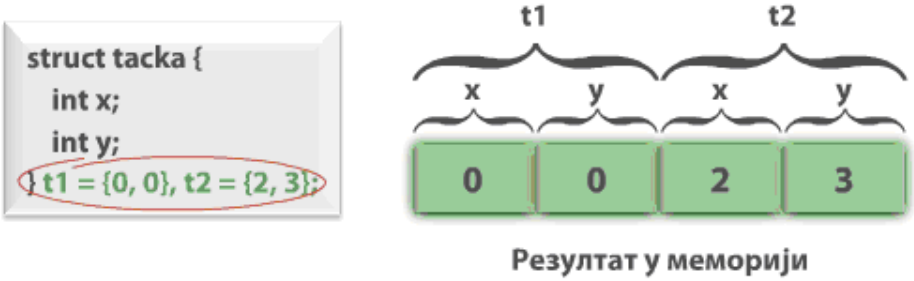
Слика 4.10 Пример декларације структура унутар дефиниције типа



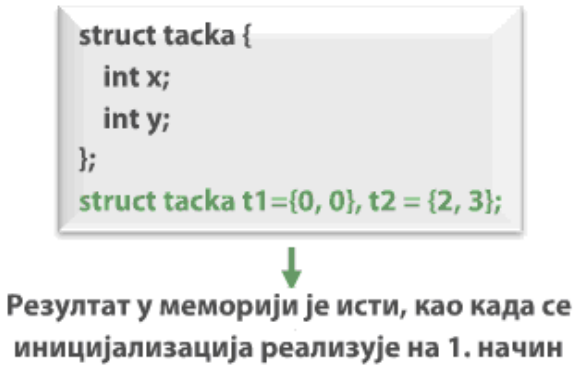
Слика 4.11 Пример декларације структура ван дефиниције типа

Иницијализација структуре

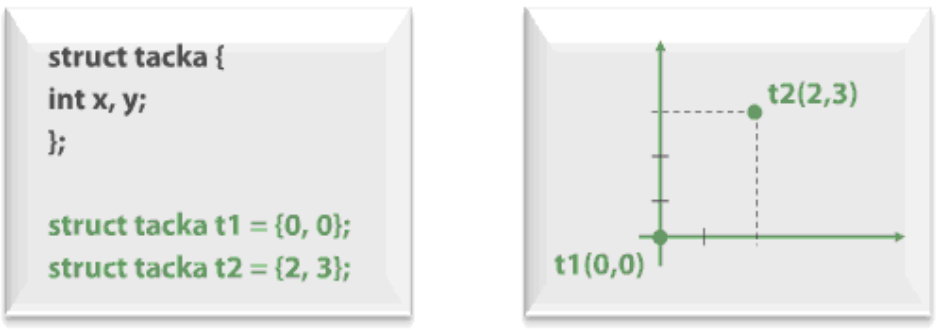
Иницијализација структуре може бити написана, као и њена декларација, унутар дефиниције типа структуре, слика 4.12, или ван ове дефиниције, слика 4.13. На пример, ако се дефинише тип структуре *tacka* који садржи два цела броја *x* и *y* (за координате једне тачке) и уведу структуре *t1* и *t2* (за две тачке), оне се у наредби декларације једноставно могу иницијализовати помоћу оператора доделе и конкретних вредности написаних између витичастих заграда, слика 4.14.



Слика 4.12 Пример иницијализације структура унутар дефиниције типа



Слика 4.13 Пример иницијализације структура ван дефиниције типа

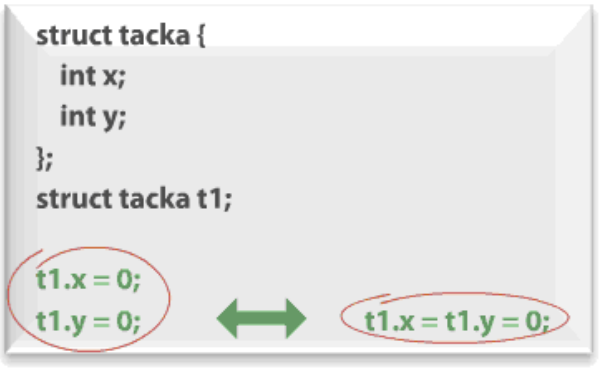


Слика 4.14 Пример примене структуре једног дефинисаног типа

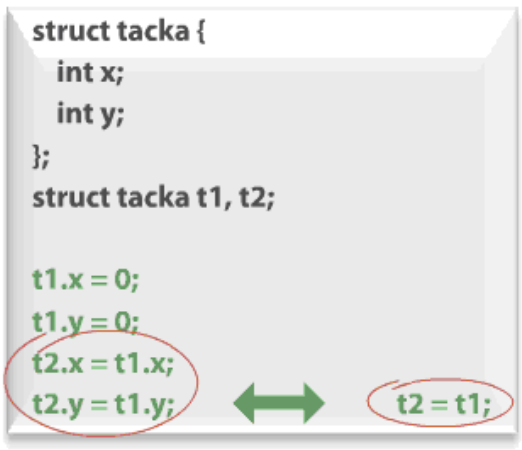
Приступ члану структуре

Приступ члану структуре може бити написан помоћу показивача на структуру или без њих. Без показивача приступ члану пише се помоћу имена структуре, "оператора тачка" и имена члана, слика 4.15 (само име члана не значи ништа преводиоцу). За две структуре (*t1* и *t2)* међусобно истог типа *(tacka)*, могући су: приступ помоћу оператора тачка, слика 4.16, додела комлетног садржаја једне структуре другој, слика 4.17, али не и примена релацијских оператора над структурама (већ само над њиховим члановима), слика 4.18.

Слика 4.15 Приступ члановима структуре помоћу "оператора тачка"



Слика 4.16 Пример приступа члановима структуре помоћу "оператора тачка"



Слика 4.17 Додела садржаја једне структуре другој структури истог типа

```
struct tacka {
   int x, y;
};
struct tacka t1, t2;
scanf("%d%d%d%d", &t1.x, &t1.y, &t2.x, &t2.y);
t1.x = t1.x + t2.x;
t1.y = t1.y + t2.y;
```

```
if(t1==t2)
   printf("Iste strukture");
```

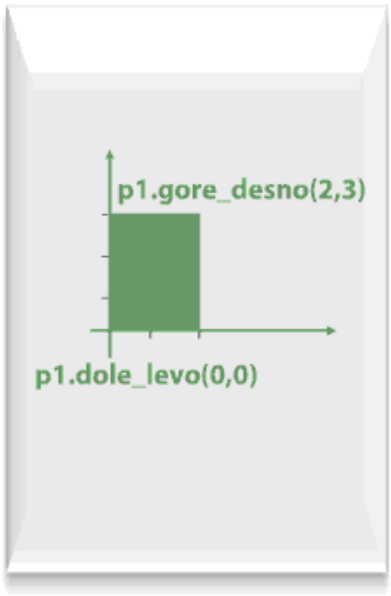
```
if(t1.x==t2.x && t1.y==t2.y)
   printf("Isti clanovi");
```

Слика 4.18 Примена појединих оператора над члановима структуре

Комплексна структура

Структура је комплексна ако садржи међу својим члановима друге структуре. Свака структура која треба да буде члан друге структуре треба да има дефиницију. Комплексна структура садржи у листи својих чланова декларацију структуре дефинисаног типа. Након декларације комплексне структуре њеним члановима се приступа помоћу: имена комплексне структуре, имена структуре која је члан комплексне и имена одређеног члана структуре која је члан комплексне. Између појединих имена је "оператор тачка", слика 4.19.

```
struct tacka {
int x;
int y; };

struct pravougaonik {
struct tacka dole_levo;
struct tacka gore_desno; };

struct pravougaonik p1;

p1.dole_levo.x = 0;
p1.dole_levo.y = 0;
p1.gore_desno.x = 2;
p1.gore_desno.y = 3;
```



Слика 4.19 Пример комплексне структуре

4.2. Низови у структурама и низови структура

Низ – члан структуре

Члан структуре може бити низ поизвољног типа. Ако је то *string* могућ је приступ помоћу његовог имена (константне адресе), само што је пре овог имена неопходно написати припадање одређеној структури (помоћу имена структуре и "оператора тачка"), слика 4.20. Код низова овог и свих осталих типова могућ је приступ сваком елементу на уобичајени начин (помоћу индексне или адресне аритметике), само опет као елементу низа који је члан одређене структуре, слика 4.21.

```
struct adresa {
  char ime_ulice[81];
   int broj_ulaza, broj_stana;  };

struct adresa a1;

gets(a1.ime_ulice);
scanf("%d%d", &a1.broj_ulaza, &a1.broj_stana);

if(strcmp(a1.ime_ulice, "NekoImeUlice")==0)
printf("U strukturi je NekoImeUlice");
```

Слика 4.20 Стринг - члан структуре

```
struct nizovi {
   int a[5];
   float b[10];
   double c[15];  };

int i;
struct nizovi n1;

for(i=0; i<5; i++)
   scanf("%d", &n1.a[i]);
for(i=0; i<10; i++)
   scanf("%f", &n1.b[i]);
for(i=0; i<15; i++)
   scanf("%lf", &n1.c[i]);
```

Слика 4.21 Низ било ког типа – члан структуре

Низ структура

Низ структура користи се као и низ било ког основног типа, само што је сада тип низа изведени (тип структуре). Као и једна структура, низ структура може бити декларисан унутар, слика 4.22, или ван дефиниције, слика 4.23. Ако структура дефинисаног типа има велики број чланова и ако је број оваквих структура у низу велики, меморија се много ефикасније користи уз алокацију меморије (декларација показивача на тип структуре, додела меморије и повезивање показивача са додељеном меморијом), слика 4.24.

```
struct  adresa {
  char ime_ulice[31];
  int broj_ulaza;
}spisak[ ] = { "Ime1", 1,
               "Ime2", 2,
               "Ime3", 3 };
int i;
for(i=0; i<3; i++)
{
   printf("%s ", spisak[i].ime_ulice);
   printf("%d ", spisak[i].broj_ulaza); }
```

Слика 4.22 Низ структура иницијализован унутар дефиниције типа структуре

```
struct tacka {
   int x;
   int y;
 };

struct tacka mnogougao[10];

int i;
for(i=0; i<10; i++)
{  scanf("%d", &mnogougao[i].x);
   scanf("%d", &mnogougao[i].y); }
```

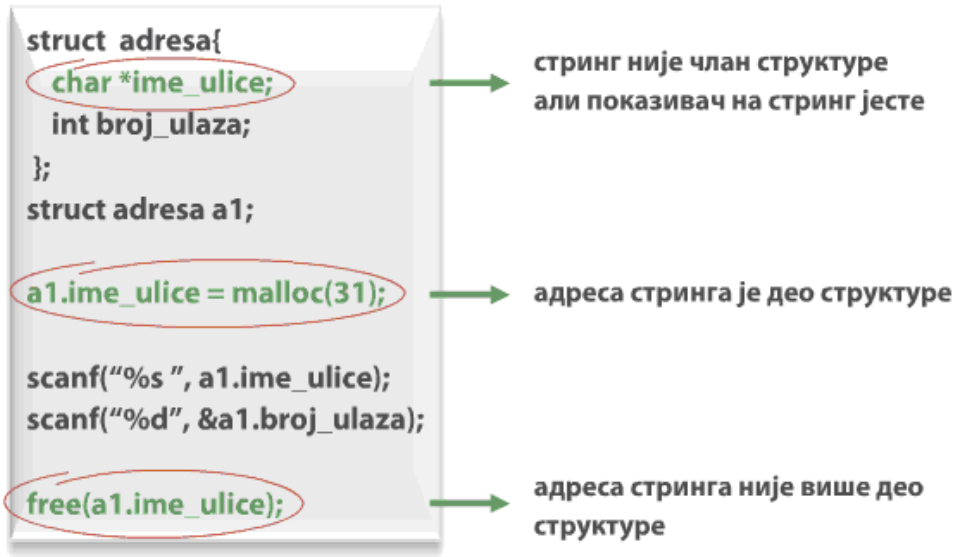
Слика 4.23 Низ структура иницијализован ван дефиниције типа структуре

Слика 4.24 Низ структура уведен динамичком доделом меморије

4.3. Структуре, показивачи и функције

Показивач – члан структуре

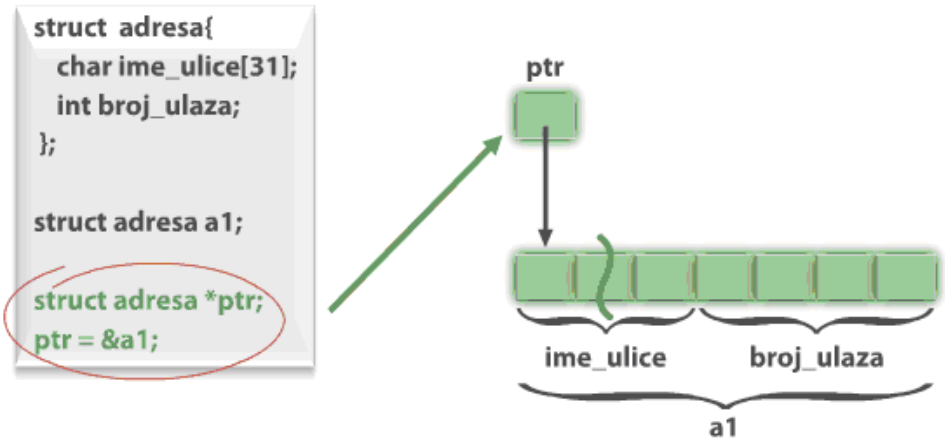
Показивач, као и променљива било ког другог типа, може бити члан структуре. Један пример је показивач који је члан структуре а иницијализује се на адресу низа који није члан структуре. Након декларације структуре дефинисаног типа показивачу члану структуре може се приступити преко имена те структуре и доделити адреса низа за који је претходно динамички додељена меморија. Даље се овом низу приступа преко имена структуре, иако низ није члан структуре (адреса низа то јесте све док се низ не укине), слика 4.25.



Слика 4.25 Показивач – члан структуре

Показивач на структуру

Показивач може бити декларисан на одређени тип структуре и иницијализован на адресу структуре тог типа, слика 4.26. Овакав показивач може бити коришћен за приступ члановима структуре на коју показује. Са уведеним показивачем приступ сваком члану пише се помоћу имена показивача и имена члана структуре а оператор може бити: оператор индиректног приступа (*) или оператор индиректног приступа члану (->), слика 4.27. Ако није уведен показивач приступ је могућ само помоћу "оператора тачка", слика 4.28.



Слика 4.26 Показивач на структуру

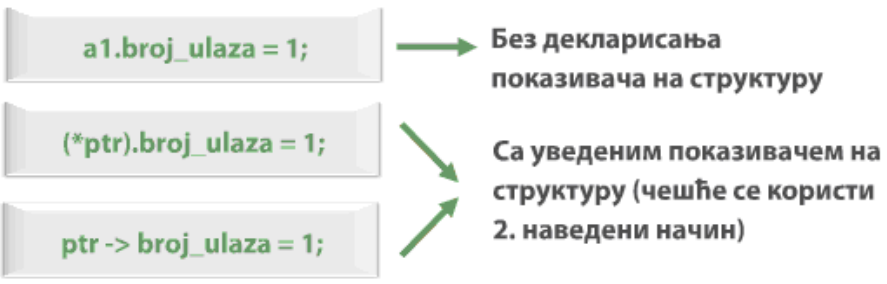
1. Помоћу оператора индиректног приступа (*):

```
strcpy( (*ptr).ime_ulice, "NekoIme" );
(*ptr).broj_ulaza = 1;
```

2. Помоћу оператора индиректног приступа члану структуре (->):

```
strcpy( ptr -> ime_ulice, "NekoDrugoIme" );
ptr -> broj_ulaza = 2;
```

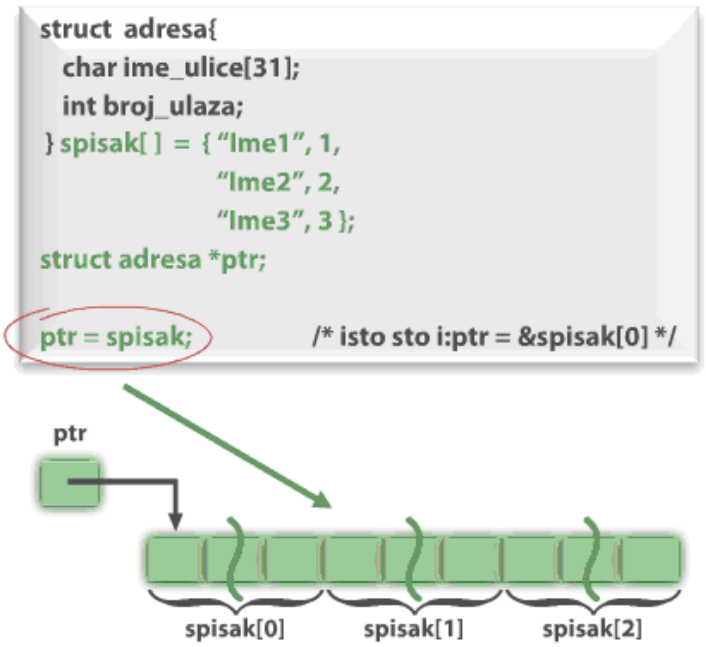
Слика 4.27 Приступ члановима структуре помоћу показивача на структуру



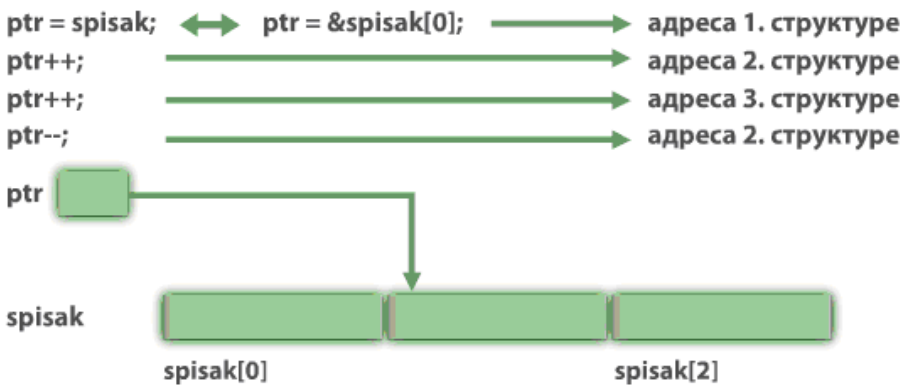
Слика 4.28 Могући начини приступа члановима структуре

Показивач на низ структура

Показивач на одређени тип структуре може бити искоришћен као показивач на низ структура истог типа. Иницијализација показивача у том случају може се написати тако што се константна адреса низа структура (само име низа) додељује показивачу, слика 4.29. Сваки инкремент показивача у том случају значи померање његовог садржаја на адресу следеће структуре у низу, а декремент на адресу претходне структуре у низу, слика 4.30. На овај начин је омогућен приступ структурама у низу ако су познати адреса и тип низа.



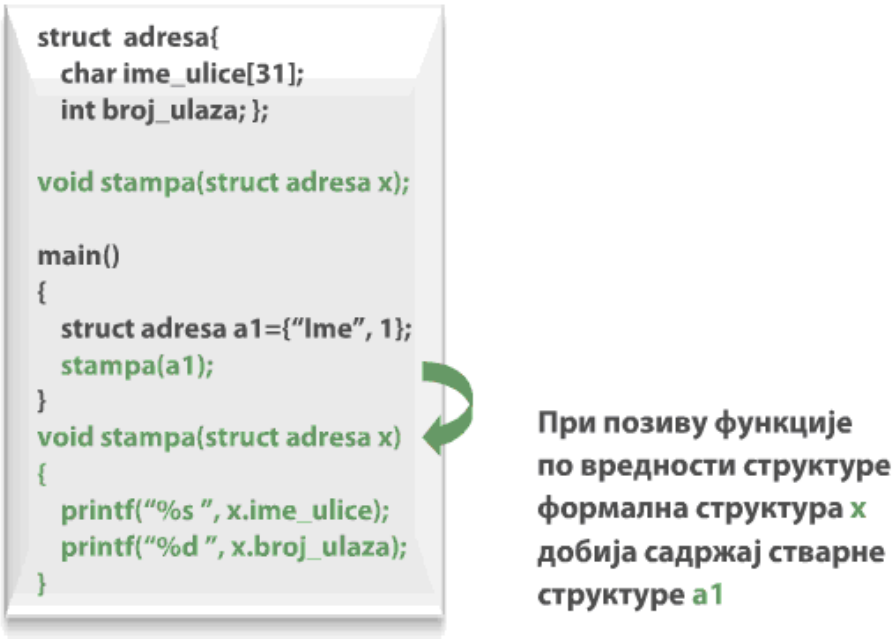
Слика 4.29 Показивач на низ структура



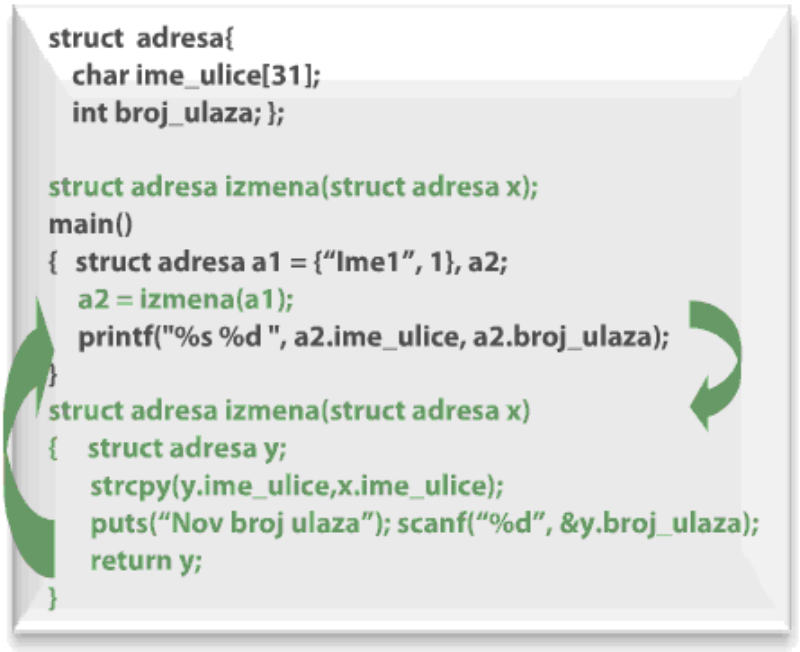
Слика 4.30 Инкремент и декремент показивача на структуру

Структуре и функције

За разлику од низа који не може бити предат функцији по вредности и не може бити повратна вредност од функције, структура све то може, јер без обзира на своју комплексност представља један објекат у програму. Аргумент функције може бити вредност структуре (функција у том случају може користити садржај структуре, али га не може ажурирати), слика 4.31. Ако је структура повратна вредност, од функције се добија садржај свих чланова структуре, слика 4.32.



Слика 4.31 Структура – аргумент функције



Слика 4.32 Структура – резултат од функције

Показивач на низ структура – аргумент и повратна вредност

Информације о низу структура функција може добити помоћу адресе на низ одређеног типа и дужине низа (као и за низ било ког другог типа). Синтакса за ово веома је слична као и код осталих низова: у декларацији се пише показивач на низ структура одређеног типа, у позиву је име конкретног низа структура а у дефиницији је омогућен приступ за читање, али и за ажурирање садржаја било које структуре у низу, слика 4.33. Показивач на низ структура може бити и повратна вредност функције.

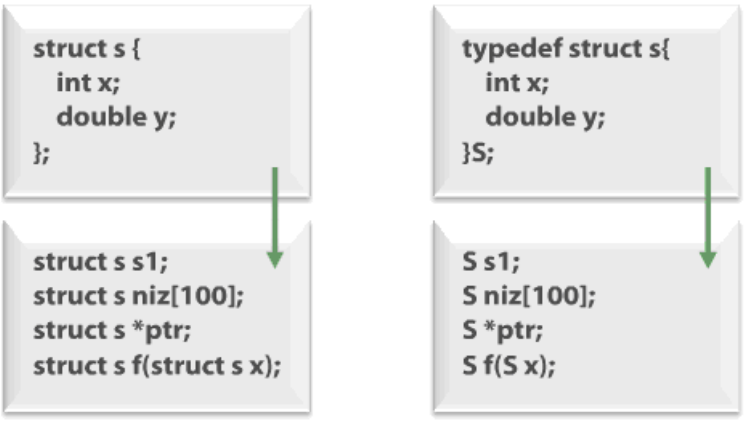
```
struct  adresa{
  char ime_ulice[31];
  int broj_ulaza; };

void stampa(struct adresa niz[], int n);
main()
{  struct adresa spisak[ ] = {"Ime1", 1, "Ime2", 2, "Ime3", 3};
   stampa(spisak, 3);
}
void stampa(struct adresa niz[ ], int n)
{  int i;
   for(i=0; i<n; i++)
   {   printf("%s ", niz[i].ime_ulice);
       printf("%d ", niz[i].broj_ulaza); }
}
```

Слика 4.33 Показивач на низ структура – аргумент функције

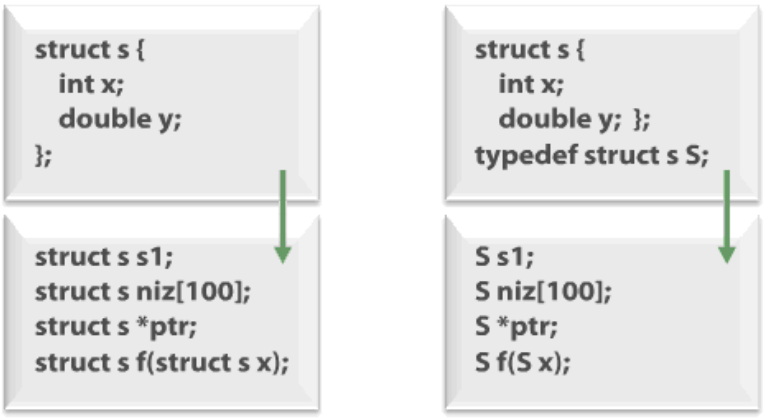
4.4. Синоним за тип структуре и тип уније

Синоним за тип структуре помоћу *typedef*

Помоћу службене речи *typedef* може се увести синоним за било који тип структуре. За ово постоје два начина писања. Један је дефинисање синонима унутар саме дефиниције типа структуре, слика 4.34, а други подразумева дефинисање типа структуре и накнадно дефинисање синонима, слика 4.35. Службену реч *struct* и ознаку типа структуре *(tip)* тако може заменити синоним који се уобичајено разликује од ознаке типа само по првом слову *(Tip)*. У даљим наредбама програма онда се уместо типа структуре пише само њен синоним.



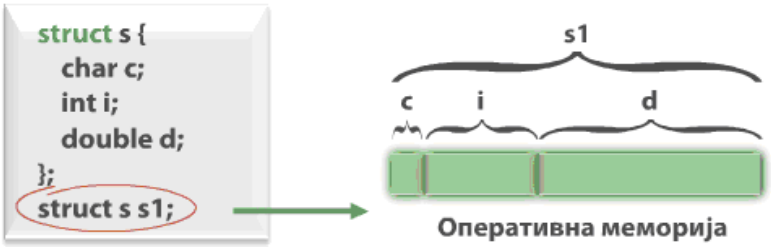
Слика 4.34 Увођење синонима за дефинисани тип структуре, начин 1.



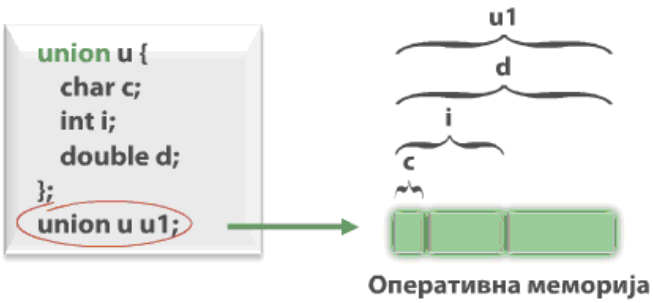
Слика 4.35 Увођење синонима за дефинисани тип структуре, начин 2.

Коришћење уније уместо структуре

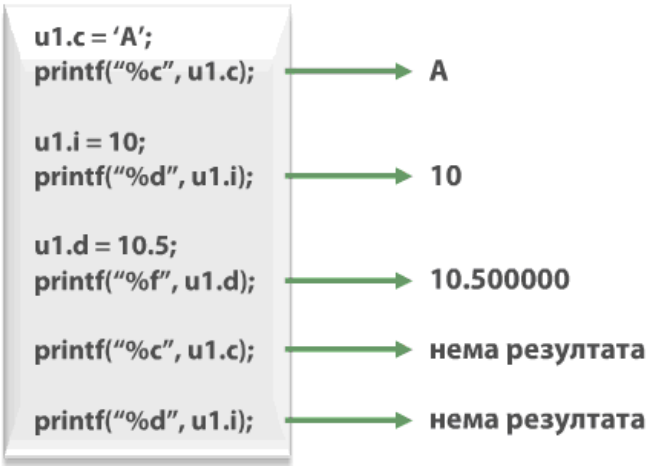
Унија је (као и структура) колекција променљивих, међусобно истих или различитих типова, које описују један објекат у програму. Разлика у односу на структуру, слика 4.36, је у службеној речи *union* уместо *struct* и то што се за унију у меморији не резервише збир бајтова потребних свим члановима, већ број бајтова највећег члана, слика 4.37. На тај начин меморија се ефикасније користи него за структуру. Сви чланови уније не могу се због тога користити истовремено, већ само један по један, како се који иницијализује, слика 4.38.



Слика 4.36 Заузеће меморије за структуру



Слика 4.37 Заузеће меморије за унију

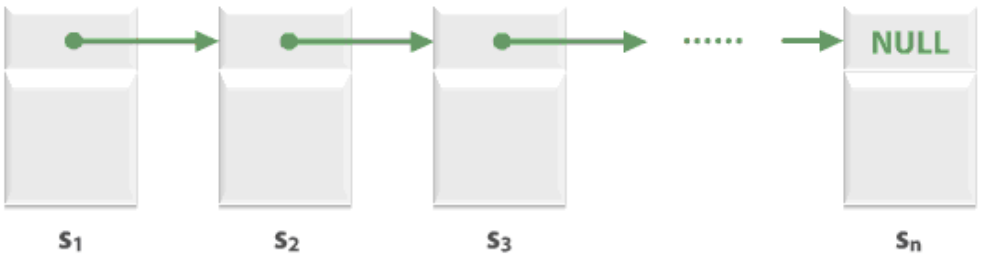


Слика 4.38 Приступ члановима уније

4.5. Динамички повезане листе

Једноструко повезане листе

Статички и динамички декларисани низови структура захтевају резервисање континуалног простора у меморији, који може бити веома велики. Код динамички повезаних листи то није случај. Једноструко повезана листа представља скуп структура, међусобно истог типа, које у меморији нису обавезно смештене једна до друге и које су повезане показивачима у једном смеру, слика 4.39. Свака структура изузев последње садржи показивач на следећу и на било ком месту у листи структура се може додати или укинути.



Слика 4.39 Једноструко повезане листе

Дефиниција чвора листе

Свака структура у листи зове се чвор листе. Структура која треба да буде искоришћена као чвор у листи садржи обавезно показивач на структуру истог тог типа, слика 4.40. Овај показивач *(sledeci)* може бити искоришћен за повезивање са неком другом структуром истог типа, тако што ће бити иницијализован на адресу те друге структуре. Број осталих чланова структуре чвора је произвољан (то су промењљиве резервисане за податке који описују један објекат у програму).

Слика 4.40 Једноструко повезане листе – дефиниција типа чвора

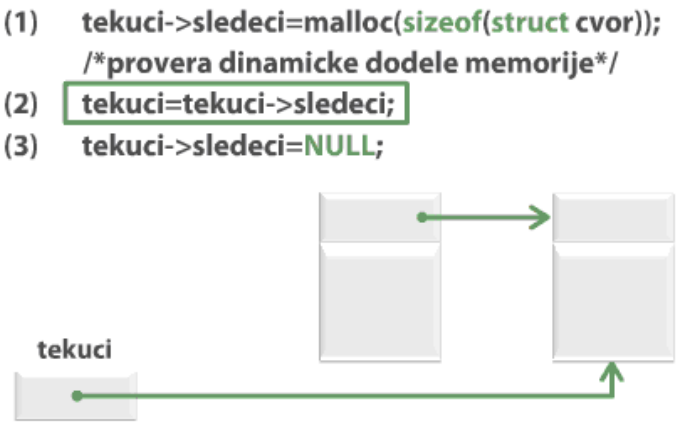
Формирање листе

Формирање једноструко повезане листе садржи следеће кораке: 1) декларацију два показивача на структуру чвор, који ће бити коришћени за адресу почетног и текућег чвора у листи; 2) алокацију меморије за чвор и доделу његове адресе првом од наведених показивача; 3) проглашавање тог почетног чвора текућим у листи; 4) на почетку нема других чворова у листи и због тога се показивач за повезивање у листу (*sledeci)* иницијализује на вредност NULL, слика 4.41.

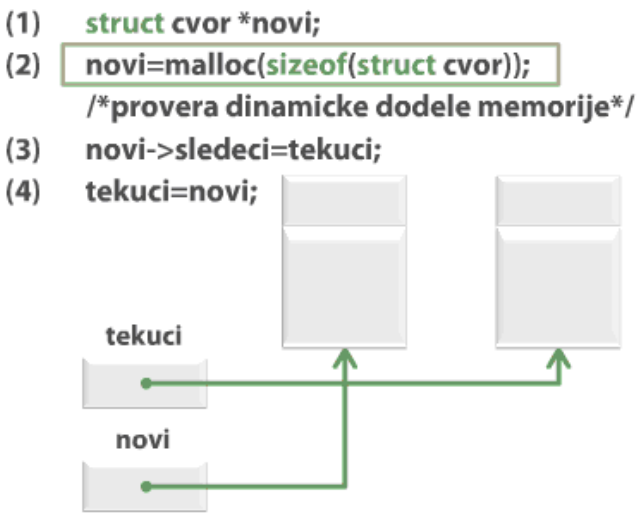
Слика 4.41 Једноструко повезане листе – формирање листе

Додавање чвора на крај и на почетак листе

Додавање чвора на крај (после текућег) обухвата: 1) алокацију меморије за нови чвор и његово повезивање са текућим; 2) проглашавање новог чвора текућим у листи; 3) иницијализацију његовог показивача (*sledeci)* на вредност NULL, слика 4.42. Додавање чвора на почетак (пре текућег чвора) обухвата: 1) декларацију новог показивача на тип чвора; 2) алокацију меморије за нови чвор и доделу његове адресе новом показивачу; 3) повезивање новог са текућим чвором; 4) проглашавање додатог чвора текућим, слика 4.43.



Слика 4.42 Једноструко повезане листе – додавање чвора на крај листе

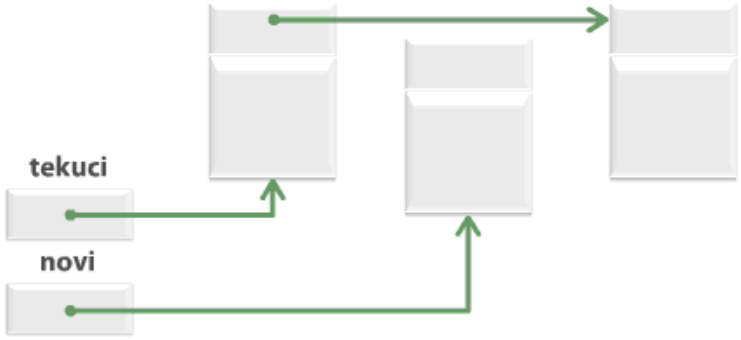


Слика 4.43 Једноструко повезане листе – додавање чвора на почетак листе

Уметање и брисање чвора после текућег у листи

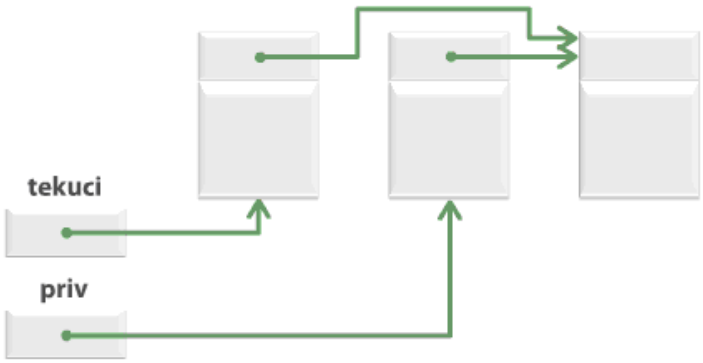
Уметање чвора после текућег обухвата: 1) декларацију новог показивача на тип чвора; 2) алокацију меморије за чвор и доделу адресе новом показивачу; 3) повезивање новог чвора са оним после текућег; 4) повезивање новог чвора са текућим; 5) проглашавање додатог чвора текућим, слика 4.44. Брисање чвора после текућег, обухвата: 1) декларацију привременог показивача на тип чвора; 2) његову иницијализацију на адресу чвора за брисање; 3) повезивање чворова који су пре и после чвора за брисање; 4) укидање меморије за чвор, слика 4.45.

```
(1)   struct cvor *novi;
(2)   novi=malloc(sizeof(struct cvor));
      /*provera dinamicke dodele memorije*/
(3)   novi->sledeci=tekuci->sledeci;
(4)   tekuci->sledeci=novi;
(5)   tekuci=novi;
```



Слика 4.44 Једноструко повезане листе – уметање чвора после текућег у листи

```
(1)   struct cvor *priv;
(2)   priv=tekuci->sledeci;
(3)   tekuci->sledeci=tekuci->sledeci->sledeci;
(4)   free(priv);
```



Слика 4.45 Једноструко повезане листе – брисање чвора после текућег у листи

4.6. Питања

1. Да ли структура података представља један или више објеката за *C* програм?
2. Да ли дефиниција *C* типа структуре резервише простор у меморији?
3. Да ли декларација примерка *C* типа структуре резервише простор у меморији?
4. Да ли члан структуре може бити само податак основног *C* типа, или податак било ког *C* типа?
5. Да ли члан структуре може бити структура претходно дефинисаног типа?
6. Да ли се примерак структуре може декларисати искључиво ван дефиниције типа структуре или и у оквиру дефиниције типа структуре?
7. Да ли тип структуре може бити тип аргумента *C* функције?
8. Да ли тип структуре може бити тип повратне вредности *C* функције?
9. Да ли се *C* функцији могу проследити информације о низу структура, само помоћу показивача на тај низ структура или не?
10. Шта је *C* функцији у аргументима неопходно, када треба да ради са низом структура, који није декларисан у тој функцији?
11. Који су кораци код динамичке декларације низа структура?
12. Која је примена службене речи typedef код структура у *C* програму?
13. Која је разлика између типа структуре и типа уније у *C* програму?
14. Која је разлика између низа структура и динамички повезаних листи?
15. Шта неопходно треба да садржи тип структуре, који ће бити чвор у динамички повезаној листи?

5. Рад са датотекама у програмима на језику C

Кратак преглед лекције

За разлику од стандардних улазно / излазних токова, који су аутоматски отворени за размену података током извршавања програма, за приступ свакој датотеци потребно је дефинисати посебан улазно / излазни ток и повезати га са датотеком. Упис / читање код текстуалних датотека реализује се на сличан начин као и улаз / излаз тастатура / екран. Упис / читање код бинарних датотека реализује се као размена података у одређеном смеру, између меморије и датотеке а на нивоу бајта.

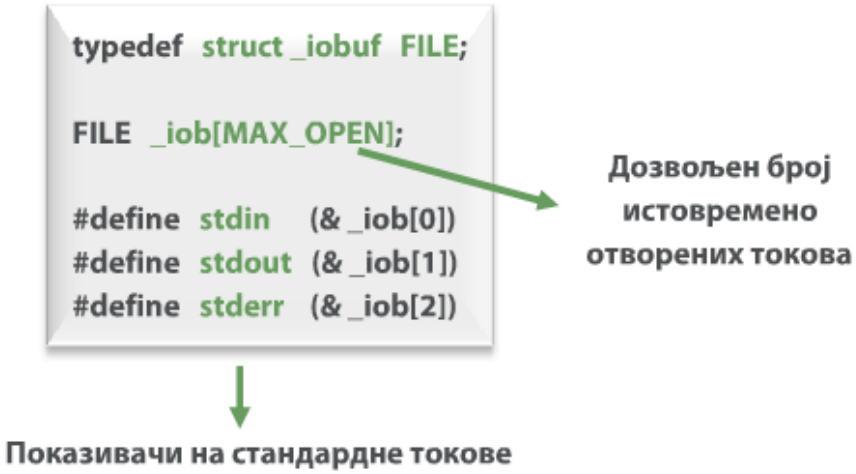
5.1. Улазно / излазни токови за рад са датотекама

У библиотеци *stdio.h* дефинисан је тип структуре *FILE* чији су чланови подаци који описују одређени улазно / излазни ток, слика 5.1. У истој библиотеци креиран је низ структура овог типа и прве три структуре у низу резервисане су за главни улазни, главни излазни и излазни ток за грешке. Имена показивача иницијализованих на адресе ове три структуре већ су позната, то су: *stdin*, *stdout* и *stderr*, слика 5.2. Број осталих структура (и одговарајућих токова), који зависи од оперативног система, расположив је за датотеке, слика 5.3.

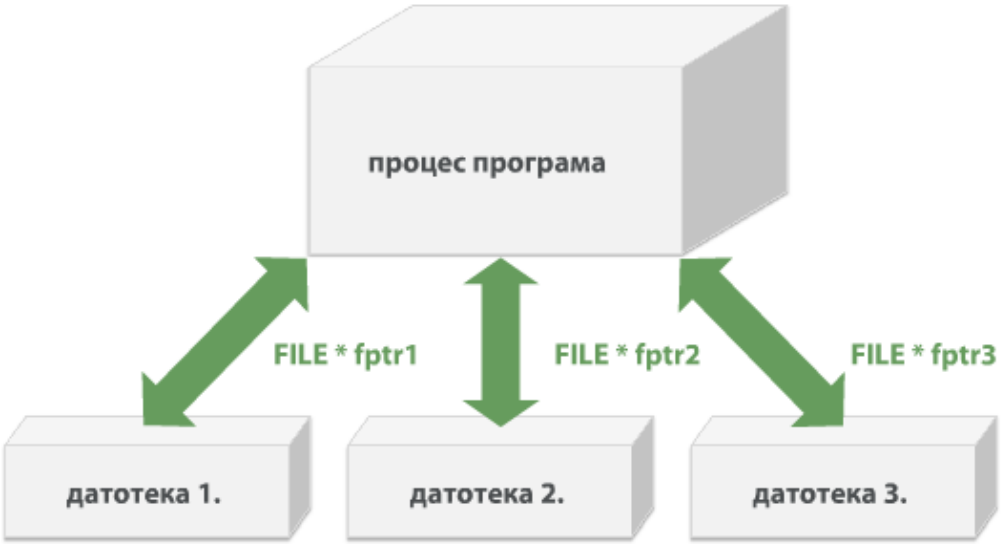
Библиотека: <stdio.h>

```
struct _iobuf {
   char *base_address;   /*adresa I/O bafera*/
   char *ptr_current;    /*trenutna pozicija u baferu*/
   int bytes_count;      /*broj bajtova u I/O baferu*/
   int flag;             /*nacin pristupa/status greske*/
   int file_description; /*Id broj datoteke*/
};
```

Слика 5.1 Структура типа *FILE*



Слика 5.2 Показивачи на структуре у низу типа FILE



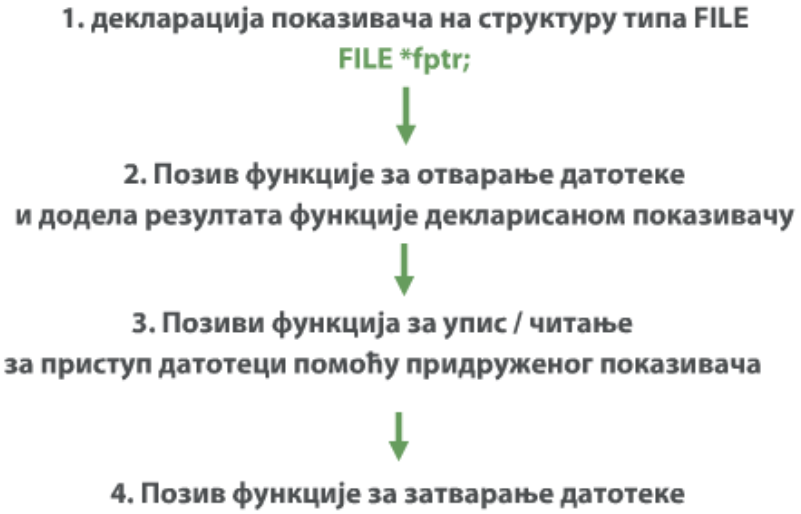
Слика 5.3 Улазно / излазни токови за датотеке

Постоје две основне врсте датотека: текстуалне (подразумевана врста) и бинарне. Текстуалне датотеке садрже форматиране знакове (штампајуће и управљачке) и због тога је у текстуалном улазно / излазном току неопходна конверзија између бинарног формата (који је заступљен у меморији) и неког другог формата (у датотеци). Бинарне датотеке садрже само бинарне цифре (исти запис као и у меморији) и због тога у бинарном улазно / излазном току нема конверзије.

5.2. Рад са датотекама

Повезивање улазно / излазног тока и датотеке

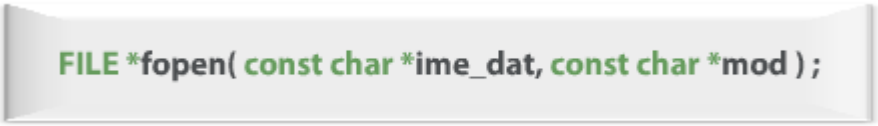
Да би био могућ приступ текстуалној или бинарној датотеци, за упис или читање, неопходно је резервисати једну структуру типа *FILE*, увести показивач на ту структуру и повезати га са датотеком. За показивач на структуру овог типа повезану са датотеком даље у тексту биће коришћен термин: показивач који одговара датотеци. За приступ датотеци било које врсте обавезно следе позиви функција: за отварање датотеке, за упис / читање садржаја и за затварање датотеке, слика 5.4.



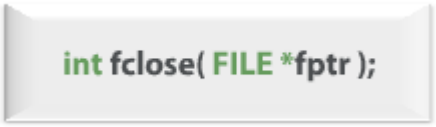
Слика 5.4 Основни кораци у раду са датотеком

Функције за отварање и затварање датотеке

Функција за отварање датотеке *fopen()* има два аргумента. Један аргумент је показивач на *string* са именом датотеке, а други је показивач на *string* са режимом отварања датотеке, слика 5.5. Резултат ове функције је показивач на структуру типа *FILE*, или *NULL* показивач ако постоји грешка при отварању датотеке. Након завршеног приступа датотеци потребан је позив функције за затварање датотеке *fclose()* која има само један аргумент: показивач на датотеку коју затвара, слика 5.6. Обе функције су из библиотеке *stdio.h*.



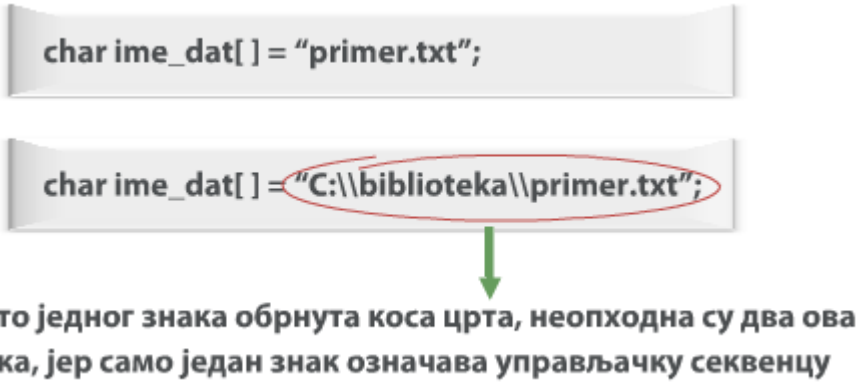
Слика 5.5 Функција за отварање датотеке



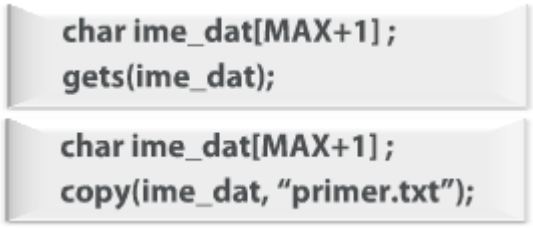
Слика 5.6 Функција за затварање датотеке

Име датотеке и режим отварања датотеке

Један од аргумената функције *fopen()* је показивач на *string* са именом датотеке. Садржај овог *stringa* може бити иницијализован при његовој декларацији, слика 5.7, или накнадно помоћу неке од *string* функција, слика 5.8. Други аргумент функције *fopen()* је показивач на *string* са режимом отварања датотеке. Неки од основних режима (за упис, за читање или за упис и читање) имају више верзија, код текстуалних, слика 5.9 и код бинарних датотека, слика 5.10.



Слика 5.7 Један могући начин задавања имена датотеке са којом се ради



Слика 5.8 Други могући начин задавања имена датотеке са којом се ради

Ознака	Значење
"w"	Отварање за упис од почетка, ако датотека не постоји, креира се
"r"	Отварање за читање, ако датотека не постоји, резултат функције fopen() је NULL
"a"	Отварање за упис у продужетку постојећег садржаја, ако датотека не постоји, креира се
"w+"	Отварање за упис од почетка и за читање после тога, ако датотека не постоји, креира се
"r+"	Отварање за читање и за упис после тога, ако датотека не постоји, резултат функције fopen() је NULL
"a+"	Отварање за упис у продужетку постојећег садржаја и за читање после тога, ако датотека не постоји креира се

Слика 5.9 Режими отварања текстуалне датотеке

Ознака	Значење
"wb"	Отварање за упис од почетка, ако датотека не постоји, креира се
"rb"	Отварање за читање, ако датотека не постоји, резултат функције fopen() је NULL
"ab"	Отварање за упис у продужетку постојећег садржаја, ако датотека не постоји, креира се
"wb+"	Отварање за упис од почетка и за читање после тога, ако датотека не постоји, креира се
"rb+"	Отварање за читање и за упис после тога, ако датотека не постоји, резултат функције fopen() је NULL
"ab+"	Отварање за упис у продужетку постојећег садржаја и за читање после тога, ако датотека не постоји, креира се

Слика 5.10 Режими отварања бинарне датотеке

Провера статуса датотеке

Након позива функције *fopen()* а пре позива функција за упис / читање, неопходно је предвидети проверу статуса датотеке. Може се десити да датотека

није отворена за операцију која се тражи у функцији *fopen()*: за упис (ако је погрешно наведено име диск уређаја или директоријума у комплетном имену датотеке), или за читање (ако датотека не постоји). Ако функција *fopen()* враћа показивач *NULL*, то значи грешку при отварању датотеке и у сваком таквом случају потребно је предвидети наредбе за обраду грешке, слика 5.11.

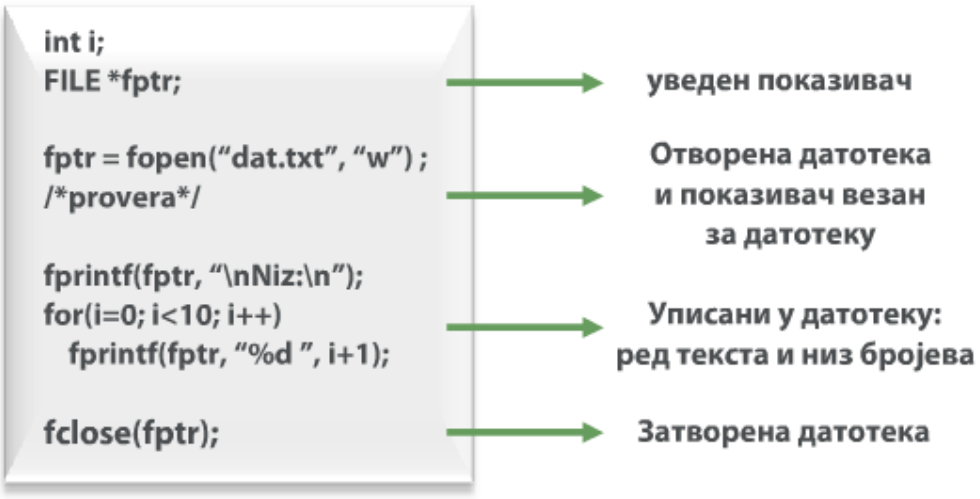
```
fptr = fopen("primer.txt", "w") ;

if(fptr == NULL)
{  fprintf(stderr, "Greska otvaranja datoteke");
   exit(1);
}

/*pristup datoteci na odgovarajuci nacin*/
```

Слика 5.11 Провера резултата функције fopen()

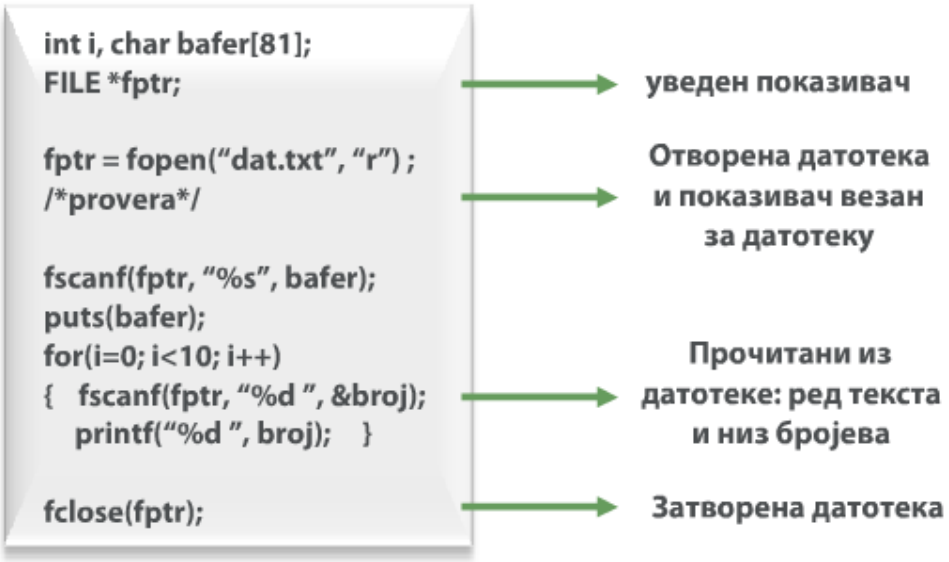
5.3. Функције за упис и читање код датотека

Форматирани упис и читање код текстуалних датотека

За форматирани упис и читање користе се функције *fprint()* и *fscanf()* које раде на исти начин као и функције *print()* и *scanf()*. Разлика је у томе што је функцијама које раде са датотеком неопходна информација о датотеци. Код функција *fprint()* и *fscanf()* променљив је број аргумената. Због тога информацију о датотеци свака од њих добија у првом аргументу и то је показивач који одговара одређеној датотеци. Сви формати за упис, слика 5.12, и читање, слика 5.13, важе као и код функција *print()* и *scanf()*.



Слика 5.12 Форматирани упис у датотеку



Слика 5.13 Форматирано читање из датотеке

Упис / читање знак по знак код текстуалних датотека

За упис и читање знак по знак код датотека се користе функције *fputc()* и *fgetc()*. Ове функције раде на исти начин као и функције *putchar()* и *getchar()*, само што свака од њих добија информацију о датотеци. Функција *fputc()* у свом последњем аргументу добија показивач који одговара одређеној датотеци, а функција *fgetc()* добија овај показивач у свом једином аргументу. Као и код тастатуре и екрана и код датотека се функције *fputc()* и *fgetc()* обично користе у комплету, слика 5.14.

Слика 5.14 Читање / упис код датотека знак по знак

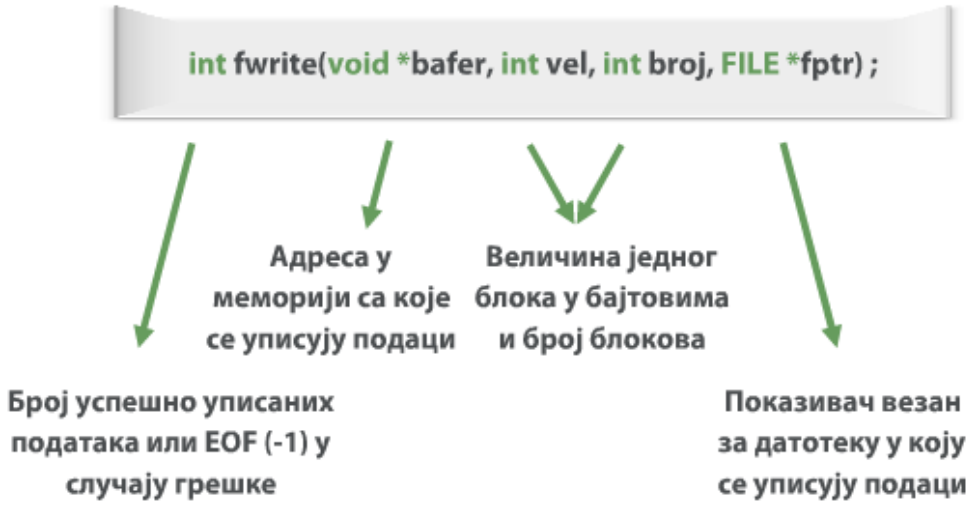
Упис / читање ред по ред код текстуалних датотека

За упис и читање ред по ред знакова код датотека се користе функције *fputs()* и *fgets()*. Ове функције раде на исти начин као и функције *puts()* и *gets()*, само што свака од њих добија информацију о датотеци. Функција *fputs()* у свом последњем аргументу добија показивач који одговара одређеној датотеци, а функција *fgets()* има у једном додатом аргументу максимални број знакова који се чита а у другом додатом аргументу показивач који одговара датотеци. И функције *fputs()* и *fgets()* обично се користе у комплету, слика 5.15.

Слика 5.15 Читање / упис код датотека ред по ред

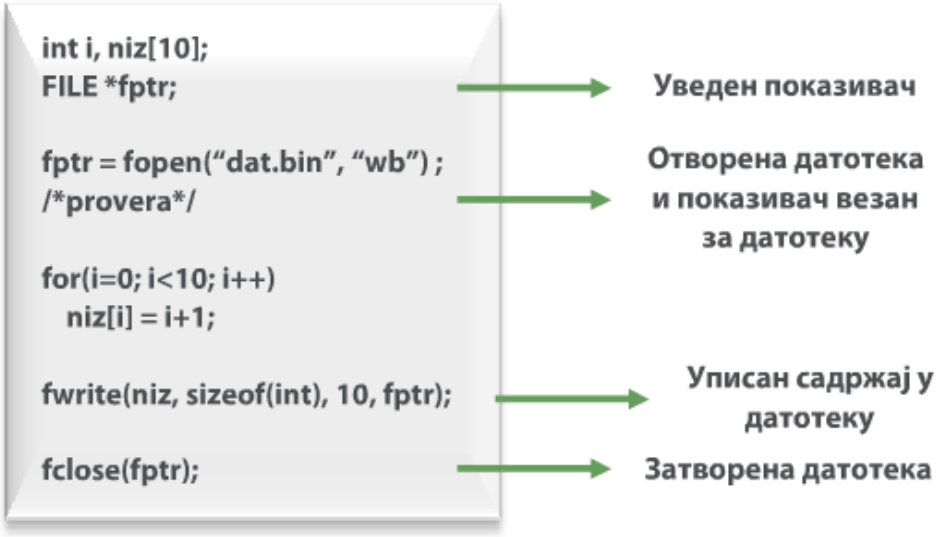
Упис / читање код бинарних датотека

Функција *fwrite()* уписује блокове бајтова из меморије у датотеку чији показивач добија у последњем аргументу, слика 5.16. Функција *fread()* ради у обрнутом смеру, али има исти редослед аргумената: учитава блокове бајтова у меморију из датотеке чији показивач добија у последњем аргументу, слика 5.17. За израчунавање броја бајтова уобичајено се користи оператор *sizeof()*, а упис низа, слика 5.18 и читање низа, слика 5.19, могу бити остварени сваки помоћу једног позива функције за упис и читање.



Слика 5.16 Функција fwrite() из библиотеке stdio.h

Слика 5.17 Функција fread() из библиотеке stdio.h



Слика 5.18 Упис у бинарну датотеку

Слика 5.19 Читање из бинарне датотеке

5.4. Директан приступ и позиционирање у датотекама

Читање тренутне позиције у датотеци

Након отварања датотеке било које врсте, за упис / читање од почетка њеног садржаја, тренутна позиција у датотеци је на вредности 0. Свака операција уписа / читања увећава вредност ове тренутне позиције за број уписаних / прочитаних бајтова. Функција *ftell()* из библиотеке *stdio.h* даје као резултат вредност тренутне позиције у датотеци чији показивач добије у аргументу, слика 5.20. Резултат ове функције може бити приказан на екрану, слика 5.21, или искоришћен у функцији за позиционирање унутар датотеке.

Слика 5.20 Функција ftell() из библиотеке stdio.h

```
int i;
FILE *fptr;

fptr = fopen("dat.txt", "w") ;
printf("\n ip=%ld", ftell(fptr));

for(i=0; i<10; i++)
   fprintf(fptr, "%d ", i+1);
printf("\n ip=%ld", ftell(fptr));

fclose(fptr);
```

ip=0

ip=40

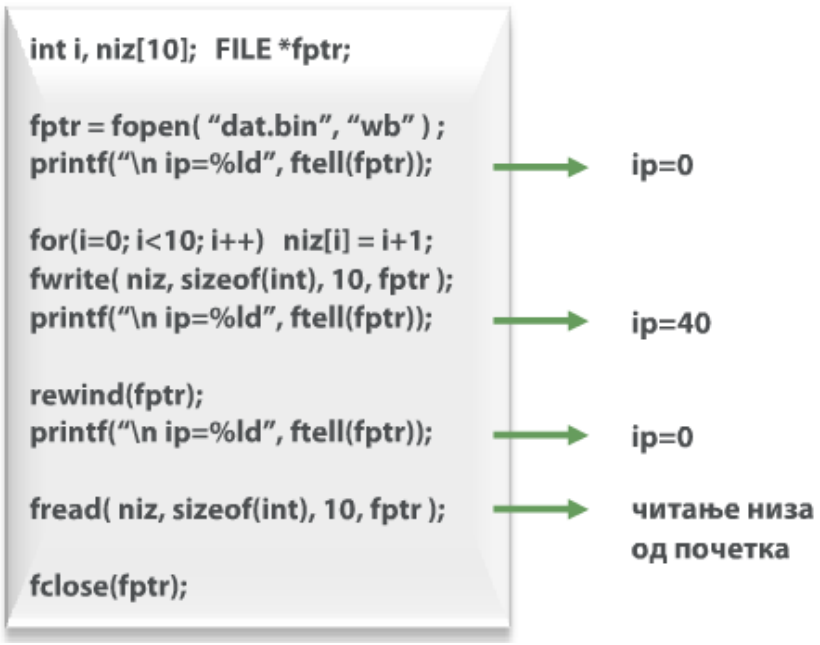
Слика 5.21 Читање тренутне позиције у датотеци

Позиционирање у датотеци

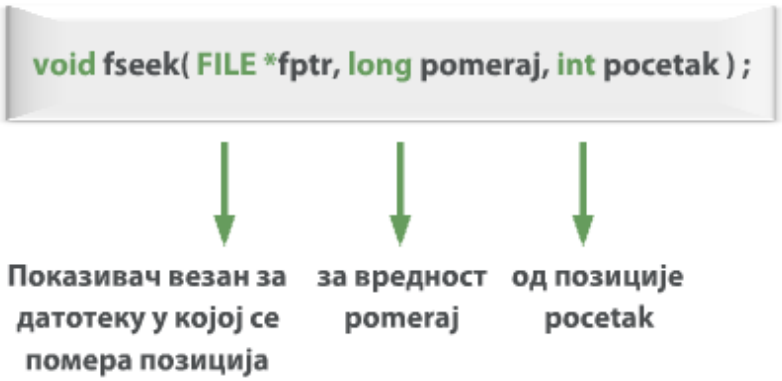
Обично је у раду са датотеком предвиђено више узастопних операција уписа и читања. Да се не би стално позивале функције *fclose()* и *fopen(),* у библиотеци *stdio.h* постоје функције за позиционирање. Функција *rewind()*, слика 5.22, враћа позицију на почетак датотеке чији показивач добије, слика 5.23. Функција *fseek(),* слика 5.24, помера позицију за одређени број бајтова и то: од почетка, од затечене позиције или с краја датотеке чији показивач добије, слика 5.25 и корисна је код наизменичног уписа и читања, слика 5.26.

```
void rewind( FILE *fptr ) ;
```

Слика 5.22 Функција rewind() из библиотеке stdio.h



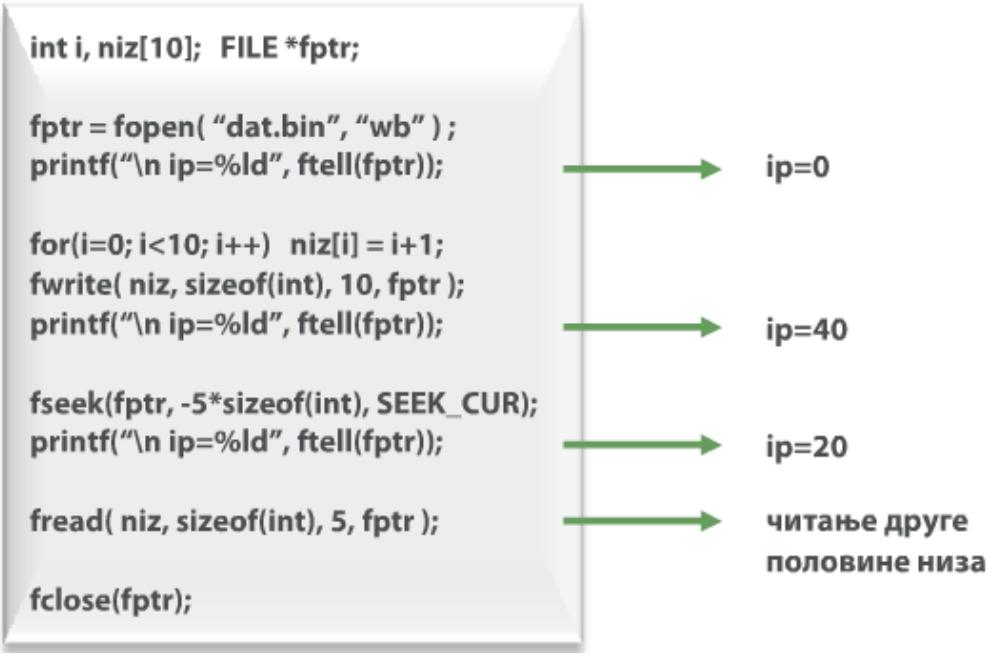
Слика 5.23 Повратак на почетак датотеке између уписа и читања садржаја



Слика 5.24 Функција fseek() из библиотеке stdio.h

име	вредност	опис
SEEK_SET	0	Померај од почетка датотеке
SEEK_CUR	1	Померај од тренутне позиције у датотеци
SEEK_END	2	Померај од краја датотеке

Слика 5.25 Константе за ознаке почетне позиције у фуннкцији fseek()



Слика 5.26 Позиционирање унутар датотеке

Детекција краја датотеке

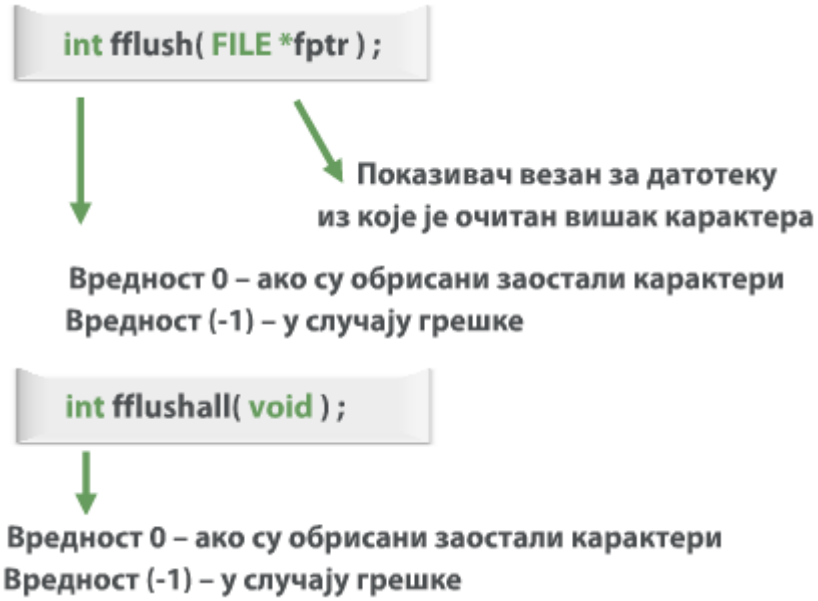
Ако се претражује комплетан садржај датотеке, а није унапред познат тачан број записа у њој, неопходно је користити један од начина за детекцију краја датотеке. Код текстуалних датотека то може бити детекција знака *EOF (EndOfFile)*. Бинарне датотеке немају овај знак и због њих је уведена, а може се користити и код текстуалних датотека, функција *feof()* из библиотеке *stdio.h*, слика 5.27. Ова функција враћа вредност различиту од 0 ако је нађен крај датотеке (ако је тренутна позиција на самом крају датотеке).

Слика 5.27 Функција feof() из библиотеке stdio.h

5.5. Решавање вишка карактера, измена имена и брисање датотека

Функције за решавање вишка карактера прочитаних из датотека

За решавање проблема карактера заосталих у улазном току а из датотека, може бити искоришћена функција *fflush()* из библиотеке *stdio.h*. Ако у аргументу добије показивач на улазни ток од датотеке ова функција брише заостале карактере са тог тока и због тога се препоручује њен позив после сваког комплета позива функције *fscanf()*. Поред ове функције, у истој библиотеци постоји и функција *fflushall()* која нема аргументе, већ брише заостале карактере са свих улазних токова, слика 5.28.



Слика 5.28 Функције fflush() и fflushall()

Функције за промену имена и брисање датотеке

Креирање датотеке из *C* програма омогућава функција *fopen()* у режиму за упис. За копирање садржаја из једне датотеке у другу не постоји функција већ се користе комбинације следећих: *fscanf()* и *fprintf()*, *fgetc()* и *fputc()*, *fgets()* и *fputs()*, *fread()* и *fwrite()*. За промену имена датотеке постоји функција *rename()*, а за брисање датотеке функција *remove()*, обе из библиотеке *stdio.h*. Прва функција користи показиваче на *stringove* са старим и новим именом, а друга функција показивач на *string* са именом датотеке коју брише, слика 5.29.

Слика 5.29 Функције rename() и remove()

5.6. Питања

1. Да ли је за приступ датотекама из C програма неопходно увођење показивача?
2. Које врсте информација чува структура типа FILE из библиотеке stdio.h?
3. Да ли се тип датотеке са којом треба да ради C програм, дефинише при позиву функције fopen(), или касније при позиву функција за упис / читање?
4. Шта од аргумената очекује C функција fopen(), да би извршила отварање одређене датотеке (односно одређеног тока за рад са датотеком)?
5. Шта од аргумената очекује C функција fclose(), да би извршила затварање одређене датотеке (односно одређеног тока за рад са датотеком)?
6. На који начин је могуће проверити да ли је C функција fopen() успешно извршила свој задатак?
7. Шта у језику C значе режими за отварање датотека: "w" / "r" / "a"?
8. Шта у језику C значе режими за отварање датотека: "w+" / "r+" / "a+"?
9. Шта у језику C значе режими за отварање датотека: "wb" / "rb" / "ab"?
10. Шта у језику C значе режими за отварање датотека: "wb+" / "rb+" / "ab+"?
11. Које C функције се могу користити за форматиран упис / читање код текст датотека?
12. Које C функције се могу користити за упис / читање знак по знак код текст датотека?
13. Које C функције се могу користити за упис / читање ред по ред код текст датотека?
14. Које C функције се могу користити за упис / читање код бинарних датотека?
15. Које се C функције могу користити за позиционирање унутар садржаја датотека, текстуалних и бинарних?

6. Модуларни програми и претпроцесорске директиве у програмима на језику C

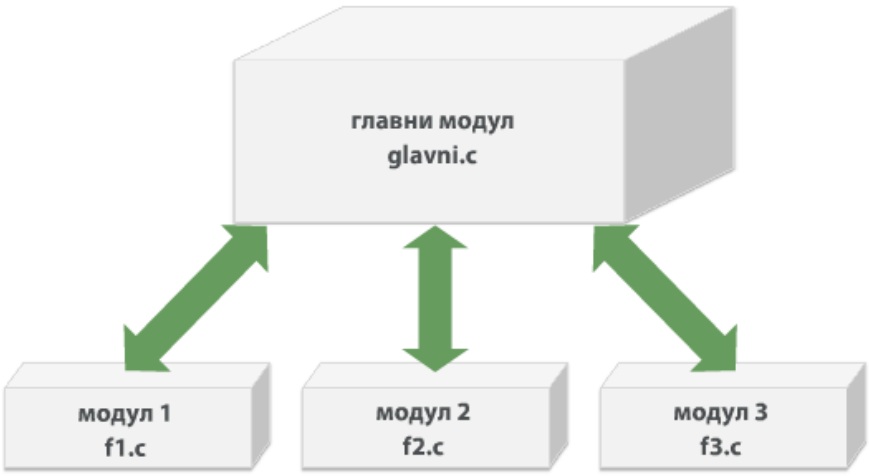
Кратак преглед лекције

Изворни кôд кратког програма може бити сачуван у једној датотеци. Код обимнијих програма ако писање, претраживање и ажурирање његових делова постане компликовано, изворни кôд се може распоредити у више датотека. Све датотеке са изворним кодом програма зову се модули програма. Једноставно коришћење функција и података у свим модулима програма омогућавају и претпроцесорске директиве. Помоћу одређених директива могу бити дефинисани макрои, као замена за функције, онда кад је критично време.

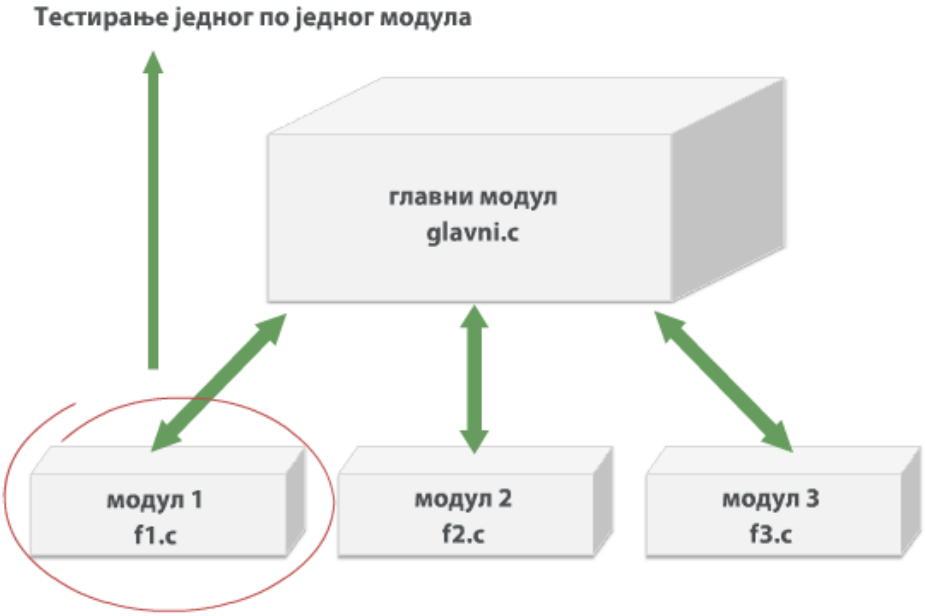
6.1. Модуларни програми

Распоређивањем изворног кода сваког обимног програма у више датотека, које се називају модули, поједностављују се пројектовање, слика 6.1, писање, слика 6.2, и тестирање рада, слика 6.3, овог програма. Модули *(source files)* једног *C* програма обавезно имају екстензију *.c*. Програми написани у модулима рашчлањени су на програмске целине, једноставнији су за развој и омогућавају коришћење својих решења у другим програмима. На тај начин сваки готов програм даје и своју библиотеку готових појединачних модула.

Слика 6.1 Пројектовање модуларног програма



Слика 6.2 Издвајање модула у фази писања програма



Слика 6.3 Тестирање модуларног програма

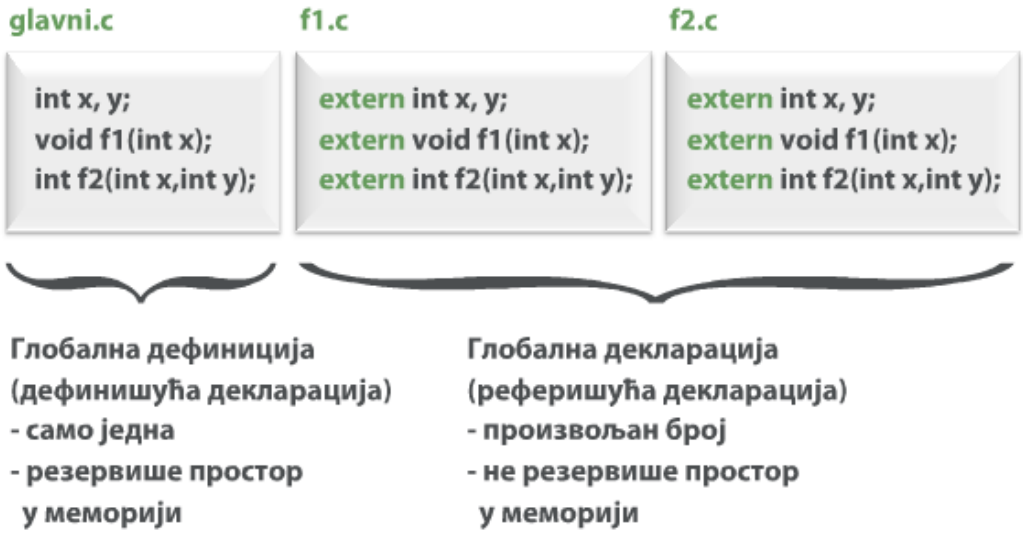
Распоређивање изворног кода у модулима

Сваки програм има један главни модул и произвољан број секундарних модула. Главни модул садржи главну функцију програма и евентуално кратке функције које не припадају ниједном другом модулу. Секундарни модули садрже функције различите од главне и у сваком од њих је једна функција или више кратких међусобно повезаних функција. *C* програмски алат за повезивање програма *(C linker)* генерише извршни кôд тако што повезује све модуле, као и све стандардне *C* библиотеке које укључују ови модули, слика 6.4.

Слика 6.4 Повезивање изворног кода из више модула

Екстерна декларација променљивих и функција

Променљиве и функције које су потребне великом броју функција у једном модулу програма, декларишу се глобално. За њихово коришћење у другим модулима истог програма користи се екстерна декларација. Само у једном модулу пише се глобална дефинишућа декларација (која резервише простор у меморији), а у произвољном броју осталих модула је екстерна реферишућа декларација (која не резервише простор у меморији, већ се само позива на постојећу декларацију у неком другом модулу програма), слика 6.5.

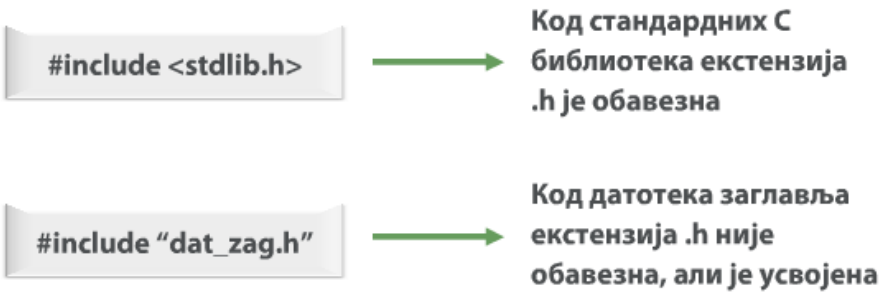


Слика 6.5 Глобално декларисане променљиве и функције

6.2. Претпроцесорска директива *#include*

Датотеке заглавља

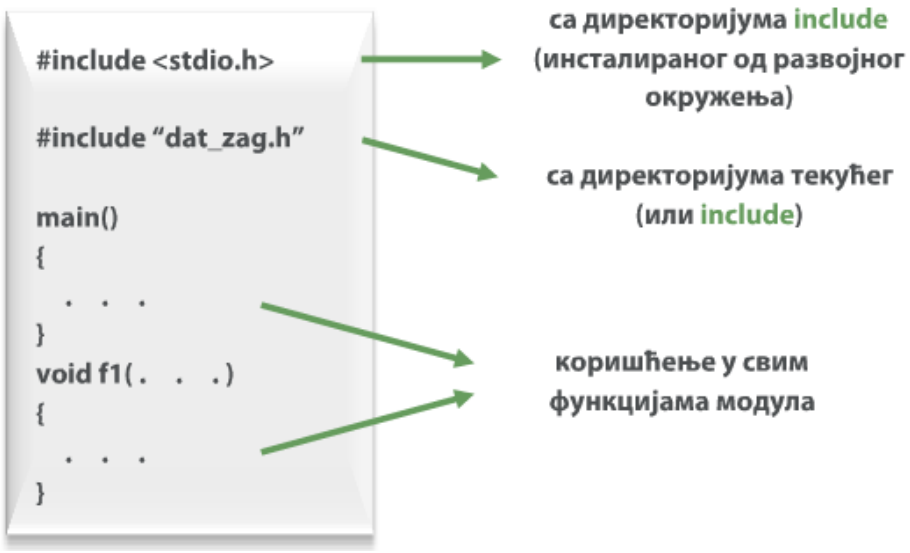
Сваки C програм укључује стандардне C библиотеке помоћу директиве *#include<ime_biblioteke.h>*. Ако се програм пише модуларно онда сваки његов модул обично укључује и датотеке заглавља *(header files)*. То су датотеке које се креирају за конкретан програм, уобичајено имају екстензију *.h* и укључују се помоћу директиве *#include "ime_datoteke.h"*, слика 6.6. Ове датотеке креирају се за глобалне и одговарајуће екстерне декларације променљивих и функција, и за дефиниције изведених типова потребних програму.



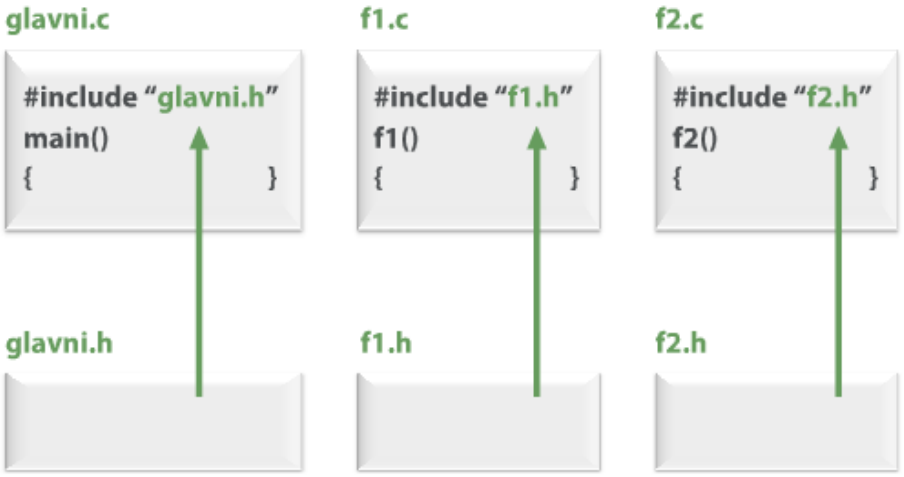
Слика 6.6 Датотеке заглавља

Коришћење датотека заглавља

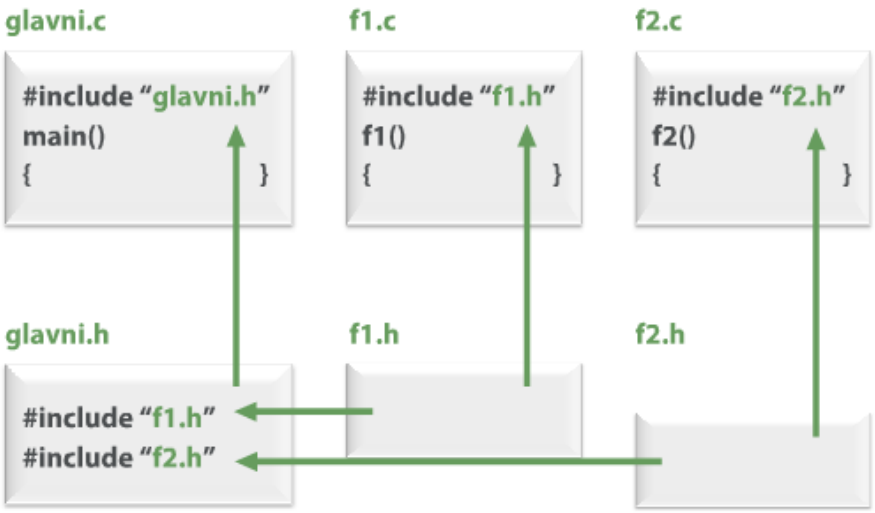
За разлику од библиотека које се укључују директивом *#include<ime_biblioteke.h>*, са директоријума *include*, датотеке заглавља укључују се са текућег директоријума пројекта (евентуално са директоријума *include*, ако их нема на текућем), слика 6.7. Датотека заглавље може бити укључена у било који модул програма, слика 6.8, или у другу датотеку заглавље, слика 6.9. Свака измена садржаја једне овакве датотеке захтева поновљено превођење модула који је укључују и повезивање програма.



Слика 6.7 Коришћење датотека заглавља



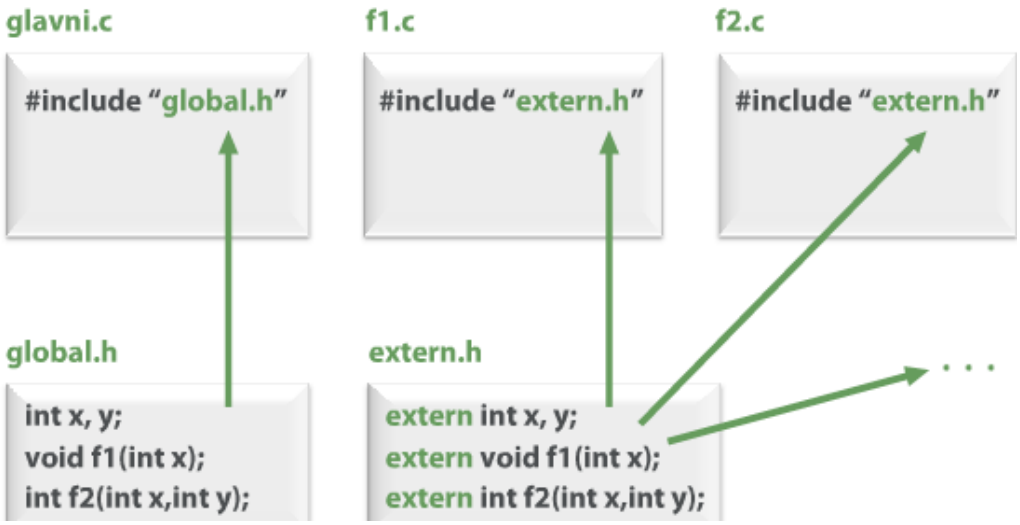
Слика 6.8 Датотеке заглавља укључене у модуле програма



Слика 6.9 Датотеке заглавља укључене у друге датотеке заглавља

Садржај датотека заглавља

Датотека заглавље обично садржи следеће компоненте: директиве *#include* које укључују библиотеке, директиве *#define* које дефинишу симболе, дефиниције изведених типова (као што су на пример структуре) и глобалне и екстерне декларације променљивих и функција. Једна датотека заглавље може садржати глобалну декларацију и онда се она укључује само у један модул програма, а друга датотека заглавље која садржи одговарајућу екстерну декларацију може се укључити произвољан број пута, слика 6.10.



Слика 6.10 Глобално декларисане променљиве и фукције у датотекама заглављима

6.3. Претпроцесорска директива *#define*

Макрои

Претпроцесорска директива *#define*, поред симболичких константи, може дефинисати и макрое. Макро представља симбол који се на почетку програма дефинише да замени одређени *C* израз, а у изворном коду програма пише се на сваком месту где је потребан израз који замењује. Макро може бити заступљен у два своја могућа облика: макро - објекат и функцијски макро, слика 6.11. Макро – објекат може се увести да замени било који *C* израз, слика 6.12, а ако је из *C* библиотеке користи се на уобичајени начин, слика 6.13.

Слика 6.11 Макрои – објекти и функцијски макрои

```
#define GRESKA1 "Nije uspela dodela"
#define GRESKA2 "Nije otvorena datoteka"

void greska(int status)
{
    if(status==1)
        printf("%s", GRESKA1);

    else if(status==2)
        printf("%s", GRESKA2);
}
```

Слика 6.12 Макро – објекат

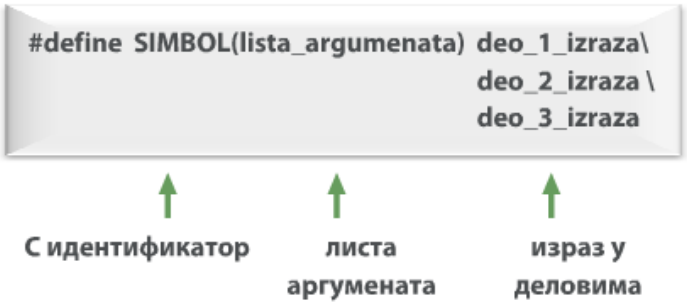
```
#include <stdio.h>

main()
{
    printf( "\n Datum: \t %s", __DATE__ );
    printf( "\n Vreme: \t %s", __TIME__);
    printf( "\n Datoteka: \t %s", __FILE__ );
}
```

Слика 6.13 Предефинисани макро - објекти

Функцијски макрои

Функцијски макро дефинише се помоћу симбола, листе формалних аргумената и блока наредби који замењује макро, слика 6.14. На сваком месту у програму

где је потребан овај блок наредби пише се позив: макро симбол са листом стварних аргумената, слика 6.15. Претпроцесор замењује макро симбол одговарајућим блоком наредби. Предност макроа у односу на функцију је у томе што се не троши време на позив а недостаци су: већи изворни кôд и одсуство провере типова аргумената и резултата позива.



Слика 6.14 Дефиниција функцијског макроа

```
#define POLOVINA(x)  (x) / 2
#define MAX(x,y)     ( (x) > (y) ) ? (x) : (y)

main()
{
   int a=10, b=20;

   printf( "%d", POLOVINA(a+b) );
   printf( "%d", MAX(a,b) );
}
```

Слика 6.15 Дефиниције и позиви функцијских макроа

Писање и коришћење готових макроа

Простор у меморији који заузима замена блока наредби на сваком месту позива макроа није занемарљива, јер је заузеће меморије увек веома битно. Недостатак провере типова аргумената и резултата, код макроа, такође није занемарљив, јер је и прецизност типова увек битна. Из ових разлога препорука је да се програм на језику *C* напише помоћу функција, а само у оним деловима програма где је

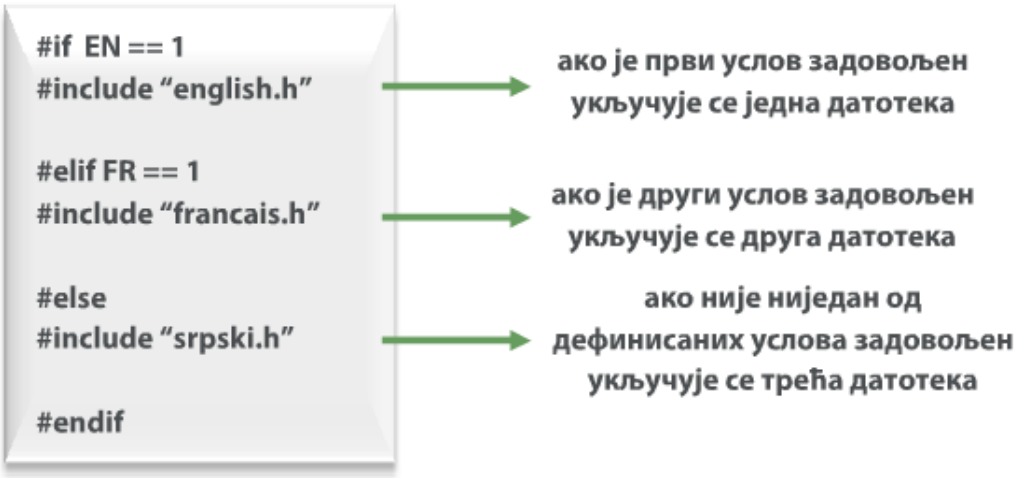
веома критична брзина функције се замењују макроима, али уз пажљиво коришћење типова аргумената и резултата.

6.4. Претпроцесорске директиве *#if, #elif, #else, #endif*

Условно превођење изворног кода

Поред директива које укључују у изворни кôд програма читаве библиотеке и датотеке и оних директива које дефинишу симболе за изразе и блокове наредби, у *C* програмима често се користе и директиве за условно превођење изворног кода. Један такав комплет чине директиве: *#if..#endif (*уз евентуалне директиве *#elif* и *#else)*, слика 6.16. Ове директиве користе се на сличан начин као и наредба *if (*са члановима *else..if* и *else)*, само што се ради о условном превођењу а не о условном извршавању кода, слика 6.17.

Слика 6.16 Претпроцесорске директиве за условно превођење изворног кода, пример 1.

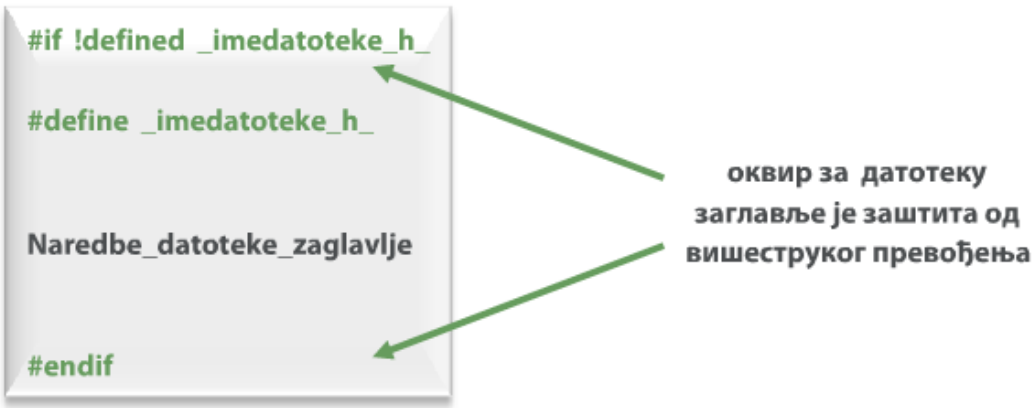


Слика 6.17 Претпроцесорске директиве за условно превођење изворног кода, пример 2.

Заштита од вишеструког превођења изворног кода

У програмима на језику *C* дозвољено је вишеструко уклапање датотека заглавља (једна датотека може укључити другу, друга трећу...) и може се направити грешка: вишеструко превођење изворног кода исте датотеке, слика 6.18. За заштиту од овакве грешке користи се комплет директива: *#if !defined _ime_h_, #define _ime_h_* и *#end*, слика 6.19. Ако ове директиве уоквире сав кôд једне датотеке заглавља на одређени начин, датотека ће бити укључена само једном, без понављања, слика 6.20.

Слика 6.18 Вишеструко превођење делова програма



Слика 6.19 Оквир за датотеку заглавље, од претпроцесорских директива за заштиту од вишеструког превођења делова програма

Слика 6.20 Претпроцесорске директиве за заштиту од вишеструког превођења делова програма, пример

6.5. Питања

1. Које су предности модуларног програмирања?
2. Које датотеке су уобичајено заступљене у модуларним програмима?
3. Шта представљају модули, а шта датотеке заглавља у програмима?
4. Коју екстензију имају модули С програма, а коју уобичајено датотеке заглавља?
5. Који је најмањи број модула за један С програм?
6. Шта уобичајено садржи главни модул С програма?
7. Да ли у једном модулу може бити једна или више функција програма?
8. Да ли је једна функција неопходно у једном модулу или може бити распоређена у више модула програма?
9. Да ли једна датотека заглавље може бити укључена само у модулима програма, или и у модулима и у другим датотекама заглављима?
10. Чему служи екстерна декларација променљивих и функција у програмима?
11. Шта значи дефинишућа, а шта референшућа глобална декларација у програмима?
12. Шта је претпроцесор и које претпроцесорске директиве су најчешће у употреби?
13. Шта су макрои и у којим случајевима је функцију оправдано заменити макроом?
14. Који су главни недостаци коришћења макроа у програмима?
15. Како раде препроцесорске директиве за условно превођење кода програма?

7. Улаз / излаз података, оператори и наредбе у програмима на језику *C++*

Кратак преглед лекције

Програмски језик *C++* је језик у коме су заступљени, сад већ опште-прихваћени, објектно оријентисани концепти. Синтакса овог језика представља проширење синтаксе језика *C*. Користе се и даље неке стандардне C библиотеке, али и *C++* библиотеке. Језик *C++* има један апстрактан тип изведен из типа структуре, то је класа. За приступ класи уводи се објекат класе. И без пројектованих класа у сваком *C++* програму заступљене су готове класе и њихови објекти, функције чланице и оператори чланови.

7.1. Основно о програмском језику C++

Разлози развоја језика C++

Проблеми који су се у прошлости појавили у програмирању на језику *C* били су: све обимнији програми, велики тимови програмера, неусаглашености појединих делова програма... Као решење Бјорн Строуструп *(Bjarne Stroustrup)* је 1980. године написао програмски језик "*C* са класама" а 1984 и језик *"C++"*. Језик је добио свој назив *C++* од: *C* и нешто више, јер има проширену синтаксу језика *C* и нове концепте у пројектовању програма. Основни концепти у језику *C++* су: објектно оријентисано пројектовање програма и модуларно програмирање.

Решење за развој обимних програма на језицима C и C++

Данас постоје решења за развој обимних програма на језицима *C* и *C++*. То је програмски алат за праћење развоја програма *SCCS (Source Code Control System)* који има разне верзије *(Clear Case за Linux, Visual Source Safe за Windows...)*. Коришћење овог програмског алата омогућава: писање програма у деловима, чување свих верзија програма, преглед разлика између појединих

верзија програма... Језик *C++* је замена за језик *C* код развоја програма који су обавезно објектно оријентисано пројектовани (као што су *Windows* апликације).

Развој програма на језику C++

Програмски језик *C++* користи објектно оријентисане концепте пројектовања програма. То значи да пројектовање програма подразумева груписање података и функција у класе и приступ помоћу објеката (класе и објекти су у даљем тексту објашњени). Заступљене су све остале фазе развоја програма, као и на језику *C*, само што су прилагођене објектно оријентисаном пројектовању. Модули *C++* програма имају екстензију *.cpp*, а библиотеке и датотеке заглавља екстензију *.h* (прве обавезно а друге препоручено).

7.2. Проширења у односу на језик *C*

Коришћење синтаксе језика *C*

У језику *C++* важи све што и у језику *C* о следећим елементима: основним и изведеним типовима променљивих, операторима, наредбама, низовима, показивачима, функцијама, структурама, динамички повезаним листама... Проширења у односу на језик *C* заступљена су код: употребе нових готових *C++* библиотека (њихових: класа, објеката, функција и оператора), нових особина функција, новог начина писања наредби декларације, новог начина рада улаза / излаза података, новог начина за динамичку доделу меморије...

Библиотеке језика *C++*

У програмима на језику *C++* користе се стандардне библиотеке језика *C*: *math.h, stdlib.h, string.h*... Нове *C++* библиотеке су: *iostream.h, fstream.h*… Поменуте библиотеке улаза / излаза, поред готових функција нуде и велики број готових оператора и поједностављених решења (иако је коришћење библиотеке *stdio.h* и у језику *C++* дозвољено). Библиотека *iostream.h* је библиотека за улаз / излаз са тастатуром и екраном а библиотека *fstream.h* обухвата улаз / излаз са тастатуром, екраном и датотекама.

Функције

У језику *C++* функције имају особине *C* функција и неке нове особине. Једна од њих је: подразумеване вредности аргумената (ако неки аргумент у формалној листи има наведену вредност, онда није неопходно да му се предаје стварна вредност). Још једна нова особина је: преклапање функција (може бити више функција са међусобно истим именом, изузетак је главна функција). Особина уграђивања функција у кôд, омогућава замену макроа (код функција, као предност у односу на макрое, постоји провера типова).

Наредбе декларације

Језик *C++* дозвољава писање наредби декларације било где у програму (тамо где су потребне) а не само на почетку функције. Декларисана променљива видљива је од места декларације до краја блока (ако је декларисана унутар блока) или до краја функције у којој је написана. Декларација било где у програму значајна је за помоћне и бројачке променљиве, а све остале променљиве обично се декларишу на једном месту у функцији (због лакшег откривања грешака).

Основне компоненте програма

И језик C++ користи директиве: #include (да укључи стандардну библиотеку или датотеку заглавље) и #define (за симболичке константе али не и за макрое). Главна функција опет има два своја облика (са аргументима, да прихвати податке из командне линије и без аргумената). У главној и осталим функцијама заступљене су наредбе разних врста које су познате из језика *C* (наредбе декларације, извршне наредбе и наредбе повратка). У језику C++ линијски коментари могу бити написани од дуплог знака коса црта (//) до краја реда.

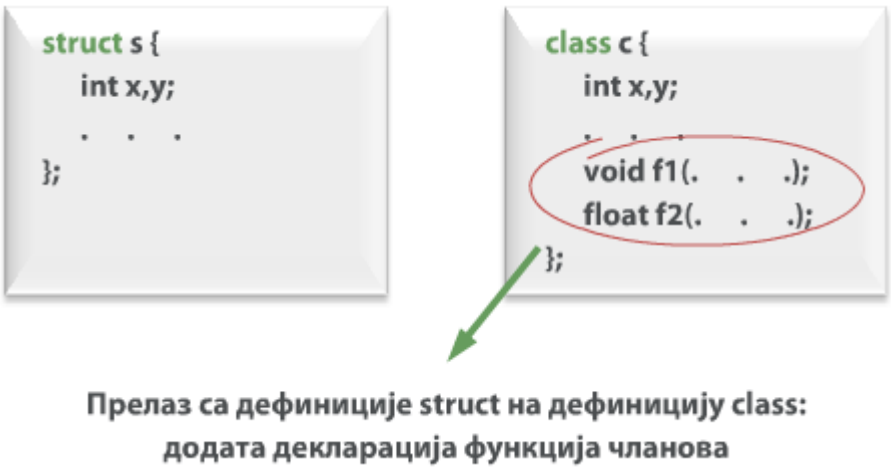
7.3. Класе и објекти потребни механизму улаза / излаза

Механизам улаза / излаза

Механизам улаза / излаза у програмима на језику *C++* базиран је на класама које су дефинисане у *C++* стандардним библиотекама. *C++* има своје библиотеке улаза / излаза и у њима: класе, објекте, функције, операторе... И програми на језику *C++* који немају пројектовање класа, за улаз / излаз података, користе неке готове објекте класа из библиотека, или креирају објекте готових класа из истих библиотека. За коришћење функција и оператора улаза / излаза неопходна су основна знања о класама.

Дефиниција класе

Класа представља проширени тип структуре. За разлику од структуре класа је група чланова – логички повезаних података и чланица – функција над тим подацима. Дефиниција класе изгледа као дефиниција структуре, слика 7.1, само што поред декларације променљивих чланова садржи и декларацију функција чланица. Дефиниција ових функција ван дефиниције класе пише се помоћу оператора досега (::) који дефинише припадање одређеној класи, слика 7.2 (уместо само *ime_funkcije* пише се *ime_klase::ime_funkcije*).



Слика 7.1 Дефиниција класе

Слика 7.2 Дефиниција функција чланица класа

Креирање објеката класе и приступ члановима класе

Објекат класе представља конкретан примерак дефинисане класе. Увођење објекта класе изгледа као и увођење примерка структуре у програм, слика 7.3, само што у језику *C++* није неопходно писање службене речи *struct* или *class* на почетку наредбе (ово важи и за даље наредбе у програму које садрже име примерка структуре или објекта класе). Свака класа у програму може добити произвољан број објеката. За приступ члановима и чланицама класе користи се оператор "тачка" (.) на следећи начин: *ime_objekta.ime_clana*, слика 7.4.

Слика 7.3 Креирање објекта класе

Слика 7.4 Приступ члановима и чланицама класе

Основне особине класа

Основне особине класа су: апстракција, енкапсулација, наслеђивање и полиморфизам. Апстракција значи могућност креирања класа без својих конкретних објеката, које су значајне у програму за извођење других класа. Енкапсулација је скривање података и функција унутар класе. Наслеђивање значи следеће: класа родитељ је општа и модел је за изведене класе, а изведене класе користе дефиниције класа родитеља, уз проширења. Полиморфизам значи да класа родитељ дозвољава преклапање неких својих функција или оператора у изведеној класи (постојање више верзија функција и оператора класе).

7.4. Улаз / излаз података

Класе и објекти улаза / излаза

Библиотека главног стандардног улаза / излаза је *iostream.h*. Класа родитељ за улаз / излаз је *ios* а њене изведене класе су: *istream* (само за улаз) и ostream (само за излаз). За сваки стандарадни улазно / излазни ток резервисан је по један објекат одговарајуће класе: *cin* за тастатуру (објекат класе *istream*), *cout* за екран (објекат класе *ostream*) и *cerr* за грешке или екран без могућности преусмеравања (такође објекат класе *ostream*). Над овим објектима примењују се функције и оператори чланице њихових класа.

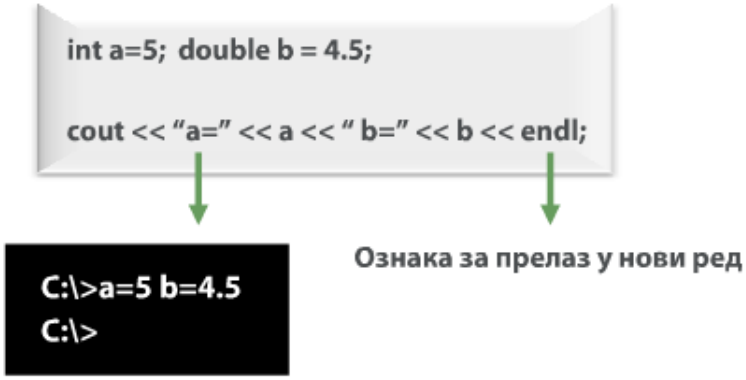
Форматиран улаз / излаз и преклапање оператора

Форматиран улаз / излаз у језику *C++* једноставнији је него у језику *C*. Постоје оператори улаза *(>>)* и излаза *(<<)* који су применљиви над објектима својих класа (*cin>>, cout<<* и *cerr<<*). Оператори се преклапају, што значи да се исти симбол користи за више оператора. Један симбол *(>>)* користи се за више оператора улаза, слика 7.5, а други за више оператора излаза, слика 7.6. Оператор зависи од типа променљиве. Постоје оператори који раде са *stringovima*, слика 7.7, а више њих користи се за комбинацију типова, слика 7.8.

Слика 7.5 Преклапање код оператора улаза

Слика 7.6 Преклапање код оператора излаза

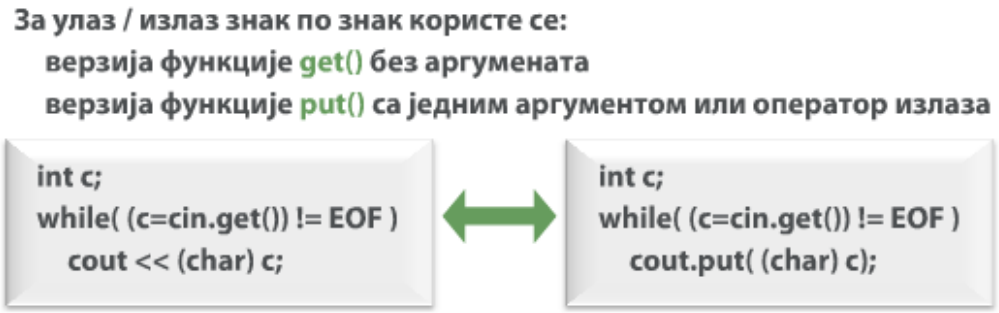
Слика 7.7 Оператор излаза за стрингове



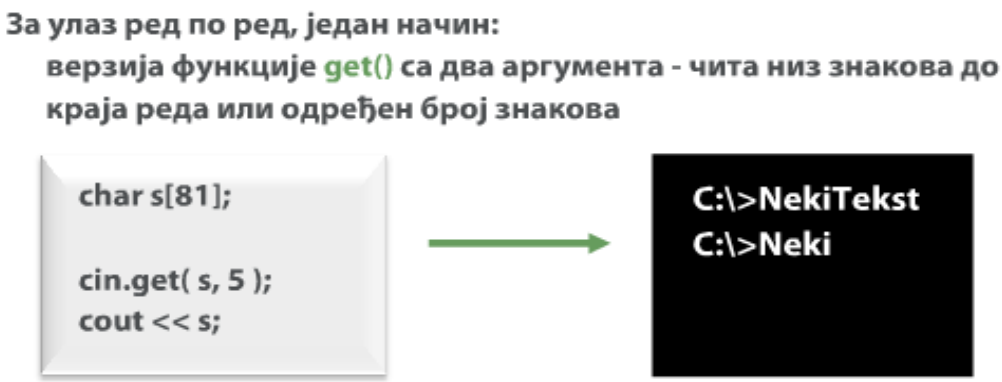
Слика 7.8 Оператор излаза за комбиноване типове података

Улаз / излаз знак по знак и ред по ред и преклапање функција

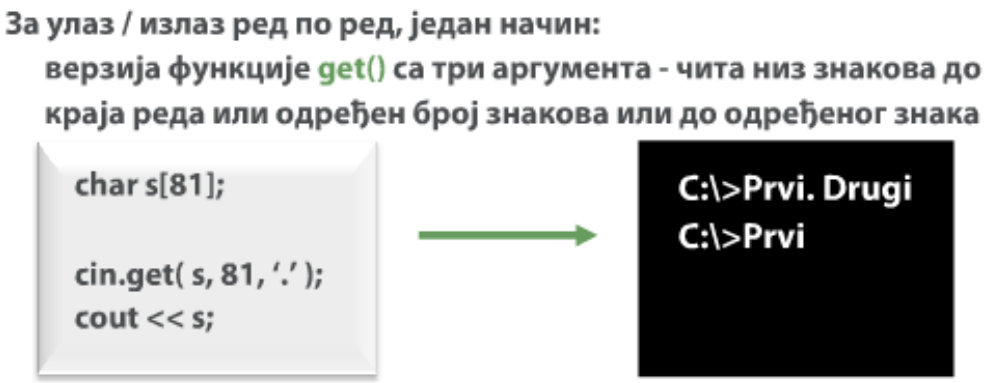
У класи *istream* постоји више функција са именом *get()*, које се међусобно разликују по листи аргумената. Функција *get()* ради као функција *getchar()* у језику *C*. У комбинацији са функцијом улаза *get()* користи се функција излаза *put()*, слика 7.9. Функција *get(string,MAX+1)* ради као функција *gets()* у језику *C*, само што чита или до знака за прелаз у нови ред, или до *(MAX+1)*-ог знака, слика 7.10. Функција *get(string,MAX+1,znak)* има још један аргумент, то је знак до ког се чита, слика 7.11. Оператор излаза (<<) користи се у свакој варијанти.



Слика 7.9 Једна верзија функције get() и функција put()



Слика 7.10 Друга верзија функције get()



Слика 7.11 Трећа верзија функције get()

Функције за постављање формата улаза / излаза

Форматирање улаза / излаза на неподразумевани начин у језику *C++* омогућавају готове функције, слика 7.12. Функција *setf()*, на пример, поставља формат који важи све до позива функције *unsetf()* за укидање истог формата, а функција *width()* дефинише формат само за прву следећу наредбу улаза / излаза. Ове функције у аргументима добијају вредности готових *flagova*, слика 7.13. Свако форматирање у језику *C*, за целобројне, слика 7.14 и реалне податке, слика 7.15, има свој еквивалент у језику *C++*.

Назив функције	Опис функције
setf()	Постављање формата
unsetf()	Укидање постављеног формата
width()	Дефинисање минималне ширине поља за приказ податка
fill()	Попуњавање празних места испред податка нулама
precission()	Дефинисање прецизности са којом се приказује податак

Слика 7.12 Функције за форматирање

flag	Опис flaga
ios::left	Равнање уз леву ивицу поља за приказ екрана
ios::oct	Приказ целог броја у окталном облику
ios:hex	Приказ целог броја у хексадецималном облику
ios::uppercase	Хексидецималан облик са великим словима
ios::fixed	Приказ реалног броја са децималном тачком и стандардном прецизношћу
ios::scientific	Приказ реалног броја са експонентом

Слика 7.13 Формат flagovi (ios::flags)

За иницијализовану целобројну променљиву x:

Наредба у језику C	Еквивалент у језику C++
printf("%d", x);	cout<<x;
printf("%010d", x);	cout.width(10); cout.fill(0); cout<<x;
printf("%-10d", x);	cout.width(10); cout.setf(ios::left); cout<<x;
printf("%o", x);	cout.setf(ios::oct); cout<<x; (cout << oct << x;)
printf("%x", x);	cout.setf(ios::hex); cout<<x; (cout << hex << x;)
printf("%X", x);	cout.setf(ios::hex | ios::uppercase); cout<<x;

Слика 7.14 *C* и *C++* еквивалентни код излаза, за целобројне променљиве

За иницијализовану реалну променљиву y:

Наредба у језику C	Еквивалент у језику C++
printf("%f", y);	cout<<y;
printf("%10f", y);	cout.width(10); cout<<y;
printf("%.2f", y);	cout.setf(ios::fixed); cout.precision(2); cout<<y;
printf("%10.2f", y);	cout.width(10); cout.setf(ios::fixed); cout.precision(2); cout<<y;
printf("%e", y);	cout.setf(ios::scientific); cout<<y;

Слика 7.15 *C* и *C++* еквивалентни код излаза, за реалне променљиве

6.5. Питања

1. Које су стандардне библиотеке улаза / излаза у C++ језику?
2. Да ли постоје и дефинисани објекти у стандардним библиотекама улаза / излаза у C++ језику?
3. Које класе су највише у употреби за рад са тастатуром и екраном, од класа стандардних библиотеке улаза / излаза у C++ језику?
4. Које класе су највише у употреби за рад са датотекама, од класа стандардних библиотеке улаза / излаза у C++ језику?
5. Шта представљају идентификатори cin, cout и cerr у C++ стандардним библиотекама?
6. Који се симбол користи за оператор улаза, из C++ стандардне библиотеке?
7. Који се симбол користи за оператор излаза, из C++ стандардне библиотеке?
8. Шта значи преклапање оператора у C++ језику?
9. Да ли је у C++ језику могуће и преклапање оператора?
10. Како раде функције get() и put() у C++ језику?
11. Која је разлика између функција get() и getline() у C++ језику?
12. Како раде функције setf() и unsetf() у C++ језику?
13. Шта се од оператора / функција користи у C++ језику за форматирани улаз / излаз?
14. Шта се од оператора / функција користи у C++ језику за улаз / излаз знак по знак?
15. Шта се од оператора / функција користи у C++ језику за улаз / излаз ред по ред?

8. Функције, динамичка додела меморије, низови и структуре у програмима на језику *C++*

Кратак преглед лекције

У програмском језику *C++* важи све што важи за функције у језику *C*. Објектно оријентисани концепти омогућили су и неке нове особине *C++* функција: подразумеване вредности аргумената, уграђивање у кôд, преклапање и позив по референци помоћу референце. За динамичку доделу меморије језик *C++* користи аритметику показивача и операторе: *new* и *delete*. И у овом језику, као и у језику *C*, заступљен је тип структуре података који је послужио као основа за развој типа класе и објектно оријентисаних концепата.

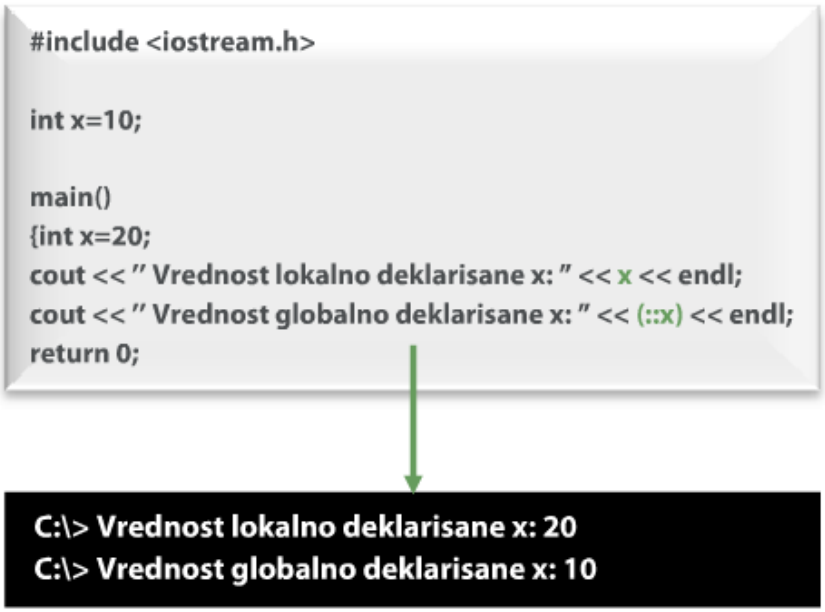
8.1. Глобални и локални досег у програму, именски простори

Језик *C++* наследио је правила досега променљивих од језика *C*: глобална екстерна променљива може бити видљива у свим модулима једног програма, глобална статичка променљива видљива је од места декларације у модулу до краја тог модула, локална променљива видљива је од места декларације у функцији до краја те функције. У *C++* програму променљива може бити декларисана било где унутар функције. Ако је декларисана унутар једног блока функције важи видљивост до краја тог блока, слика 8.1.

Слика 8.1 Правила досега променљивих

Оператор резолуције досега

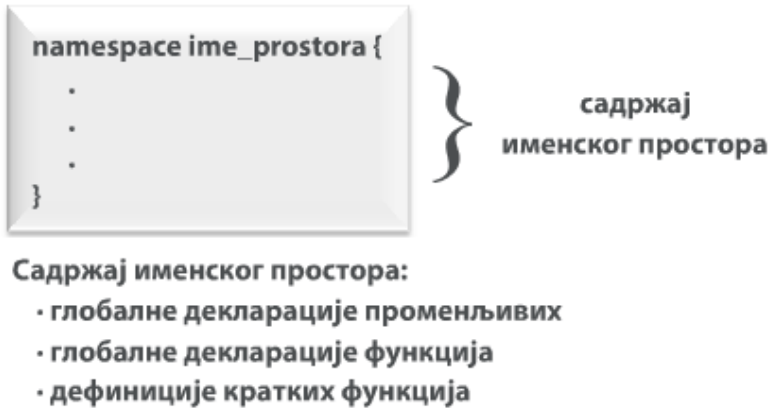
Језик *C++* дозвољава, као и језик *C*, да две променљиве, глобална и локална, имају међусобно исто име. У овом случају локална декларација доминантна је над глобалном (унутар функције увек важи локална декларација променљиве). Језик *C++* дозвољава да се унутар функције приступи глобалној променљивој, која има исто име као локална, помоћу оператора резолуције досега (::). Овај оператор написан испред имена променљиве (::x) значи: приступ глобалној променљивој под тим именом, слика 8.2.



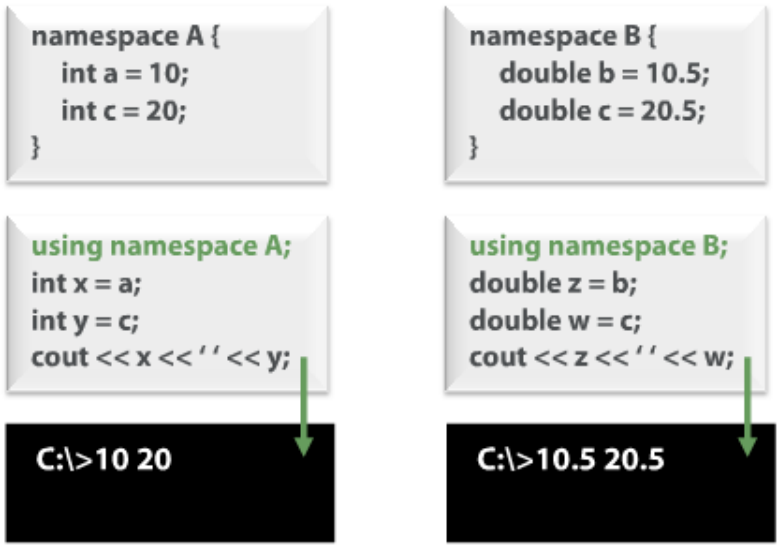
Слика 8.2 Оператор резолуције досега

Именски простори

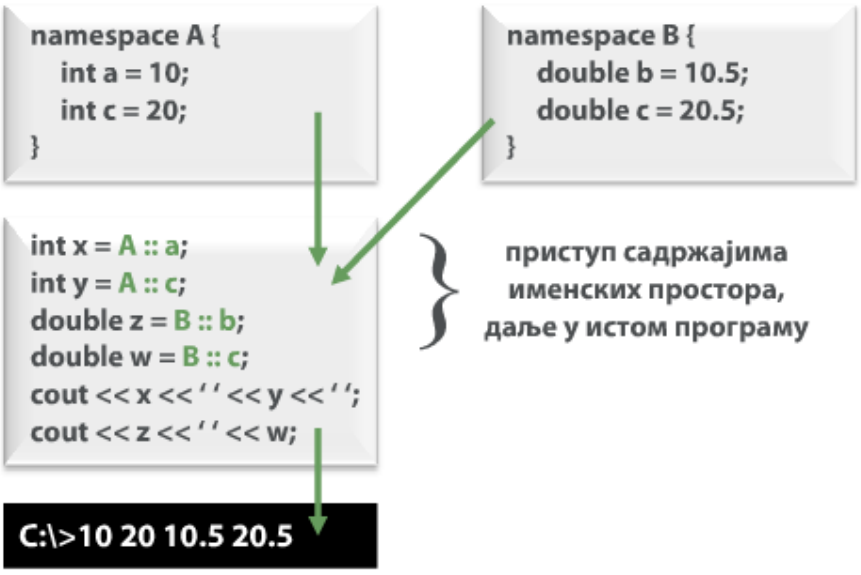
Оператор резолуције досега користи се и код именских простора. Именски простор је група глобално декларисаних променљивих и функција, која је заштићена од екстерне декларације, слика 8.3. Приступ комплетном садржају таквог простора могућ је и ван њега (*using namespace ime_prostora)*, слика 8.4. За приступ појединим деловима именског простора користи се оператор резолуције досега (*ime_prostora::*), слика 8.5. Грешку при коришћењу именских простора, слика 8.6, може елиминисати употреба оператора, слика 8.7.



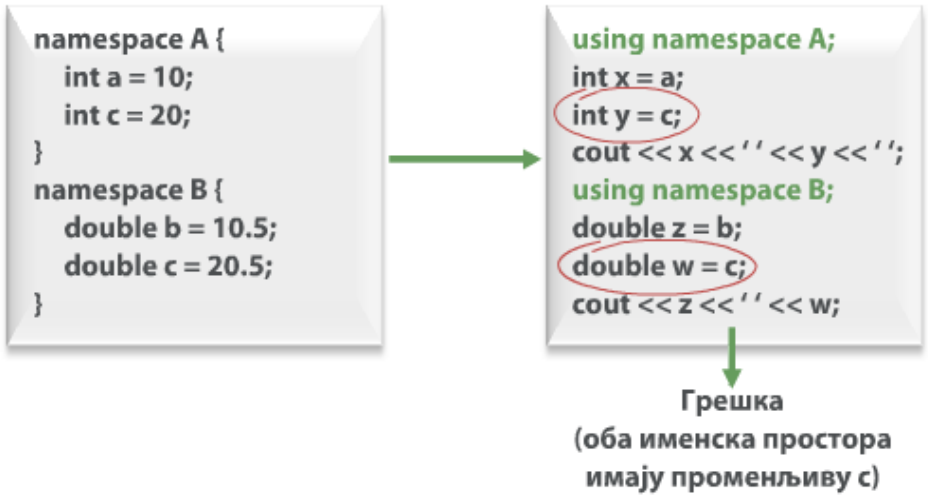
Слика 8.3 Дефиниција именског простора



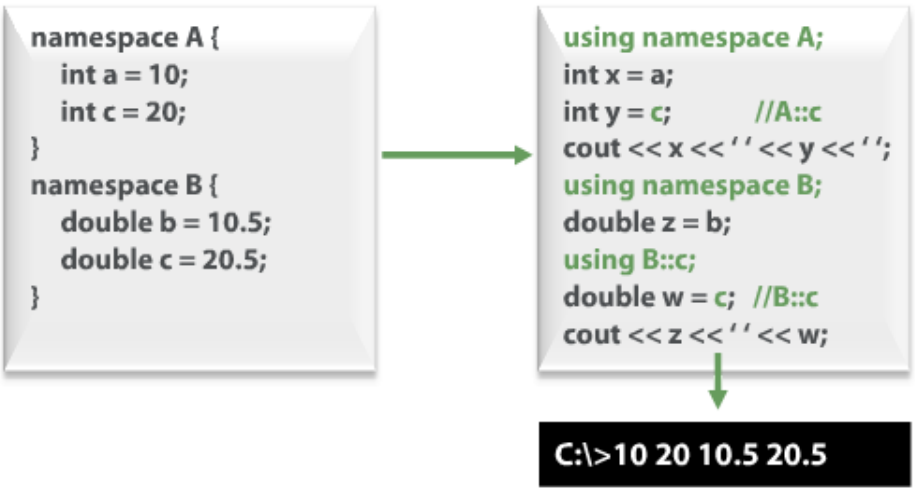
Слика 8.4 Приступ комплетном садржају именског простора



Слика 8.5 Приступ деловима именског простора



Слика 8.6 Коришћење именских простора, пример 1.



Слика 8.7 Коришћење именских простора, пример 2.

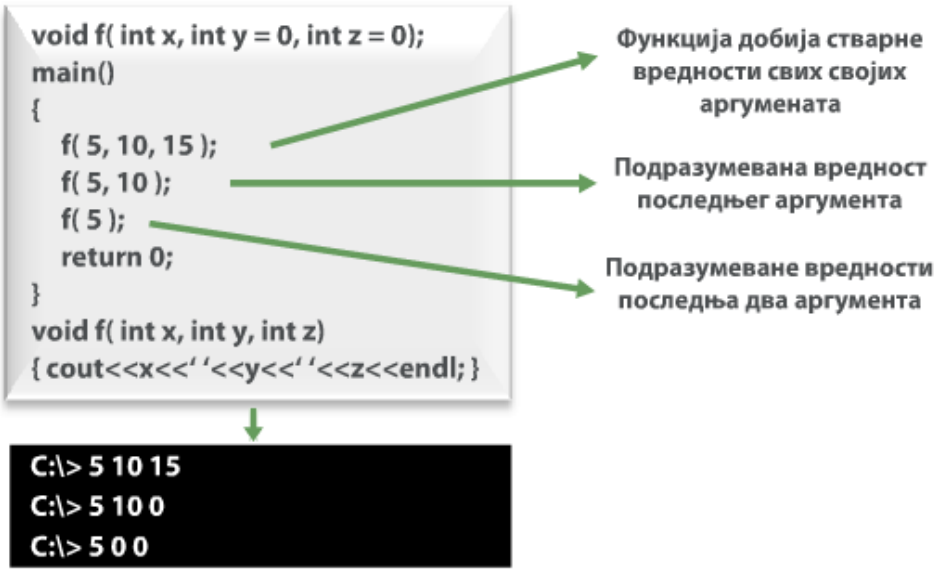
8.2. Функције са подразумеваним вредностима аргумената

У језику *C* број стварних аргумената функције исти је као и број формалних аргумената, или информацију о броју стварних аргумената садржи један од аргумената са којима се функција декларише (промењљива листа аргумената). У језику *C++* могуће су и функције са подразумеваним вредностима аргумената. У листи формалних аргумената, при декларацији, сваком аргументу може бити додељена вредност. Ако при позиву функције недостаје стварна вредност неког аргумента, аутоматски се додељује подразумевана.

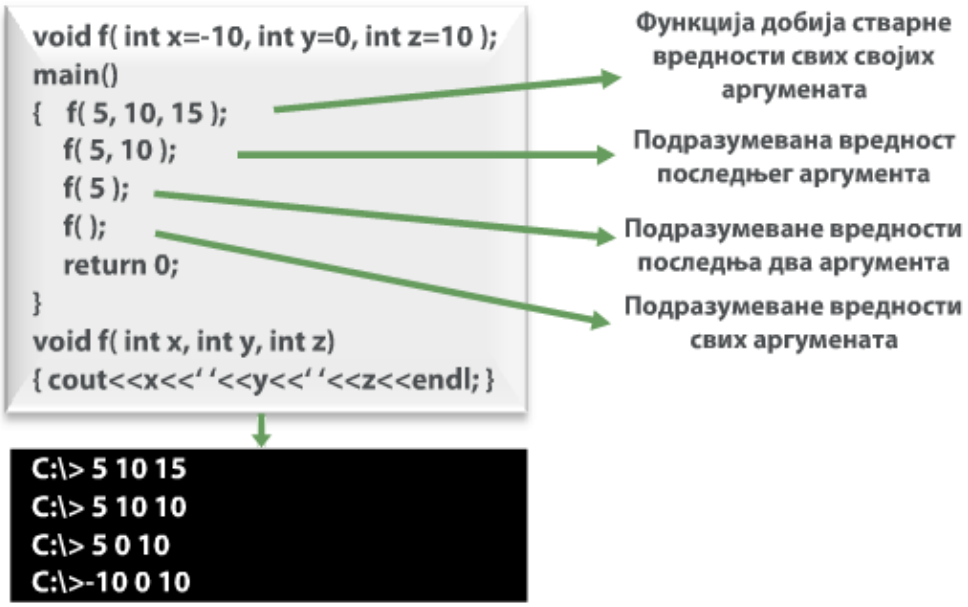
Правила код подразумеваних вредности аргумената функција

Функција у језику *C++,* може имати подразумеваних вредности аргумената, колико и аргумената. При томе, подразумевани могу бити само последњи аргументи: један последњи, слика 8.8, претпоследњи и последњи, слика 8.9,..., или сви аргументи, слика 8.10. Правило да се редослед аргумената поштује тако што се подразумевају аргументи с краја формалне листе, слика 8.11, а не од почетка, или из средине листе, слика 8.12, неопходно је да би при позиву функције било познато које су стварне а које подразумеване вредности.

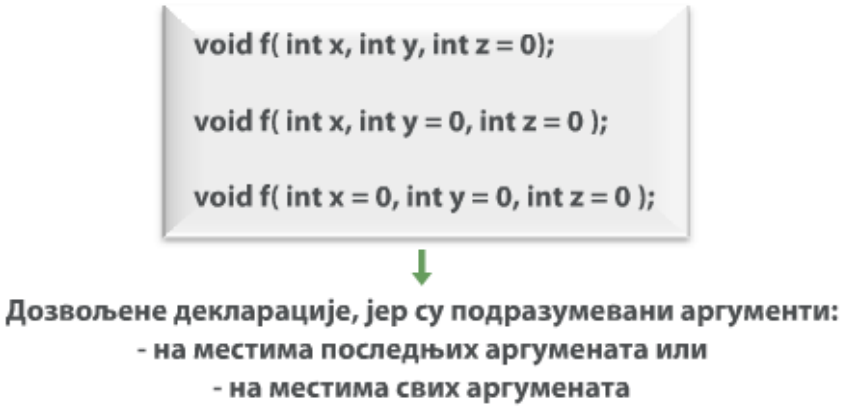
Слика 8.8 Подразумеване вредности аргумената, пример 1.



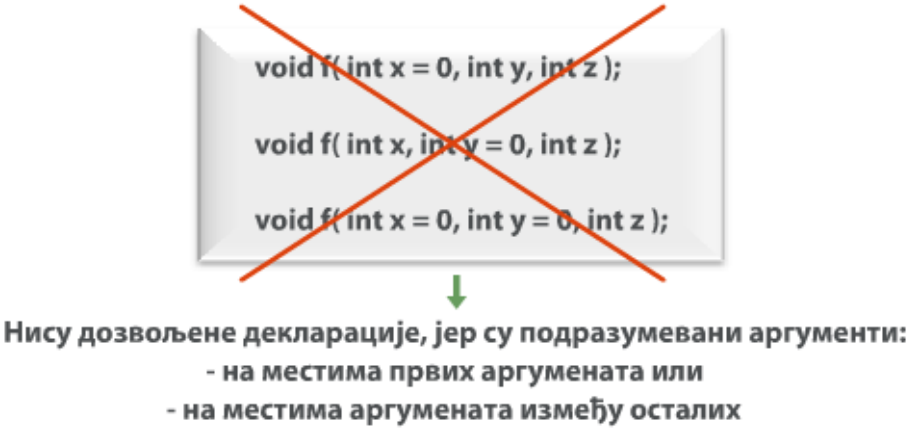
Слика 8.9 Подразумеване вредности аргумената, пример 2.



Слика 8.10 Подразумеване вредности аргумената, пример 3.



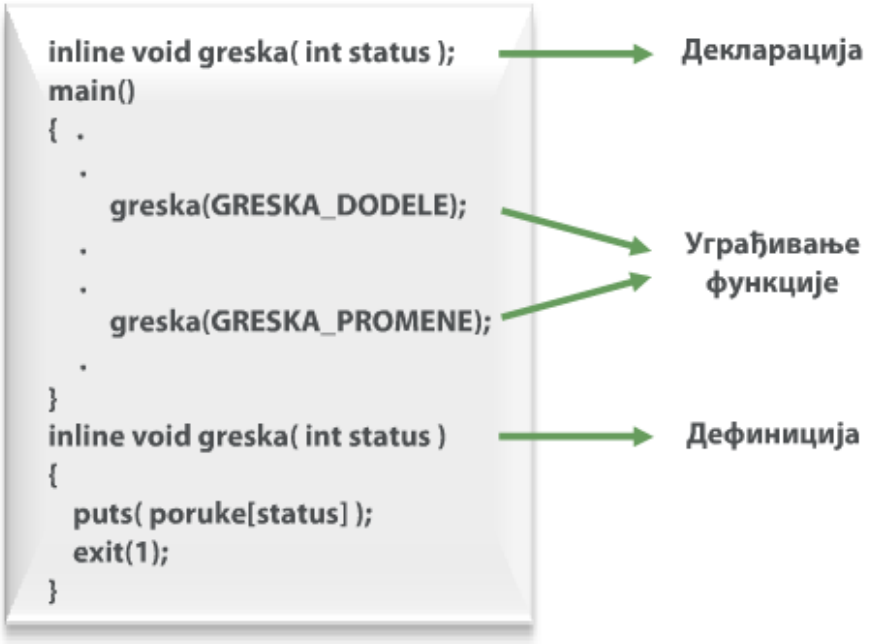
Слика 8.11 Подразумеване вредности аргумената, могући случајеви



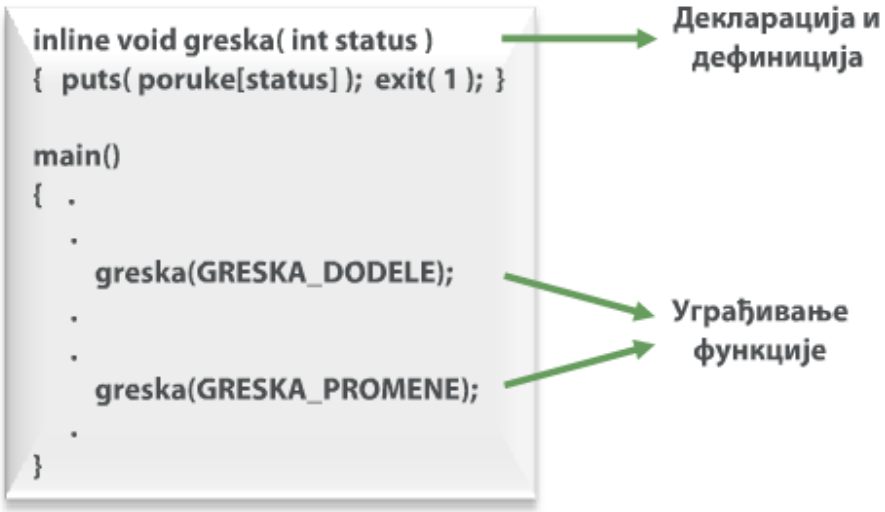
Слика 8.12 Подразумеване вредности аргумената, немогући случајеви

8.3. Функције уграђене у кôд

Функције уграђене у кôд представљају замену за макрое (решење за бројне недостатке који су заступљени код макроа). Ако се на почетку декларације функције напише службена *C++* реч *inline*, она значи захтев преводиоцу да се функција угради на свако место њеног позива, слика 8.13. Оправдано је тражити ово само за кратке функције, са учесталим позивима. Ако се одобри овај захтев нема трошења времена на позив / повратак функције. Дефиниција *inline* функције може бити написана унутар наредбе декларације, слика 8.14.



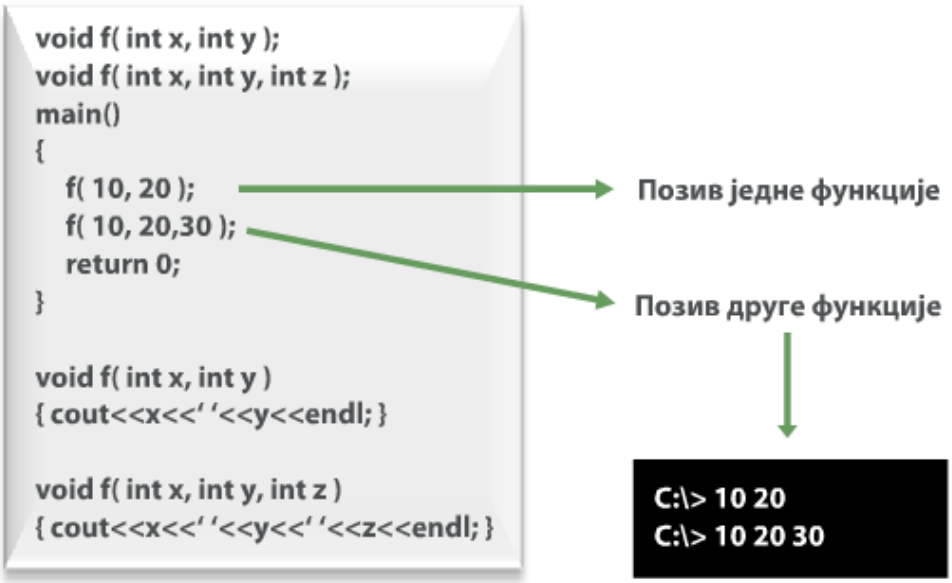
Слика 8.13 Функције уграђене у изворни код



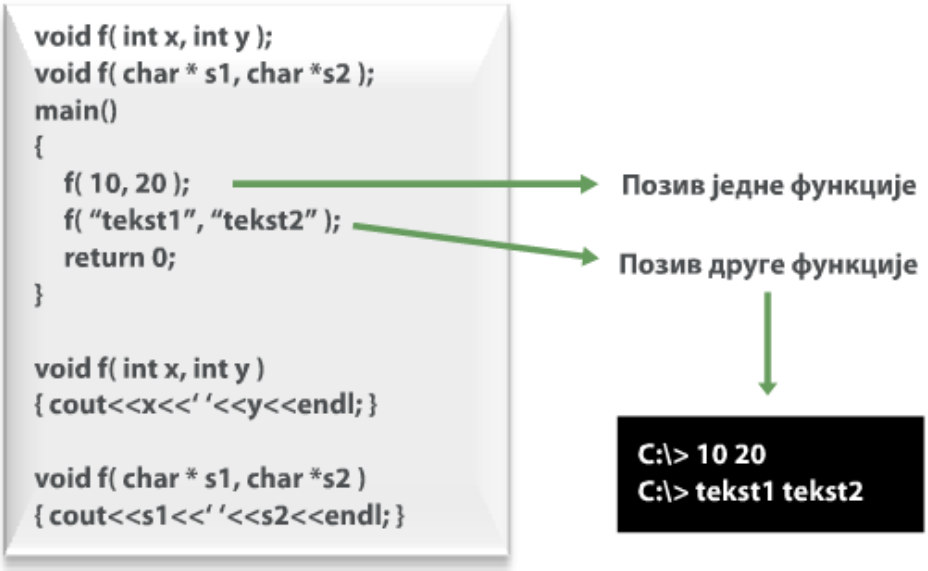
Слика 8.14 Функције уграђене у изворни код, дефинисане унутар декларације

8.4. Преклопљене функције

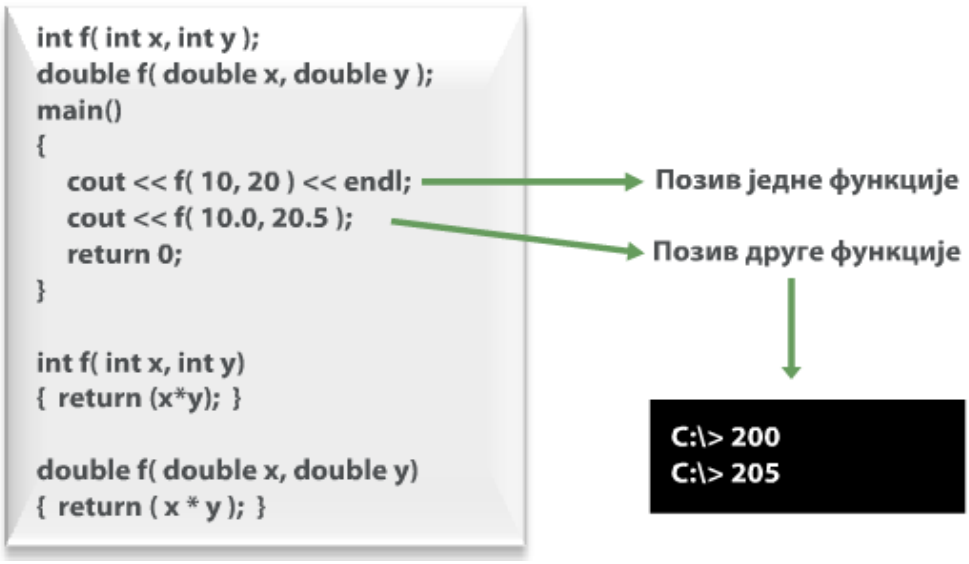
Преклопљене функције су функције истог програма које имају међусобно исто име У том случају неопходна је разлика међу њима: у броју аргумената, слика 8.15, или у типу бар једног аргумента, слика 8.16. Може бити разлике и у типу повратне вредности, слика 8.17. На основу броја и типова аргумената при позиву бира се једна од преклопљених функција. Разлика између две истоимене функције не може бити само у типу повратне вредности, слика 8.18, јер онда не би био могућ избор функције.



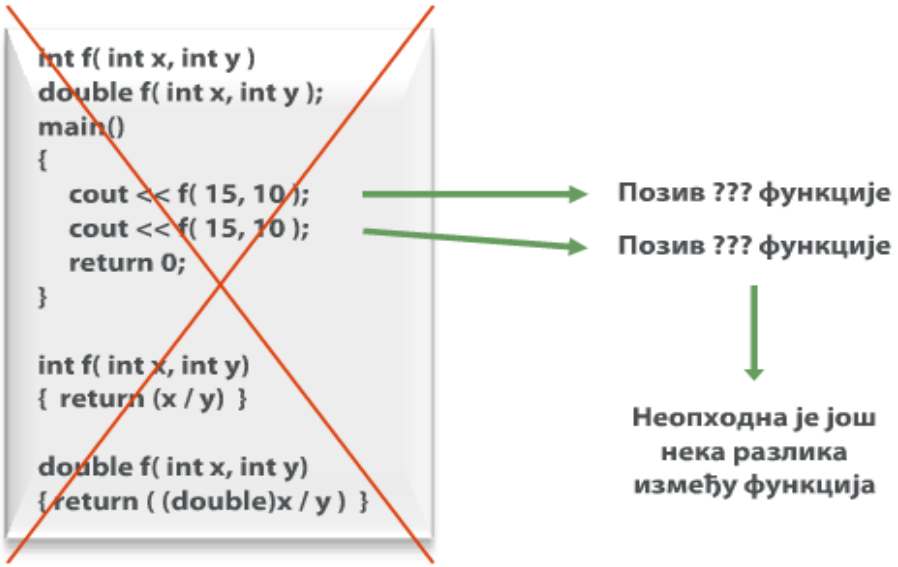
Слика 8.15 Преклопљене функције, разлика у броју аргумената



Слика 8.16 Преклопљене функције, разлика у типовима аргумената



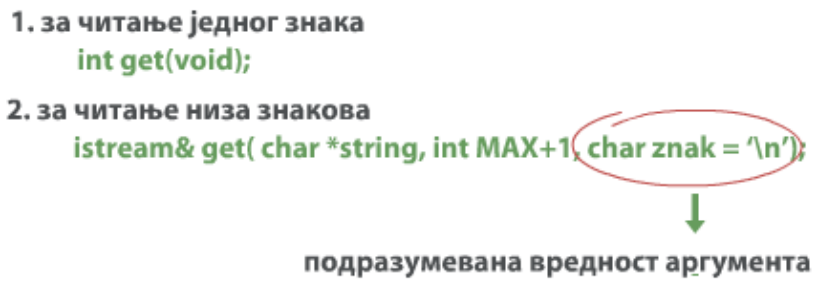
Слика 8.17 Преклопљене функције, разлика у типовима аргумената и повратне ведности од функције



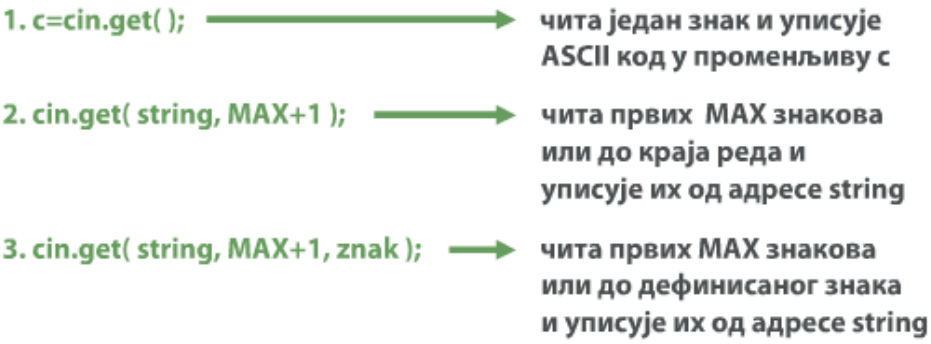
Слика 8.18 Немогући случајеви преклапања функција

Преклопљена функција *get()*

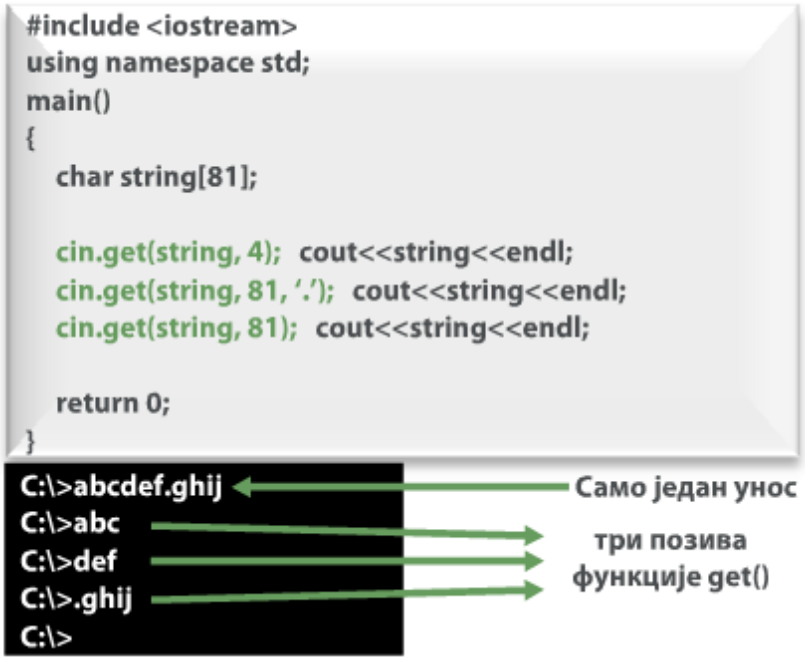
У стандардним *C++* библиотекама постоји велики број преклопљених функција. Једна од њих је функција *get()* која има две преклопљене верзије, слика 8.19, а у другој верзији и подразумевани аргумент. Ова функција може се користити за читање: 1) једног знака, 2) одређеног броја знакова, или до знака за прелаз у нови ред и 3) одређеног броја знакова, или до дефинисаног знака, или до знака за прелаз у нови ред, слика 8.20. Функција *get()* припада класи *istream*. Ако се позове над објектом *cin*, чита са тастатуре, слика 8.21.



Слика 8.19 Функција get() са преклапањем и подразумеваним аргументима



Слика 8.20 Могући облици позива функције get()



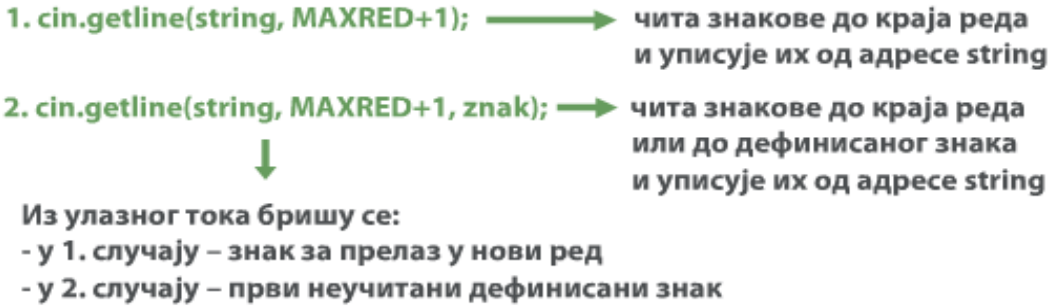
Слика 8.21 Примери позива функције get()

Функција *getline()*

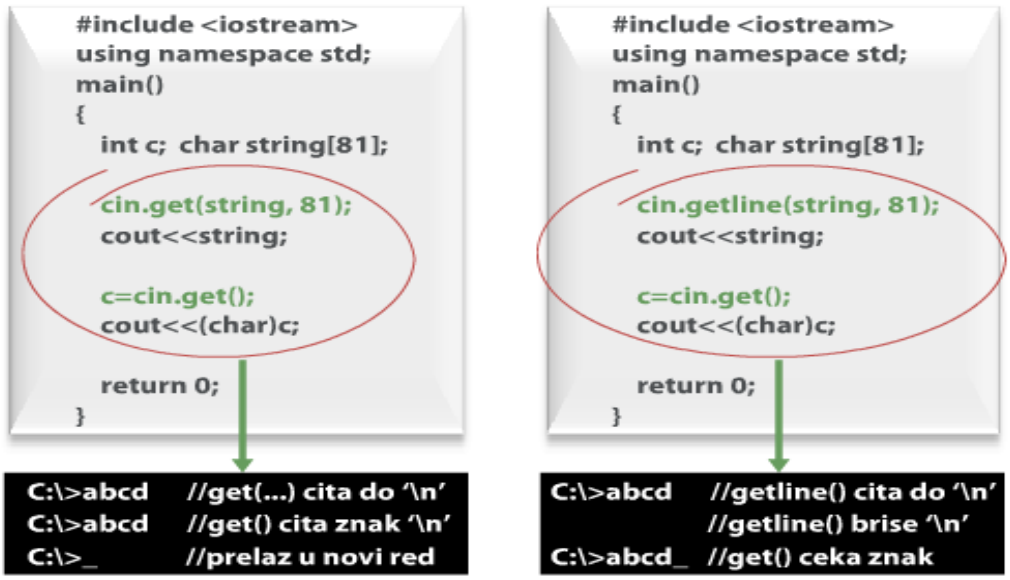
Функција *getline()* нема преклапање, али је нашла овде место јер се користи као замена за функцију *get()* ако је потребно да се са улазног тока избрише и знак за прелаз у нови ред (потврда уноса података). Ова функција има подразумеван

последњи аргумент, слика 8.22, и може се користити за читање: 1) одређеног броја знакова или до знака за прелаз у нови ред и 2) одређеног броја знакова или до дефинисаног знака или до знака за прелаз у нови ред, слика 8.23. Ако се позове над објектом *cin* чита са тастатуре, слика 8.24.

Слика 8.22 Функција getline() са подразумеваним вредностима аргумената



Слика 8.23 Могући облици позива функције getline()

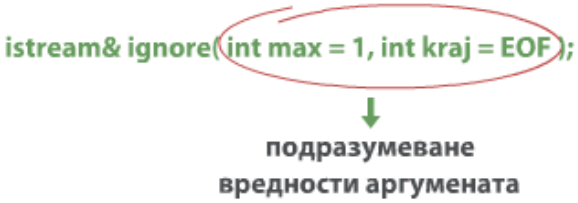


Слика 8.24 Примери позива функција get() и getline()

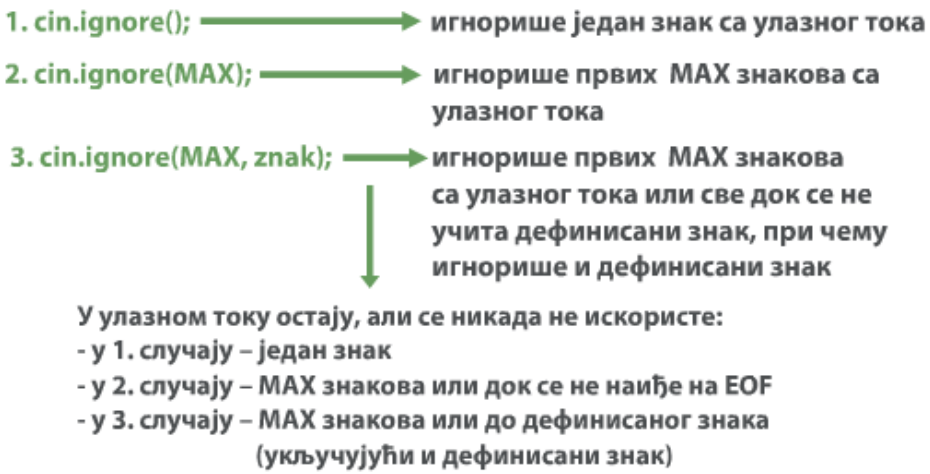
Функција *ignore()*

Функција *ignore()* такође нема преклапање, али се уобичајено користи у комбинацији са функцијама *get()* и *getline()* ако је потребно са улазног тока

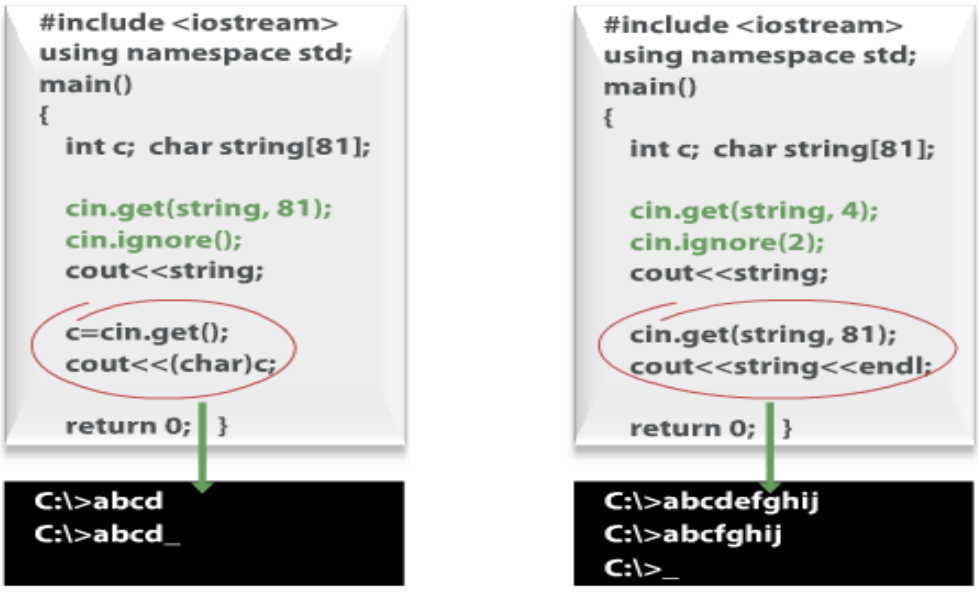
игнорисати један или више знакова (остају у улазном току али се више не користе). Ова функција има подразумевана оба аргумента, слика 8.25, и може се користити за игнорисање: 1) једног знака, 2) одређеног броја знакова и 3) до дефинисаног знака, слика 8.26. Ако се позове над објектом *cin* функција игнорише знакове са тастатуре, слика 8.27.



Слика 8.25 Функција ignore() за игнорисање података са улазног тока



Слика 8.26 Могући облици позива функције ignore()

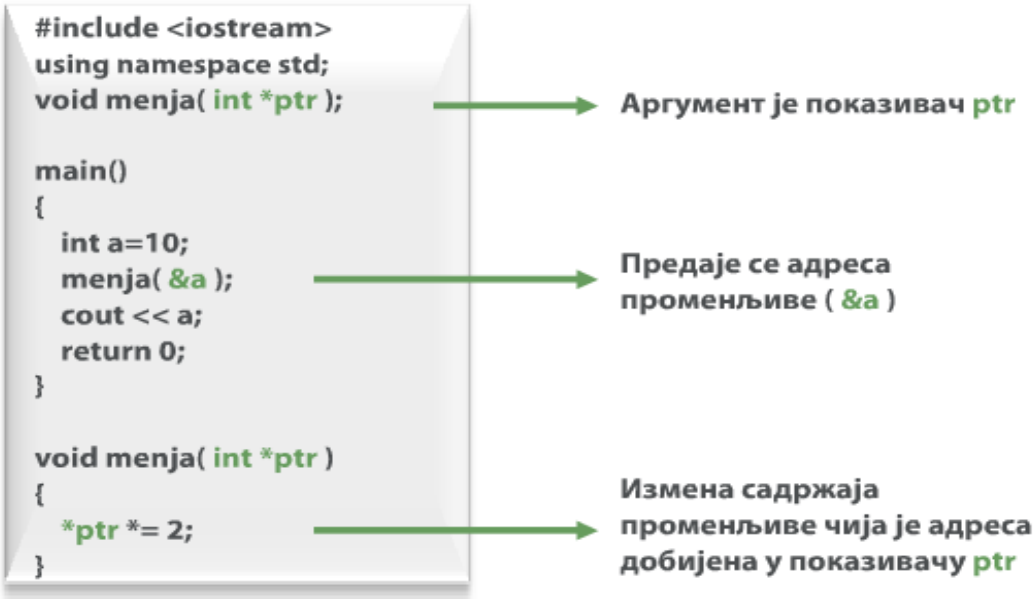


Слика 8.27 Примери позива функција get() и ignore()

8.5. Функције позване по референци

Позив функције по референци помоћу показивача

Функција може бити позвана по референци помоћу показивача. Функција у том случају може користити и изменити садржај меморијске локације чију адресу добија у показивачу. Писање овакве функције, као и у језику *C,* захтева коришћење оператора индиректног приступа *(*)* и адресног оператора *(&)*, што некад даје доста сложенији изглед наредбама саме функције, слика 8.28. За функције позване по референци, језик *C++* има и алтернативни начин писања: позив функције по референци помоћу референце.

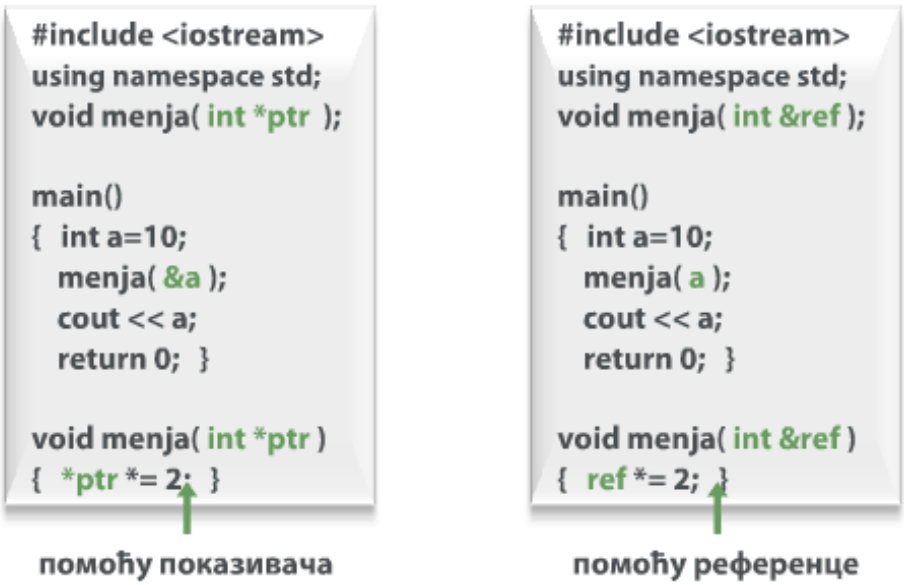


Слика 8.28 Позив функције по референци помоћу показивача

Позив функције по референци помоћу референце

Функција у језику *C++* може бити позвана по референци помоћу референце. Разлика у односу на позив по референци помоћу показивача само је у нешто једноставнијем начину писања. У формалној листи пише се адресни оператор и име референце *(&ref)* која се придружује аргументу, а у самој функцији приступа се аргументу помоћу ове референце *(ref)*. При позиву функције пише се стварно име аргумента, слика 8.29. Овај начин писања је једноставнији, слика 8.30, али није применљив код функција које добијају адресе низова.

Слика 8.29 Позив функције по референци помоћу референце придружене основном типу



Слика 8.30 Позив функције по референци помоћу показивача и референце

8.6. Динамичка додела меморије

Оператори *new* и *delete*

За динамичку доделу меморије језик *C++* има операторе. Уместо функције *malloc()* користи се оператор *new*, уместо функције *free()* оператор *delete* а функција *realloc()* нема одговарајућег оператора. Оператор *new* додељује

тражени меморијски простор ако је слободан, а на основу броја тражених објеката даје као резултат адресу додељеног простора. Додељен простор може бити иницијализован ако се не ради о низу. Оператор *delete* ослобађа меморијски простор само ако је динамички додељен.

Динамичка додела меморије за променљиву

Ако се тражи додела меморије за променљиву основног типа, или типа структуре, после оператора *new* пише се само ознака типа, а оператор аутоматски одређује број тражених бајтова. Ако је овај број бајтова слободан од неке адресе у меморији резервише се. Резултат примене оператора *new* потребно је сачувати у одговарајућем показивачу и проверити (да није *NULL* вредност). При ослобађању меморије, после оператора *delete* пише се име показивача са адресом од које је динамички додељена меморија, слика 8.31.

```
adresa = new tip_promenljive;                  //za osnovni tip
adresa = new tip_promenljive(pocetna_vrednost);

adresa = new tip_strukture;                    //za strukturu
adresa = new tip_strukture[ velicina_niza];    //za 1d niz struktura

delete adresa;              //za pojedinacne promenljive i 1d nizove
```

Слика 8.31 Динамичка додела и ослобађање меморије

Динамичка додела меморије за низ

При додели меморије за низ после оператора *new* пише се ознака типа низа и између угластих заграда дужина низа. Ако се ради о вишедимензионалном низу за сваку димензију додаје се пар заграда и димензија унутар заграда. При ослобађању меморије за једнодимензионалан низ после оператора *delete* пише се име показивача са додељеном адресом, а за вишедимензионалан низ празан пар угластих заграда и тек онда име показивача, слика 8.32. Пре коришћења меморије потребно је проверити њену доделу, слика 8.33.

```
adresa = new tip_niza[ velicina_niza];                  //za 1d niz
adresa = new tip_niza[ broj_nizova][velicina_niza];  //za 2d niz

delete [ ] adresa;      //za pojedinacne promenljive i 1d nizove
delete [ ] adresa;      //samo za visedimenzionalne nizove,
                        //jedan par zagrada bez obzira na dimenzije!
```

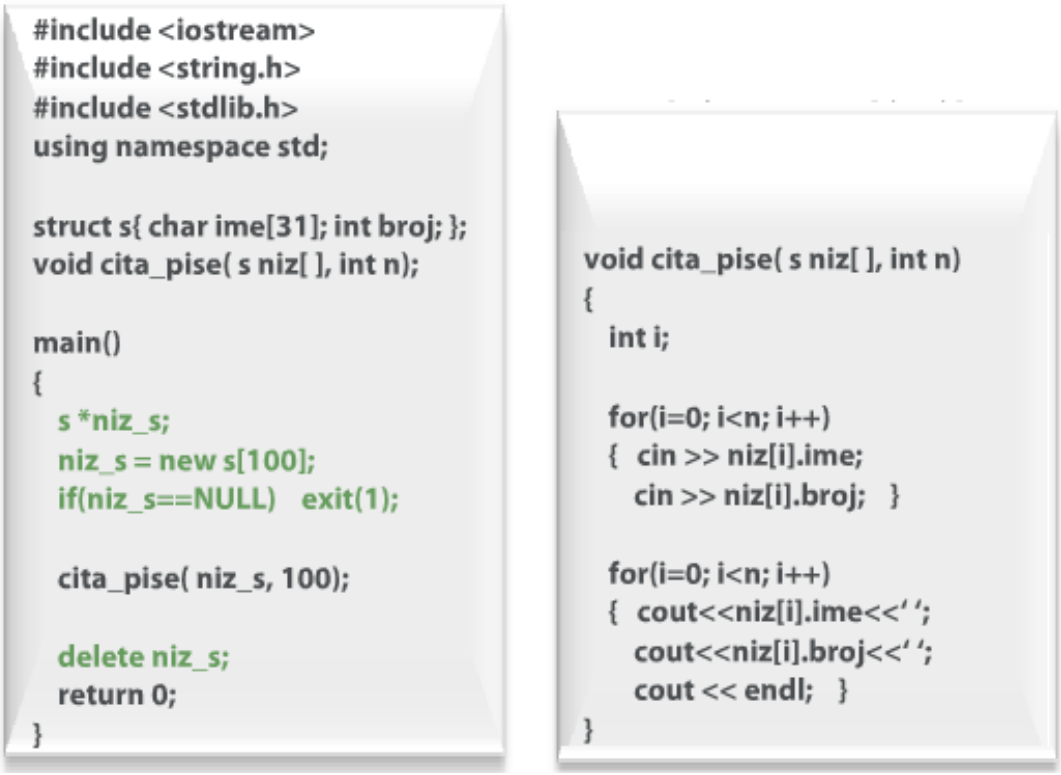
Слика 8.32 Динамичка додела и ослобађање меморије за низ

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

main()
{  int i, *niz;                    → Статичка декларација показивача

   niz = new int[10];               → Динамичка додела за низ

   for(i=0; i<10; i++)
   {  cin >> niz[i];
      cout << niz[i] << ' ';  }

   delete [] niz;                   → Укидање низа
   return 0;
}
```

Слика 8.33 Динамичка додела и ослобађање меморије за низ основног типа

Динамичка додела меморије за низ структура

Структуре у језику *C++* могу се дефинисати и декларисати на исти начин као и у језику *C*. Ново на језику *C++* је што је могућ једноставнији запис: на сваком месту у програму где је потребно назначити тип структуре може се изоставити на службена реч *struct* (уместо *struct s s1;* може се писати *s s1;* или уместо *void f(struct s s1);* може се писати *void f(s s1);*). Динамичка додела меморије за структуру или низ структура реализује се помоћу оператора *new*, слика 8.34.



```
#include <iostream>
#include <string.h>
#include <stdlib.h>
using namespace std;

struct s{ char ime[31]; int broj; };
void cita_pise( s niz[ ], int n);

main()
{
  s *niz_s;
  niz_s = new s[100];
  if(niz_s==NULL)   exit(1);

  cita_pise( niz_s, 100);

  delete niz_s;
  return 0;
}
```

```
void cita_pise( s niz[ ], int n)
{
  int i;

  for(i=0; i<n; i++)
  {  cin >> niz[i].ime;
     cin >> niz[i].broj;   }

  for(i=0; i<n; i++)
  {  cout<<niz[i].ime<<' ';
     cout<<niz[i].broj<<' ';
     cout << endl;   }
}
```

Слика 8.34 Динамичка додела и ослобађање меморије за низ структура

8.7. Питања

1. Која је примена оператора резолуције досега у C++ језику?
2. Шта представља именски простор у C++ језику?
3. Да ли је доминантнија глобална или локална декларација у C++ функцији?
4. Шта значи подразумевање аргумената код C++ функција?
5. Да ли C++ функција може имати подразумеване аргументе, без обзира на то колико аргумената она има?
6. Који аргументи могу бити подразумевани код C++ функције?
7. Да ли се код међусобно преклопљених C++ функција може разликовати само број аргумената?
8. Да ли се код међусобно преклопљених C++ функција могу разликовати само типови аргумената?
9. Да ли се код међусобно преклопљених C++ функција може разликовати само тип повратне вредности?
10. Шта представља inline функција у C++ језику?
11. Како се све могу проследити аргументи C++ функцији?
12. Да ли се C++ функцији може проследити структура само по вредности, само по референци, или по вредности и по референци?
13. Који оператор се користи у C++ језику за динамичку доделу меморије?
14. Како ради и шта враћа као резултат, оператор за динамичку доделу меморије у C++ језику?
15. Који оператор се користи у C++ језику за ослобађање динамички додељене меморије?

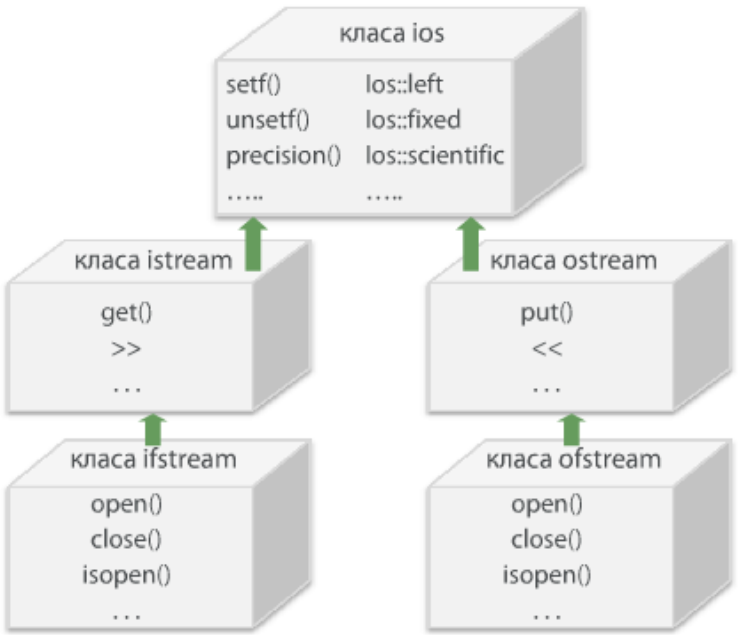
9. Рад са датотекама у програмима на језику C++

Кратак преглед лекције

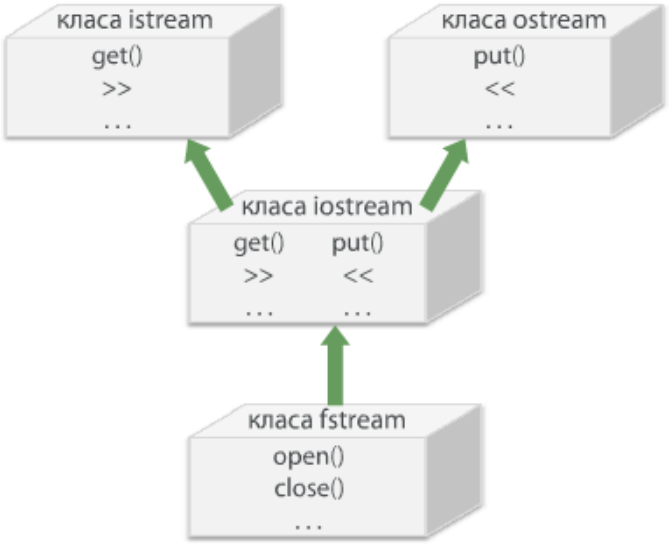
Рад са датотекама у програмима на језику *C++* разликује се од оног у програмима на језику *C*. Рад са датотекама, као и у случају улаза / излаза тастатура / екран, базиран је на класама и њиховим објектима. За рад са сваком датотеком потребно је дефинисати објекат одговарајуће класе (само улаза, само излаза или улаза / излаза) и повезати га са датотеком. И у програмима на језику *C++* користе се повезане листе за рад са великим бројем структура података.

9.1. Улазно / излазни токови за рад са датотекама

У *C++* хијерархији класа, слика 9.1, из класа *istream* (комплетног улаза) и *ostream* (комплетног излаза) изведене су класе *ifstream* (за читање из датотека) и *ofstream* (за упис у датотеке). За рад са датотекама користе се функције чланице ових класа. Особине класа *ifstream* и *ofstream* обједињене су у класи *fstream* (за упис / читање код датотека), слика 9.2. Ако су потребне класе за рад са датотекама укључује се библиотека *fstream.h*, која омогућава комплетан улаз / излаз (нема потребе за укључивањем библиотеке *iostream.h*).



Слика 9.1 Хијерархија класа улаза / излаза, део 1.

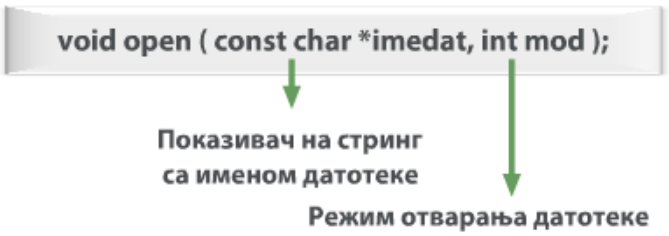


Слика 9.2 Хијерархија класа улаза / излаза, део 2.

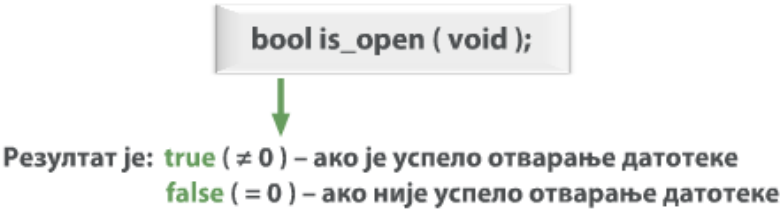
Приступ датотекама преко улазно / излазних токова

Рад са датотекама у *C++* програму садржи ове кораке: 1) креирање објекта одговарајуће класе (само улазног, само излазног или улазно / излазног тока), слика 9.3; 2) отварање датотеке (улазно / излазног тока) и повезивање датотеке са током помоћу функције *open()*, слика 9.4; 3) проверу статуса датотеке на један од могућих начина: помоћу функције *is_open()*, слика 9.5, помоћу функције *fail()*, слика 9.6, или провером креираног објекта, слика 9.7; 4) упис / читање и 5) затварање датотеке помоћу функције *close()*, слика 9.8.

Слика 9.3 Креирање објекта улазно / излазног тока



Слика 9.4 Функција за отварање датотеке



Слика 9.5 Једна функција за проверу статуса датотеке

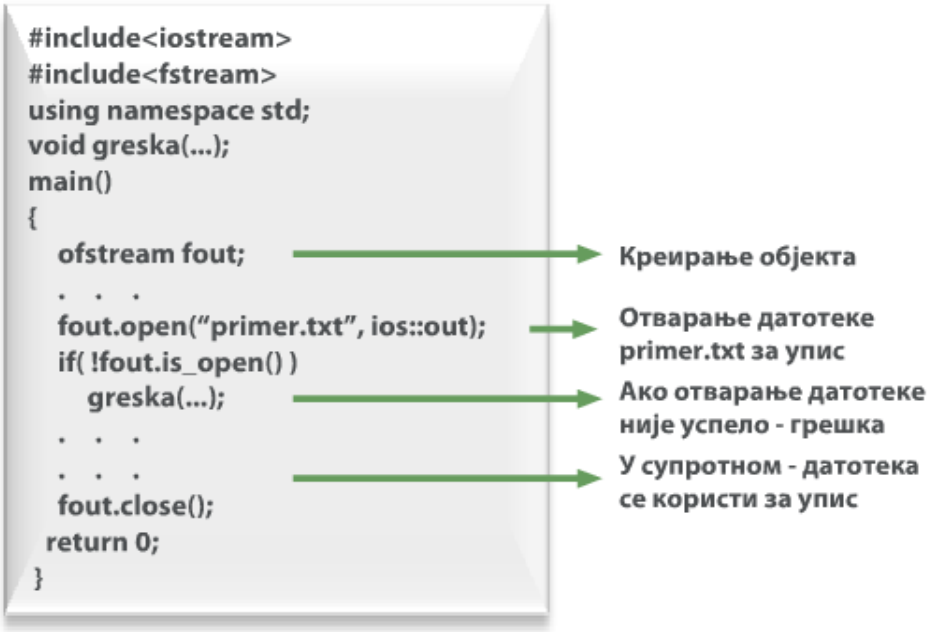
Слика 9.6 Друга функција за проверу статуса датотеке

Слика 9.7 Три начина за проверу статуса датотеке

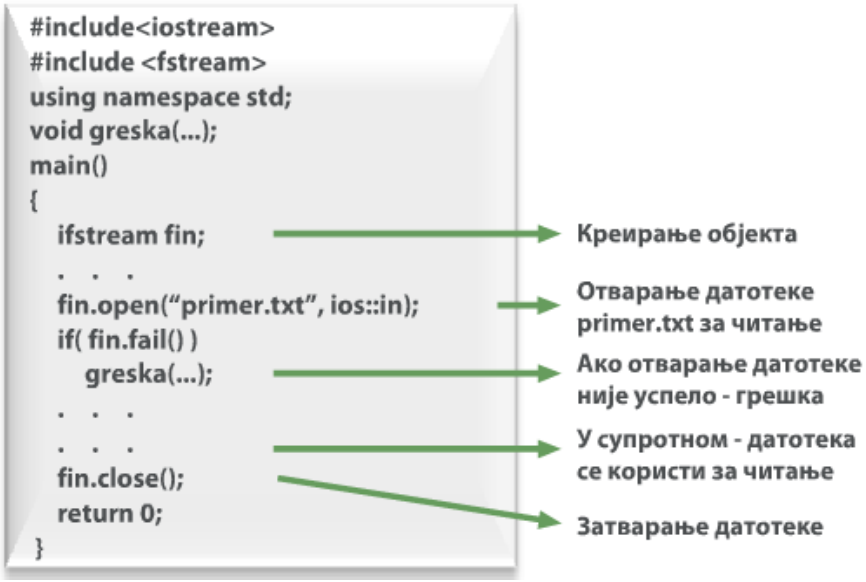
Слика 9.8 Функција за затварање датотеке

9.2. Режими отварања улазно / излазних токова за рад са датотекама

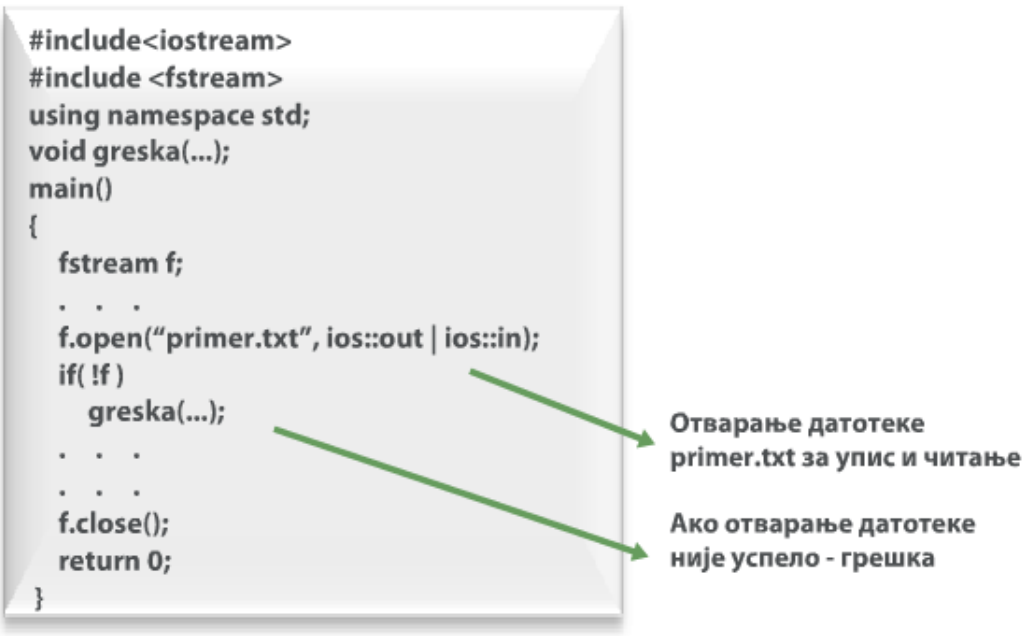
Без обзира на то да ли се у програму тражи отварање датотеке за упис, слика 9.9, за читање, слика 9.10, или за упис / читање, слика 9.11, пре приступа датотеци неопходно је дефинисати њено име и проверити да ли је могуће отварање у траженом режиму. Функција за отварање датотеке *open()* има два аргумента. Један аргумент функције *open()* је показивач на *string* са именом датотеке. Садржај овог *stringa* може бити иницијализован при његовој декларацији, слика 9.12, или даље у програму, слика 9.13.



Слика 9.9 Отварање текстуалне датотеке за упис



Слика 9.10 Отварање текстуалне датотеке за читање



Слика 9.11 Отварање текстуалне датотеке за упис и читање

```
char ime_dat[ ] = "primer.txt";
```

```
char ime_dat[ ] = "C:\\biblioteka\\primer.txt";
```

Слика 9.12 Експлицитно задавање имена датотеке

```
char ime_dat[MAX+1] ;
cin.get(ime_dat, MAX+1);
```

Слика 9.13 Задавање имена датотеке са тастатуре

Режим отварања текстуалне датотеке

Други аргумент функције *open()* је константа из библиотеке *fstream.h*, која садржи информацију о режиму отварања датотеке. Подразумевана врста датотеке је текстуална, слика 9.14. На месту другог аргумента може бити више константи које су међусобно повезане оператором "или на нивоу бита" (|), слика 9.15. Ако се за рад са датотеком уводи објекат класе *ofstream*, подразумева се

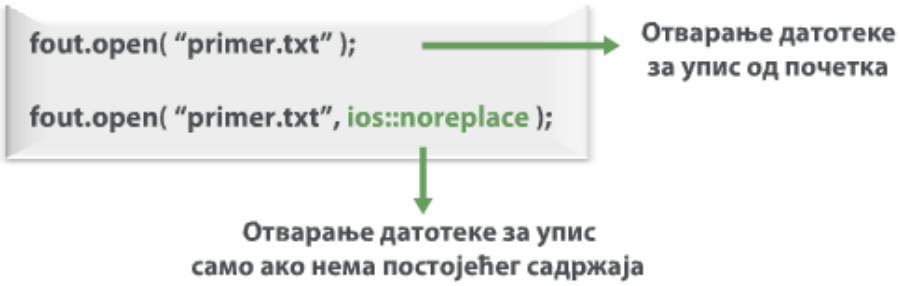
основни режим уписа, слика 9.16. Исто важи и за читање, ако је објекат класе *ifstream* подразумева се основни режим читања, слика 9.17.

Константа	Значење
ios::out	Отварање за упис од почетка, постојећи садржај се преписује
ios::noreplace	Ако постоји садржај не преписује се, већ је резултат отварања датотеке грешка
ios::app	Отварање за упис у продужетку постојећег садржаја
ios::in	Отварање за читање, ако датотека не постоји креира се
ios::nocreate	Ако датотека не постоји не креира се, већ је резултат отварања датотеке грешка

Слика 9.14 Режими отварања текстуалне датотеке

```
f.open( "primer.txt", ios::out );

f.open( "primer.txt", ios::out | ios::noreplace );

f.open( "primer.txt", ios::app );

f.open( "primer.txt", ios::in );

f.open( "primer.txt", ios::in | ios::nocreate );

f.open( "primer.txt", ios::out  | ios::in );

f.open( "primer.txt", ios::app | ios::in );
```

Слика 9.15 Примери отварања текстуалне датотеке



Слика 9.16 Отварање текстуалне датотеке само за упис

Слика 9.17 Отварање текстуалне датотеке само за читање

Режим отварања бинарне датотеке

За отварање бинарне датотеке важе исти режими као код текстуалне, са додатком константе *(ios::binary)* која се односи на бинарни режим, слика 9.18. И у овом случају више константи може бити међусобно повезано оператором "или на нивоу бита" (|), слика 9.19. Ако се за рад са датотеком уводи објекат класе излаза *ofstream* подразумева се основни режим уписа, слика 9.20, а ако је објекат класе улаза *ifstream* подразумеван је основни режим читања, слика 9.21, тако да није неопходно наводити ове основне режиме.

Ознака	Значење
ios::out | ios::binary	Отварање за упис од почетка, постојећи садржај се преписује
ios::noreplace | ios::binary	Ако постоји садржај не преписује се, већ је резултат отварања датотеке грешка
ios::app | ios::binary	Отварање за упис у продужетку постојећег садржаја
ios::in | ios::binary	Отварање за читање, ако датотека не постоји креира се
ios::nocreate | ios::binary	Ако датотека не постоји не креира се, већ је резултат отварања датотеке грешка

Слика 9.18 Режими отварања бинарне датотеке

```
f.open( “primer.bin”, ios::out | ios::binary );

f.open( “primer.bin”, ios::out | ios::noreplace | ios::binary );

f.open( “primer.bin”, ios::app | ios::binary );

f.open( “primer.bin”, ios::in | ios::binary );

f.open( “primer.bin”, ios::in | ios::nocreate | ios::binary );

f.open( “primer.bin”, ios::out  | ios::in | ios::binary );

f.open( “primer.bin”, ios::app | ios::in | ios::binary );
```

Слика 9.19 Примери отварања бинарне датотеке

```
fout.open( “primer.bin”, ios::binary);

fout.open( “primer.bin”, ios::noreplace | ios::binary);
```

Слика 9.20 Отварање бинарне датотеке само за упис

```
fin.open( “primer.bin”, ios::binary);

fin.open( “primer.bin”, ios::nocreate | ios::binary);
```

Слика 9.21 Отварање бинарне датотеке само за читање

9.3. Функције за упис / читање

Упис / читање код текстуалних датотека

За форматиран упис и читање код текстуалних датотека користе се оператори улаза (>>) и излаза (<<). Уместо формата у *C* програмима код оператора се размена података реализује на много једноставнији начин: на једној страни оператора пише се име објекта а на другој име променљиве, слика 9.22. За упис / читање знак по знак користе се функције *put() / get()*, слика 9.23, а за упис / читање ред по ред функција *get()* и оператор улаза (<<), слика 9.24. Све ове функције и оператори користе се над објектима који су везани за датотеке.

```
char c;  int i;  double d;
fin >> c >> i >> d;
fout << c << ' ' << i << ' ' << d << endl;
```

```
char string[81];
fin >> string;
fout << string << endl;
```

Слика 9.22 Форматиран упис / читање код текстуалних датотека

```
int c;

c = fin.get();

fout.put( (char)c );
```

Слика 9.23 Упис / читање знак по знак код текстуалних датотека

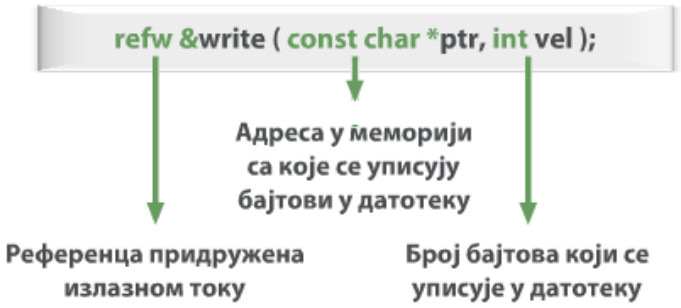
```
char string1[81];
fin.get(string1, 81);
fout << string1 << endl;
```

```
char string2[81];
fin.get(string2, 81, '.');
fout << string2 << endl;
```

Слика 9.24 Упис / читање ред по ред код текстуалних датотека

Упис / читање код бинарних датотека

Функција *write()* уписује одређени број блокова, сваки одређене величине, из меморије у датотеку, слика 9.25. Функција *read()* ради у обрнутом смеру, истим редоследом аргумената учитава одређени број блокова, сваки одређене величине, у меморију из датотеке, слика 9.26. На месту првог аргумента је показивач на тип *char*, тако да је неопходно написати конверзију у показивач на тип *char*, било да се ради о адреси основног типа, слика 9.27, или адреси низа, слика 9.28. Функције се позивају над објектима везаним за датотеке.



Слика 9.25 Функција за упис код бинарних датотека

Слика 9.26 Функција за читање код бинарних датотека

```
int x;
fin.read( (char *)&x, sizeof(int) );
fout.write( (char *)&x, sizeof(int) );
```

```
int niz[5];
fin.read( (char *)niz, 5*sizeof(int) );
fout.write( (char *)niz, 5*sizeof(int) );
```

Слика 9.27 Упис / читање података основног типа код бинарних датотека

```
struct s{ char c; int broj; } s1;
fin.read( (char *)&s1, sizeof(s) );
fout.write( (char *)&s1, sizeof(s) );
```

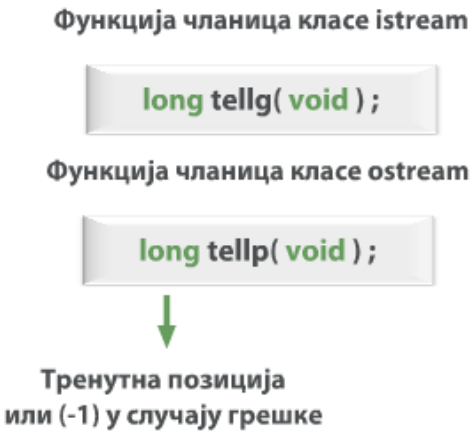
```
s niz_s[10];
fin.read( (char *)niz_s, 10*sizeof(s) );
fout.write( (char *)niz_s, 10*sizeof(s) );
```

Слика 9.28 Упис / читање структура података код бинарних датотека

9.4. Директан приступ и позиционирање у датотекама

Читање тренутне позиције у датотеци

И у програмском језику *C++*, као и у језику *C*, постоје функције које дају вредност тренутне позиције у датотеци и померају ову позицију за потребе операција уписа / читања , слика 9.29. Функција *tellg()* даје као резултат вредност тренутне позиције у датотеци отвореној за читање, а функција *tellp()* у датотеци отвореној за упис, слика 9.30. Ако је датотека отворена за упис / читање може се користити било која од ових функција (у том случају је уобичајено да се непосредно пре читања користи *tellg()*, а пре уписа *tellp()*).



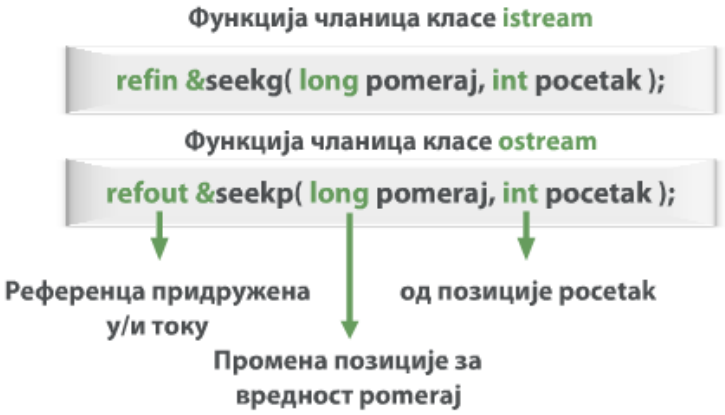
Слика 9.29 Читање тренутне позиције у датотеци

```
ifstream fin;
ofstream fout;

fin.open("dat1.bin", ios::in | ios::binary);
cout << "ip=" << fin.telg();          ip=0
. . .
fout.open("dat2.bin", ios::out | ios::binary);
cout << "ip=" << fout.telp();         ip=0
. . .
fin.close();
fout.close();
```

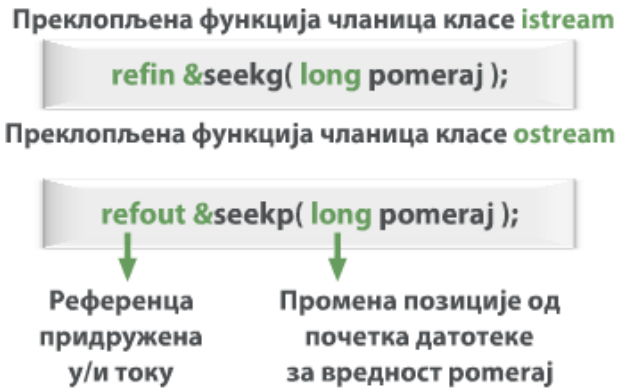
Слика 9.30 Читање тренутне позиције у датотеци

Позиционирање у датотеци

Функције за позиционирање: *seekg()* / *seekp()*, слика 9.31, померају позицију за одређени број бајтова и то: од почетка, од затечене позиције и с краја датотеке отворене за читање / упис. Ако се у овим функцијама изостави аргумент са информацијом о почетној позицији то је подразумевано почетак датотеке, слика 9.32. Други аргумент у свакој од ових функција је константа са информацијом о почетној позицији, слика 9.33. Функције за позиционирање користе се код наизменичних операција уписа / читања, слика 9.34.



Слика 9.31 Позиционирање унутар датотеке, начин 1.

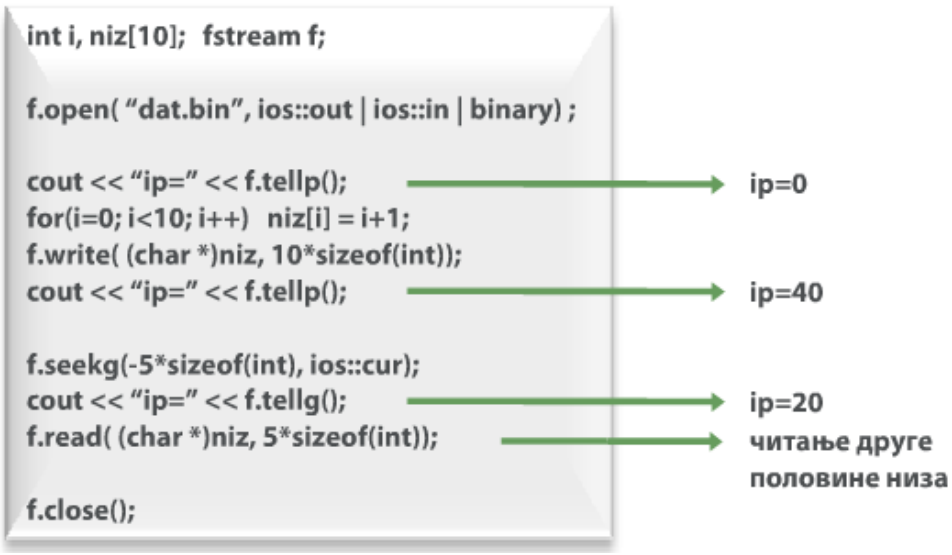


Слика 9.32 Позиционирање унутар датотеке, начин 2.

Константе које се користе на месту 3. аргумента у функцијама seekg и seekp, као ознаке за почетну позицију

константа	вредност	опис
ios::beg	0	Померај од почетка датотеке
ios::cur	1	Померај од тренутне позиције
ios::end	2	Померај од краја датотеке

Слика 9.33 Дефинисање почетне позиције код позиционирања

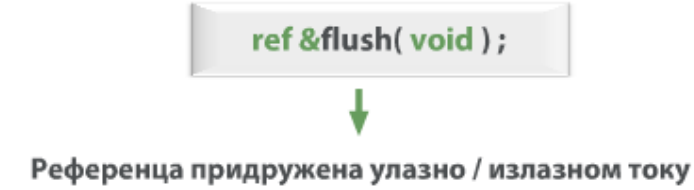


Слика 9.34 Пример позиционирања унутар датотеке

Детекција краја датотеке и читање заосталих карактера

Код текстуалних датотека за детекцију краја датотеке може се користити знак *EOF (EndOfFile)*. Бинарне датотеке немају овај знак и због њих је уведена, а може се користити и код текстуалних датотека, функција *eof()*, слика 9.35. Ова функција враћа вредност различиту од 0 ако је нађен крај датотеке. За решавање проблема карактера заосталих у улазном току, а из датотека, може бити искоришћена функција *flush()*, позвана над објектом тока који одговара датотеци, слика 9.36.

Слика 9.35 Детекција краја текстуалне и бинарне датотеке



Слика 9.36 Брисање заосталих карактера

9.5. Питања

1. Које су класе из стандардних C++ библиотека потребне за рад са датотекама?
2. Која стандардна C++ класа је задужена за рад са датотекама, искључиво у режиму читања њиховог садржаја?
3. Која стандардна C++ класа је задужена за рад са датотекама, искључиво у режиму уписа њиховог садржаја?
4. Која стандардна C++ класа је задужена за рад са датотекама, у режиму уписа и читања њиховог садржаја?
5. Која се C++ функција користи за отварање улазно / излазног тока за рад са сваком датотеком?
6. Да ли се C++ функција за отварање улазно / излазног тока датотеке, позива над одговарајућим објектом стандардне класе улаза / излаза?
7. Како се у C++ програму може проверити статус улазно / излазног тока датотеке (да ли је одређена датотека отворена у траженом режиму)?
8. Са којом врстом датотеке C++ програм подразумевано ради?
9. Који се C++ оператори могу користити за упис / читање код текстуалних датотека?
10. Које се C++ функције могу користити за упис / читање код текстуалних датотека?
11. Које се C++ функције могу користити за упис / читање код бинарних датотека?
12. Које се C++ функције могу користити за читање позиције у датотекама?
13. Које се C++ функције могу користити за позиционирање унутар садржаја датотека, текстуалних и бинарних?
14. Како у C++ програму може бити детектован крај текстуалне датотеке?
15. Како у C++ програму може бити детектован крај бинарне датотеке?

10. Пројектовање класа и њихова имплементација на језику C++

Кратак преглед лекције

Као што се у *C* и *C++* језику због ефикасности програма, променљиве међусобно истог типа групишу у низове, променљиве међусобно истог или различитог типа у структуре, а структуре међусобно истог типа у низове или листе, у *C++* језику се подаци и функције са међусобном логичком везом могу груписати у класу. За приступ подацима и функцијама класе, у програму се креирају примерци - објекти класе. Вредности података тако постају атрибути који описују стања објеката а функције постају методе за измене стања објеката током извршавања програма.

10.1. Појам и основне особине класе и објеката класе

Појам класе

Језик *C++* наследио је од језика *C* тип структуре, као и креирање примерака - објеката дефинисаних типова структура. Тип класе у језику *C++* изграђен је на основу типа структуре, са додатком функција и уведеним додатним особинама типа класе у односу на тип структуре. Пројектовање и синтакса класа и објеката заступљени су у свим језицима објектно оријентисаног програмирања. Ако се пројектовање и писање класа научи на једном језику, релативно је једноставно применити то и на остале језике објектно оријентисаног програмирања.

Објектно оријентисани концепти подразумевају пројектовање класа у програму и у свакој од класа: обједињавање логички повезаних података – чланова класе и функција – чланица класе; могућност скривања тако обједињених података и функција; могућност извођења класа из претходно дефинисане класе – које значи могућност наслеђивања података и функција; као и могућност дефинисања начина приступа одређеним групама података и функција претходно дефинисане класе, из других класа и појединих делова програма;

Појам објекта класе

У *C++* програму, на једној страни дефинише се класа као тип са члановима подацима и чланицама функцијама и дефиниција класе користи се као шаблон објекте. На другој страни, објекти класе уводе се у програму као конкретни примерци класе, јер се креирају по њеном опису. Након креирања објекта класе, вредности података чланова постају атрибути објекта (и описују стање објекта) а функције постају методе објекта (користе се за измене стања објекта).

Основне особине класе

Апстракција – значи могућност креирања класе, изузетно без конкретних објеката. Оваква класа резервисана је за обједињавање апстрактних метода значајних програму, али без дефиниције ових метода, дефиниција метода оставља се изведеним класама. Сврха дефинисања апстрактне класе је да се из ње изводе друге класе и да изведене класе имају своје дефиниције за апстрактне методе и своје конкретне објекте за приступ овим методама.

Енкапсулација - у објектно оријентисаном програмирању, груписање података и функција у класи омогућава им разне начине приступа, скривање, заштиту или јавност. За ово се користе три службене речи: *private*, *protected* и *public*. Ако је испред декларације написана реч *private* чланови су ограничени на саму класу. Реч *protected* значи ограничавање чланова на класу и класе које су из ње изведене. Ако су чланови означени са *public* јавни су и доступни ван класе.

Наслеђивање - у објектно оријентисаном програмирању кључни је особина класа. Класом "родитељем" сматра се основном класом која треба да буде искоришћена као модел. Класа "потомак" која се изводи из основне и садржи све њене особине али и неке само своје, које се накнадно дефинишу. Наслеђивање омогућава коришћење дефиниције класе на више начина без великих измена изворног кода и тако доста олакшава одржавање програма.

Полиморфизам - се заснива на особини наслеђивања и значи да свака изведена класа може на свој начин користити дефиницију основне класе родитеља. Изведена класа подразумевано има функције своје основне класе у оригиналној верзији, али и могућност да има преклопљене верзије ових функција. Ова особина класе базирана је на значајном концепту преклапања функција у програмима на језику *C++*.

10.2. Дефиниција класе и креирање објеката класе

Правила дефинисања приступа члановима класе

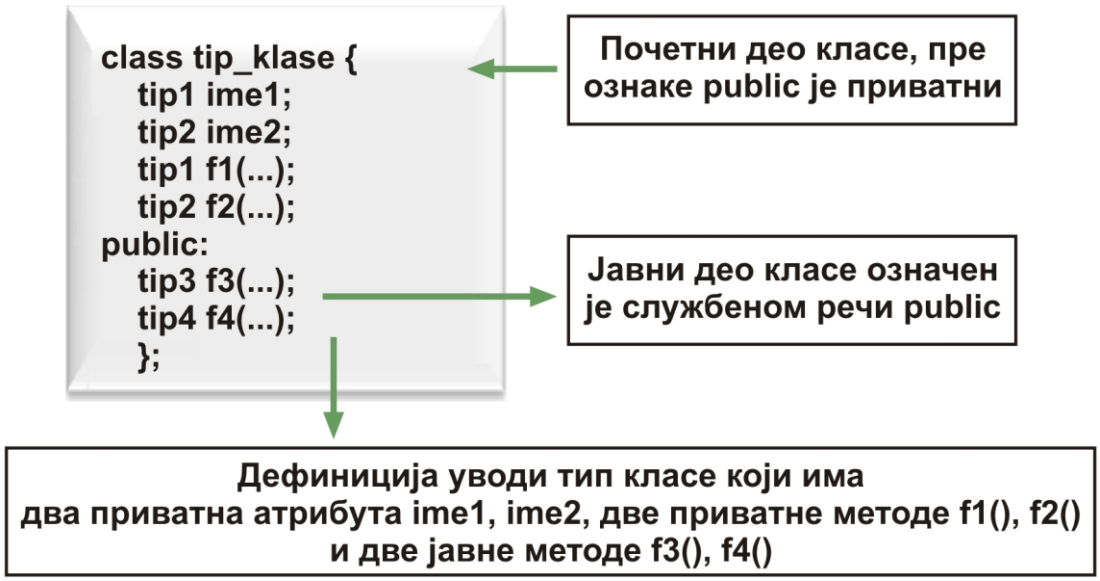
У зависности од тога каква је видљивост потребна за податке и функције класе, касније атрибуте и методе објеката, у дефиницији класе, користе се службене речи: private – за приступ приватним атрибутима и методама искључиво из саме класе (подразумевани тип); protected – за приступ заштићеним атрибутима и методама из саме класе и њених наследника; public – за потпуно јавни приступ атрибутима и методама класе, без ограничења.

Дакле, правила приступа су следећа: Приватне атрибуте једне класе – могу мењати методе те класе; Приватне методе једне класе – могу позивати друге методе те исте класе; Јавне атрибуте једне класе – могу мењати и методе ван те класе; Јавне методе једне класе – могу позивати и методе ван те класе. Заштићеним атрибутима и методама једне класе могућ је приступ из унутрашњости класе и класа које је наслеђују.

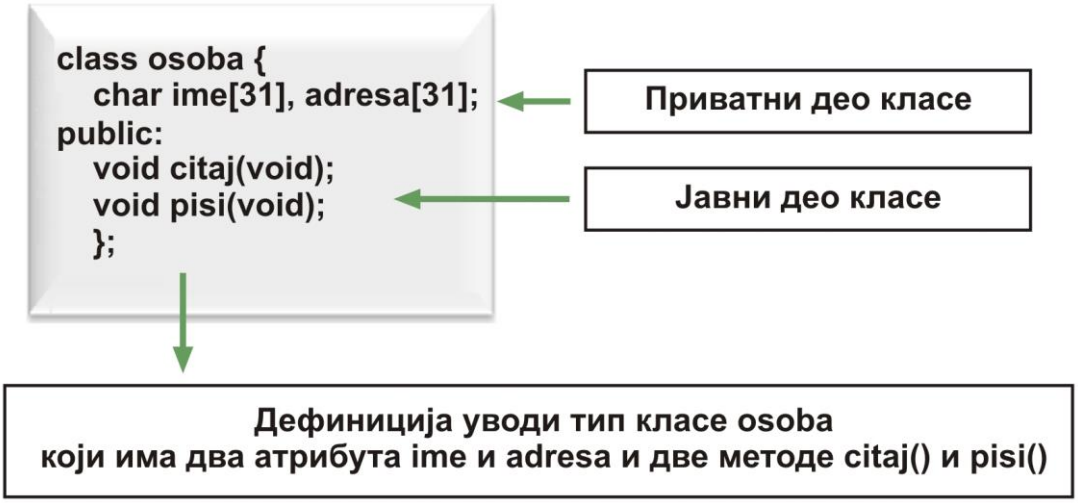
При пројектовању класа, уобичајено је знати и следеће: За атрибуте објеката – углавном се у дефиницији класе предвиђају приватни подаци чланови; За методе објеката – у дефиницији класе припремају се приватне и јавне функције чланице. Концепт јавних метода омогућава да се преко њих дође до приватних атрибута и метода исте класе: из других делова програма ван класе, као што је већ претходно поменуто, могу бити позване искључиво јавне методе, а јавне методе могу приступити приватним атрибутима и методама своје класе.

Дефиниција типа класе

Дефиниција типа класе слична је дефиницији типа структуре. Након службене речи *class* на почетку, слети tag - ознака типа класе (идентификатор) и у продужетку између витичастих заграда пишу се декларације података чланова и функција чланица. За приватне податке и функције није потребно ништа написати ако су на почетку дефиниције (у супротном писало би *private*). За остале чланове неопходно је нагласити испред њихове декларације, на пример *public* ако треба да буду потпуно јавни, слике 10.1. и 10.2.



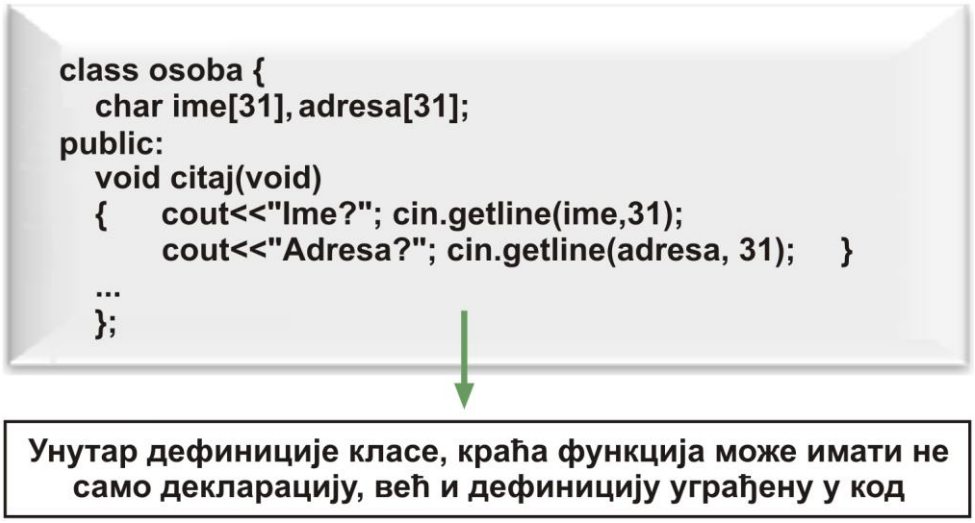
Слика 10.1 Дефиниција типа класе, општи облик



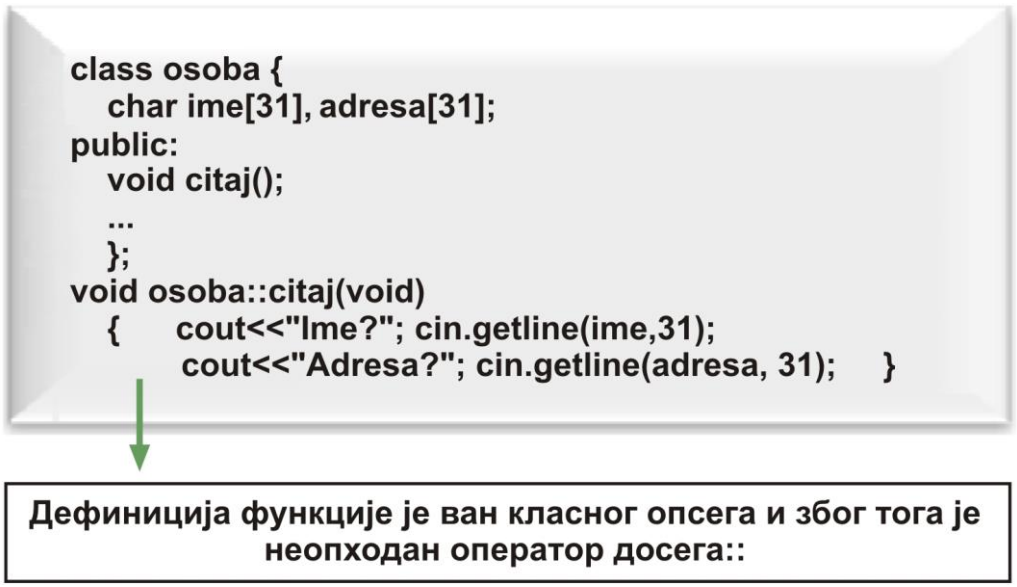
Слика 10.2 Дефиниција типа класе, пример

Дефиниција функција чланица класе

Дефиниција функције која је чланица може бити написана унутар дефиниције класе, слика 10.3, или ван ње, слика 10.4. У овом другом случају неопходно је испред имена функције написати припадање класи, име класе и оператор досега (::). Посебан случај су евентуално потребне пријатељске функције класе, које нису чланице класе али се унутар класе пише њихова дефиниција са службеном речи friend на почетку и то говори да оне могу евентуално приступити и приватним члановима класе којој су пријатељи.



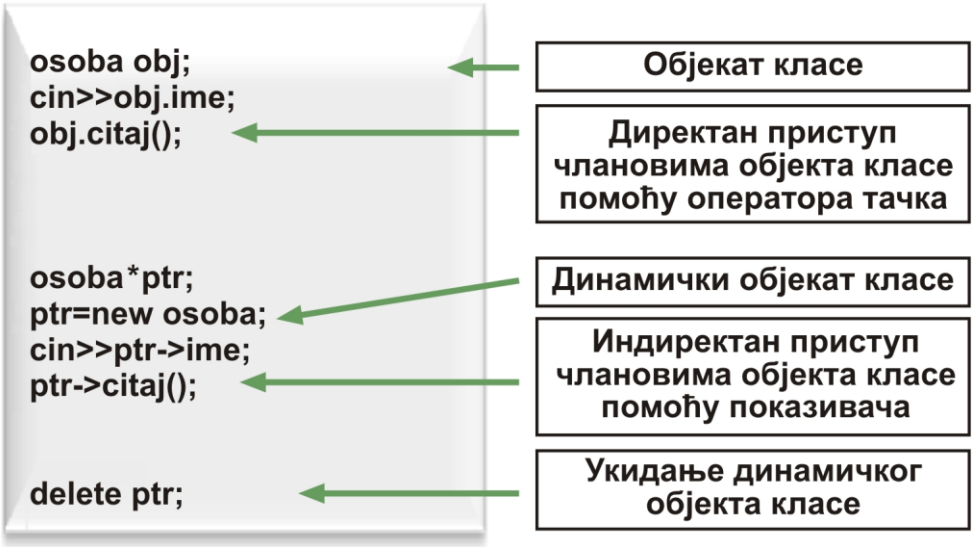
Слика 10.3 Дефиниција функције унутар дефиниције класе



Слика 10.4 Дефиниција функције ван дефиниције класе

Креирање објеката класе

Објекат класе може бити креиран статички (објекат или низ објеката), слика 10.5 или динамички помоћу показивача / низа показивача и оператора new (низ објеката или листа објеката), слика 10.6. У зависности од начина креирања објеката, приступ атрибутима и методама објаката може бити остварен директно (.) или индиректно (->). Креирањем сваког објекта једне класе, креира се комплет свих његових атрибута у меморији, док креирањем првог објекта класе, свака његова метода добија место у меморији и услед позива ових метода над било којим објектом исте класе, користи се исти примерци метода.



Слика 10.5 Креирање објекта класе, статичко

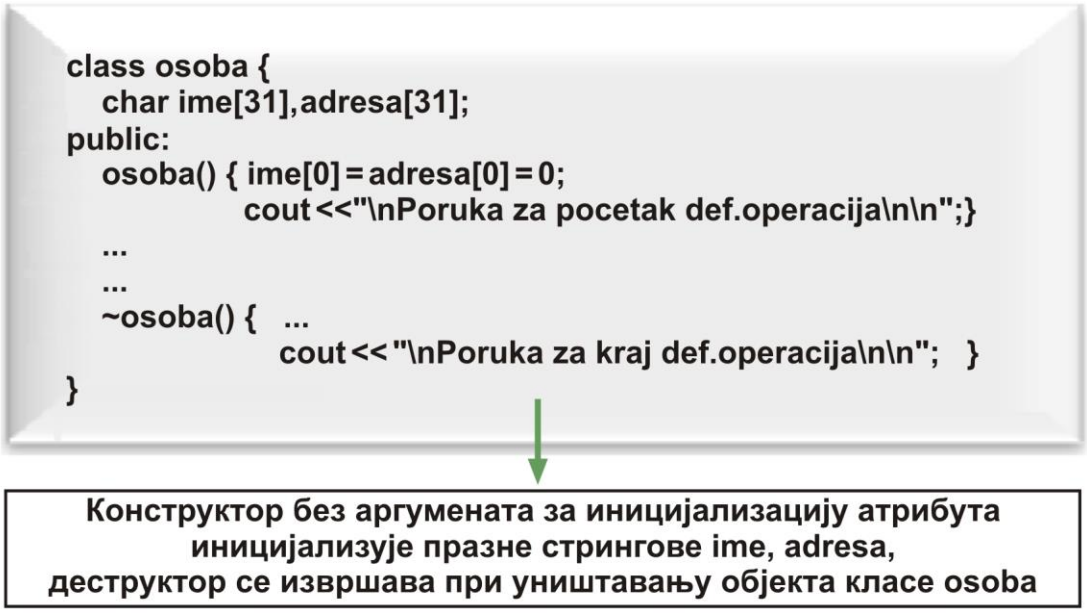
Слика 10.6 Креирање објекта класе, динамичко помоћу низа показивача

10.3. Конструктори и деструктори класа

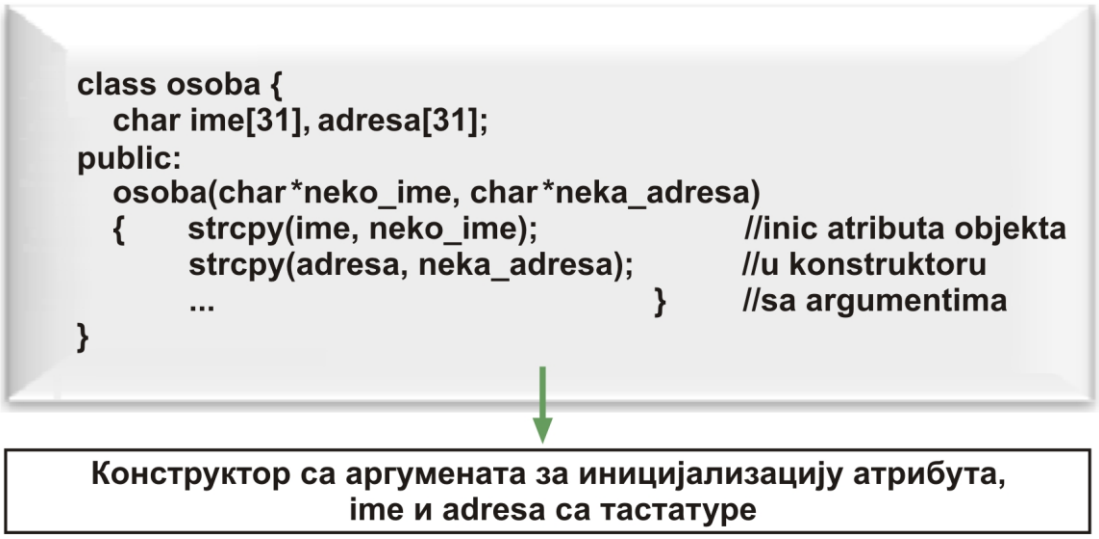
Конструктор класе је опциона функција класе, која има исто име као и класа. Ако се дефинише у класи, позива се аутоматски након креирања сваког објекта класе (статичког или динамичког) и користи се као механизам за иницијализацију: динамичку доделу меморије, иницијализацију атрибута објекта..., слика 10.7. (Изузев статичких атрибута означених са static и тако иницијализованих на 0, остали атрибути немају аутоматску иницијализацију.)

Деструктор класе је такође опциона функција класе, која има исто име као и конструктор класе, само са префиксом ~, јер је комплемент конструктору. Ако се дефинише у класи, позива се аутоматски при укидању сваког објекта класе и користи се као механизам за регуларан крај извршавања појединих операција у програму, на пример за ослобађање динамички додељене меморије. Деструктор класе не може имати аргументе ни повратну вредност, слика 10.7.

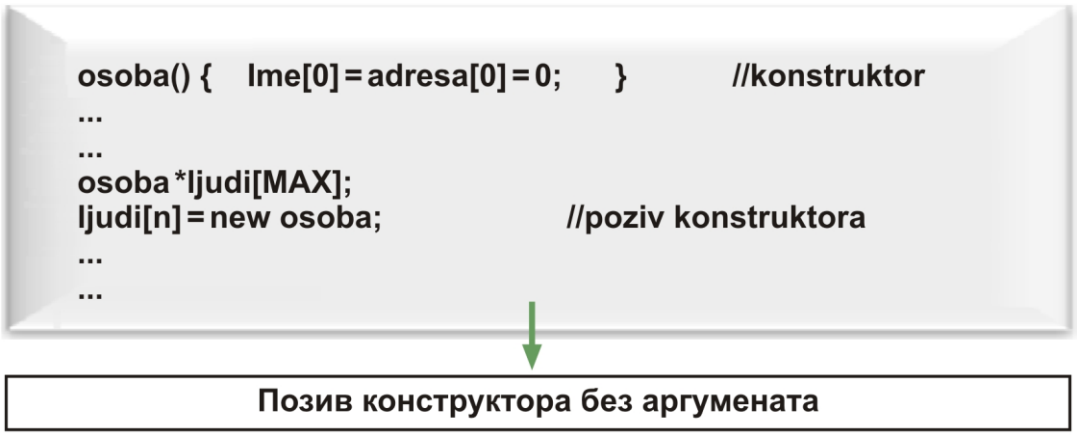
Поред тога што је конструктор класе опциона функција, она опционо користи аргументе. Ако се при иницијализацији користе унапред дефинисане вредности, конструктор класе нема потребе за аргументима, слика 10.7. У неким случајевима потребно је да, у оквиру иницијализације користи команде и вредности који се задају током извршавања програма и онда треба аргументе, слика 10.8. Од ове функција нема повратне вредности. У зависности од тога да ли има или не аргументе, конструктор се при креирању било ког објекта класе, аутоматски позива на одговарајући начин, слике 10.9 и 10.10.



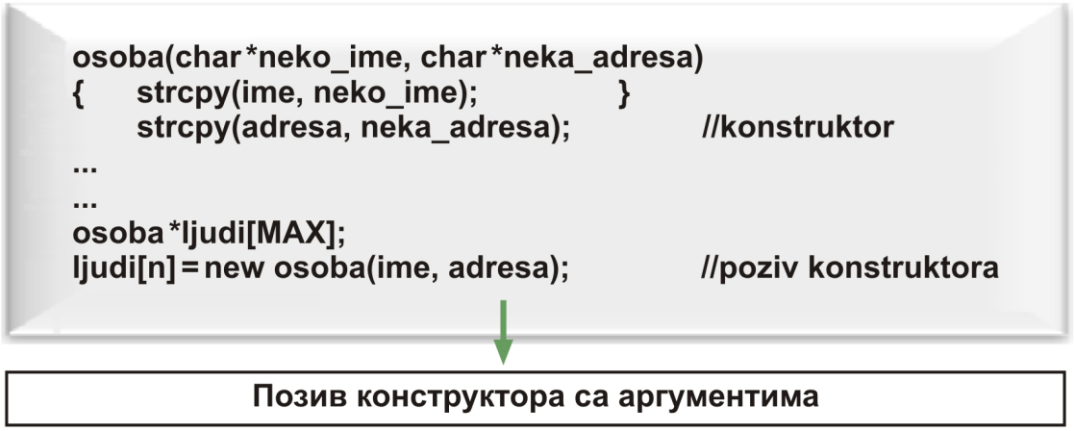
Слика 10.7 Дефиниција конструктора без аргумената и деструктора класе



Слика 10.8 Дефиниција конструктора класе са аргументима



Слика 10.9 Позив конструктора класе без аргумената



Слика 10.10 Позив конструктора класе са аргументима

10.4. Наслеђивање, интерфејси класа, полиморфне и апстрактне класе

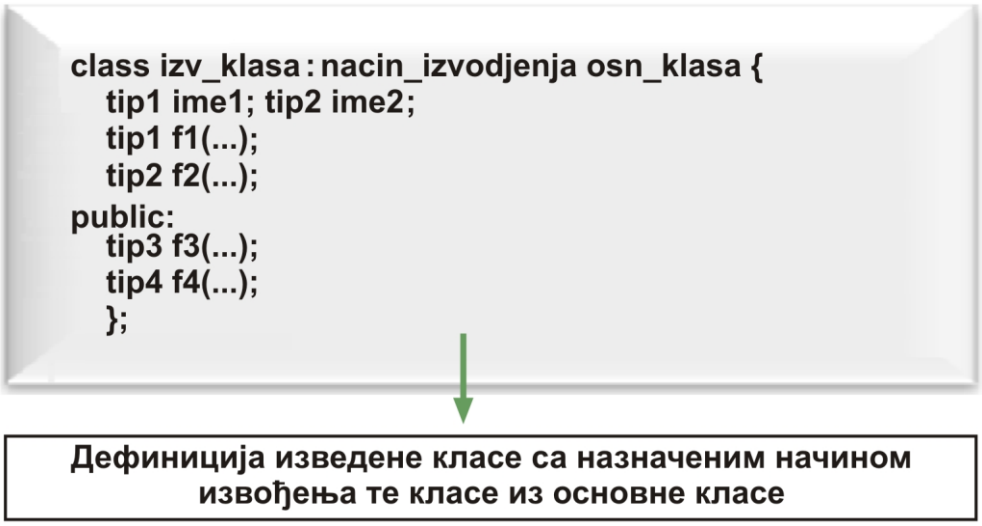
Наслеђивање код класа

Група објеката сваке класе има заједничке особине те класе. Неке групе објеката могу имати даља груписања у подгрупе објеката, што се решава у програмима наслеђивањем класа. Основна класа (родитељ) је полазна и описује општу групу објеката, док изведене класе (деца) описују подгрупу објеката са додатним специфичним особинама у односу на општу групу објеката. Основна класа за неку изведену класу и сама може бити изведена из других класа.

Постојање основне класе из које је изведена класа, означава се у дефиницији изведене класе двема тачкама (:), између типа изведене и типа основне класе, слика 10.11. Изведена класа наслеђује све чланове основне класе, изузев конструктора и деструктора, који се генеришу посебно. У дефиницији изведене класе, дефинише се и начин извођења класе из основне, који значи ниво приступа члановима основне класе, преко објеката изведене, слика 10.12. Углавном је предвиђен јавни начин извођења класе, као на слици 10.13.

Креирање објекта сваке изведене класе прате следећи кораци: извршавање конструктора основних класа, по редоследу навођења основних класа у дефиницији изведене класе; извршавање конструктора изведене класе.

Уништавање објекта сваке изведене класе прате кораци овим редом: извршавање деструктора изведене класе; извршавање деструктора основних класа, по обрнутом редоследу од редоследа позива конструктора ових класа.



Слика 10.11 Дефиниција изведене класе

Извођење	Члан основне класе		
	јавни	заштићени	приватни
јавно	јавни	заштићени	приватни
заштићено	заштићени	заштићени	приватни
приватно	приватни	приватни	приватни

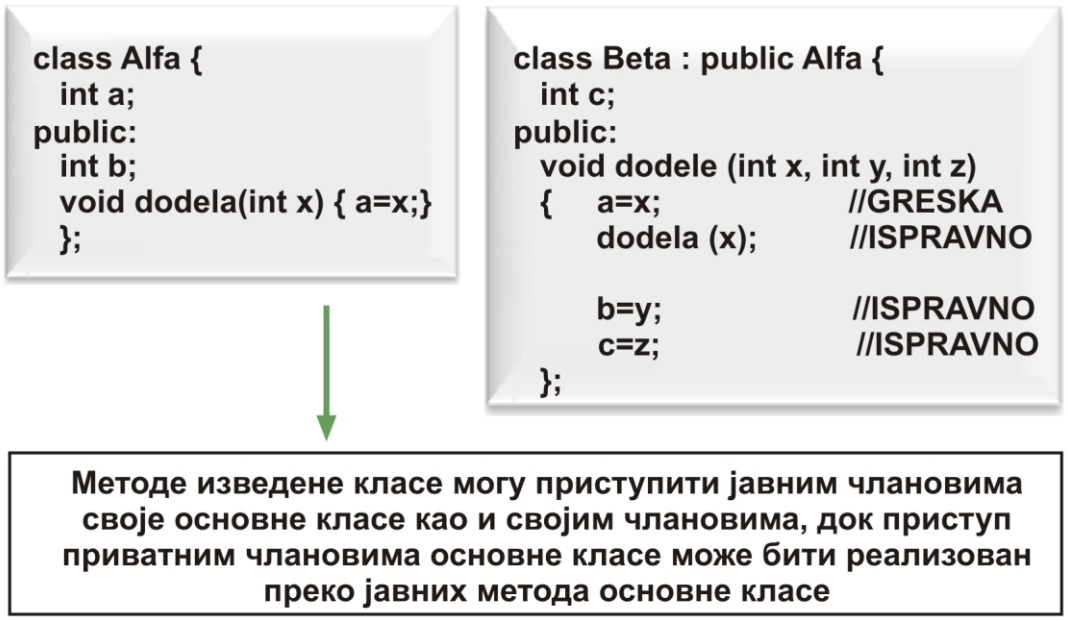
Слика 10.12 Могући начини извођења и њихово значење (ниво приступа члановима основне класе из изведене)

```
class ucenik : public osoba {
   char skola[31], razred[31];
public:
   ucenik() : osoba() { skola[0]=razred[0]=0; }
   void citaj(void);
   void pisi(void);
   };
```

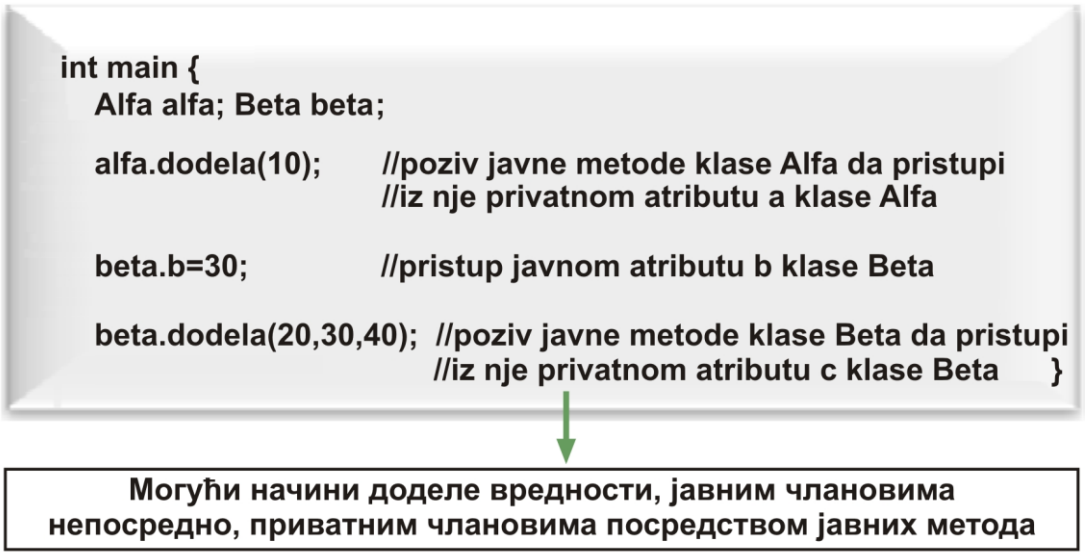
Слика 10.13 Јавни начин извођења класе

Интерфејси класа

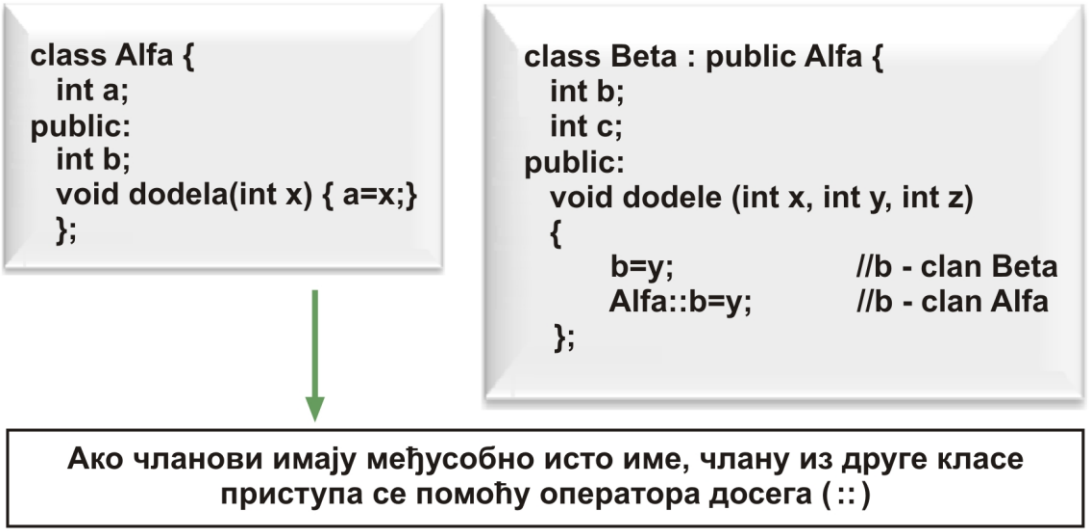
Интерфејс класе са остатком програма, па и са њеним изведеним класама, представљају њене јавне методе. Методе било које изведене класе, могу непосредно приступити јавним методама основне класе, а посредством јавних метода основне класе могућ је приступ свим приватним члановима исте основне класе, слике 10.14 и 10.15. Ако чланови разичитих класа имају међусобно исто име, приступ се решава помоћу оператора досега, слика 10.16.



Слика 10.14 Приступ члановима основне класе из изведене класе



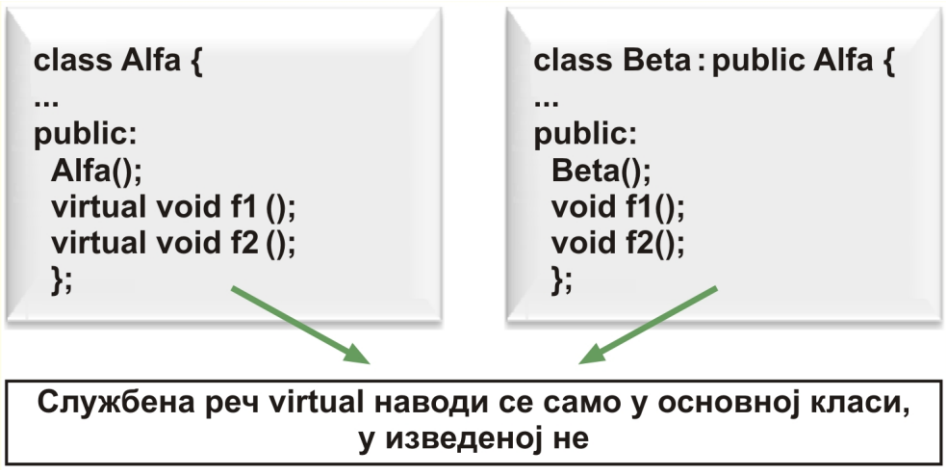
Слика 10.15 Интерфејси класе: приступ искључиво јавним члановима било које класе из главне функције, а приватним члановима посредством јавних метода



Слика 10.16 Приступ члановима основне и изведене класе, са међусобно истим именима

Виртуелне методе и полиморфне класе

Механизам виртуелних метода омогућава реализацију одређене методе над објектима међусобно различитих типова. Виртелна метода је основне класе може бити замењена истоименом методом изведене класе и може такође бити редефинисана у изведеној класи, слика 10.17. Декларације виртуелних метода у основној и изведеним класама - требају бити са међусобно истим бројем и типовима аргумената и истим типом повратне вредности. Класа која има бар једну виртуелну методу је полиморфна класа, слика 10.18.



Слика 10.17 Дефиниција виртуелних метода

```
class osoba{                                        //osnovna klasa
   char ime[31], adresa[31];
public:
   virtual void stampaj()                           //virtuelna metoda
   {cout<<ime<<adresa;}
};
class ucenik{                                       //izvedena klasa
   char skola[31], razred[31];
public:
   void stampaj()                                   //redefinicija
   {    osoba::stampaj(); cout<<skola<<razred;    }
};
```

Слика 10.18 Дефиниција полиморфне класе, са бар једном виртуелном методом

Апстрактне методе и апстрактне класе

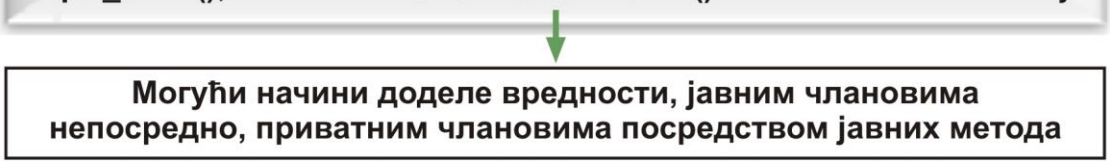
Апстрактна метода је виртуелна метода која није дефинисана у основној класи, слика 10.19. Класа која садржи бар једну апстрактну методу је апстрактна класа. Апстракне класе немају своје објекте, већ им је сврха да се из њих изводе друге класе и да изведене класе имају дефиниције за апстрактне методе основне класе, слика 10.20. Класа која је изведена из апстрактне класе, а не садржи дефиницију бар за једну апстрактну методу основне класе и сама је апстрактна.

```
class Alfa {
  public:
    virtual void f1()=0;              //Apstraktna metoda
    virtual void f2(){....}           //Obicna virtuelna metoda
    };
class Beta:public Alfa {
  public:
    void f1(){....}                   //Definicija apstraktne metode
    };
```

Основна класа на крају декларације апстрактне методе има =0, а изведена класа има дефиницију исте апстрактне методе

Слика 10.19 Дефиниција апстрактне методе

```
int main {
   Alfa a;                  //GRESKA, apstraktna klasa nema objekte
   Beta b;                  //Objekat klase Beta

   Alfa *ptr_a;             //NIJE GRESKA, ali samo ako se inicijalizuje
   ptr_a=&b;                //kao pokazivac na objekat izvedene klase
   Beta *ptr_b;             //Pokazivac na objekat klase Beta

   ptr_a->f1();             //Poziva se Beta::f1()
   ptr_a->f2();             //Poziva se Alfa::f2()                    }
```



Слика 10.20 Креирање објеката класе изведене из апстрактне

Пример динамичког креирања објеката и наслеђивања код класа

Као што је претходно поменуто, динамички може бити креирана и листа објеката класе: дефиницијом типа основне и изведених класа, слика 10.21; декларацијом низа показивача на тип класе; применом оператора new над типом класе и доделом резултата оператора new показивачима редом у низу, у једној петљи, слика 10.22. Ако је основна класа апстрактна, слике 10.21, показивачи се декларишу на њен тип, али се оператор new користи над објектима изведених класа, слика 10.22.

```
class predmet{                                  //osnovna klasa
   double spec_tezina;
public:
   virtual double v()=0;                        //apstraktna metoda
};
class lopta:public predmet{                     //izvedena klasa
   double r;
public:
   double v() { return 4.0/3*(r*r*r)*3.14;}     //definicija
};
class kocka:public predmet{                     //izvedena klasa
   double a;
public:
   double v() { return a*a*a;}                  //definicija
};
```

Слика 10.21 Пример дефиниције основне и изведених класа

```
int main()
{   predmet*predmet[MAX];                  //niz pokazivaca
    ...
   int n=0;
   while(1){...
   predmeti[n++]=new lopta;                //dinamicko kreiranje objekta
   predmeti[n++]=new kocka;                //dinamicko kreiranje objekta
   if(n==n_zadato) break; ...
   }
   for(i=0; i<n; i++)
         predmeti[i]->stampaj();           //poziv metoda nad objektima
   ...
   for(i=0; i<n; i++)
         delete predmeti[i];               //unistavanje objekata
};
```

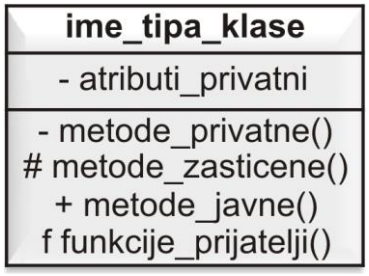
Слика 10.22 Пример динамичког креирања објеката и приступа јавним методама из главне функције програма

Модули и датотеке заглавља

Као и код структурно пројектованих програма и код објектно оријентисано пројектованих, сав код може бити у једном главном модулу. Код обимнијих програма: дефиниције класа смештају се у датотеке заглавља; методе класа уобичајено су у секундарним модулима, док је за главну функцију предвиђен главни модул програма. И овде, програм може имати један главни модул и произвољан број секундарних модула и датотека заглавља.

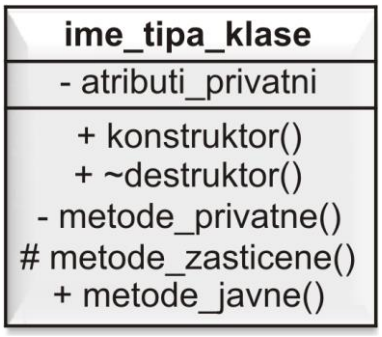
10.5. Класни дијаграми

Класни дијаграм је граф за приказ класа и односа међу њима. Сваки чвор графа представља класу помоћу правоуганих поља: прво поље резервисано је за име класе, друго за атрибуте, а треће за методе класе. Уобичајени изглед класног дијаграма, за опис једне класе, слика 10.?? има следеће ознаке: атрибути су углавном приватни (- ознака испред имена), док методе могу бити: приватне (- ознака испред имена), заштићене (# ознака испред имена), јавне (+ ознака испред имена) или пријатељи класе (f ознака испред имена), слика 10.23.



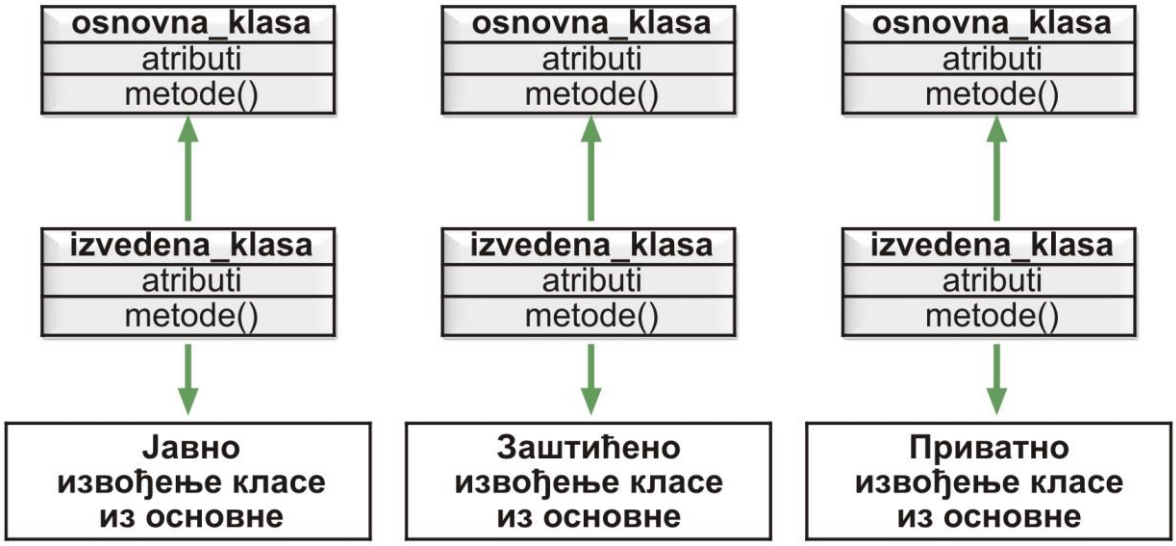
Слика 10.23 Општи изглед класног дијаграма

Ако су у класи дефинисане функције конструктор и деструктор, оне су јавне и потребно је представити и њих на графу, слика 10.24. Могуће је да атрибути изостану из класе, у том случају граф изгледа као на слици 10.25. Везе између класа представљају се линијама, а начин извођења класа из основне класе родитеља представљају се појединим врстама стрелица: празна стрелица користи се за јавно извођење, сива стрелица за заштићено, а попуњена стрелица за приватно извођење класе из основне, као што је приказано на слици 10.26.



Слика 10.24 Представљање конструктора и деструктора

Слика 10.25 Представљање класе без атрибута



Слика 10.26 Представљање начина извођења класа из основне класе

10.6. Питања

1. Шта представља класа у објектно оријентисаном програмирању?
2. Шта представља објекат класе у објектно оријентисаном програмирању?
3. Да ли се објекти могу креирати само као статички или и као динамички?
4. Да ли је класа шаблон за израду објекта, или је објекат шаблон за класу?
5. Шта значи приватност чланова класе?
6. Шта значи заштићеност чланова класе?
7. Шта значи потпуна јавност чланова класе?
8. Да ли је дефиниција функција чланица могућа само унутар дефиниције типа класе или и ван ње?
9. Шта представљају атрибути а шта методе за објекат класе?
10. Шта су изведене класе у односу на основну класу?
11. Шта представља интерфејс једне класе?
12. Шта су полиморфне класе?
13. Шта су апстрактне класе а шта апстрактни објекти?
14. Који су основни елементи класних дијаграма?
15. Које се све ознаке користе на класним дијаграмима, за приватне, заштићене и јавне чланове класе?

Индекс појмова

Литература

1. Л. Краус, Програмски језик C са решеним задацима, Академска мисао, Београд, 2014.

2. Л. Краус, Програмски језик C++ са решеним задацима, Академска мисао, Београд, 2019.

3. A. Hansen, Програмирање на језику C – потпуни водич за програмски језик C, Микро књига, Београд, 1991.

4. B. Kernighan, D. Ritchie, The C Programming Language - ANSI C, Prentice Hall Software Series, 1988.

5. C. Pappas, W. Murray, C/C++ Водич за програмере, Микро књига, Београд 1996.

6. С. Ђенић, Основи програмирања 2, електронски уџбеник, ВИШЕР, Београд, 2014.

7. С. Ђенић, Ј. Митић, С. Штрбац, Програмирање на језику C и основи програмирања на језику C++, збирка примера и задатака, ВИШЕР, Београд, 2019.