Слободанка Ђенић

Основи програмирања

Висока школа електротехнике и рачунарства
струковних студија
Београд, 2018.

Аутор: Слободанка Ђенић

Рецензенти: Јелена Митић, Светлана Штрбац Савић

Корице: Јелена Лабус Грујић (MASSVision)

Издавач: Висока школа електротехнике и рачунарства струковних студија у Београду

Штампа: Развојно-истраживачки центар графичког инжењерства ТМФ, Београд

Тираж: 30

Издање: 1.

ISBN 978-86-7982-277-2

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.42 (075.8)
004.432.2C (075.8)

ЂЕНИЋ, Слободанка, 1965 -
Основи програмирања / Слободанка Ђенић. - 1. изд. - Београд : Висока школа електротехнике и рачунарства струковних студија, 2018 (Београд : Развојно-истраживачки центар графичког инжењерства ТМФ). - 125 стр. : илустр. ; 30 cm

Тираж 30. – Регистар. – Библиографија: стр. 125.

ISBN 978-86-7982-277-2

a) Програмирање b) Програмски језик "C"
COBISS.SR-ID 262491148

Предговор

Циљ наставе из предмета Основи програмирања у Високој школи електротехнике и рачунарства струковних студија у Београду, за који је написан овај уџбеник, јесте да на једноставан начин уведе у програмирање апсолутне почетнике у овој области али и да пружи основе инжењерског приступа програмирању. База знања рачунарског инжењерства обавезно садржи основне концепте програмирања, као и увод у програмирање на једном програмском језику. У овом случају то је програмски језик *C* који је послужио за развој великог броја осталих тренутно заступљених програмских језика.

Лекције уџбеника обухватају теоријски део наставе из предмета Основи програмирања, кроз следеће тематске целине:

- фазе развоја програма и алгоритми основних програмских структура,
- основне компоненте програма, основни типови података и оператори,
- наредбе једноструке и вишеструке селекције, циклуса и скокова,
- нумерички и знаковни низови, сортирање и претраживање низова,
- показивачи, директан и индиректан приступ низовима и функције.

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, 2018. године Аутор

Садржај

Уводна лекција

Кратак преглед лекције

Пројектовање програма веома је значајна фаза у развоју програма, која претходи фазама писања и превођења програма и даје идејно решење. У овој фази раде се алгоритми програма који описују све његове структуре и могуће токове. У фази извршавања и тестирања испитује се да ли програм ради на предвиђени начин, да ли је добро испројектован. Појава логичких грешака захтева измене у пројекту програма.

Фазе развоја програма

Развој програма не може бити успешно остварен без пројектовања програма, које претходи писању наредби, и без тестирања преведеног програма. Један програм пролази кроз следеће своје фазе: дефинисање и анализа проблема, пројектовање, писање изворног кода, превођење, повезивање преведених делова, тестирање, документовање и одржавање, слика 1. Пројекат програма сматра се најзначајнијом фазом, јер што је боље урађен мање је проблема при писању и тестирању програма.

Слика 1. Дијаграм превођења и повезивања програма

Дефинисање програмског задатка

На почетку развоја програма неопходно је дефинисати програмски задатак, прикупити све потребне информације за решавање задатка и донети одлуку о методи пројектовања и о програмском језику. Избор програмског језика зависи од тога: која је намена програма, који се језици користе код сличних програма, за које радно окружење се развија, која су искуства програмера... Сваки програмски језик одговара одређеној методи пројектовања, тако да је избором језика и метода изабрана.

Пројектовање програма

Пре него што се приступи писању програма битно је испројектовати га. Пројекат програма обухвата описе свих делова програма и њихове међусобне повезаности, као и опис свих могућих токова извршавања програма. Актуелне су као основне методе пројектовања: структурна и објектно-оријентисана. Избор методе пројектовања повезан је са избором програмског језика: језик *C* користи се за писање структурно пројектованих програма, а на језицима *C++, Java* и *C#* пишу се програми који се објектно-оријентисано пројектују.

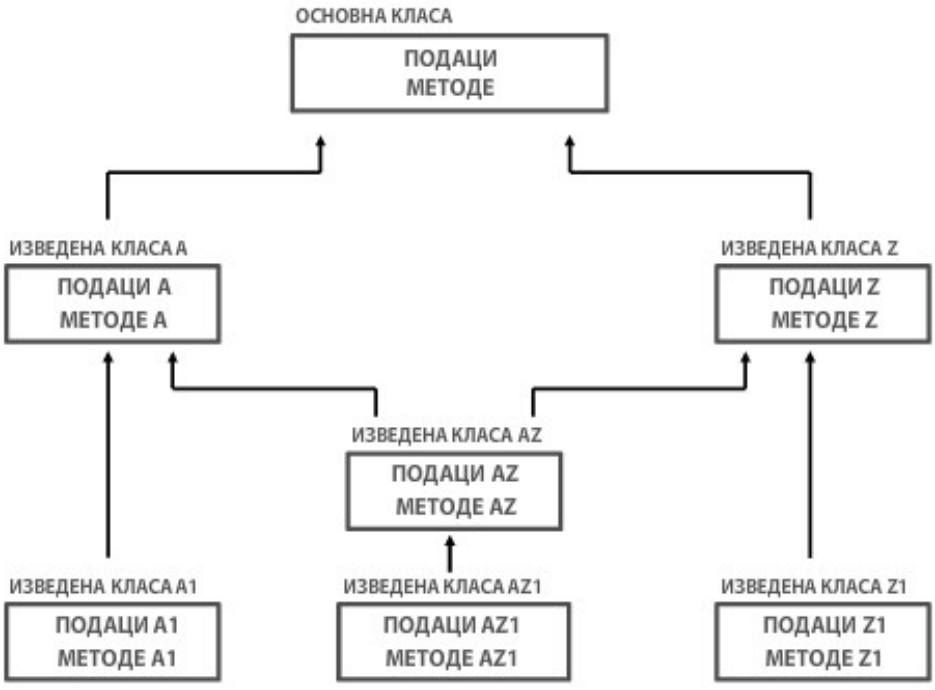
Структурна метода пројектовања програма

Структурно пројектовање значи рашчлањивање главног задатка на подзадатке и решавање сваког од њих помоћу потпрограма, слика 2. Потпрограм представља сегмент програма који садржи потребне операције над подацима, самостално решава свој подзадатак, али је и повезан са осталим потпрограмима. Почетак програма је почетак главног програма, из кога се (непосредно или посредно) позивају сви потпрограми (приступ "одозго надоле"). Решења свих потпрограма дају глобално решење главног програма.

Слика 2. Структурни модел пројектовања програма

Објектно оријентисана метода пројектовања програма

Када се објектно оријентисано пројектује програм се гради "нагоре" од своје основе коју чине подаци потребни програму, груписани у логичке целине. Основни појам овде је класа. Класу чини скуп података и операција (метода) применљивих над скупом података (подаци и операције су заједно), слика 3. Да би операције над подацима једне класе биле могуће класа (апстрактни тип) добија своје објекте (конкретне примерке). Подацима и методама приступа се преко објеката њихових класа.



Слика 3. Објектно оријентисани модел пројектовања програма

Писање изворног кода програма

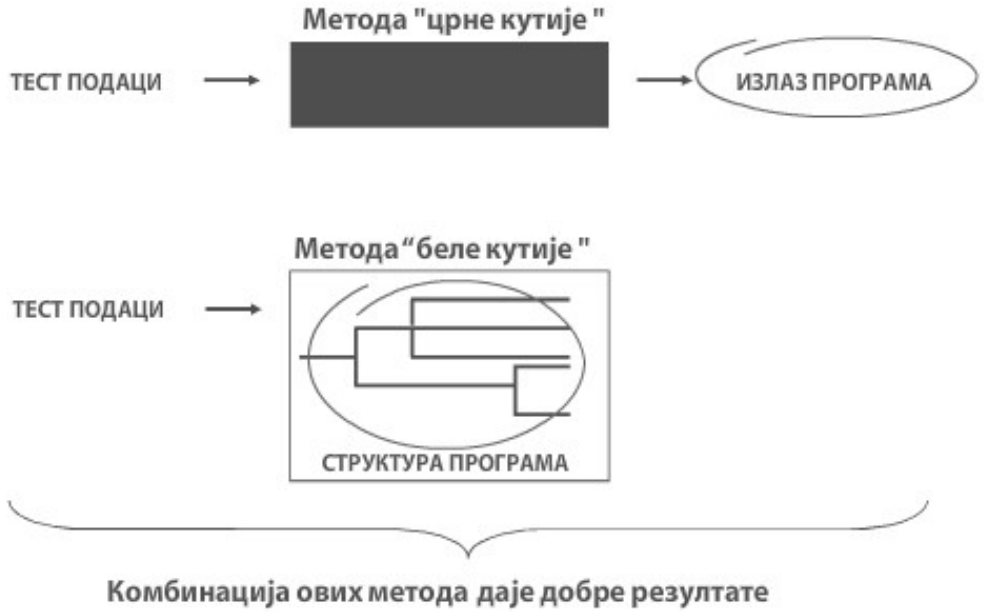
Изворни кôд програма чине наредбе написане по правилима синтаксе програмског језика. Писање изворног кода програма подразумева следеће: унос наредби на одређеном програмском језику помоћу неког едитора текста (сваки савремени компајлер садржи свој едитор текста), и памћење написаних наредби у једној или у више датотека (ако је програм обиман). Свака датотека са изворним кодом програма и екстензијом резервисаном за одређени програмски језик представља један модул програма.

Превођење, повезивање и тестирање рада програма

Свако савремено развојно окружење садржи и пратеће програмске алате (програме који омогућавају креирање извршне верзије програма). Програм за превођење програма на машински језик, компајлер *(compiler)* од изворног кода генерише објектни (машински) кôд. Програм за повезивање, линкер *(linker)* повезује објектне кодове свих делова програма и генерише извршни кôд програма. У оквиру развојног окружења, програм који помаже програмеру при откривању логичких грешака у раду програма је дибагер *(debugger)*.

Методе тестирања програма

Без обзира на избор методе, тестирање увек захтева план теста (некада и писање комплексних програма за тестирање), као и тест податке (податке приближно стварних вредности који покривају све изузетке у програму). Могуће су две методе за тестирање рада програма: метода црне кутије, која тестира само функцију програма (излаза на основу тест података на улазу), и метода беле кутије за тестирање само структуре програма (могућих токова програма), слика 4. У пракси се обавезно користи комбинација поменутих метода.



Слика 4. Методе тестирања рада програма

Документовање и одржавање програма

Израда документације програма почиње у фази пројектовања програма (опис пројекта), заступљена је у току писања изворног кода (текстуални коментари између наредби програма) и у току тестирања извршног кода (опис тестова) а комплетира се након добијене коначне верзије програма. Одржавање програма обухвата додавање и измене изворног кода, услед додатних и измењених захтева корисника или услова рада. Добар пројекат поједностављује тестирање и одржавање програма.

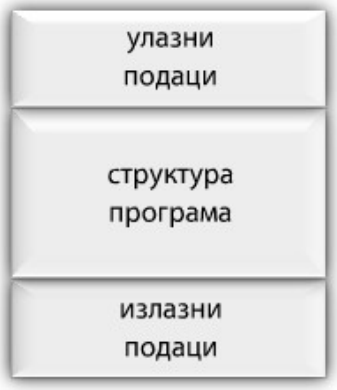
Алгоритми основних програмских структура

Програм се по покретању извршава кроз своје наредбе а наредбе се извршавају на начин на који је њихов изворни кôд написан: у простој линијској структури (једна за другом, редом) или под контролом посебних наредби за контролу тока, у структурама селекције, циклуса или скока. Честе су и комбинације поменутих структура (уклапање једне структуре унутар друге, друге унутар треће...). Знати један програмски језик не значи само познавати његову основну синтаксу, већ и могуће програмске структуре на том језику.

Алгоритми

Алгоритам представља прецизан опис свих структура у једном програму, односно свих корака при извршавању наредби једног програма. Један од начина представљања алгоритама је њихов графички приказ и то помоћу стандардног или структурног дијаграма тока. Стандардни дијаграм тока има почетну и крајњу тачку, улаз и излаз података и између блокове који описују поједине структуре у програму, слика 5. Структурни дијаграм нема почетну и крајњу тачку већ је на нивоу блокова, слика 6.

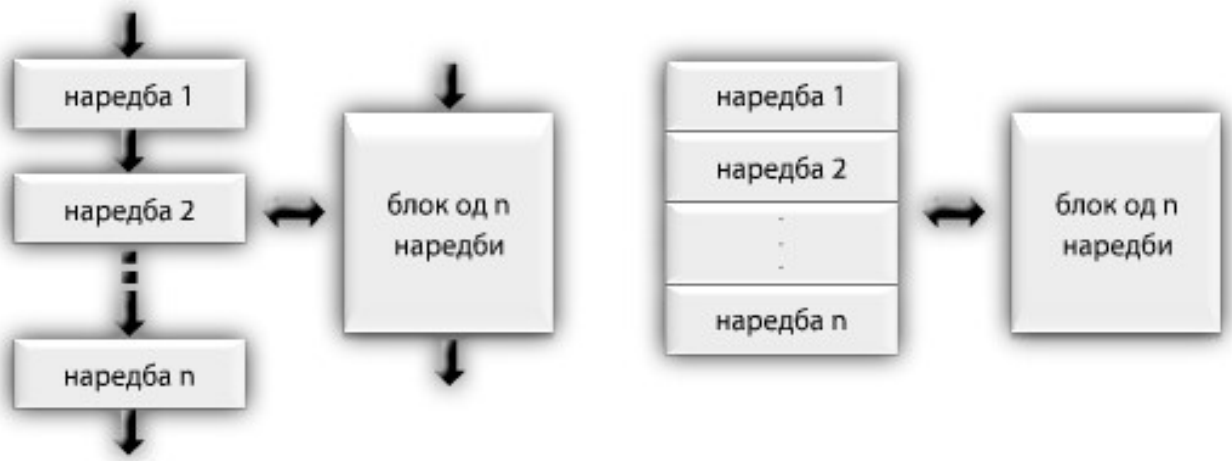
Слика 5. Стандардни дијаграм тока



Слика 6. Структурни дијаграм тока

Проста линијска структура

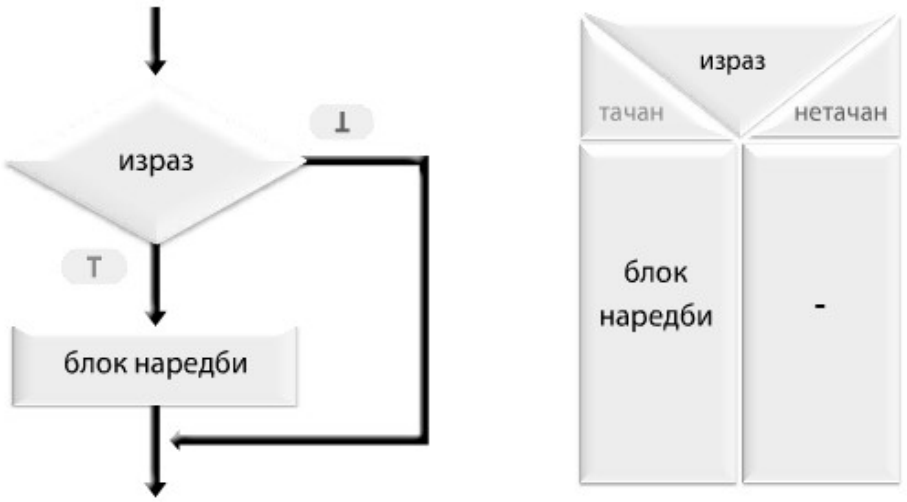
Проста линијска структура у програму значи секвенцијално извршавање наредби (једне за другом), слика 7. Секвенца од две или више наредби представља у програму блок наредби. Блок наредби третира се у програму као целина (могућа је замена једне наредбе блоком или блока једном наредбом). Сваки програмски језик има у својој синтакси ознаке за почетак и за крај блока. Обично програм поред простих линијских структура у појединим својим деловима садржи и остале контролне структуре.



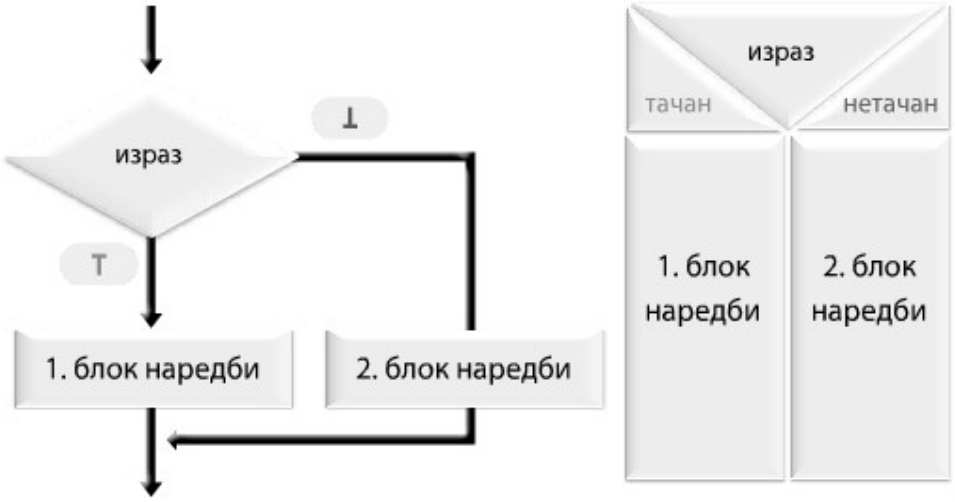
Слика 7. Проста линијска структура

Селекција (одлука у програму)

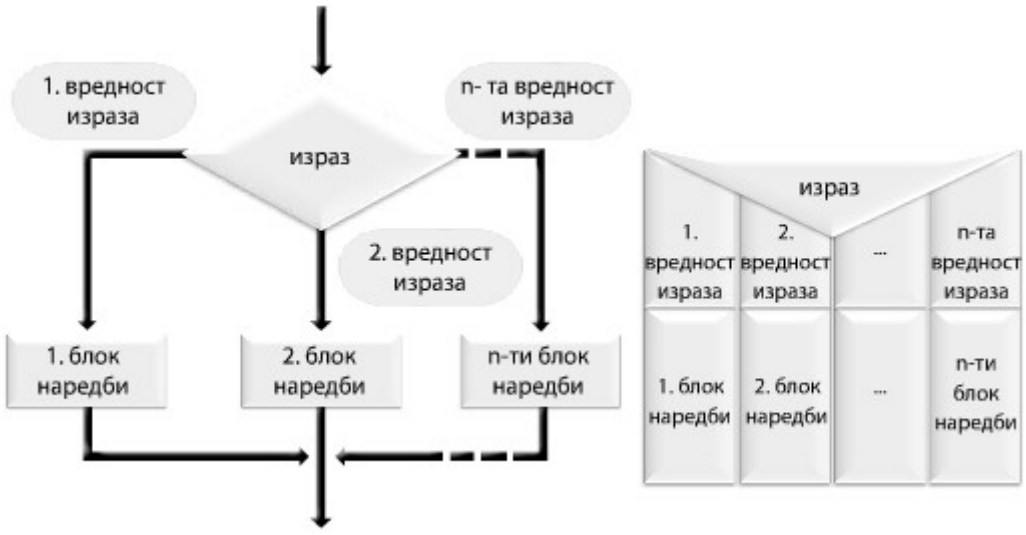
Селекција у програму испитује испуњеност одређеног услова , односно тачност израза који услов садржи. Ако је израз услова тачан бира се један ток извршавања програма, ако није онда програм иде другим током, где се може испитати тачност неког другог израза и одлучити даљи ток програма... Код селекције у програму могу се издвојити следећи случајеви: избор једног или ниједног блока наредби, слика 8, избор једног од два предвиђена блока наредби, слика 9 и избор једног од више блокова наредби, слика 10.



Слика 8. Селекција једног или ниједног блока наредби



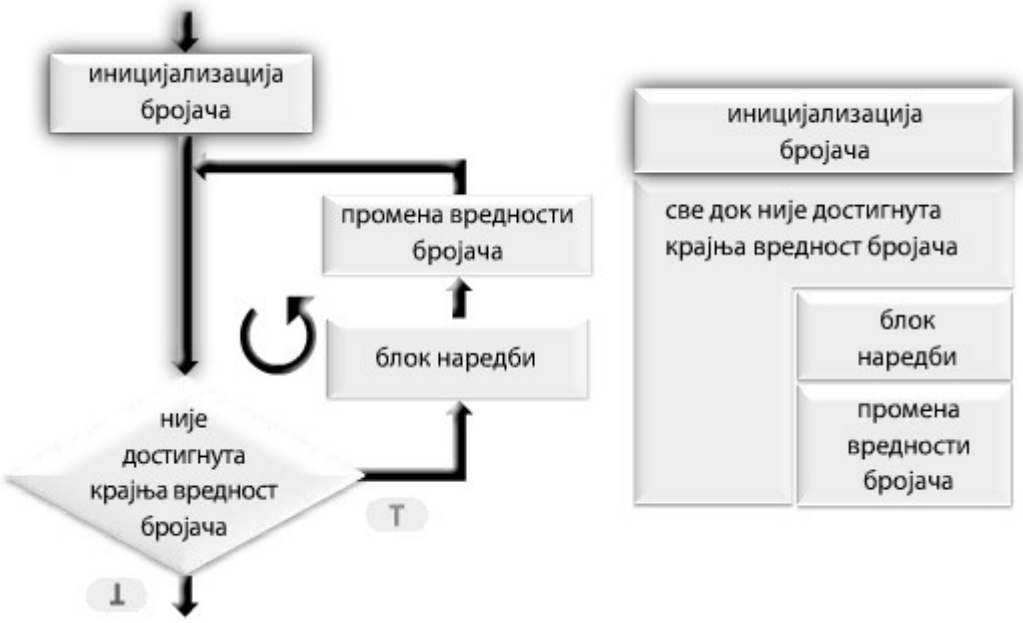
Слика 9. Селекција једног од два блока наредби



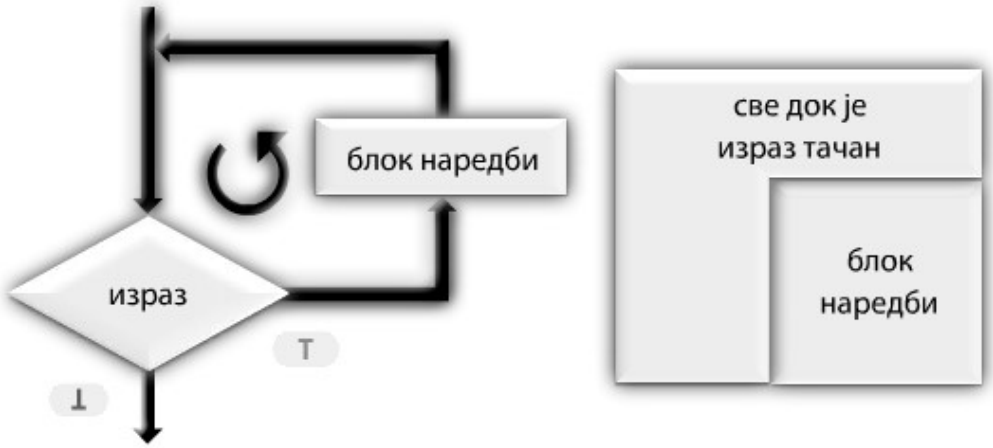
Слика 10. Селекција једног од више блокова наредби

Циклус (петља у програму)

Циклус у програму је коначан број понављања блока наредби. Постоје три основне врсте циклуса. Код бројачког циклуса задаје се почетна вредност бројача, извршава се блок наредби док се не достигне крајња вредност и бројач се мења са задатим кораком, слика 11. Код циклуса са излазом на врху поставља се услов за извршавање блока наредби (у првој и свакој даље итерацији), слика 12, а код циклуса са излазом у дну блок наредби изврши се једном, а услов постоји за свако понављање, слика 13.



Слика 11. Бројачки циклус



Слика 12. Циклус са излазом на врху

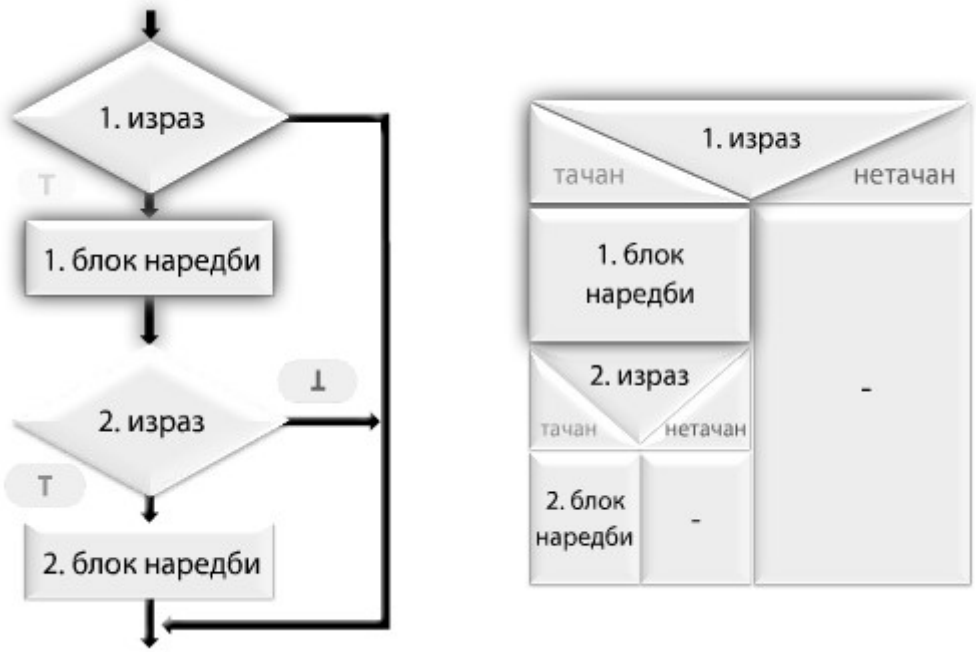
Слика 13. Циклус са излазом у дну

Скок у програму

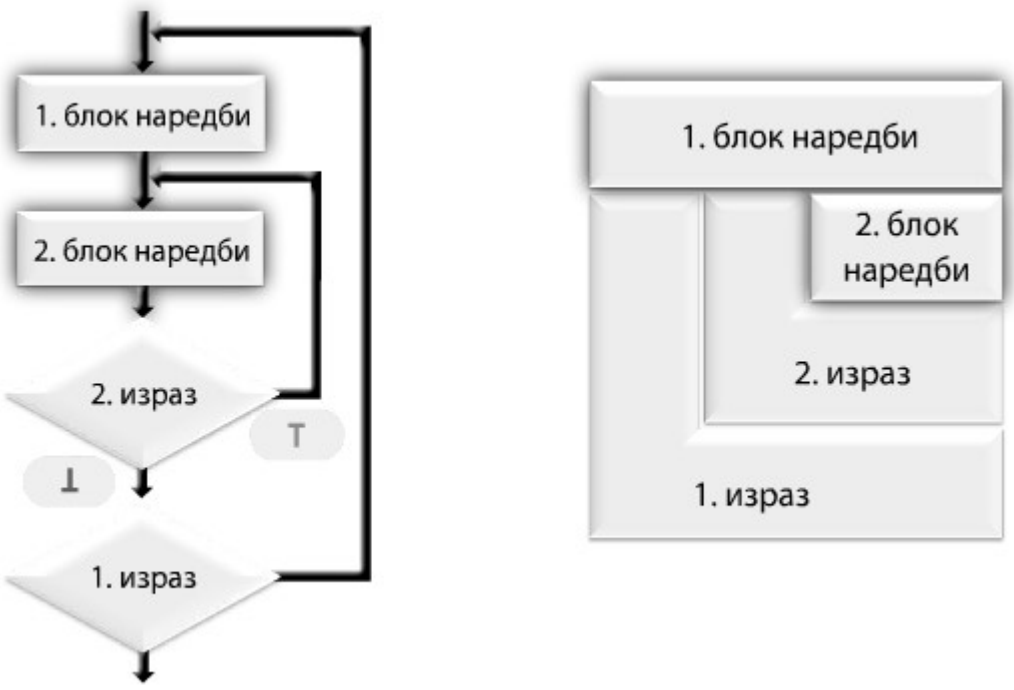
Скок у програму постоји да би извршавање програма било прекинуто у једној и настављено од неке друге наредбе истог програма. Наредба од које се наставља програм може бити пре или после наредбе скока. Ток програма са скоковима некада може бити нејасан и због тога се скокови избегавају у свим случајевима (замењују се другим могућим врстама структура) изузев за излазе из циклуса. Дијаграми тока немају стандардне графичке симболе за скок, тако да се описују у овим дијаграмима текстуално.

Уклопљене структуре

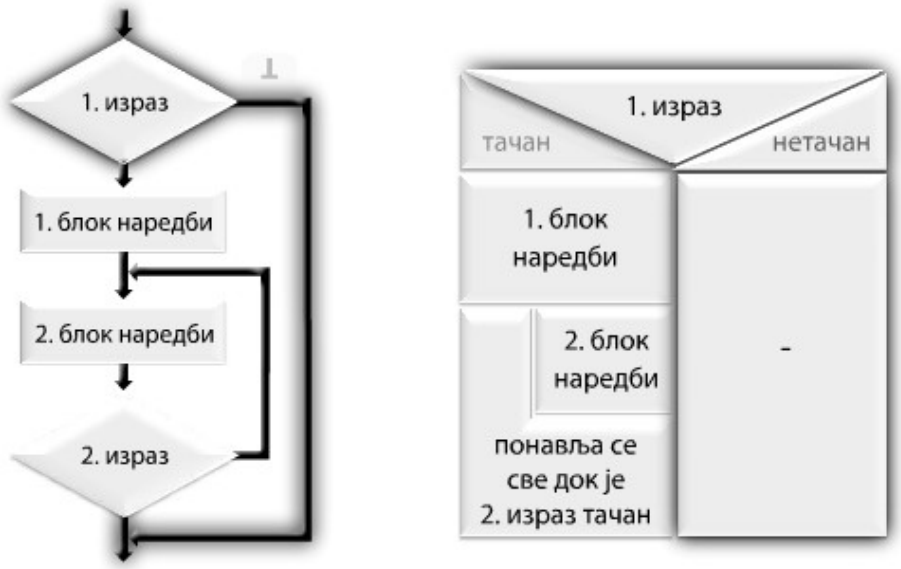
У програмима се често користе уклопљене структуре, које представљају структуре написане једна унутар друге. За број нивоа уклапања нема ограничења, али је некада једноставније заменити непрегледно вишеструко уклапање неком другом структуром. Уклапање је дозвољено код структура међусобно истог типа, више селекција један унутар друге, слика 14, или више циклуса један унутар другог, слика 15, али и код међусобно различитих структура, слика 16.



Слика 14. Уклопљене структуре, селекција унутар селекције



Слика 15. Уклопљене структуре, циклус унутар циклуса



Слика 16. Уклопљене структуре, циклус унутар селекције

Питања

1. Кроз које основне фазе пролази развој сваког програма?
2. Које се фазе сматрају кључним у току равоја сваког програма?
3. Шта све обухвата дефинисање програмског задатка?
4. Која фаза развоја програма је неопходна пре почетка писања изворног кода?
5. Од чега зависи избор методе пројектовања програма?
6. Које су основне карактеристике структурне методе, а које објектно оријентисане методе пројектовања програма?
7. Који програмски алат се уобичајено користи за писање изворног кода програма?
8. Који су задаци програмских алата компајлера, линкера и дибагера?
9. Које су основне методе тестирања рада програма?
10. Шта све обухвата одржавање програма?
11. Шта представљају алгоритми за програме и како се они могу представљати?
12. Које су основне структуре заступљене у програмима?
13. Где се у програмима користи структура селекције?
14. Какве су структуре петљи заступљене у програмима?
15. Које се програмске структуре могу уклапати једне унутар других?

1. Основне компоненте програма

Кратак преглед лекције

Програмски језик *C* користи се у системском и процесорском програмирању. Развој програмских алата под разним оперативним системима незамислив је без овог језика, а програмирање на језицима *C++, C#* и *Java,* много је лакше након познавања језика који је послужио као основа за развој свих наведених програмских језика. Један *C* програм пролази кроз одређене фазе развоја које ће даље бити укратко описане, а обично има велики број компоненти које су искомбиноване на одговарајући начин. За почетак биће једноставно описане само основне компоненте једног *C* програма.

1.1. Програмски језик *C*

Програмски језик који је претходио језику *C* звао се *B* јер је развијен у *Bell* телефонским лабораторијама у Њу Џерзију (New Jersey). У истој лабораторији Денис Ричи (*Denis Ritchie)* је 1972. године написао језик *C* са циљем да на једном језику, који је за разлику од претходнх погодан за системско програмирање, развије оперативни систем *UNIX*. Ричи је успео у томе и језик *C* је постао популаран у системском програмирању. Године 1988. Амерички Институт за стандарде усвојио је *ANSI* дефиницију језика *C,* која и данас важи.

Синтакса језика *C* дозвољава више начина за писање већине наредби на овом језику, неки су више разумљиви а неки мање за програмере почетнике. У почетној фази учења овог језика увек се препоручује писање на најједноставнији могући начин а касније је дозвољено комбиновање појединих елемената у наредбама, као и самих наредби, али само до граница разумљивости програма. Сви делови програма који садрже веома сложене наредбе треба да буду описани и објашњени помоћу коментара.

1.2. Фазе развоја програма

Пројектовање програма

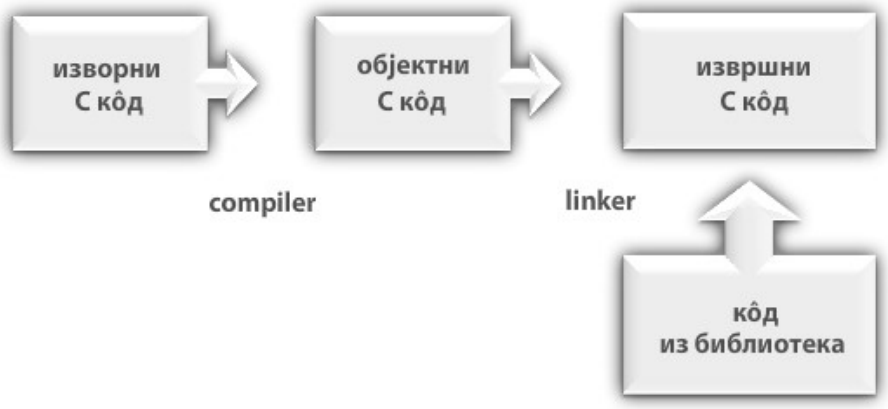
За пројектовање програма на језику *C* користи се структурни модел, који подразумева решавање задатка по деловима. У главном програмском задатку препознају се основни задаци и за сваки од њих предвиди се посебан потпрограм који се позива из главног прогама. Алгоритми (идејна решења) свих потпрограма, уклапају се у алгоритам главног програма. Програмски језик C дозвољава груписање и распоређивање потпрограма једног програма по различитим датотекама (модулима) које све заједно чине један пројекат.

Писање изворног кода програма

Изворни кôд програма на језику *C* чине наредбе написане по *C* синтакси, састављене од речи на енглеском језику (или њихових скраћеница), симбола и заграда. Свако савремено окружење за развој *C* програма има свој едитор текста за унос и измену изворног кода. Овакав едитор синтаксно боји кôд (има резервисане боје за поједине врсте наредби) и пријављује поруке о евентуалним синтаксним грешкама. Изворни кôд програма на језику *C* је само кôд сачуван у једној или у више датотека, од којих свака има екстензију *.c.*

Превођење и повезивање програма

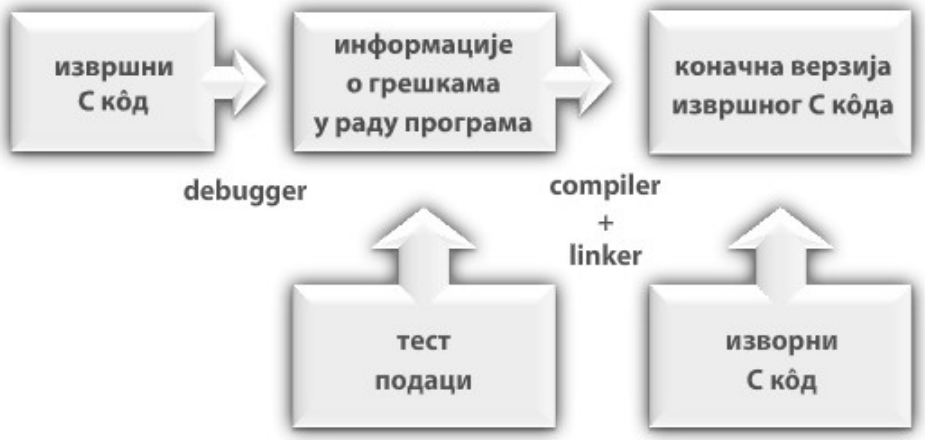
Део *C* развојног окружења за превођење изворног кода у објектни (машински) кôд зове се преводилац *(C compiler)*. Део *C* развојног окружења за повезивање *(C linker)* од објектног кода генерише извршни кôд тако што повезује све модуле, као и све стандардне *C* библиотеке које се укључују у модуле једног програма, слика 1.1. Генерисање извршног кода значи да је програм написан по свим правилима синтаксе. У супротном, програмер добија поруке о синтаксним грешкама, њихове описе и места у изворном коду.



Слика 1.1 Дијаграм превођења и повезивања програма

Тестирање рада програма

Извршавање *C* програма подразумева да су по правилима *C* синтаксе написане све наредбе, укључене потребне стандардне *C* библиотеке и повезани сви предвиђени модули. Одсуство грешака при превођењу и повезивању програма не значи да програм ради без грешака. Део *C* развојног окружења за откривање грешака при раду програма *(C debugger)*, слика 1.2, омогућава: извршавање програма корак по корак, постављање тачака прекида и праћење садржаја меморијског простора који се користи из истог програма.



Слика 1.2 Дијаграм тестирања рада програма

Документовање програма

Документовање *C* програма обухвата следеће:

- израду документа који описује намену програма, пројектне захтеве и ресурсе,

- израду алгоритама који описују рад програма (у облику текстуалних описа, псеудокода и графичких дијаграма),
- израду интерне документације у изворном коду програма (*C* коментара),
- израду докумената тестирања рада програма (са описима тест података, изузетака које обухватају тест подаци и резултата тестирања).

1.3. Синтакса и основне наредбе програма

Програмски језик *C* нема велики број резервисаних речи (то су речи од којих свака има једнозначну примену), слика 1.3, што значи да нема ни велики број наредби. Разлог овоме је постојање великог броја стандардних *C* библиотека које нуде готова решења програмских задатака и које се интензивно користе. Код додела разних имена језик C разликује велика и мала слова (код резервисаних речи дозвољена су само мала слова, а код имена које се уводе у програм дозвољена је само величина слова која је дефинисана на почетку).

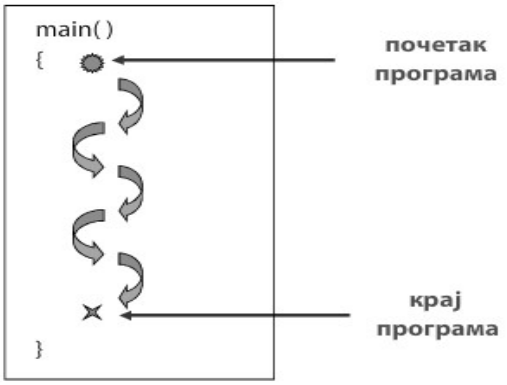
Резервисане речи језика "C"			
auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Слика 1.3 Речник језика *C*

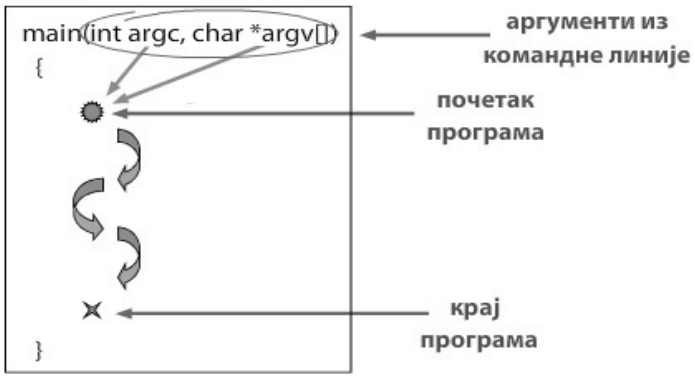
Главна функција

Функција је компонента *C* програма која се пише да реши одређен задатак унутар главног програма. Свака функција има своје име, може имати аргументе (податке које користи од осталих делова програма), има своје наредбе и може враћати резултат на место у програму одакле је позвана. Из стандардних *C* библиотека позива се велики број готових функција које решавају једноставне програмске задатке. За остале функције неопходно је дефинисати наредбе. Од функција које се пишу, на почетку је објашњена само главна функција.

Сваки *C* програм има само једну главну функцију, из које се непосредно или посредно позивају све остале функције. Ова функција може бити написана у две верзије. После имена функције *main* пишу се заграде ( ), које могу бити празне, слика 1.4, што ће бити случај у почетку, или могу садржати имена аргумената, података од оперативног система под којим се позива програм, слика 1.5. Следе витичасте заграде { } а између њих наредбе програма. Извршавање програма почиње са првом наредбом главне функције.



Слика 1.4 Једна верзија



Слика 1.5 Друга верзија главне функције

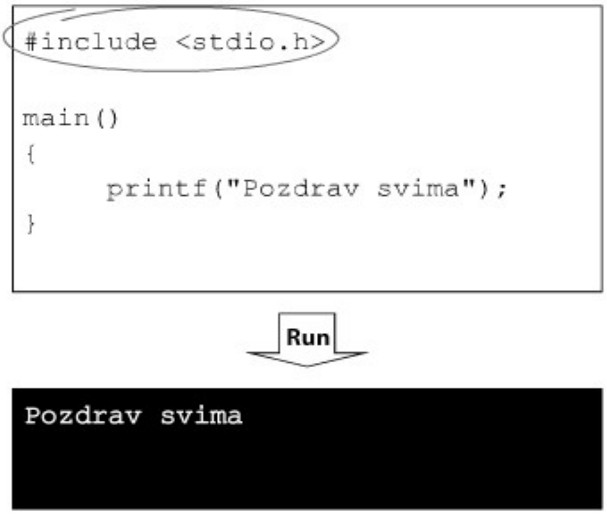
Претпроцесорске директиве

Претпроцесор је компонента *C* развојног окружења која припрема изворни кôд за превођење. Претпроцесорске директиве представљају посебан облик наредби. То су упутства за рад претпроцесору. Симбол # означава почетак било које овакве директиве. Свака претпроцесорска директива пише се у посебном реду и није дозвољено писање неке друге директиве или наредбе у њеном продужетку. Ово је изузетак, све остале компоненте у *C* програму могу бити написане једна у продужетку друге.

Претпроцесорска директива #include

И најједноставнији *C* програм, који само штампа на екрану текст, користи стандардну *C* библиотеку. Претпроцесорска директива *#include* представља упутство претпроцесору да у програм, на место на ком је написана, укључи *C* библиотеку чије је име написано у продужетку речи *include.* Када се зада команда за превођење програма, претпроцесор преузима изворни кôд и по

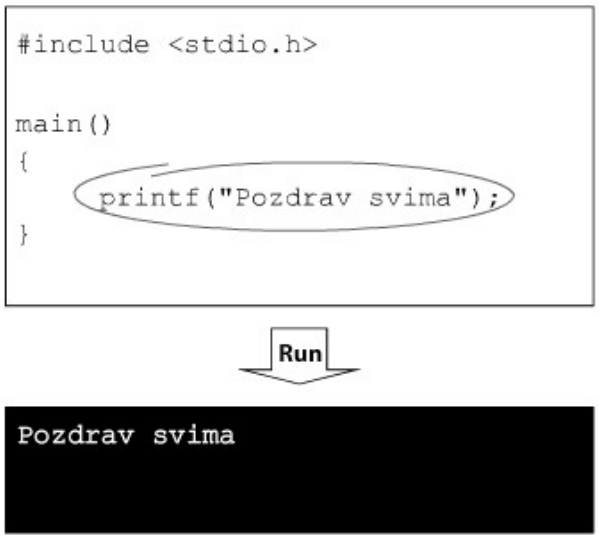
директиви *#include <stdio.h>* укључује у програм са директоријума *include* библиотеку улаза / излаза *stdio.h.*, слика 1.6.



Слика 1.6 Програм помоћу директиве #include користи библиотеку

Функције из стандардних *C* библиотека

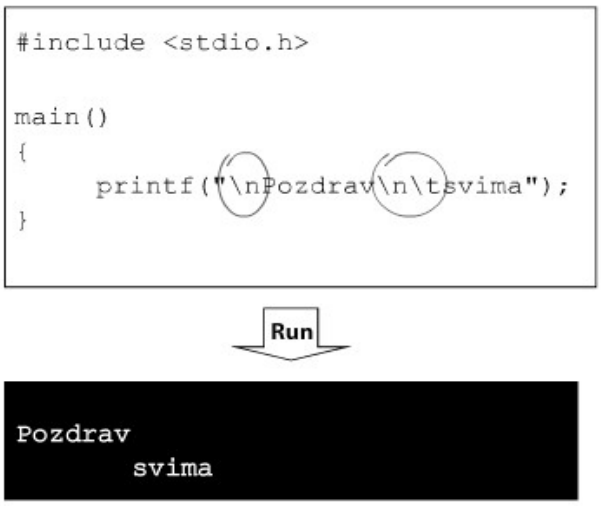
Из стандардних *C* библиотека позива се велики број готових функција које решавају једноставне програмске задатке. Инсталација сваког *C* развојног окружења омогућава коришћење овакве функције, под условом да је у програм укључена библиотека која садржи ту функцију. Групе готових функција су: математичке, за рад са знаковима, за рад са низовима знакова, за рад са улазно / излазним уређајима... У првим примерима заступљена је само функција *printf()* за излаз података на екран, слика 1.7, из библиотеке *stdio.h*.



Слика 1.7 Програм користи функцију *printf()* из библиотеке *stdio.h*

Стандардна функција форматираног излаза

Функција *printf()* из библиотеке *stdio.h* користи се за форматиран излаз текстуалних и нумеричких података на екран. То значи да штампајуће знакове приказује онако како је задато између знакова наводника, у првом аргументу при позиву функције. Од управљачких (нештампајућих) знакова (коса црта уназад и одговарајуће слово) највише се користе прелаз у нови ред (\n) и хоризонтални табулатор (\t), слика 1.8. Специјални знакови (\ и “) приказују се на екрану на посебан начин, слика 1.9.



Слика 1.8 Формат стринг функције *printf()* садржи управљачке знакове

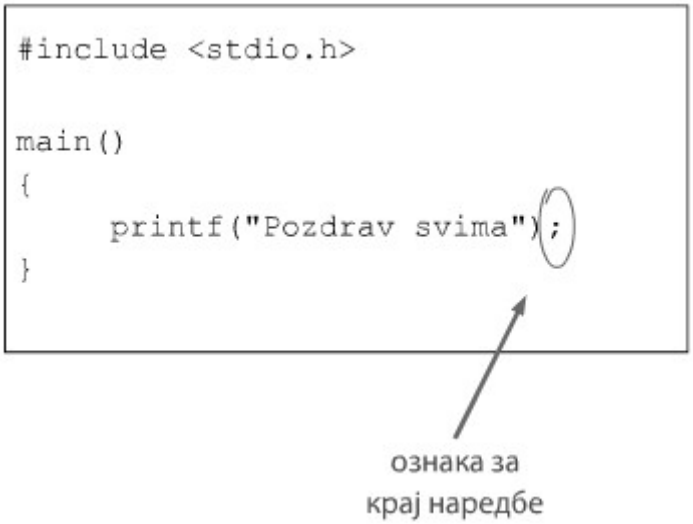
Секвенца	Опис
\n	Постављање курсора на почетак следећег реда
\t	Хоризонтални померај до следећег tab stopa
\a	Звоно упозорења
\\	Испис обрнуте косе црте
\"	Испис наводника

Слика 1.9 Контролне секвенце управљачких знакова

Наредбе

Функција C програма извршава се кроз своје наредбе. Наредба је је израз написан по синтакси језика C, са знаком тачка-зарез (;) на крају, слика 1.10. Само име операнда (податка над којим треба да се изврше операције)

представља најједноставнији израз. Формирање наредбе није могуће само од овог, најједноставнијег израза. За наредбу је потребан сложен израз који се добија применом оператора (дозвољених операција) над операндима, једним или више њих, слика 1.11.



Слика 1.10 Програм "Pozdrav svima" има једну наредбу

Слика 1.11 Начини писања наредби у програму

Врсте наредби

Извршавање *C* програма почиње првом наредбом функције *main*. У било којој функцији програма заступљене су три основне групе наредби. Прву групу чине наредбе декларације које у програм уводе имена операнада и резервишу за њих

потребан простор у меморији. Другој групи припадају извршне наредбе које садрже операције над операндима (између осталих и позиве функција). Трећу групу чини само наредба повратка која враћа резултат функције. Свака од ових група наредби детаљно је описана у некој од следећих лекција.

Коментари

Иако није неопходна компонента, коментар је обично заступљен у изворном коду било ког програма да појасни синтаксу програмског језика и опише програм. Ова компонента нема утицаја на извршавање програма јер је преводилац игнорише. У *C* програму коментар се пише између симбола /* и симбола */. Један коментар може бити написан у једном реду, слика 1.12, или у више редова, слика 1.13, а место му је на почетку датотеке, на почетку функције, на почетку групе наредби или у продужетку неке наредбе програма.

```
#include <stdio.h>

main()
{
     printf("Pozdrav svima"); /*Stampanje teksta*/
}
```

Слика 1.12 Линијски коментар у програму

```
#include <stdio.h>

main()
{
     /*Stampanje teksta
      pomocu funkcije
      formatiranog izlaza*/

     printf("Pozdrav svima!");
}
```

Слика 1.13 Блоковски коментар у програму

1.4. Питања

1. Који се модел пројектовања користи за развој програма на језику *C*?
2. Да ли се код идентификатора језика *C* прави разлика велика / мала слова?
3. Које су врсте компоненти обавезно заступљене у програмима на језику *C*?
4. Које се ознаке користе за почетак и крај тела (наредби) сваке *C* функције?
5. Који се симбол користи за почетак претпроцесорске директиве у *C* програму?
6. Које значење има претпроцесорска директива #include?
7. Под којим условом *C* програм може користити функције из једне стандардне библиотеке?
8. Која стандардна *C* функција ради форматирани излаз на екран?
9. За које врсте података се може користити *C* функција форматираног излаза на екран?
10. Која је ознака за крај наредбе у *C* програму?
11. Које су све врсте наредби заступљене у *C* програмима?
12. Да ли више наредби може бити написано у једној линији изворног кода *C* програма?
13. Да ли су коментари обавезне компоненте *C* програма и које значење имају?
14. Да ли се један коментар може написати у наставцима, у више линија изворног кода *C* програма?
15. Који је минимални број линија изворног кода *C* програма, који на екрану приказује реч "pozdrav"?

2. Основни типови података

Кратак преглед лекције

У програмима се често користе разни типови података. У *C* програмском језику дефинисани су само нумерички основни типови података. Помоћу основних *C* нумеричких типова могу се представити у програму и сачувати у меморији нумерички и текстуални подаци (при чему једни и други могу бити промењиви и фиксни). При додели простора за одређени податак у меморији, програмер треба да изабере *C* тип који обезбеђује довољан број бајтова, али без непотребног заузећа меморије.

2.1. Промењиве

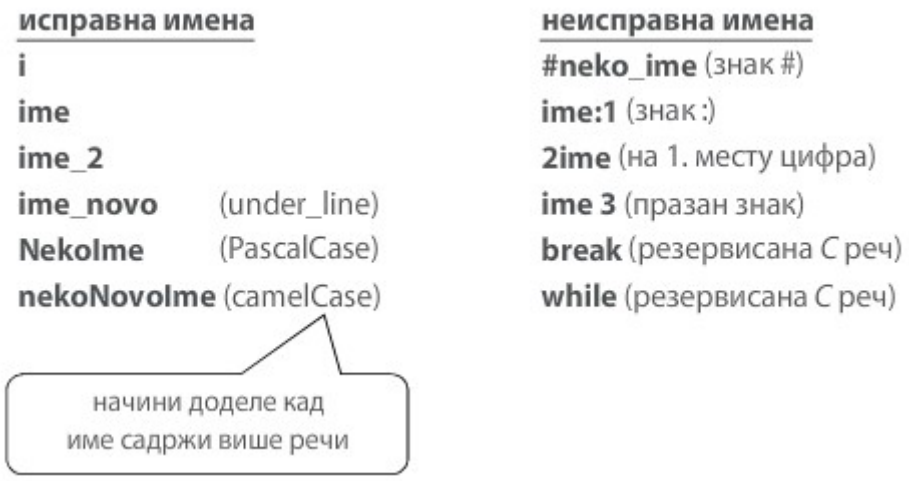
Промењива је локација у оперативној меморији која се резервише из једног *C* програма а користи се за чување податка чија се вредност може мењати током извршавања истог програма. За увођење промењиве у програм користи се наредба декларације. Декларација промењиве треба да пружи информације о њеном типу (да би у меморији био издвојен потребан број бајтова) и имену (да би се разликовала од осталих промењивих у програму), а евентуално и о иницијалној вредности промењиве, ако је од саме декларације потребна.

Имена промењивих

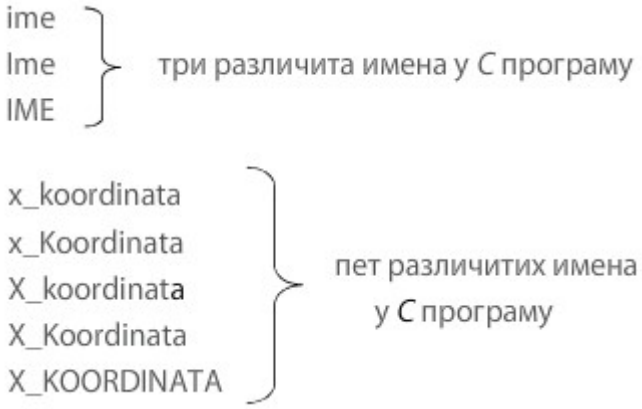
За доделу имена промењивама у језику *C* важе правила за идентификаторе:

1. име може садржати само слова, цифре и знак доња црта (_),
2. на месту првог знака у имену може бити слово или знак доња црта (_),
3. у имену се прави разлика: мала / велика слова,
4. име не може бити нека од резервисаних речи језика *C*, слика 2.1.

Постоје и следеће препоруке: користити мала слова (изузетак су почетна слова од речи, ако их има више у имену) и бирати имена која имају смисла, слика 2.2.



Слика 2.1 Примери исправно и неисправно додељених имена



Слика 2.2 Примери са разликама мала / велика слова у именима

Типови промењљивих

У језику *C* дефинисани су само нумерички типови промењљивих и то две основне групе: целобројни и реални. Свака од ових група има своје типове а избор зависи од потребног опсега вредности (код целобројних) или од потребне прецизности (код реалних). Један од оператора језика C је оператор *sizeof(tip)*, који као резултат даје величину у бајтовима оног типа промењљиве чија је ознака написана унутар заграда. Примена овог оператора биће приказана над ознакама конкретних *C* типова.

Целобројни типови промењљивих

За чување једног знака или целог броја величине 1 бајта (*1B*) довољно је резервисати тип *char*. За чување целог броја користе се тип *short, int* или *long*.

Величина типа *short* је *2B,* величина типа *int* зависи од имплементације, *2B* је код 16-битног а *4B* код 32-битног режима у рачунару, док је величина типа *long* увек *4B*. Сваки од наведених типова, слика 2.3, може имати модификатор *unsigned* (*unsigned char, unsigned short, unsigned int* и *unsigned long)* и значити опсег исте величине, али померен од вредности 0 (само позитиван), слика 2.4.

Тип	Ознака	Величина (у бајтовима)	Опсег
Character	char	1	-128 до 127
Short	short	2	-32768 до 32767(1)
Integer	int	2/4	(1)/(2)
Long	long	4	-2,147,483,648 до 2,147,438,647(2)

Слика 2.3 Основни означени целобројни типови података

Тип	Ознака	Величина (у бајтовима)	Опсег
Unsigned character	Unsigned char	1	0 do 255
Unsigned Short	Unsigned short	2	0 do 65535 (3)
Unsigned Integer	Unsigned	2/4	(3)/(4)
Unsigned long	Unsigned long	4	0 до 4,294,967,295 (4)

Слика 2.4 Основни неозначени целобројни типови података

Реални типови променљивих

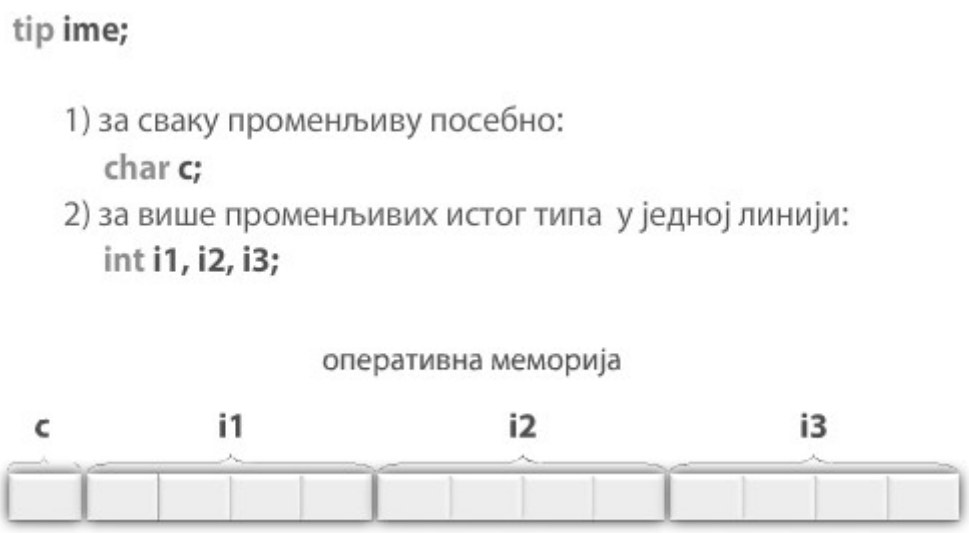
За чување реалног броја користе се типови *float* и *double*. Величина типа *float* је *4B* и омогућава чување реалне вредности са прецизношћу од првих 7 тачних цифара после децималне тачке, док је величина типа *double 8B* и прецизност код овог типа је првих 16 тачних цифара после децималне тачке, слика 2.5. За тачно чување неке реалне вредности, поред резервисања реалног типа тражене прецизности, неопходно је обезбедити у програму тачно учитавање/израчунавање ове вредности.

Тип	Ознака	Величина (у бајтовима)	Опсег
Single-precision Floating-point	float	4	1.2E-38 до 3.4E38 (прецизност=7 цифара)
Double-precision Floating-point	double	8	2.2E-308 до 1.8E308 (прецизност=16 цифара)

Слика 2.5 Основни реални типови података

Декларација промењљивих

Наредба декларације једне промењљиве садржи ознаку типа промењљиве на основу кога се резервише у оперативној меморији одговарајући број бајтова, као и име које ће даље бити коришћено. Место овим наредбама је на почетку сваке функције програма. У једној наредби може бити написана декларација више промењљивих међусобно истог типа. У том случају једном на почетку наредбе пише се заједнички тип, а у продужетку листа имена међусобно одвојених знаком зарез, слика 2.6.

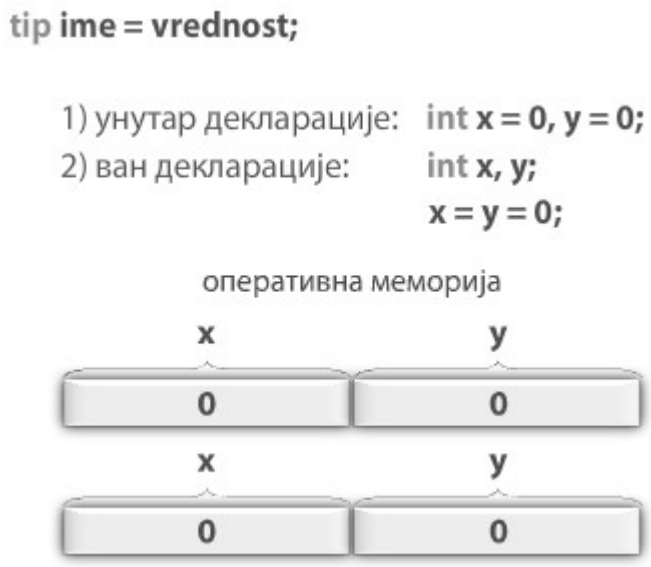


Слика 2.6 Декларација промењљивих

Иницијализација промењљивих

Само декларација промењљиве резервише простор у оперативној меморији, али не уписује у овај простор вредност, већ је то неки случајно затечен садржај. Иницијализација промењљиве значи доделу почетне вредности промењљивој.

Иницијализација једне променљиве може бити изведена унутар њене декларације (ако је ово потребно на самом почетку функције) или ван декларације променљиве (у некој другој наредби, где је потребна вредост те променљиве), слика 2.7.



Слика 2.7 Иницијализација променљивих

2.2. Константе

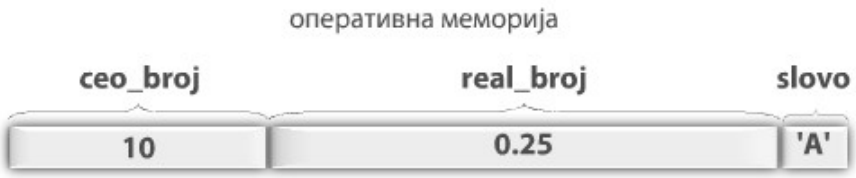
Константе могу бити уведене у *C* програм на два начина. Један начин је резервисана локација у оперативној меморији, из програма а за чување податка чија је вредност фиксна током извршавања истог програма. За увођење овакве константе у програм користи се наредба декларације. Други начин је дефиниција симбола константе помоћу претпроцесорске директиве и код овог начина нема резервисања простора у меморији, симбол се замењује фиксном вредношћу и "не стиже" до преводиоца.

Декларација константи

Наредба декларације једне константе треба да садржи *C* службену реч *const* на почетку, онда ознаку типа на основу кога се резервише у оперативној меморији одговарајући број бајтова, име које ће даље бити коришћено, као и саму фиксну вредност. Ако је написан целобројни тип вредност може бити цео број или знак

написан између апострофа (опет цео број који одговара знаку у *ASCII* табели), док за реалан тип вредност може бити реалан број написан са децималном тачком или експонентом, слика 2.8.

```
const tip ime = vrednost;

const int ceo_broj = 10;          целобројна
const float real_broj = 0.25;     реална
const char slovo = 'A';           знаковна
```



Слика 2.8 Декларација константи

Дефиниција константи

Претпроцесорска директива *#define* може пружити информације о симболу који замењује константу и о самој вредности константе, слика 2.9. Ова константа користи се тако што се у програму пише симбол свуда где постоји потреба за вредношћу константе коју мења. У фази припреме програма за превођење претпроцесор замењује сваки овај симбол вредношћу. У овом случају нема резервисања простора у меморији, као код декларације константе, али нема ни провере типа константне вредности при њеном коришћењу.

```
#define SIMBOL vrednost

#define CEO_BROJ 10       /*celobrojna konstanta*/
#define REAL_BROJ 0.25    /*realna konstanta*/
#define SLOVO 'A'         /*znakovna konstanta*/

#define TAB  '\t'          /*jos jedna znakovna konstanta*/
#define EOL  '\n'          /*i jos jedna znakovna konstanta*/

/*Konstantan niz znakova napisan izmedju navodnika*/
#define PORUKA  "Greska u otvaranju Datoteke!"
```

Слика 2.9 Дефиниција константи

Набројане константе

Више целобројних константи дефинисаних помоћу директиве *#define* могуће је заменити набрајањем. У том случају пишу се: *C* службена реч набрајања *enum*, име набрајања и између витичастих заграда листа симбола, међусобно одвојених знаком зарез. Постоји могућност дефинисања вредности за сваки симбол. За дефинисану вредност само првом симболу подразумева се да је свака следећа вредност за 1 већа од претходне. Ако вредности симбола нису дефинисане, подразумевано су вредности: 0, 1, 2... слика 2.10

```
1) enum ime_nabrajanja {IME1=vrednost1,
                        IME2=vrednost2,
                              ⋮
                        IMEn=vrednostn};
2) enum ime_nabrajanja {IME1=vrednost1,
                        IME2,
                        IME3,
                          ⋮
                        IMEn};
3) enum ime_nabrajanja {IME1,IME2,...,IMEn};
```

Слика 2.10 Набројане константе

2.3. Форматирани улаз / излаз података

Главни стандардни улаз / излаз

За форматиран улаз / излаз података у *C* програмима користе се функције *scanf()* и *printf()* из стандардне библиотеке <*stdio.h*> слика 2.11. Помоћу ових функција нумерички и текстуални подаци учитавају се са тастатуре у оперативну меморију и приказују из меморије на екрану, у оном формату који се наведе при позиву одговарајуће функције. Овакав улаз / излаз подразумева конверзије (из задатог улазног формата са тастатуре у бинарни у меморији и из бинарног у тражени излазни формат на екрану).

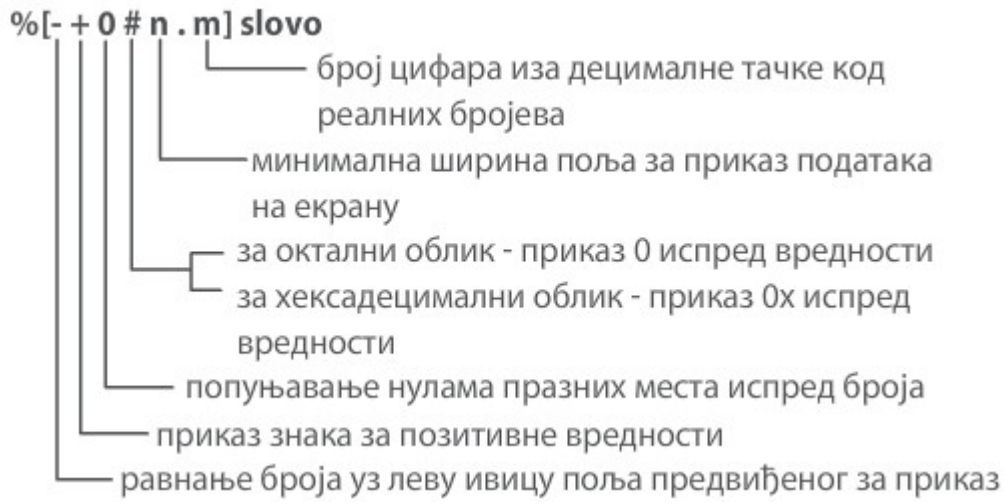
Слика 2.11 Форматирани улаз / излаз података

Форматирање главног стандардног излаза

За форматиран излаз на екрану користи се функција са промењивим бројем аргумената *printf()*. Једини обавезан аргумент ове функције је формат стринг који се пише између знакова навода и овај аргумент је једини кад функција приказује на екрану само текст. За приказ сваког податка из меморије на екрану, стринг формат има знак % и слово које одговара типу податка, слика 2.12. Између знака % и слова могу бити потребно још знакова, слика 2.13. После формат стринга, наводи се име промењиве која чува податак.

%c (char)
%d (int - децимални облик)
%u (unsigned int - децимални облик)
%o (int/unsigned int - октални облик)
%x(X) (int/unsigned int - хексадецимални облик)
%hd,%ho,%hx (short int у сва три облика)
%ld,%lo,%lx (long int у сва три облика)
%f (float/double - облик са децималном тачком)
%e (float/double - облик са експонентом)
%g (float/double - најједноставнији облик)

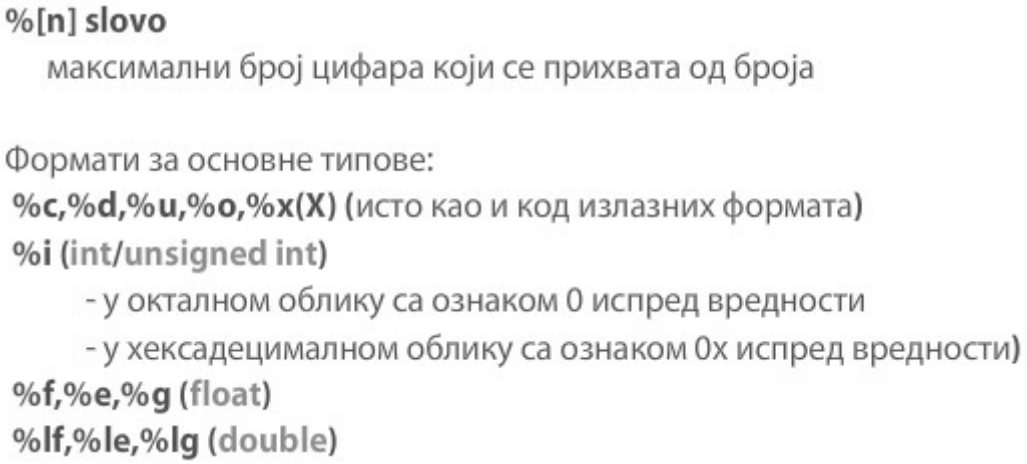
Слика 2.12 Формати за основне типове



Слика 2.13 Форматирани излаз података

Форматирање главног стандардног улаза

За форматиран улаз са тастатуре користи се функција са променљивим бројем аргумената *scanf()*. За упис података са тастатуре у меморију сваки податак треба да има у формат стрингу свој формат (знак %, слово и евентуално између њих одговарајући знак) слика 2.14, као и још један аргумент после формат стринга, са адресом променљиве у меморији у коју се уписује податак. За добијање адресе променљиве користи се симбол адресног оператора *&* написан испред имена променљиве (*&x* представља адресу променљиве *x*).



Слика 2.14 Форматирани улаз података

2.4. Питања

1. Шта су промењљиве, а шта константе?
2. Које врсте знакова могу садржати *C* идентификатори?
3. Који су целобројни *C* типови промењљивих и константи највише у употреби?
4. Који реални *C* типови промењљивих и константи постоје?
5. Да ли декларација промењљивих аутоматски иницијализује њихове садржаје?
6. Који начини постоје за увођење константи у програм?
7. По чему се међусобно разликују декларације промењљивих и константи?
8. Да ли дефиниција симболичких константи резервише простор у меморији?
9. За које врсте константи се може увести тип њиховог набрајања *enum*?
10. Који је минимални број аргумената код функције *printf()*, а који код функције *scanf()*?
11. Које се ознаке формата могу користити код целобројних типова *char* и *int* у формат стрингу код функције *printf()*, а које код функције *scanf()*?
12. Које се ознаке формата могу користити код реалних типова *float* и *double* у формат стрингу код функције *printf()*, а које код функције *scanf()*?
13. Да ли се код приказа на екрану садржаја реалних промењљивих типова *float* и *double* може дефинисати различита врста приказа од подразумеваног?
14. Да ли форматирани улаз / излаз обавезно подразумева конверзије формата података који се учитавају у меморију / из меморије?
15. Да ли функције форматираног улаза / излаза могу радити са мешовитим типовима података?

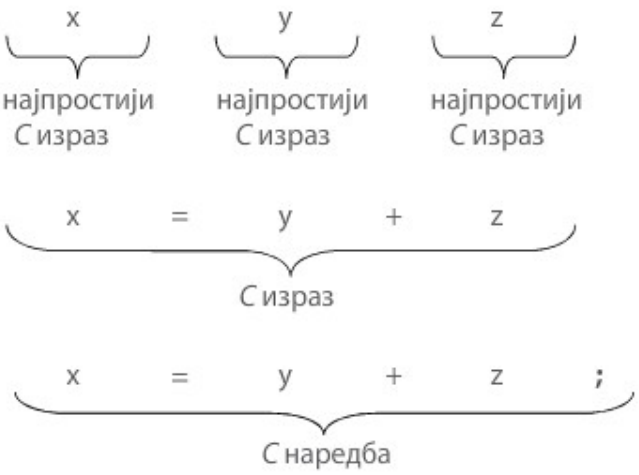
3. Оператори, изрази и наредбе селекције

Кратак преглед лекције

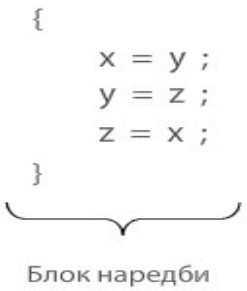
Већина наредби у *C* програму садржи операторе који одређују шта треба урадити са садржајима променљивих и константи, које представљају операнде у наредби. У зависности од броја операнада над којима се примењују у језику C заступљени су унарни, бинарни и тернарни оператори. За сваки оператор синтакса језика има: симбол (или симболе), типове података над којима је применљив, тип резултата и ниво приоритета извршавања. Да би били боље објашњени поједини оператори овде се уводи и наредба селекције *if*.

3.1. Оператори и њихова примена код наредби

Израз је део изворног кода програма који као резултат може имати нумеричку вредност. Додавањем знака тачка-зарез на било који израз, од најпростијег (само име променљиве, име константе или константна вредност) до веома сложеног (добијеног применом оператора над простијим изразима), добија се наредба *C* програма, слика 3.1. Некад је у програму потребно издвојити две или више наредби које представљају програмску логичку целину, ова група наредби написана између витичастих заграда чини блок наредби, слика 3.2.



Слика 3.1 Израз и наредба у програму



Слика 3.2 Блок наредби у програму

Оператор доделе и бинарни аритметички оператори

Оператор доделе (=) је бинарни оператор који израчунава вредност израза (*y)* који је написан на његовој десној страни и резултат додељује променљивој (*x*) чије је име написано на његовој левој страни (*x=y*). Међу бинарним аритметичким *C* операторима има математичких оператора, а симболи неких од њих прилагођени су *C* синтакси: сабирање (+), одузимање (-), множење (*), дељење (/) и остатак дељења (%). У зависности од типа операнада, оператор дељење (/) даје реалну или целобројну вредност количника, слика 3.3.

За декларисане и иницијализоване целобројне променљиве x и y:
x/y - вредност израза је реална вредност количника само ако је бар један операнд реалног типа
x%y - вредност израза је остатак дељења операнда x операндом y

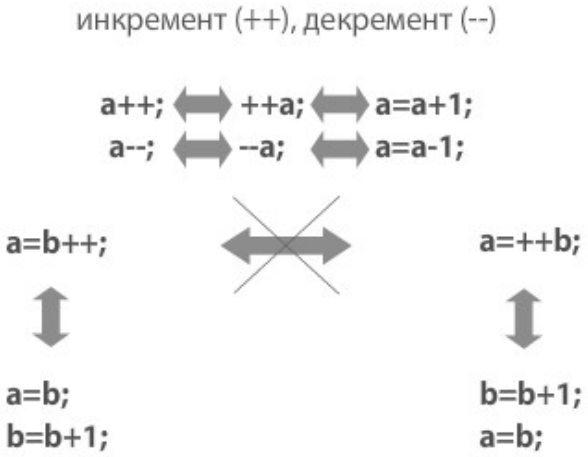
Операција	Алгебарски израз	C израз
сабирање	a + b	a + b
одузимање	a − b	a - b
множење	a · b	a * b
дељење	a : b	a / b
остатак дељења	a **mod** b	a **%** b

Слика 3.3 Бинарни аритметички оператори

Унарни аритметички оператори

Аритметички оператори промена знака (-) и без промене знака (+) пишу се испред својих операнада (-x и +x) као и у математици. Оператор инкремент (++) увећава вредност променљиве чије је име написано (на левој или десној страни овог оператора) за вредност 1. Оператор декремент (--) умањује вредност

променљиве чије је име написано (на левој или десној страни овог оператора) за вредност 1. У сложеном изразу, кад инкремент (или декремент) није једини оператор, битно је на којој страни операнда је написан, слика 3.4.



Слика 3.4 Унарни аритметички оператори

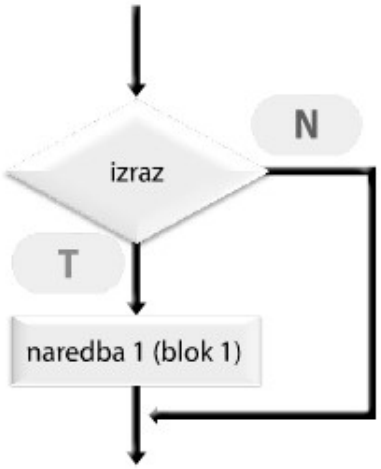
Релацијски оператори и наредба селекције

Релацијски оператори користе се као и у математици, али са новим симболима. Ови оператори су бинарни, сваки од њих проверава одређену релацију између израза на левој и десној страни и као резултат даје вредност различиту од 0 (логичку јединицу) или вредност 0 (логичку нулу), у зависности од тога да ли је релација тачна или не, слика 3.5. *C* програми користе резултате примене следећих релацијских оператора: еквиваленција (==), негација еквиваленције (!=), мање (<), веће (>), мање или једнако (<=) и веће или једнако (>=).

Релација	Релацијски C оператор	C израз	Вредност израза је логичка јединица
=	==	x == y	ако су вредности x и y еквивалентне
≠	!=	x != y	ако вредности x и y нису еквивалентне
<	<	x < y	ако је вредност x мања од вредности y
≤	<=	x <= y	ако је вредност x мања или једнака од вредности y
>	>	x > y	ако је вредност x већа од вредности y
≥	>=	x >= y	ако је вредност x већа или једнака од вредности y

Слика 3.5 Релацијски оператори

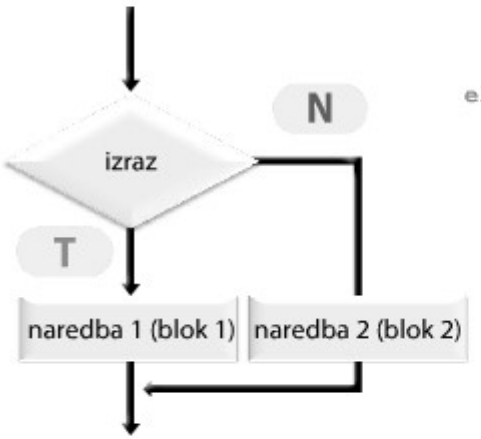
Да би кроз примере били објашњени логички оператори неопходно је увођење наредбе селекције. У *C* програмима наредба *if* омогућава селекцију једног блока наредби под одређеним условом или ниједног блока ако исти услов није испуњен, слика 3.6. Наредба *if* може имати члан *else* за селекцију другог блока наредби ако услов наредбе *if* није испуњен. Члан *else* повезан је са најближом наредбом *if*, слика 3.7. Наредба *if* може бити уклопљена унутар друге наредбе *if*, слика 3.8, или члана *else* друге наредбе *if*, слика 3.9.




```
if (izraz)
    naredba1;
```

```
if (izraz)
{
    naredba1;
    naredba2;
       :
       :
    naredban;
}
```

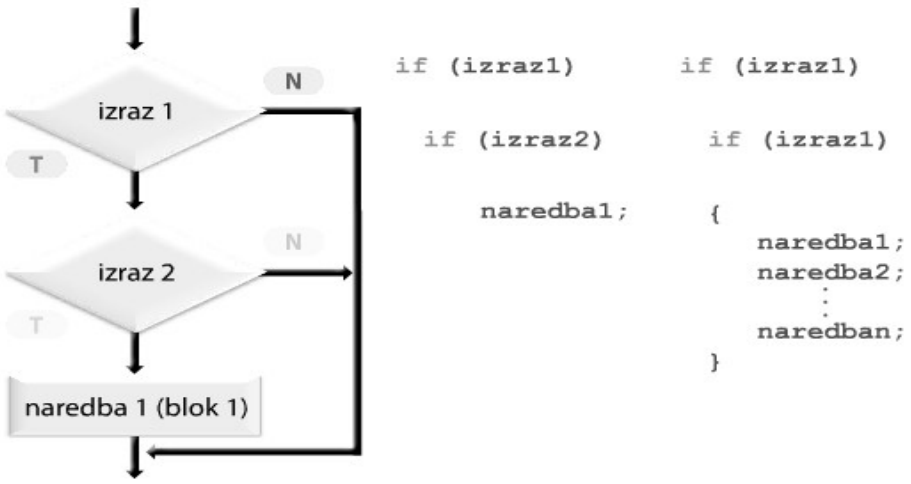
Слика 3.6 Наредба селекције *if*



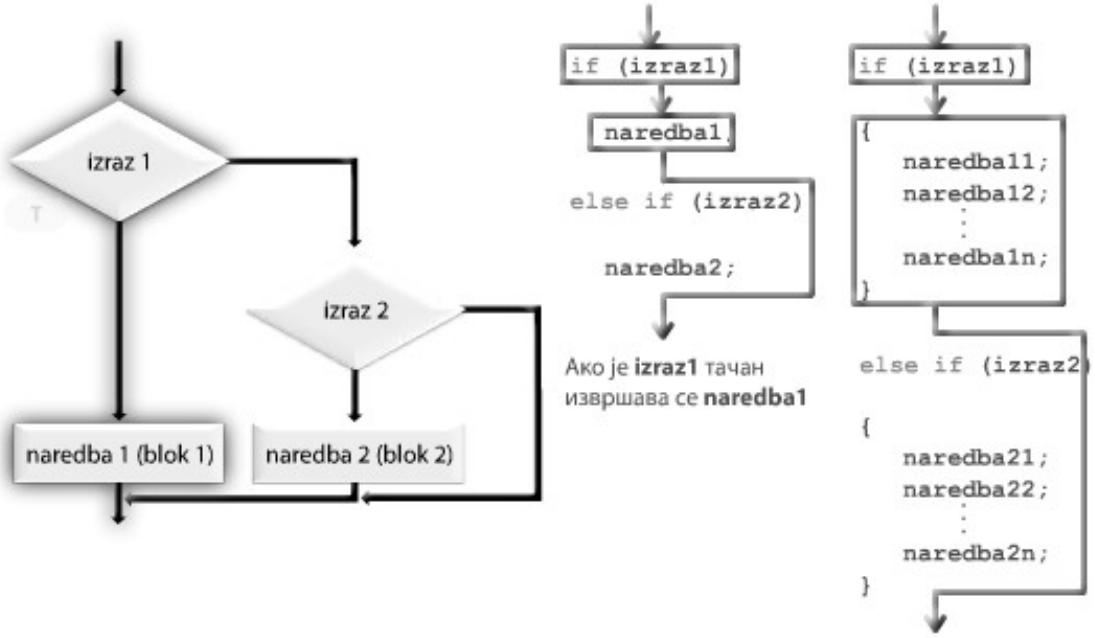

```
if (izraz)
    naredba1;
else
    naredba2;
```

```
if (izraz)
{
    naredba11;
    naredba12;
       :
       :
    naredba1n;
}

else

{
    naredba21;
    naredba22;
       :
       :
    naredba2n;
}
```

Слика 3.7 Алтернативни члан *else* наредбе *if*



Слика 3.8 Наредба *if* угњеждена унутар друге наредбе *if*



Слика 3.9 Наредба *if* угњеждена унутар члана *else* друге наредбе *if*

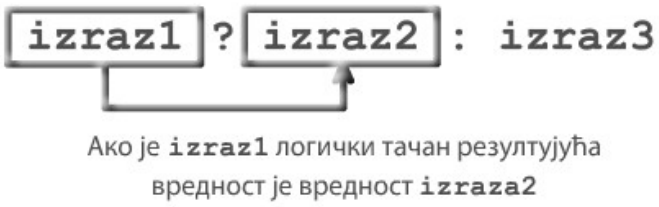
Логички оператори

Логички оператори користе се да смање број наредби, а дају резултат: логичку јединицу или логичку нулу. То су оператори: логички И (&&), логички ИЛИ (||) и логички НЕ (!). Прва два набројана оператора су бинарна, сваки од њих користи се да повеже два *C* израза: први оператор даје логичку јединицу само ако су оба израза логички тачна, а други оператор даје логичку јединицу ако је било који од израза логички тачан. Трећи оператор је унаран, пише се испред израза и даје логичку јединицу само ако је израз логички нетачан, слика 3.10.

Слика 3.10 Логички оператори

Условни оператор

Условни оператор је једини тернарни *C* оператор. Овај оператор садржи два симбола, знак питања и знак двотачка (? и :). Ова два симбола одвајају три С израза и омогућавају следећу одлуку у програму: ако је први написан израз логички тачан резултат примене оператора је вредност другог израза, ако први израз није логички тачан онда је резултат примене оператора вредност трећег израза, слика 3.11. Овај оператор користи се често у С програмима као замена за *if else* наредбу (увек доста краћи али и разумљив запис).



Слика 3.11 Условни оператор

Оператори на нивоу бита

У језику С дефинисани су логички оператори који раде са битовима од својих целобројних операнада. Оператор И на нивоу бита (&) даје вредност 1 на месту

сваког бита на коме оба његова операнда имају вредност 1. Оператор ИЛИ на нивоу бита (|) даје вредност 1 на месту сваког бита на коме бар један од његових операнада има вредност 1, слика 3.12. Оператор ИСКЉУЧИВО ИЛИ на нивоу бита (^) даје вредност 1 на месту сваког бита на коме оба његова операнда имају међусобно различиту вредност. КОМПЛЕМЕНТ (~) мења вредност сваком биту у операнду написаном на десној његовој страни (једини је унарни у овој групи), слика 3.13. Оператори померај улево (<<) и померај удесно (>>) померају битове у првом операнду (на левој страни оператора), за број места које чува други операнд (на десној страни оператора), у одређеном смеру, слика 3.14. Битови чије се вредности померају у било ком смеру, допуњују се нулама. Ови оператори омогућавају: проверу вредности, постављање на вредност 1 или 0 и измену вредности појединачних и група битова, слика 3.15.

& логички I

1. бит операнд \ 2. бит операнд	0	1
0	0	0
1	0	1

Оператор **&** се користи за "маскирање" (бирање) задатог бита у податку

| логички ILI

1. бит операнд \ 2. бит операнд	0	1
0	0	1
1	1	1

Оператор | се користи за "сетовање" (постављање) задатог бита у податку.

Слика 3.12 Оператори логички И и логички ИЛИ на нивоу бита

^ логички искључиво ILI

1. бит операнд \ 2. бит операнд	0	1
0	0	1
1	1	0

Оператор ^ се користи за промену задатог бита у податку.

~ комплемент

бит операнд	0	1
комплемент	1	0

Слика 3.13 Оператори логички ИСКЉУЧИВО ИЛИ на нивоу бита и КОМПЛЕМЕНТ

Оператор << помера битове податка 1. операнда у лево за вредност 2. операнда (крајње десне битове поставља на 0)

Оператор >> помера битове податка 1. операнда у десно за вредност 2. операнда (битови са леве стране се одбацују)

За декларисане целобројне променљиве x и y:

```
x = 0x23;          (00100011)
x << 2 => 0x8c     (10001100)
y = 0xaa;          (10101010)
y >> 2 => 0x2a     (00101010)
```

Слика 3.14 Оператори померај улево и померај удесно на нивоу бита

```
if( x & ( 1 <<( n-1 ) ) )
  printf("%d. bit ima vrednost 1",n);
```
} Провера вредности n-тог бита у податку x

```
x = x | ( 1 <<( n-1 ) );
```
} Постављање n-тог бита у податку x на вредност 1

```
x = x & ~( 1 <<( n-1 ) );
```
} Постављање n-тог бита у податку x на вредност 0

```
x = x ^ ( 1 << ( n-1 ) );
```
} Измена вредности n-тог би у податку x (1 у 0 или 0 у 1)

Слика 3.15 Читање регистара и њихово програмирање

Оператори сложене доделе

У *C* програмима могуће су комбинације оператора доделе (=) са једним од следећих оператора: бинарним аритметичким (+, -, *, / и %) и бинарним на нивоу бита (&, |, ^, << и >>). Када садржај једне променљиве треба да се измени применом неког од поменутих оператора, и тако измењен садржај додели истој променљивој, овај оператор комбинује се са оператором доделе, слика 3.16. На

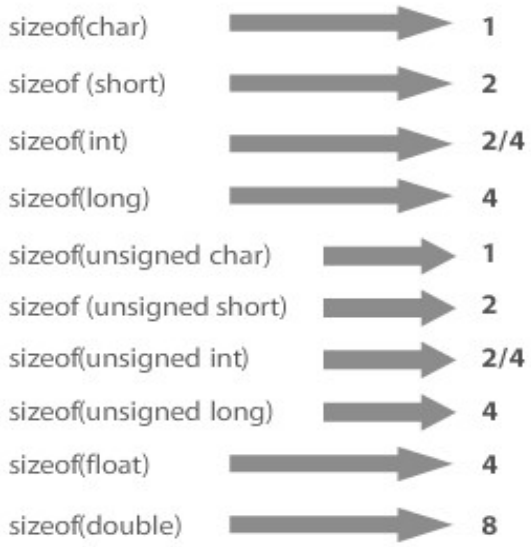
овај начин избегава се дуплирање имена променљиве, којој се додељује измењен првобитни садржај исте променљиве и скраћује се наредба.

Сложени C оператор	C израз	Еквивалентан C израз
+=	x += y	x = x+ y
-=	x -= y	x = x - y
*=	x *= y	x = x * y
/=	x /= y	x = x / y
%=	x %= y	x = x % y
&=	x &= y	x = x & y
|=	x |= y	x = x | y
^=	x ^= y	x = x ^ y
<<=	x <<=y	x = x << y
>>=	x >>= y	x = x >> y

Слика 3.16 Оператори сложене доделе

Оператор величина типа

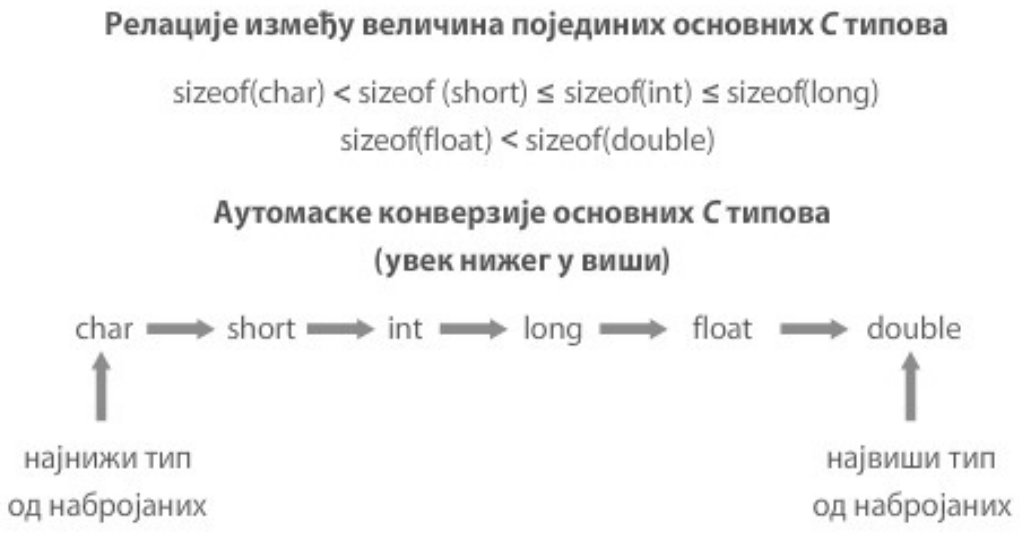
Оператор величина типа даје величину у бајтовима оног *C* типа чија је ознака написана унутар заграда, у продужетку резервисане *C* речи *sizeof*. Овај оператор може дати величину било ког основног *C* типа (*sizeof*(*char), sizeof(short), sizeof(int), sizeof(long), sizeof(float), sizeof(double)...*), слика 3.17. Исти оператор може дати величину било ког изведеног *C* типа (показивача, низа, структуре..) и променљиве било ког типа. На овај начин у програму се израчунава величина меморијског блока резервисаног из истог програма.



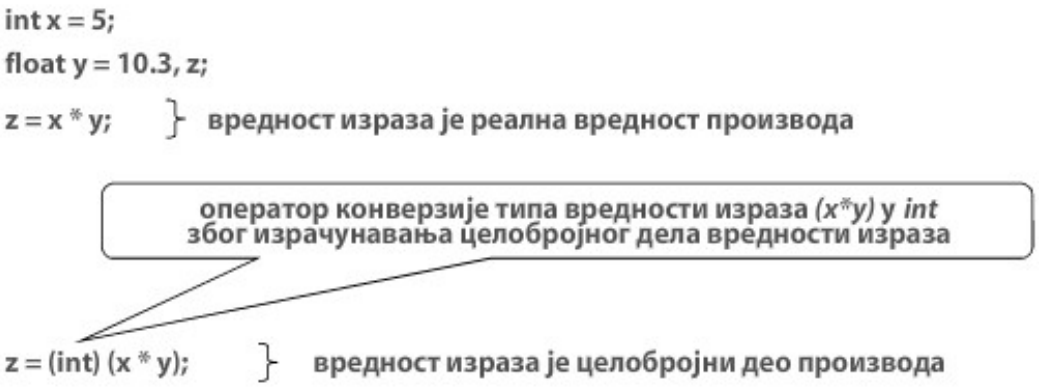
Слика 3.17 Оператор величине типа

Оператор конверзија типа

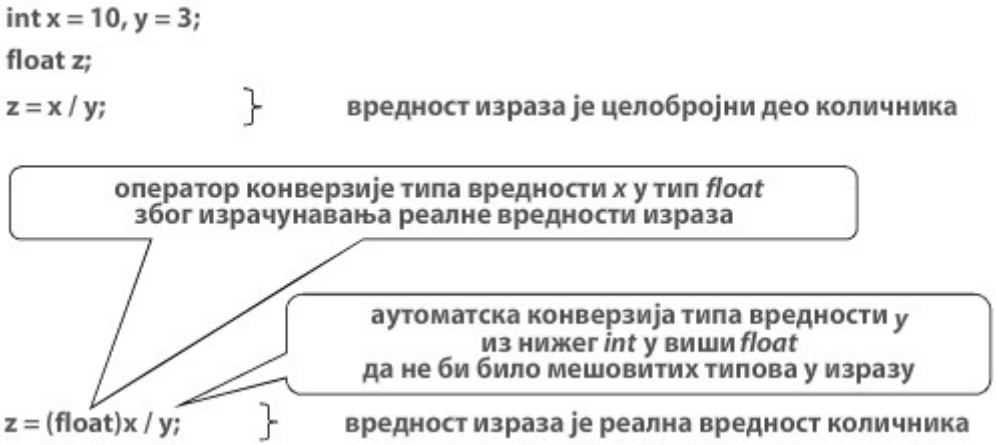
У језику C дозвољена је употреба операнада међусобно различитих типова у истој наредби. Онда се у наредби реализује аутоматска конверзија, увек нижег типа података у виши, слика 3.18. Ова конверзија не условљава измене у начину чувања података у оперативној меморији. Оператор конверзија типа *(tip)* написан пре операнда неопходан је у обрнутом случају, за конверзију вишег у нижи тип, слика 3.19, или за конвертовање типа једног операнда због касније аутоматске конверзије свих осталих у тражени тип, слика 3.20.



Слика 3.18 Оператор конверзије типа



Слика 3.19 Пример први примене оператора конверзије типа



Слика 3.20 Пример други примене оператора конверзије типа

Остали оператори

Оператори индекс у низу [ ] и позив функције ( ) интензивно се користе у *C* програмима, али они су детаљно описани код низова и функција. Има и оператора који се користе код показивача и структура и због тога је њихов опис за сада изостављен, то су: адресни оператор (&), оператор индиректног приступа (*), оператор приступа члану структуре (.), оператор индиректног приступа члану структуре (->). Оператор зарез (,) не припада ниједној од претходно описаних група оператора а скраћује запис наредбе, слика 3.21.

```
& адресни оператор
* индиректни оператор
. оператор приступа члану структуре
-> оператор приступа члану структуре помоћу показивача
[] оператор индекс у низу
() оператор позив функције
, оператор зарез
x = (++y, ++z); <=> y = y + 1;
                    z = z + 1;
                    x = z;
```

Слика 3.21 Остали оператори језика *C*

Листа приоритета *C* оператора

Као и у математици, у сваком програмском језику постоје правила у вези редоследа извршавања оператора, кад их је више у једном изразу. На првом месту редослед извршавања *C* оператора одређују заграде (овде су дозвољене само заграде), на другом месту је табела приоритета *C* оператора, слика 3.22, а на трећем смер извршавања оператора слева удесно. Правилном употребом приоритета оператора могу се избећи у програму синтаксне али и логичке грешке.

Приоритет	Број операнада	Оператор
15	2	[] () . ->
14	1	! ~ ++ -- + - * & (тип) sizeof
13	2	* / %
12	2	+ -
11	2	<< >>
10	2	< <= > >=
9	2	== !=
8	2	&
7	2	^
6	2	|
5	2	&&
4	2	||
3	3	?:
2	2	= += -= *= /= %= &= ^= |= <<= >>=
1	2	,

Слика 3.22 Листа приоритета *C* оператора

3.2. Стандардне *C* библиотеке

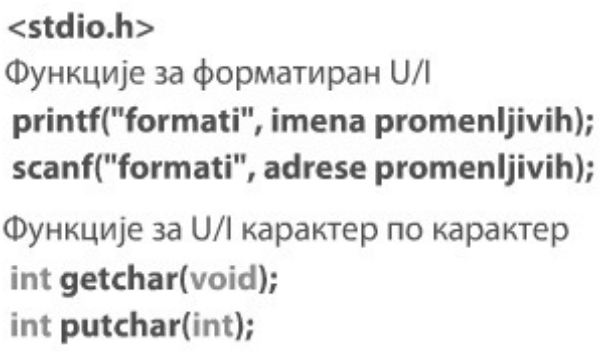
Функције из стандардних *C* библиотека извршавају одређене програмске задатке, слика 3.23, тако да представљају проширење скупа C оператора. Слично операторима, свака оваква функција извршава се кад се позове у некој наредби програма и при томе јој се предају аргументи а добија се од ње резултат. За разлику од *C* оператора који могу имати један, два или три операнда, функције могу имати нула или неограничен број аргумената. Стандардне *C* функције укључују се из библиотека помоћу директиве *#include.*

име библиотеке	опис функција које садржи библиотека
stdio.h	Функције улаза / излаза
ctype.h	Функције које раде са знаковима
math.h	Математичке функције
string.h	Функције које раде са стринговима
time.h	Функције са рад са датумима и временом
stdlib.h	Функције за: конверзију текста у број, и обрнуто, доделу меморије, генерисање случајних бројева …

Слика 3.23 Стандардне библиотеке које се често користе у *C* програмима

Стандардна библиотека *stdio.h*

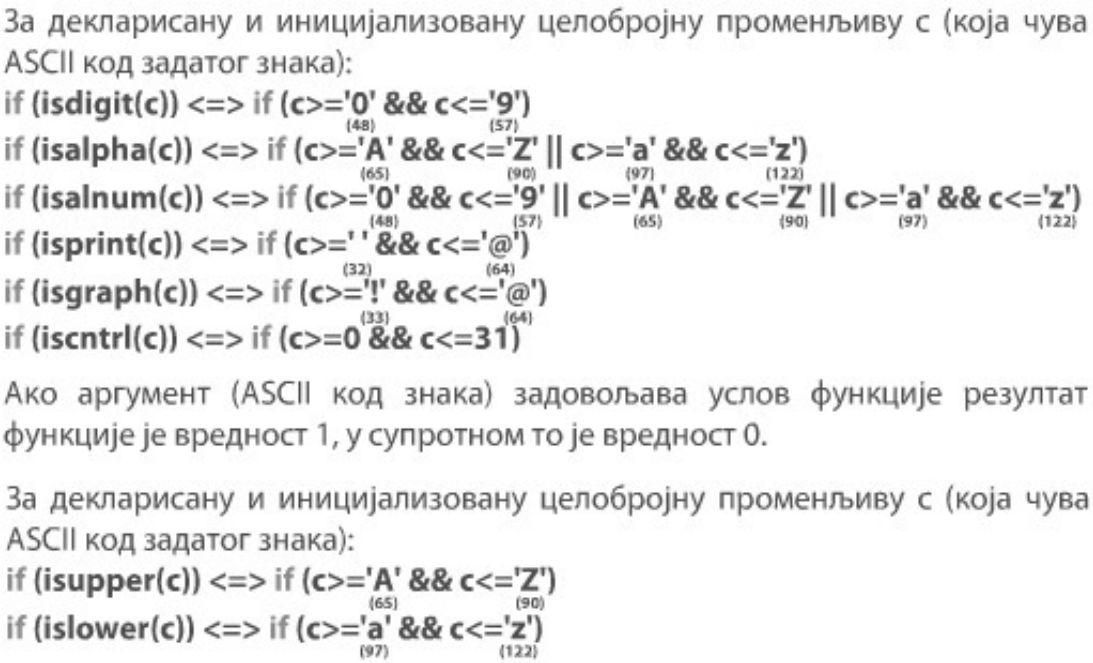
Библиотека *stdio.h* садржи функције за улаз / излаз података. Функције форматираног улаза / излаза *scanf()* и *printf()* објашњене су на самом почетку. За читање са тастатуре и приказ на екрану података, али карактер по карактер, користе се функције *getchar()* и *putchar()*, слика 3.24. Функција *getchar()* чита знак са тастатуре (нема аргументе) и враћа као резултат *ASCII* вредност учитаног знака. Функција *putchar()* добија у аргументу *ASCII* вредност знака који треба да прикаже на екрану и враћа *ASCII* вредност приказаног знака.



Слика 3.24 Функције за улаз / излаз знак по знак

Стандардна библиотека *ctype.h*

Библиотека *ctype.h* садржи групу функција за тестирање знакова: *isdigit(), isalpha(), isalnum(), isprint(), isgraph(), iscntrl(), isupper(), islower()...* Свака од ових функција користи *ASCII* вредност знака, и ако је задовољен услов који је описан именом функције даје логичку јединицу а у супротном логичку нулу, слика 3.25. Функције *toupper()* и *tolower()* конвертују: мало слово у исто велико и велико слово у исто мало. Свака од ових функција треба да добије *ASCII* кôд знака који конвертује, а резултат је *ASCII* кôд конвертованог знака, слика 3.26.



Слика 3.25 Функције за испитивање знакова

Ако је аргумент ASCII код малог слова може се конвертовати у ASCII код истог великог слова:

```
c = toupper(c) <=> if (c>='a' && c<='z')
                       c = c - ('a' - 'A');
                   (97)  (122)   (32)
```

Ако је аргумент ASCII код великог слова може се конвертовати у ASCII код истог малог слова:

```
c = tolower(c) <=> if (c>='A' && c<='Z')
                       c = c + ('a' - 'A');
                   (65)  (90)    (32)
```

Слика 3.26 Функције за конверзију знакова

Стандардна библиотека *math.h*

У библиотеци *math.h* спаковане су математичке функције. Свака од функција као што су *sin(), cos(), tan(),* даје резултат једне тригонометријске функције. Ту су и функције које израчунавају експонент, логаритам за природну основу, логаритам за основу 10, степен, квадратни корен, апсолутну вредност: *exp(), log(), log10(), pow(), sqrt(), fabs()*... Аргументи и резултат набројаних функција реални су бројеви (типа *double*), слика 3.27. Позиви математичких функција у неким случајевима су неопходни у наредбама *C* програма, слика 3.28.

Математичке функције облика:

```
double ime_funkcije(double)
```

sin(x)	sin(x)
cos(x)	cos(x)
tan(x)	tan(x)
exp(x)	e^x
log(x)	ln(x), x>0
log10(x)	$\log_{10}(x)$, x>0
pow(x,y)	x^y
sqrt(x)	$\sqrt{x}$, x≥0
fabs(x)	|x|

Слика 3.27 Математичке функције

математички израз	израз у C програму
$\frac{a + b - \cos(x)}{a - b}$	(a+b-cos(x))/(a-b)
$\frac{(x + y)^2}{2x - y + 1}$	pow((x+y),2)/(2*x-y+1)
$c = \sqrt{a^2 + b^2}$	c=sqrt(pow(a,2)+pow(b,2))
$d = \sqrt{(x1-x2)^2 + (y1-y2)^2}$	d=sqrt(pow((x1-x2),2)+pow((y1-y2),2))

Слика 3.28 Примери примене математичких функција у програму

3.3. Питања

1. Шта све дефинише синтакса програмског језика за сваки свој оператор?
2. Које врсте оператора постоје у *C* језику, у зависности од броја операнада?
3. Да ли заграде могу променити приоритет оператора у програмима?
4. Које значење има аритметички оператор инкремент, а које декремент?
5. Који је разлог примене логичких оператора, у комбинацији са релацијским?
6. У којим случајевима је у програму потребно употребити наредбу *if*?
7. Примену које наредбе може елиминисати условни *C* оператор?
8. Примена којох оператора на нивоу бита омогућава проверу вредности појединих битова у подацима?
9. Примена којох оператора на нивоу бита омогућава сетовање / ресетовање / промену вредности појединих битова у подацима?
10. Код којих типова података се може применити оператор величине типа?
11. У којим случајевима се може користити оператор конверзије типа?
12. Како раде функције *getchar()* и *putchar()*?
13. Помоћу којих се стандардних функција из библиотеке *ctype.h* може проверити да ли је знак: слово / децимална цифра / штампајући знак...
14. Помоћу којих се стандардних функција из библиотеке *ctype.h* може конвертовати мало слово у исто велико / велико слово у исто мало?
15. Шта је све потребно знати у вези стандардне функције из одговарајуће библиотеке, пре него што се предвиди њено коришћење?

4. Наредбе циклуса

Кратак преглед лекције

Једна наредба за контролу тока програма већ је објашњена код релацијских и логичких оператора, то је наредба једноструке селекције, у свом основном или проширеном облику. Значајну групу наредби за контролу тока чине наредбе циклуса (или петљи): *for, while* и *do..while*. Како и сам назив ове групе наредби говори, оне омогућавају да се део програма извршава у циклусу, већи број пута. Обавезно је дефинисање услова који одређује прекид извршавања циклуса. Овај услов садржи један *C* израз. Услов је испуњен ако је израз који је написан у њему логички тачан.

4.1. Бројачки циклус

У програмима су често заступљени циклуси који представљају понављања појединих делова програма, увек коначан број пута. На почетку пројектовања једног циклуса у програму потребно је дефинисати услов под којим се тај циклус прекида. Ако услов има везе са бројањем (променом неког податка од почетне до крајње вредности, са кораком промене) циклус је бројачки.

Понављање неке наредбе, или блока наредби, у програму често прати промену вредности неког податка. Да би програмски циклус пратио ове промене променљива се прогласи бројачем (бројачком променљивом) и дефинишу се за ову променљиву следеће вредности: почетна, крајња и корак промене. У овом случају могуће је предвидети по једно понављање наредбе или блока наредби, у сваком кораку промене бројача, од почетне до крајње вредности истог бројача.

Ток извршавања наредбе *for*

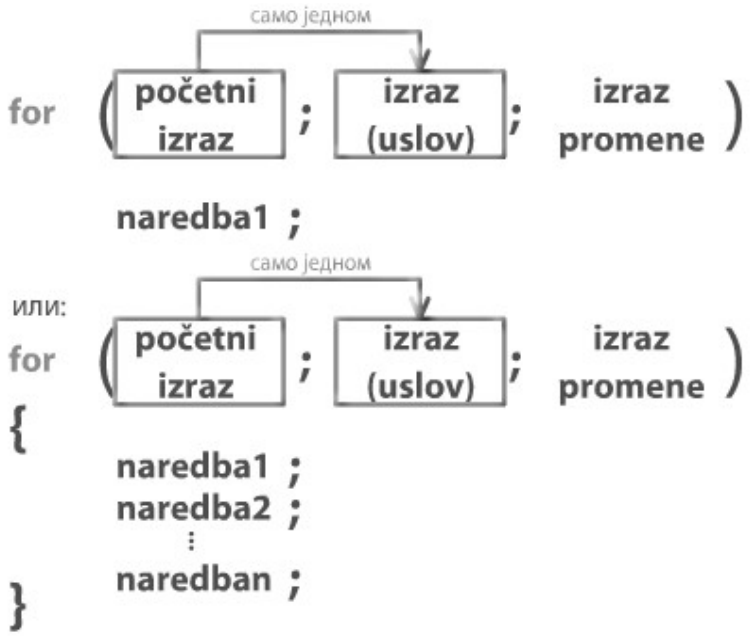
За реализацију бројачког циклуса у *C* програму користи се наредба *for*. Следи опис тока ове наредбе. Само једном, на почетку наредбе, иницијализује се

бројач (почетни израз циклуса), а даље се проверава испуњеност услова циклуса (може се проверавати да ли бројач још није достигао крајњу вредност). У сваком пролазу у коме је услов испуњен извршава се тело циклуса (то је наредба или блок наредби) и мења се вредност бројача (израз промене). У првом пролазу у коме услов није испуњен завршава се циклус, слика 4.1.

Слика 4.1 Ток извршавања наредбе *for*

Структура наредбе *for*

Код наредбе *for* сва три израза (почетни, услов и промена) пишу се у продужетку службене речи *for*, између заграда и одвојени су међусобно знаком тачка-зарез (;), слика 4.2. Сваки од ових израза може бити сложен. Израз услов може садржати више услова повезаних логичким операторима. У почетном изразу и у изразу промене може бити више израза и онда се они одвајају помоћу знака зарез. После израза пише се само тело циклуса (само једна наредба или блок наредби уоквирен витичастим заградама).



Слика 4.2 Структура наредбе *for*

Уклапање наредбе *for*

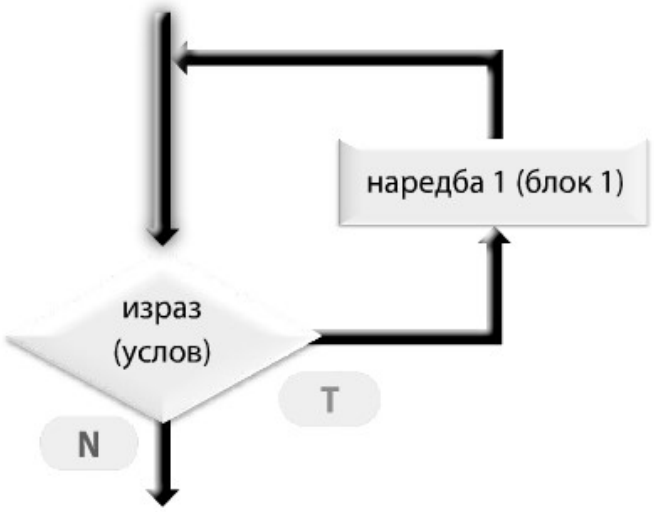
Једна наредба *for* некада није довољна у програму, кад програмски задататак захтева две димензије (редови и колоне матрице, редови и речи текста...). Овакав задатак решавају уклопљене наредбе *for* (једна унутар друге). У том случају свака наредба *for* има свој бројач. У сваком пролазу спољне наредбе *for* (за вредности првог бројача, од почетне до крајње) реализују се сви пролази унутрашње наредбе *for* (други бројач добија све своје могуће вредности).

4.2. Циклус са излазом на врху

За реализацију циклуса у С програму нису увек неопходни бројачи. Када је потребно да се испуни неки услов у програму за почетак и за трајање циклуса решење је излаз на врху. Испуњеност/неиспуњеност услова за почетак/крај овог циклуса могу бити добијени са неког улазног уређаја или из наредби истог програма. (На пример: читање са тастатуре било ког знака различитог од знака за прелаз у нови ред представља услов за почетак а уједно и услов за трајање циклуса. Кад се прочита знак за прелаз у нови ред, циклус се завршава.)

Ток извршавања наредбе *while*

Када је у С програму потребно да се услов за извршавање циклуса провери на самом почетку користи се наредба *while*. Ако услов на почетку није испуњен, то ће бити крај наредбе *while*. У супротном, у сваком пролазу у коме је услов испуњен, извршава се тело циклуса (наредба или блок наредби). У пролазу у коме услов није испуњен завршава се циклус, слика 4.3. Дакле, код ове наредбе број пролаза кроз циклус није обавезно унапред познат, као што је то случај код наредбе *for*.



Слика 4.3 Ток извршавања наредбе *while*

Структура наредбе *while*

Код наредбе *while* заступљен је само један израз услов који се пише у продужетку службене речи *while*, између заграда, слика 4.4. Израз услов може садржати више услова повезаних логичким операторима. После израза пише се само тело циклуса (само једна наредба или блок наредби уоквирен витичастим заградама). И наредба *while* може бити уклопљена са осталим наредбама за контролу тока, уз пажљиво писање услова и осталих израза у свакој од уклопљених наредби.

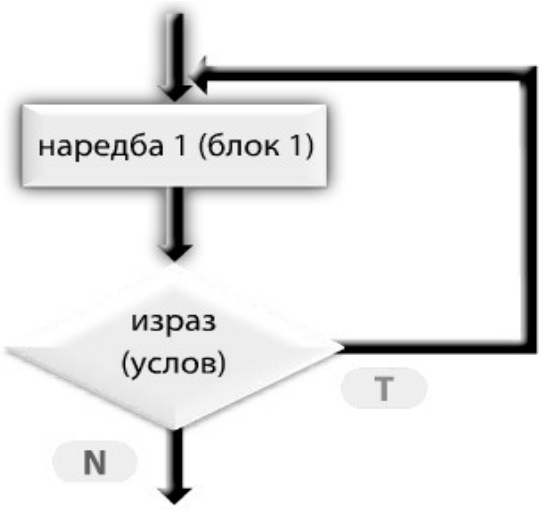
Слика 4.4 Структура наредбе *while*

4.3. Циклус са излазом у дну

Када је у програму предвиђено извршавање наредбе или блока наредби по први пут обавезно, а понављање ако је испуњен одређени услов, решење је излаз на дну. Испуњеност/неиспуњеност услова за сваки пролаз кроз циклус, изузев за први, могу бити добијени са улазног уређаја или из наредби истог програма. (На пример: читање комплета бројева са тастатуре је обавезан део наредбе, услов за понављање овог читања је да нема нула међу учитаним бројевима. Кад буде прочитан комплет бројева са нулом, циклус се завршава.)

Ток извршавања наредбе *do..while*

Када је у *C* програму потребно да се тело циклуса (наредба или блок наредби) изврши једном без икаквог услова, а да се услов за понављање провери на самом крају, користи се наредба *do..while*. У првом пролазу извршава се једном тело циклуса. Ако услов на крају није испуњен, то ће бити крај наредбе *do..while*. У супротном тело циклуса извршава се још једном и услов се проверава на крају циклуса. У пролазу после кога услов није испуњен завршава се циклус, слика 4.5.



Слика 4.5 Ток извршавања наредбе *do..while*

Структура наредбе *do..while*

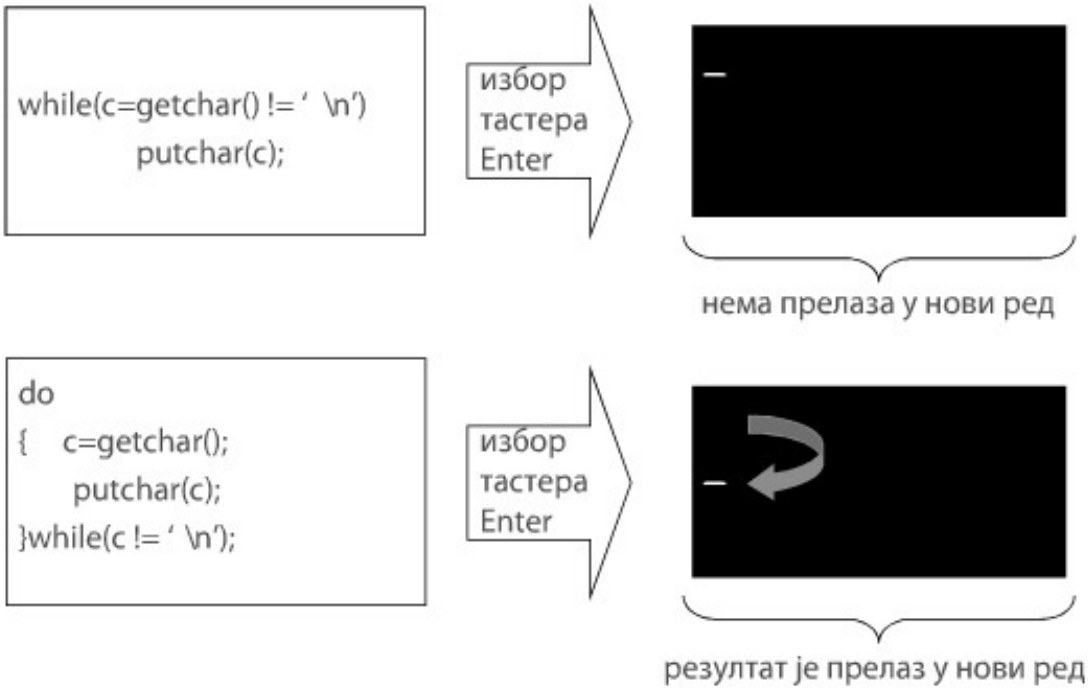
Код наредбе *do..while* пише се прво службена реч *do*, следи тело циклуса (само једна наредба или блок наредби уоквирен витичастим заградама). На крају је службена реч *while* и израз услов који се пише у продужетку ове службене речи, између заграда, слика 4.6. Дакле, тело циклуса уоквирује се речима *do* и *while.* Израз услов наредбе *do..while* је на крају и због тога се наредба обавезно завршава знаком тачка-зарез. И код ове наредбе могуће је уклапање.

Слика 4.6 Структура наредбе *do..while*

Разлика између наредби *while* и *do..while*

Кад један део програма треба да се извршава у циклусу, а нема потребе за бројачем, биће предвиђена једна од две описане наредбе: *while* и *do..while*. Ако

и за први пролаз кроз циклус треба да буде испуњен одређени услов, треба предвидети наредбу *while* која испитује услов на врху. За део програма који треба да се изврши бар једном, без икаквих услова, а да буде испуњен одређени услов за свако понављање истог дела програма, одговарајућа је наредба *do..while* која испитује услов на дну, слика 4.7.



Слика 4.7 Разлика између наредби *while* и *do..while*

4.4. Питања

1. Шта представља циклус у програму?
2. Шта је неопходно дефинисати на почетку пројектовања било које наредбе циклуса у програму?
3. Које наредбе циклуса испитују услов на врху, а које у дну наредбе?
4. У којим случајевима се користи наредба *for* и која је њена структура?
5. Како се може описати ток извршавања наредбе *for*?
6. Како се може описати ток извршавања наредбе *while*?
7. Како се може описати ток извршавања наредбе *do..while*?
8. Који од три израза наредбе *for* (почетни израз / израз услов / израз промене) може бити сложен (садржати у себи више израза)?
9. Колико пута се у току *for* циклуса може извршити почетни израз?
10. Да ли корак промене бројача у for наредби може бити само позитиван / само негативан / позитиван или негативан?
11. Која врста израза може бити написана као услов наредбе *while*?
12. У којим случајевима се користи наредба *while*, а у којим наредба *do..while*?
13. Који је минимални број итерација код *while*, а који код *do..while* циклуса?
14. Унутар које наредбе се може уклопити било која од постојећих наредби циклуса?
15. Да ли наредба циклуса може бити написана унутар наредбе селекције и да ли може бити и обрнуто?

5. Наредбе скокова и вишеструке селекције

Кратак преглед лекције

Поред наредби селекције и циклуса, у *C* програмима заступљене су и наредбе скокова: *break* и *continue* (наредба *goto* не препоручује се у језику *C*). Ова група наредби користи се у наредбама циклуса: за крај циклуса или за крај једног пролаза кроз циклус. Наредба скока *break* користи се и у наредби вишеструке селекције *switch.* Наредба *switch* може бити замена за више уклопљених наредби једноструке селекције, а помоћу наредбе *break* остварује се крај било које селекције од предвиђених вишеструких.

5.1. Наредба скока из петље

Извршавање било које наредбе скока у програму значи да се после ове наредбе не извршава наредба која је написана прва следећа по реду, већ нека друга наредба у истом програму. Сама наредба скока говори о томе где је написана та друга наредба. У програмима на језику *C* заступљени су само условни скокови, тако да је неопходно написати и услов под којим се свака од ових наредби извршава (то је најчешће услов наредбе *if* који и овде може бити произвољно сложен логички израз).

Наредба скока *break* може бити написана унутар било које наредбе циклуса: *for, while* или *do..while*. Писање наредбе *break* има смисла само ако је дефинисан услов под којим треба да се изврши ова наредба скока. У продужетку службене речи *if* пише се један израз услов, између заграда. Ако је потребно да више услова буде испуњено за крај циклуса, израз услов и овде може садржати више њих повезаних логичким операторима. После израза услова пише се наредба скока *break;* слика 5.1

```
for (brojac=poc; brojac<=kraj; brojac++)
{       ...
        if(izraz1)
                break;
        ...
}
```

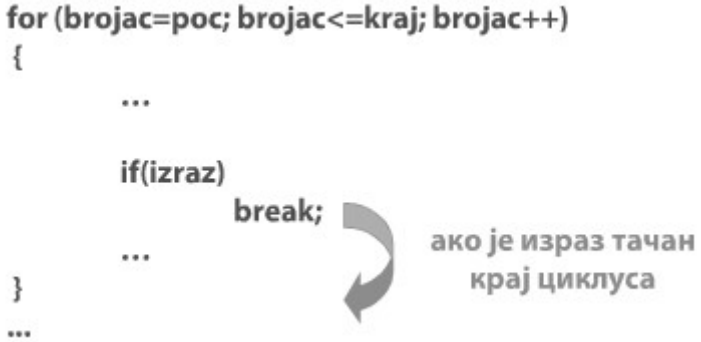
```
while (izraz)
{       ...
        if(izraz2)
                break;
        ...
}
```

```
do
{       ...
        if(izraz3)
                break;
        ...
} while (izraz);
```

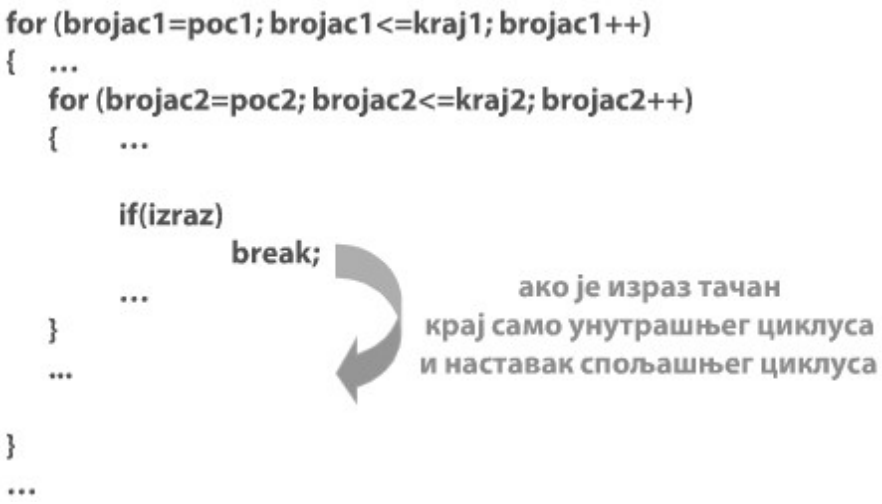
Слика 5.1 Писање наредбе *break* унутар наредбе циклуса

Извршавање наредбе *break*

Наредба *break* користи се за превремени крај извршавања циклуса. Наредбе тела циклуса извршавају се до ове наредбе скока, извршавање наредбе скока значи прескакање свих даље наредби истог циклуса и скок на наредбу која је написана прва после наредбе циклуса, слика 5.2. Наредба *break* може бити уклопљена (у циклусу који је унутар другог) и онда значи прекид само најближег циклуса (који је непосредно обухвата) не и осталих спољних (наредба *break* "пробија" само један ниво циклуса), слика 5.3.



Слика 5.2 Извршавање наредбе *break* написане унутар наредбе циклуса



Слика 5.3 Извршавање наредбе *break* написане
унутар угњеждене наредбе циклуса

Скок из наредбе циклуса са стално тачним изразом (*while(1)*)

Некад у програму не постоји дефинисан услов за излаз, на врху или на дну, из неког циклуса (нема написаног услова наредбе *while* или услова наредбе *do..while*). Најчешће се користи наредба *while* тако што се напише *while(1)* (на месту израза вредност логичке јединице која значи увек испуњен услов), а међу наредбама тела циклуса наредба *break*, слика 5.4. Циклус *while(1)* био би бесконачан без наредбе *break* која се у овом случају обавезно користи да обезбеди крај циклуса.

```
while ( 1 )
{       ...
        if(izraz2)
                break;
        ...
}
```

```
do
{       ...
        if(izraz3)
                break;
        ...
} while ( 1 );
```

Слика 5.4 Крај циклуса са стално тачним изразом

5.2. Наредба скока из итерације петље

Код писања наредбе скока *continue* важе слична правила као и код претходно описане наредбе. Наредба *continue* може бити написана само унутар наредбе циклуса: *for, while* или *do..while*. Писање наредбе *continue* има смисла само ако је дефинисан услов под којим треба да се изврши ова наредба скока. У продужетку службене речи *if* пише се један израз услов, између заграда. И овде израз услов може бити произвољне сложености. После израза услова пише се наредба скока *continue;* слика 5.5.

```
for (brojac=poc; brojac<=kraj; brojac++)
{       ...
        if(izraz1)
                continue;
        ...
}
```

```
while (izraz)
{       ...
        if(izraz2)
                continue;
        ...
}
```

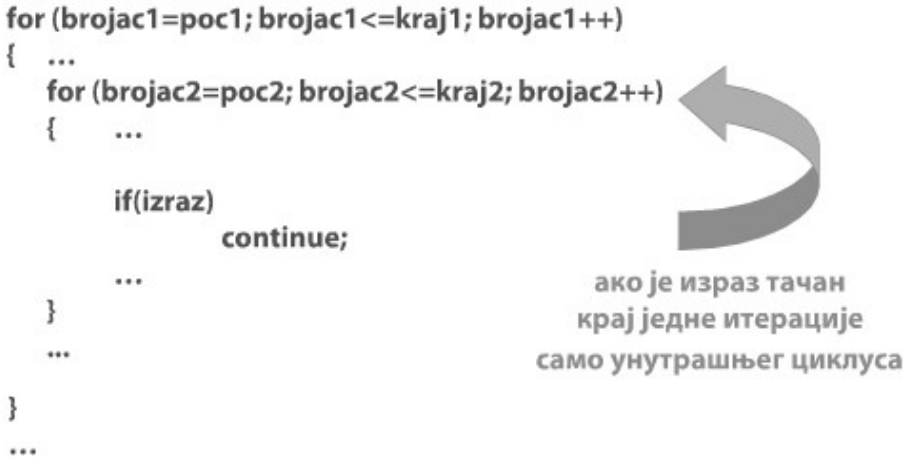
```
do
{       ...
        if(izraz3)
                continue;
        ...
} while (izraz);
```

Слика 5.5 Писање наредбе *continue* унутар наредбе циклуса

Извршавање наредбе *continue*

Наредба *continue* користи се за превремени крај једног пролаза (итерације) кроз циклус. Наредбе тела циклуса извршавају се до ове наредбе скока, извршавање наредбе скока значи прескакање свих даље наредби исте итерације циклуса и скок на почетак његове следеће итерације, слика 5.6. Наредба *continue* може бити уклопљена (у циклусу који је унутар другог) и онда значи прекид само текуће итерације од најближег циклуса (који је непосредно обухвата) а не и осталих спољних циклуса, слика 5.7.

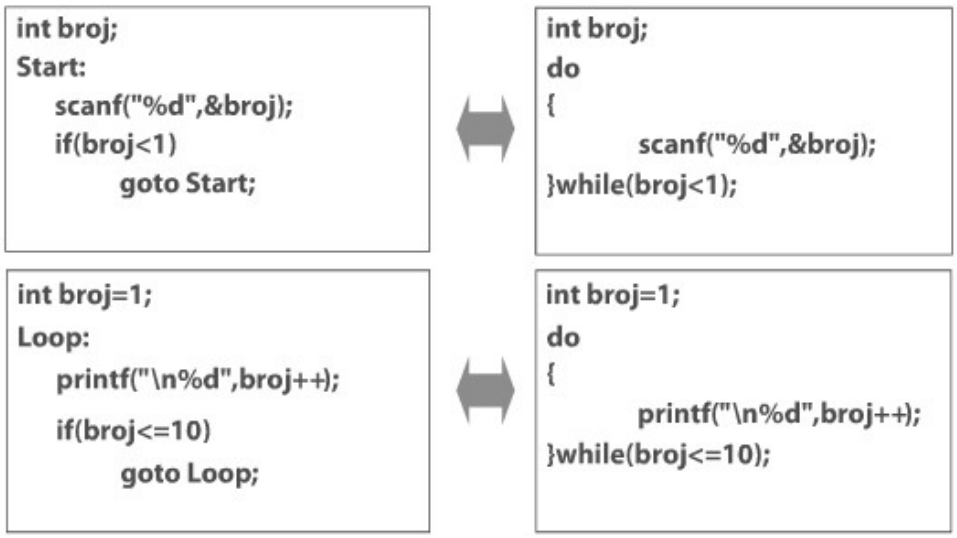
Слика 5.6 Извршавање наредбе *continue* написане унутар наредбе циклуса



Слика 5.7 Извршавање наредбе *continue* написане унутар угњеждене наредбе циклуса

Замена наредбе *goto* другим наредбама

Наредба *goto* значи скок на одредиште у програму, које се дефинише у самој наредби скока. Наредба *goto* је наредба која је заостала у језику *C* од претходних програмских језика и може се наћи само у уџбеницима. Решење било ког програмског задатка које је на језицима који су претходили било реализовано помоћу наредбе *goto*, у C програму може бити реализовано помоћу неке друге наредбе, слика 5.8, што поједностављује изворни кôд и смањује могућност грешке код уклопљених наредби.



Слика 5.8 Замена наредбе *goto* другим наредбама

5.3. Наредба вишеструке селекције

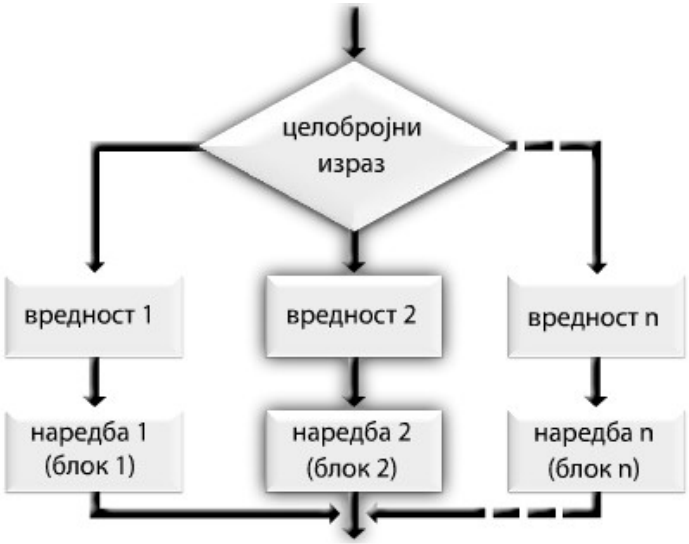
Наредба *switch* је наредба вишеструке селекције. Ова наредба омогућава извршавање једне наредбе (или блока наредби) од више предвиђених наредби (или блокова наредби) у зависности од вредности целобројног израза који тестира. За коришћење наредбе *switch* неопходно је дефинисати скуп могућих константних целобројних вредности, предвидети случај избора било које од ових вредности, као и неке непредвиђене вредности. Наредба *switch* може се користити као замена за низ уклопљених наредби *if*, слика 5.9.

```
if (celobrojni_izraz == vrednost1)
  naredba1;
else if (celobrojni_izraz == vrednost2)
       naredba2;
.
.
.
else
  naredbaN;
```

Слика 5.9 Више угњеждених *if* наредби које може заменити једна *switch* наредба

Ток извршавања наредбе *switch*

Наредба *switch* тестира вредност свог целобројног израза да би одрадила скок на одговарајући свој случај. Ако израз има једну од дефинисаних константних целобројних вредности извршава се једна наредба (или блок наредби) тог случаја, слика 5.10. Може бити произвољан број случајева и један од њих треба да буде подразумевани. Наредба *switch* обезбеђује скок на овај подразумевани случај ако вредност целобројног израза није ниједна од предвиђених осталим случајевима наредбе.



Слика 5.10 Ток извршавања наредбе *switch*

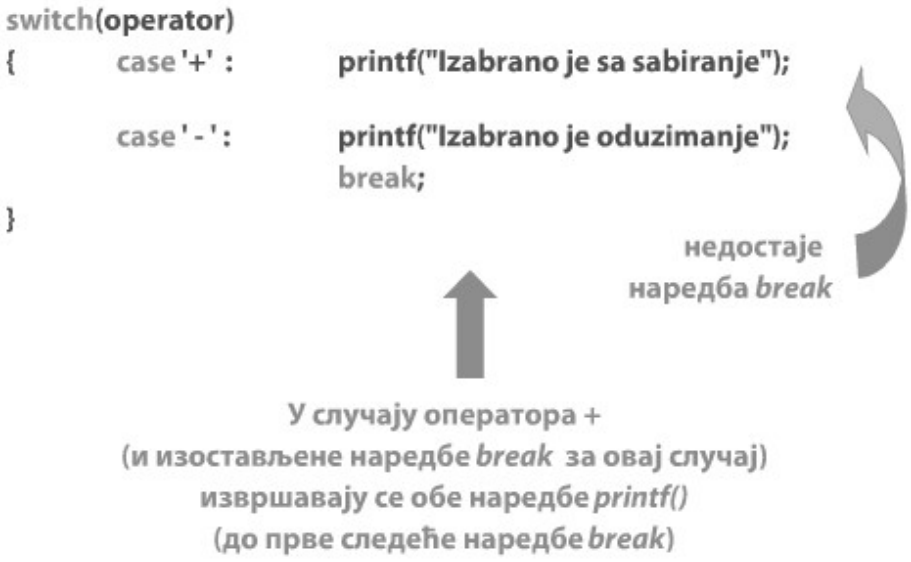
Структура наредбе *switch*

Наредба *switch* почиње службеном речи *switch*, следи целобројни израз написан између заграда, а онда између витичастих заграда написани сви случајеви наредбе. Сваки случај, издвојен је помоћу службене речи *case*, целобројне константе која одговара том случају и садржи блок наредби који се извршава за изабран случај. На крају случаја је наредба *break*. Последњи треба да буде подразумевани случај *default*, слика 5.11. За блок наредби сваког случаја није обавезан оквир од витичастих заграда.

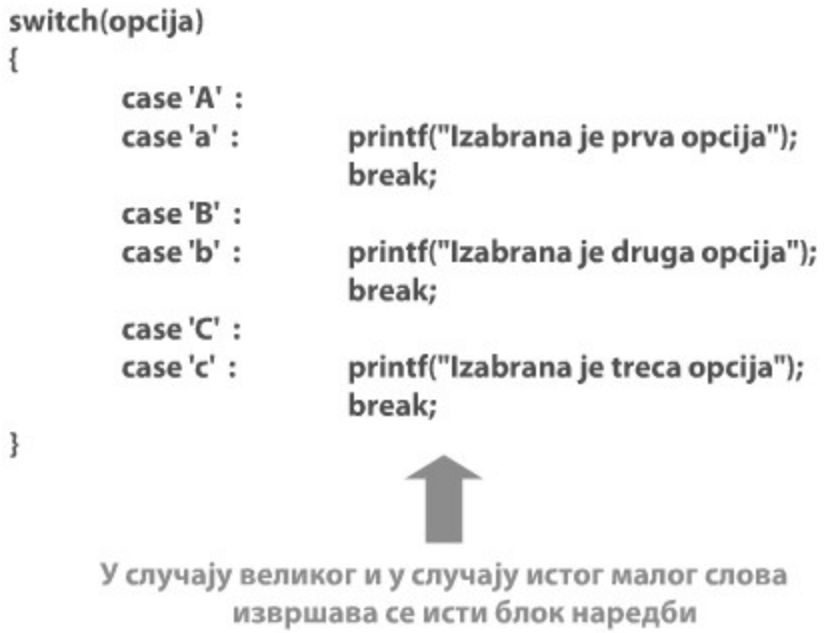
Слика 5.11 Структура наредбе *switch*

Примена наредбе *break* у наредби *switch*

Наредба *break* обавезна је на крају сваког случаја да би обезбедила крај наредбе *switch.* Ако се на крају неког случаја изостави наредба *break* извршавају се даље наредбе (следећег случаја, ако овај постоји), што може условити грешку, слика 5.12. Описано "пропадање из случаја у случај" корисно је једино онда када је потребно извршавање истог блока наредби у више различитих случајева наредбе *switch*, слика 5.13. Подразумевани случај *default* обично се пише на крају и онда није неопходно писање његове наредбе *break.*



Слика 5.12 Изостанак наредбе *break* код случаја наредбе *switch*



Слика 5.13 Извршавање истог блока наредби у више случајева наредбе *switch*

5.4. Питања

1. Шта значи наредба скока у програму?
2. Да ли су у *C* програмима заступљени условни или безусловни скокови?
3. На којим местима у програму може бити написана наредба скока?
4. У којим се све случајевима може користити наредба *break*?
5. Да ли је за излаз из циклуса помоћу *break* наредбе неопходно постојање *if* наредбе за проверу услова?
6. Да ли наредба *break* у сваком случају значи дефинитиван излаз из циклуса?
7. У којим се све случајевима може користити наредба *continue*?
8. Да ли је за излаз из итерације циклуса помоћу *continue* наредбе неопходно постојање *if* наредбе за проверу услова?
9. Како изгледа структура наредбе *switch*?
10. Како ради наредба *switch*?
11. Коју врсту израза може тестирати наредба *switch*?
12. Како се наредба *break* примењује код наредбе *switch*?

13. Да ли број случајева код наредбе *switch* може бити произвољан и да ли је неопходан подразумевани случај?
14. Како се остварује "пропадање" из случаја у случај код наредбе *switch*?
15. Које наредба може бити замењена помоћу наредбе *switch*?

6. Нумерички низови

Кратак преглед лекције

Коришћење само основних типова промењљивих у програмима не би било практично. За рад са великим бројем података међусобно истог типа, да се не би именовала свака промењљива појединачно, уводи се низ промењљивих. За чување нумеричких и знаковних низова података са улазних уређаја, *C* програм резервише само нумеричке типове низова промењљивих. Низови могу бити једнодимензионални и вишедимензионални. Без обзира на број димензија за сваки низ у програму пишу се: декларација и опционо иницијализација.

6.1. Основно о низовима

Низ се сматра изведеним типом податка у језику *C*. У меморији низ представља континуално уређење промењљивих (једна до друге промењљиве) међусобно истог типа. У програму се за поједине промењљиве у једном низу користе: заједнички тип, заједничко име а међусобно различити индекси, да би се разликовале једна од друге. Сви наведени појмови биће даље објашњени. Промењљиве једног низа груписане су у њему због једноставнијег приступа, али свакој од њих, ако је потребно, може се приступити и појединачно.

Низови и показивачи

Приступ појединим промењљивим у једном низу може бити остварен на два начина: директно и индиректно. Директан (непосредан) приступ значи приступ промењљивој у низу помоћу имена низа и додељеног индекса (броја). Други, индиректан приступ захтева чување адресе промењљиве (то је показивач) и приступ истој промењљивој помоћу сачуване адресе (помоћу показивача). На почетку биће објашњене примене низова које не захтевају познавање показивача, а у следећим лекцијама биће примера са њиховом применом.

6.2. Једнодимензионални низови

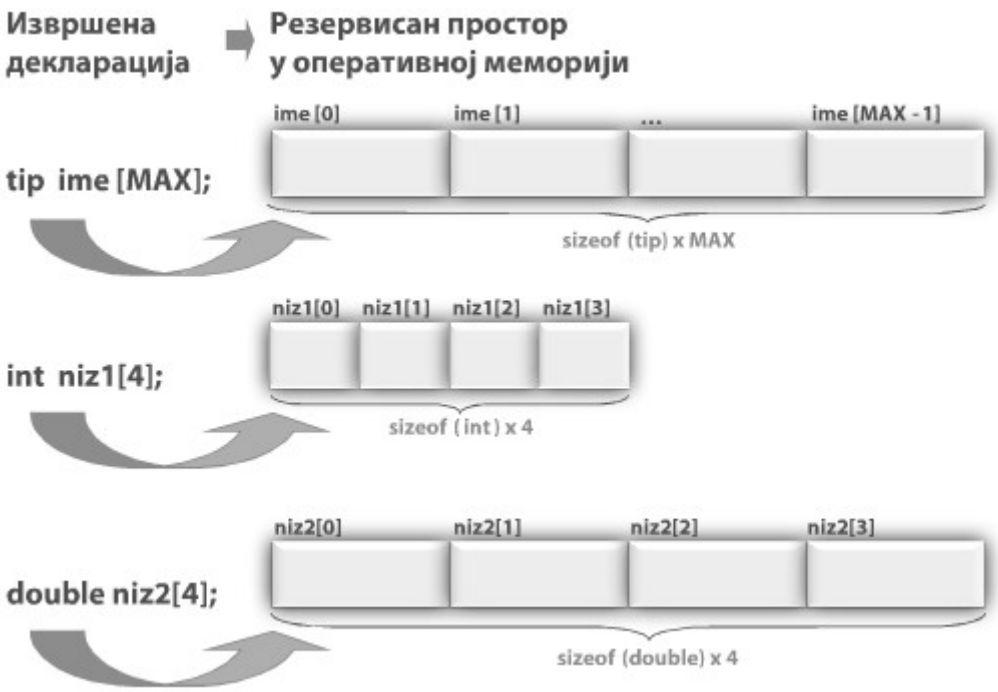
Име низа је заједничко за све његове променљиве и додељује се по правилима за идентификаторе (исто као и код доделе имена променљивим). Име једног низа може се користити у програму као константна адреса првог елемента низа (код индиректног приступа низу). Тип низа такође је заједнички за све његове променљиве и може бити било који основни *C* тип *(char, int, float, double…)* или тип изведен за потребе конкретног *C* програма (као на пример низ показивача са сачуваним адресама). Све наведено важи и за вишедимензионалне низове.

Елементи једнодимензионалног низа и њихови индекси

Свака променљива у низу представља један елемент тог низа. Индекс елемента у једнодимензионалном низу је број који је додељен свакој његовој променљивој и пише се између угластих заграда [ ]. Сваком елементу низа може се приступити као и појединачној променљивој, само што се уместо имена променљиве пишу: име низа и између угластих заграда индекс по коме се зна о ком елементу се ради. Над једним елементом низа могу се применити сви C оператори који су применљиви над променљивом истог типа.

Декларација једнодимензионалног низа

Декларација једнодимензионалног низа неопходна је пре његове употребе у програму јер пружа информације о: типу, имену и броју елемената једног низа (дужини низа), слика 6.1. На месту дужине низа може бити само целобројна константа. За дужину низа *n* индекси његових елемената су у опсегу од *0* до *n-1* (први елемент има индекс *0*, а последњи индекс *n-1*). Након извршене наредбе декларације низа у меморији се резервише континуалан простор, чија је величина једнака производу броја бајтова једног елемента низа и дужине низа.



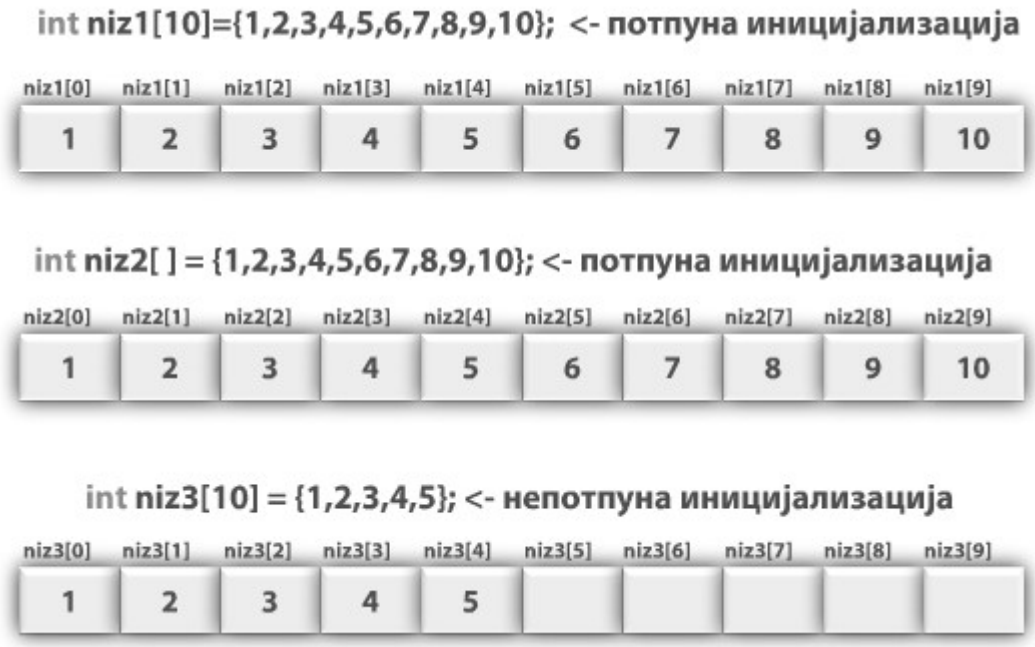
Слика 6.1 Декларација једнодимензионалног низа

Иницијализација једнодимензионалног низа

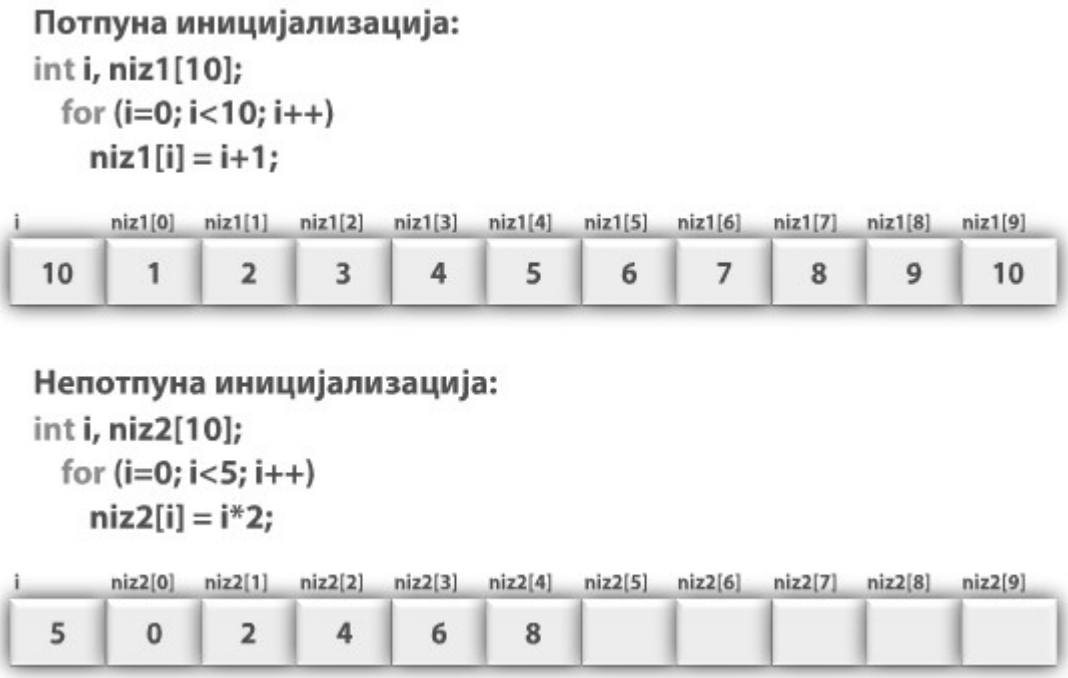
Иницијализација низа (додела почетних вредности елементима) може се реализовати на два начина: један је унутар а други ван декларације истог низа. У првом случају почетне вредности елемената наводе се у продужетку декларације. Само у овом случају није неопходно навођење дужине низа у декларацији јер може бити одређена бројем почетних вредности, слика 6.2. Други начин је иницијализација низа тамо где је потребна у програму, за групу његових елемената у циклусу или за било који елемент појединачно, слика 6.3.

Приступ елементима једнодимензионалног низа

Приступ елементима низа могућ је помоћу њихових индекса (директан приступ) или помоћу показивача који чувају адресе елемената (индиректан приступ). На почетку биће коришћен само први начин. Када је у програму потребан приступ неком елементу низа на овај начин, на месту индекса (између угластих заграда) неопходно је написати одговарајући целобројни израз. Вредност овог целобројног израза треба да буде у границама низа (за дужину низа n у опсегу од 0 до n-1), слика 6.4.



Слика 6.2 Иницијализација једнодимензионалног низа унутар декларације



Слика 6.3 Иницијализација једнодимензионалног низа ван декларације

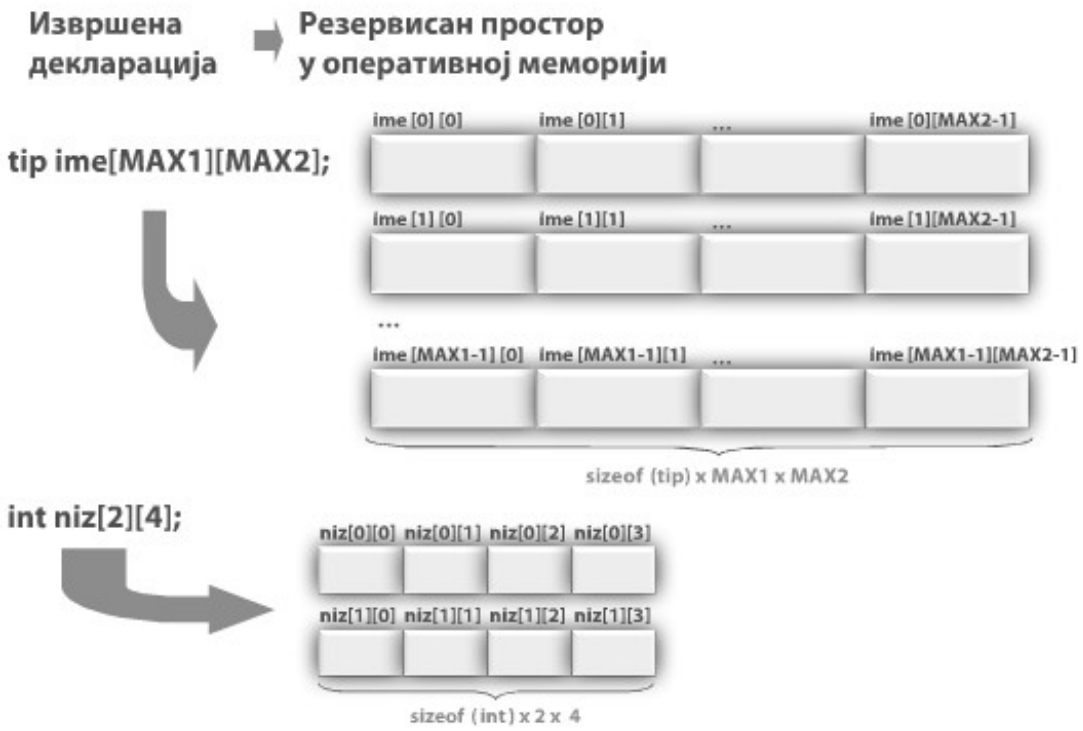
Слика 6.4 Иницијализација једнодимензионалног низа ван декларације

6.3. Вишедимензионални низови

Вишедимензионалан низ је низ низова, то је низ чији је сваки елемент такође низ. Ако овај низ садржи на пример низове променљивих основног типа низ је дводимензионалан. Ако низ садржи низове низова онда је то тродимензионалан низ... Следи објашњење дводимензионалних низова а на основу тога је могућа анализа низова и са више димензија (чији број није ограничен). Дводимензионалан низ има две индексне листе: први је индекс низа у низу низова, а други је индекс елемента у сваком од низова.

Декларација дводимензионалног низа

Декларација дводимензионалног низа пружа информације о: типу, имену, броју низова и о броју елемената сваког низа, слика 6.5. На местима броја низова и броја елемената сваког низа могу бити само целобројне константе. За број низова *n* индекси његових елемената низова су у опсегу од *0* до *n-1*, а за број елемената сваког низа *m* индекси елемената сваког низа су у опсегу од *0* до *m-1*. Након извршене наредбе декларације дводимензионалног низа у меморији се резервише континуалан простор чија је величина једнака производу броја бајтова једног елемента и броја елемената сваког низа и броја низова.



Слика 6.5 Декларација дводимензионалног низа

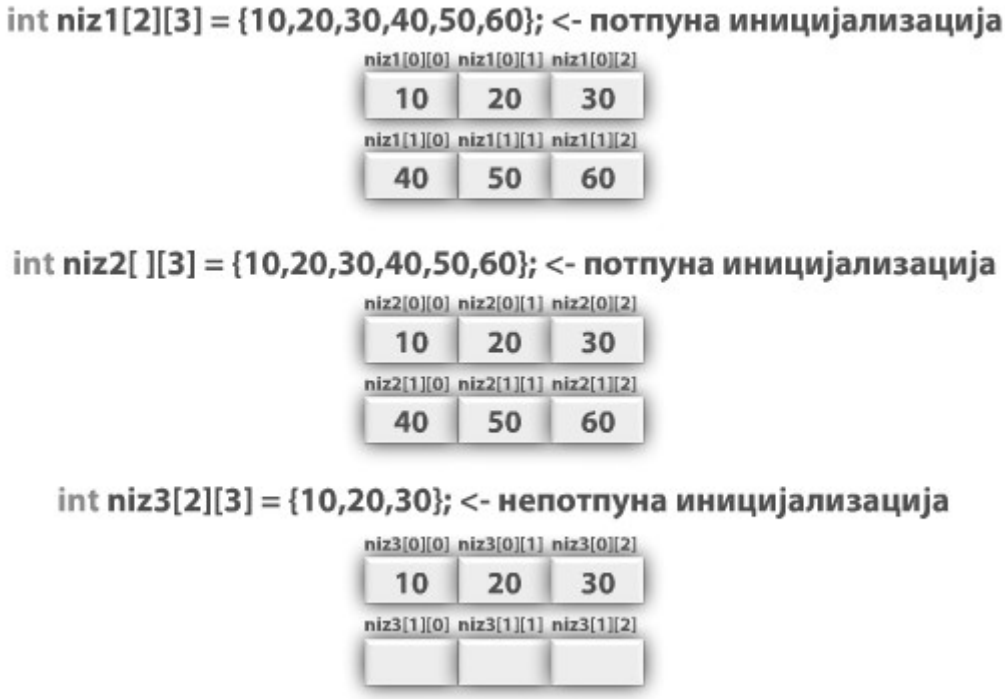
Иницијализација дводимензионалног низа

Иницијализација дводимензионалног низа може се реализовати на два начина, као и код једнодимензионалних: један је унутар а други ван декларације. У првом случају почетне вредности елемената наводе се у продужетку декларације, најчешће груписане по низовима. У овом случају није неопходан број низова у декларацији јер може бити одређена бројем почетних вредности, слика 6.6. Други начин је иницијализација низа низова даље у програму, када се може реализовати помоћу циклуса у циклусу, слика 6.7.

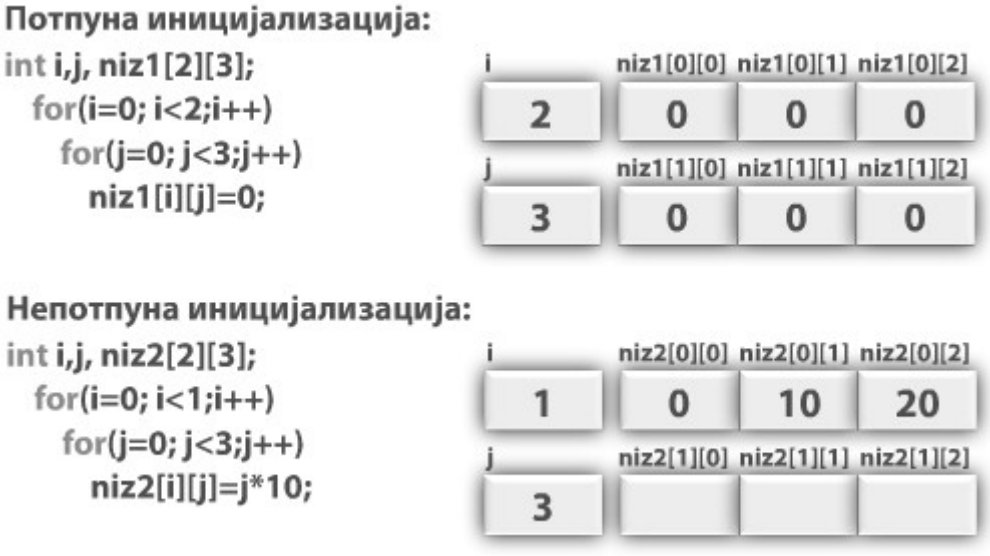
Приступ елементима дводимензионалног низа

Директан приступ елементима низа низова могућ је помоћу индекса. На месту индекса (у једној и другој индексној листи) неопходно је написати одговарајуће целобројне изразе. Вредности ових целобројних израза треба да буду у границама низа (за број низова *n* први индекс у опсегу од 0 до *n*-1, а за дужину сваког низа *m* други индекс у опсегу од 0 до *m-1*), слика 6.8. И код овог низа

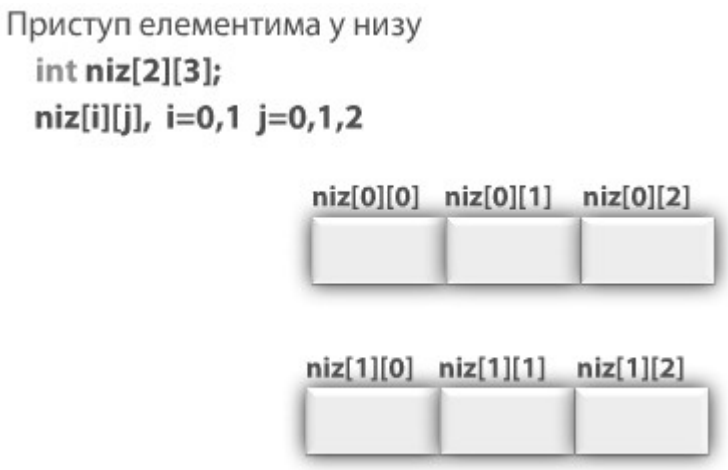
могућ је приступ свим елементима редом сваког од низова (помоћу циклуса у циклусу), као и сваком елементу појединачно.



Слика 6.6 Иницијализација дводимензионалног низа унутар декларације



Слика 6.7 Иницијализација дводимензионалног низа ван декларације



Слика 6.8 Приступ елементима дводимензионалног низа

6.4. Питања

1. Који је разлог увођења низова у програму?
2. Шта представља низ у програму?
3. Које врсте приступа низовима постоје?
4. Шта је заједничко, а шта различито појединим елементима истог низа?
5. Ког типа може бити елемент у низу?
6. Ког типа може бити дефинисана вредност дужине низа при његовој декларацији, а ког типа вредност индекса било ког елемента у низу?
7. У којим случајевима изузетно може бити изостављена дужина низа при његовој декларацији?
8. Како се означава индекс елемента у низу?
9. Коју вредност може узимати индекс у низу који је дужине *n*?
10. Да ли декларација низа обавезно аутоматски и иницијализује низ?
11. Како се може написати иницијализација код једнодимензионалног а како код вишедимензионалног низа?
12. Како се може написати приступ елементу једнодимензионалног, а како елементу дводимензионалног низа?
13. Која врста наредби циклуса се најчешће користи код низова?
14. Примена којих оператора је могућа над елементима низа одређеног типа?
15. Да ли се сваки елемент низа може сматрати посебним објектом за програм?

7. Знаковни низови

Кратак преглед лекције

Програми често користе знакове (слова, цифре и остало) као и низове знакова (речи, реченице, редове текста...). За чување текстуалних података *C* програм дефинише само нумеричке типове поменљивих: целобројни тип *char* за чување једног знака а низ типа *char* за чување низа знакова произвољне дужине. Знак *null* (вредност '\0') на крају низа типа *char* означава овај низ као *string*. *String* није *C* тип, то је само синоним за низ знакова са знаком *null* на крају, тако да га могу препознати и користити стандардне *C string* функције.

7.1. Знакови у програмима

У *ASCII* табели, слика 7.1, заступљене су основне групе знакова. Почетак ове табеле (вредности од 0 до 31) као и сам крај (вредност 127) резервисани су за управљачке знакове (знак за хоризонтални табулатор има вредност 9, знак за нов ред има вредност 10...), слика 7.2. Вредност 32 одговара празном знаку, а опсег вредности од 33 до 126 резервисан је за штампајуће знакове: од 48 до 57 за децималне цифре (од 0 до 9), од 65 до 90 за велика слова абецеде (од A до Z), а од 97 до 122 за иста мала слова, слика 7.3.

Чување у меморији и представљање у *C* програмима

Када је из *C* програма потребно резервисати променљиву за чување једног знака онда је то целобројни тип *char* величине *1 B*. У програму је довољно при додели вредности написати сам знак између апострофа (на пример: 'A' или '%') да би у променљивој била сачувана *ASCII* вредност истог знака. На исти начин се *ASCII* вредност додељује и знаковној константи која може бити декларисана и заузети место у меморији, а на сличан начин знаковна константа може бити дефинисана помоћу претпроцесорске директиве као симбол, слика 7.4.

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	NUL	32	20	Space	64	40	@	96	60	`
1	1	SOH	33	21	!	65	41	A	97	61	a
2	2	STX	34	22	"	66	42	B	98	62	b
3	3	ETX	35	23	#	67	43	C	99	63	c
4	4	EOT	36	24	$	68	44	D	100	64	d
5	5	ENQ	37	25	%	69	45	E	101	65	e
6	6	ACK	38	26	&	70	46	F	102	66	f
7	7	BEL	39	27	'	71	47	G	103	67	g
8	8	BS	40	28	(	72	48	H	104	68	h
9	9	TAB	41	29	)	73	49	I	105	69	i
10	A	LF	42	2A	*	74	4A	J	106	6A	j
11	B	VT	43	2B	+	75	4B	K	107	6B	k
12	C	FF	44	2C	,	76	4C	L	108	6C	l
13	D	CR	45	2D	-	77	4D	M	109	6D	m
14	E	SO	46	2E	.	78	4E	N	110	6E	n
15	F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EM	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	|
29	1D	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Слика 7.1 Основна *ASCII* табела

ASCII кôд	Значење нештампајућег знака	Начин представљања у C програмима
8	Повратак за једно место	'\b'
9	Хоризонтални померај	'\t'
10	Прелаз у нов ред	'\n'
11	Вертикални померај	'\v'
12	Прелаз на нову страну	'\f'
13	Повратак на почетак реда	'\r'

Слика 7.2 Поједини управљачки знакови у *ASCII* табели

ASCII кôд	Штампајући знак	Начин представљања у C програмима
32		' '
33	!	'!'
48	0	'0'
57	9	'9'
65	A	'A'
97	a	'a'

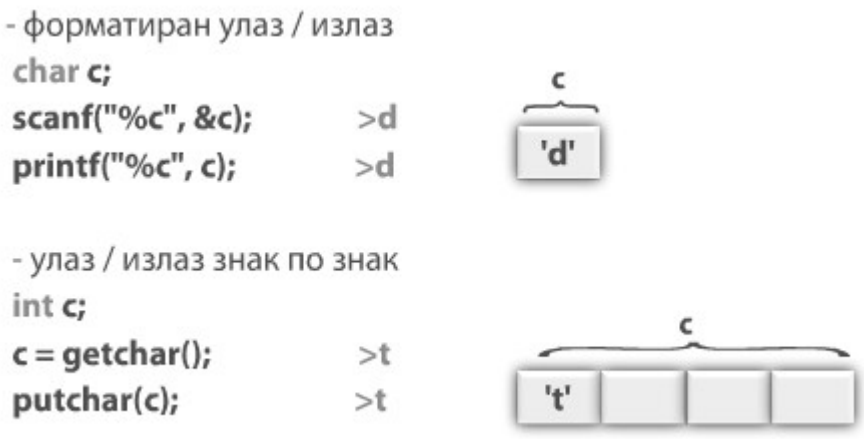
Слика 7.3 Поједини штампајући знакови у *ASCII* табели

Слика 7.4 Коришћење промењљивих и константи за знакове

Функције за улаз / излаз знак по знак у *C* програмима

Улаз са тастатуре и излаз на екран могу бити реализовани знак по знак. Један начин је форматиран улаз / излаз помоћу функција *scanf()* и *printf()* при чему је формат %c. Други начин омогућавају функције *getchar()* и *putchar()*, такође из библиотеке <*stdio.h*>. Функција *getchar()* даје *ASCII* вредност учитаног знака, а функција *putchar()* приказује знак, на основу *ASCII* вредности коју добије у

аргументу, слика 7.5. Ако је, на пример, учитан знак децимална цифра *ASCII* вредности *n*, може се израчунати вредност ове цифре као: *n - '0'*, слика 7.6.



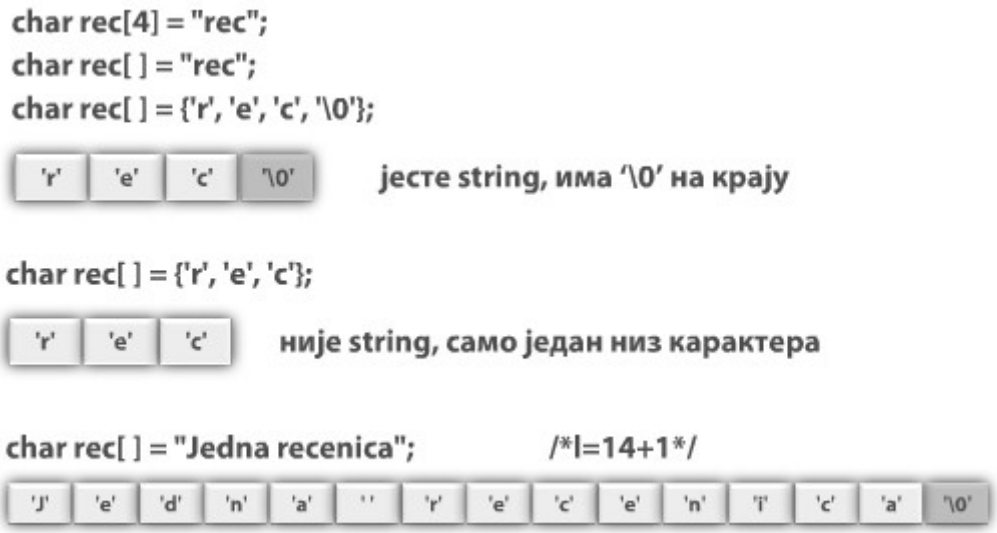
Слика 7.5 Функције за улаз / излаз: форматирани и знак по знак

Слика 7.6 Препознавање слова или цифре у знаку

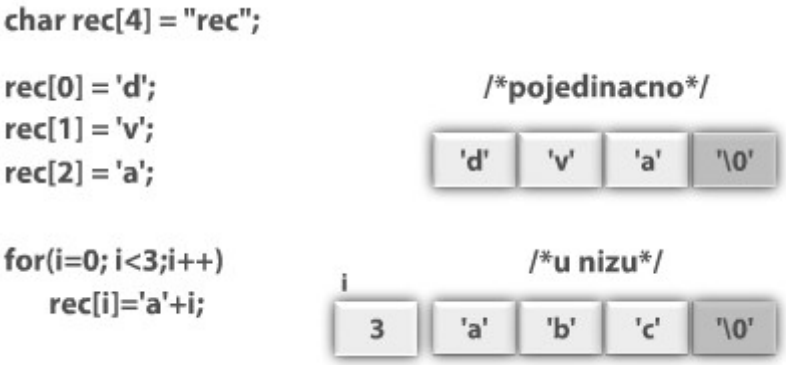
7.2. Знаковни низови и *stringovi*

String је низ карактера који се користи за чување знаковног низа, са знаком *null* на крају (ознака '\0' у програму, вредност 0 у меморији). Знак *null* не види се код

приказа садржаја *stringa,* али је потребно декларисати низ од *n+1* карактера за чување низа од *n* знакова. Код иницијализације *stringa* у декларацији довољно је написати знакове између наводника (на пример: *"rec"*) да би у низу биле сачуване *ASCII* вредности знакова и уписана вредност '\0' на крају, слика 7.7. Из програма је могућ и приступ сваком елементу *stringa* појединачно, слика 7.8.



Слика 7.7 Декларација и иницијализација stringa

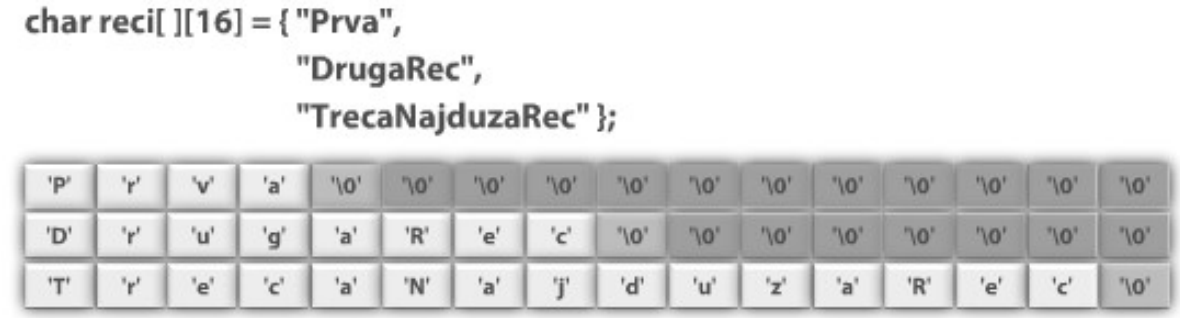


Слика 7.8 Приступ елементима *stringa*

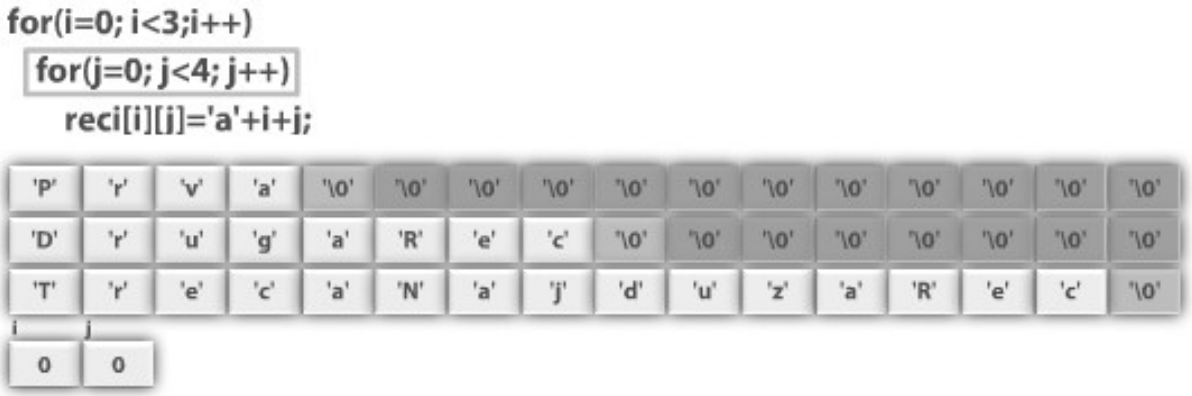
Низови *stringova*

Низ *stringova* је дводимензионалан низ који се користи за чување више знаковних низова. Сваки *string* у низу декларише се на исту дужину, то је дужина најдужег *stringa* у низу. Декларација низа *stringova* има облик декларације дводимензионалног низа типа *char*. При иницијализацији унутар

декларације знаковни низови (сваки између наводника) одвајају се зарезом и уоквирују витичастим заградама, слика 7.9. Из програма је могућ приступ било ком елементу у било ком *stringu* овог низа, слика 7.10.



Слика 7.9 Декларација и иницијализација низа *stringova*



Слика 7.10 Приступ елементима *stringova* у низу

Улаз / излаз код *stringova*

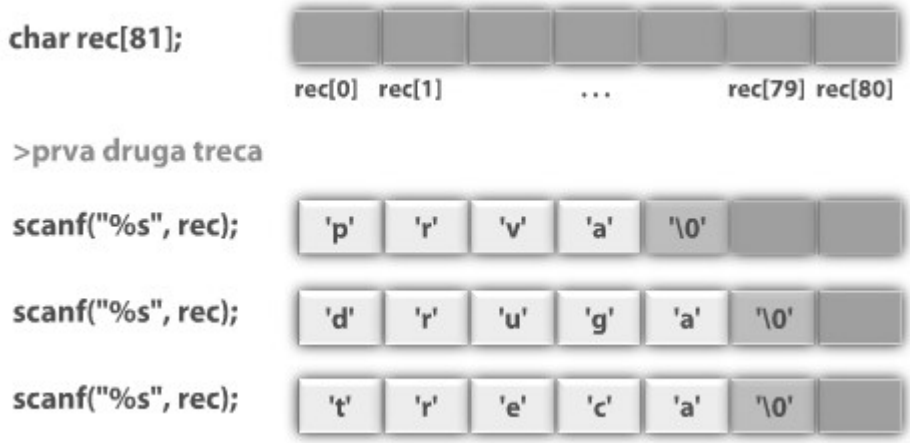
Стандардна библиотека <*stdio.h*> између осталих садржи и функције које се користе за улаз са тастатуре и излаз на екран код *stringova.* Једној групи ових функција припадају функције за форматиран улаз / излаз, *scanf()* и *printf()*, при чему *string* као специфичан низ има своју ознаку у формату *%s*. Функција *scanf()* третира *string* као низ знакова до првог белог знака. Другу групу чине функције за улаз / излаз ред по ред, *gets()* и *puts()*. Функција *gets()* прихвата низ знакова до знака за прелаз у нов ред као један *string*, слика 7.11.

```
char string[81];

- форматиран улаз / излаз
  scanf("%s", string);   читање са тастатуре знаковног низа,
                         задатог до белог знака и упис учитаног
                         знаковног низа у string
  printf("%s", string);  приказ на екрану садржаја stringa

- улаз / излаз ред по ред
  gets(string);          читање са тастатуре знаковног низа до знака
                         за прелаз у нов ред и упис учитаног знаковног
                         низа у string
  puts(string);          приказ на екрану садржаја stringa и прелаз у
                         нов ред на крају
```

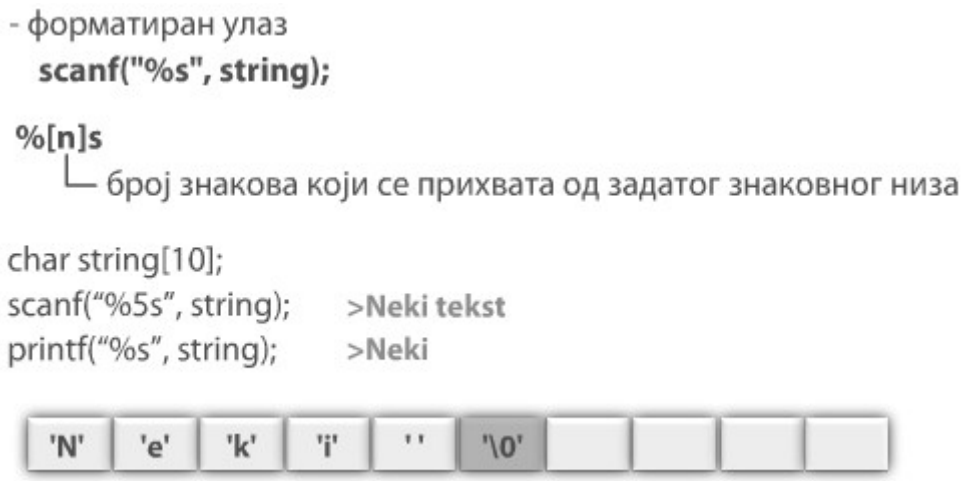
Слика 7.11 Функције за улаз / излаз тастатура / екран код *stringova*

Читање са тастатуре низа знакова

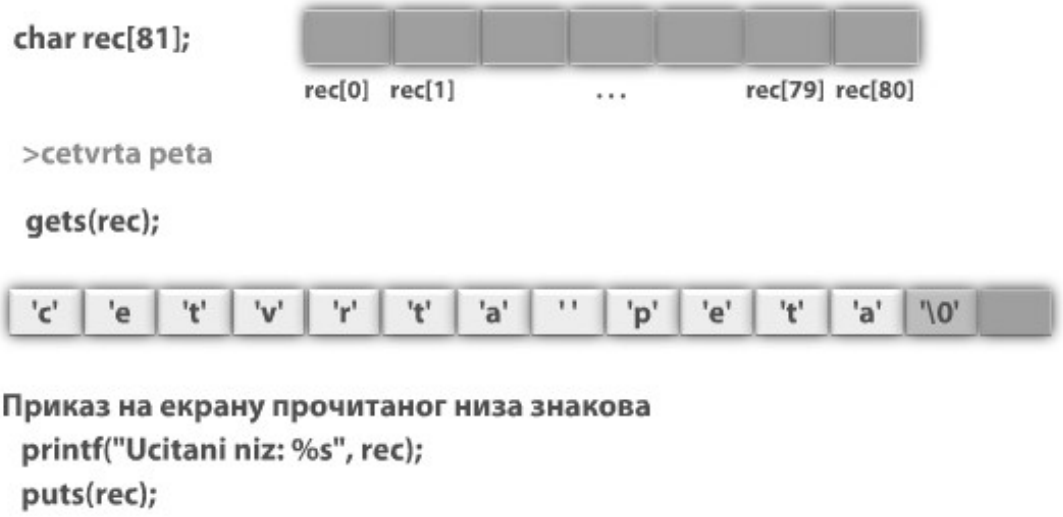
Ако је крај знаковног низа било који бели знак (празан, табулатор или прелаз у нов ред) за читање низа користи се функција *scanf()* са ознаком %s за *string* (изузетак од правила у формату има ознака само за основни тип), слика 7.12. Формат може имати и додатне опције за читање *stringa* на одређен начин, слика 7.13. Ако се као крај знаковног низа третира само прелаз у нов ред онда се користи функција за читање реда *gets()*, која у аргументу добија име *stringa* (то је показивач на *string*) у који треба да буде уписан ред знакова, слика 7.14.



Слика 7.12 Читање са тастатуре низа знакова до првог белог знака



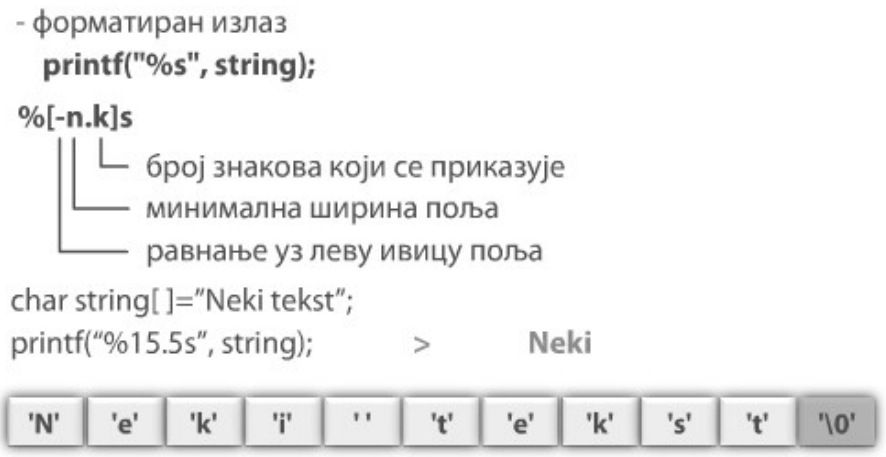
Слика 7.13 Могући формати код читања знакова помоћу функције *scanf()*



Слика 7.14 Читање са тастатуре низа знакова до првог прелаза у нови ред

Приказ на екрану низа знакова

За приказ на екрану низа знакова може се користи функција *printf()* са ознаком %s, уз евентуалне опције за приказ на одређен начин, слика 7.15. Функција *printf()* приказује знаковни низ без прелаза за нов ред на крају и обично се користи за приказ овог низа у комбинацији са подацима других типова. За приказ знаковног низа са прелазом у нов ред на крају користи се функција *puts()*, која у аргументу добија име *stringa* који чува знаковни низ или знаковни низ између наводника (у сваком случају показивач на *string*), слика 7.16.

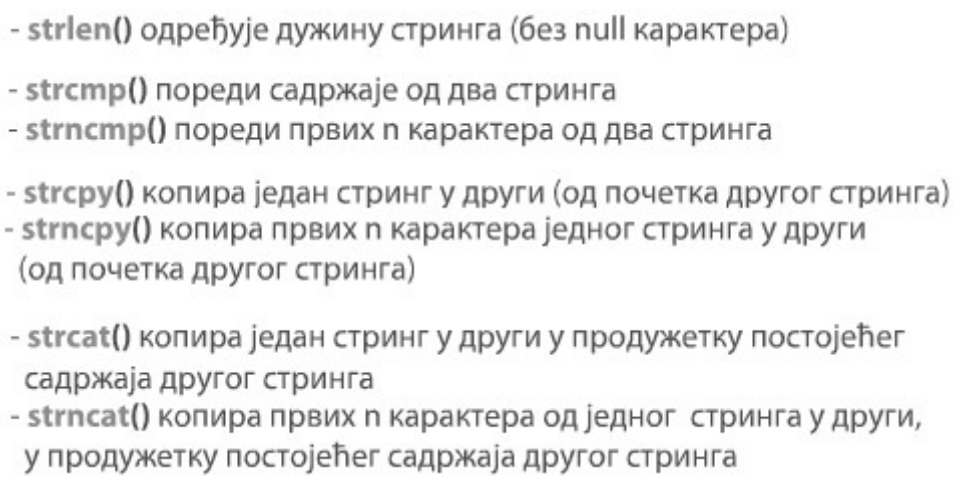


Слика 7.15 Могући формати код функције *printf()*

Слика 7.16 Приказ на екрану низа знакова помоћу функција *printf() и puts()*

7.3. Операције са *stringovima*

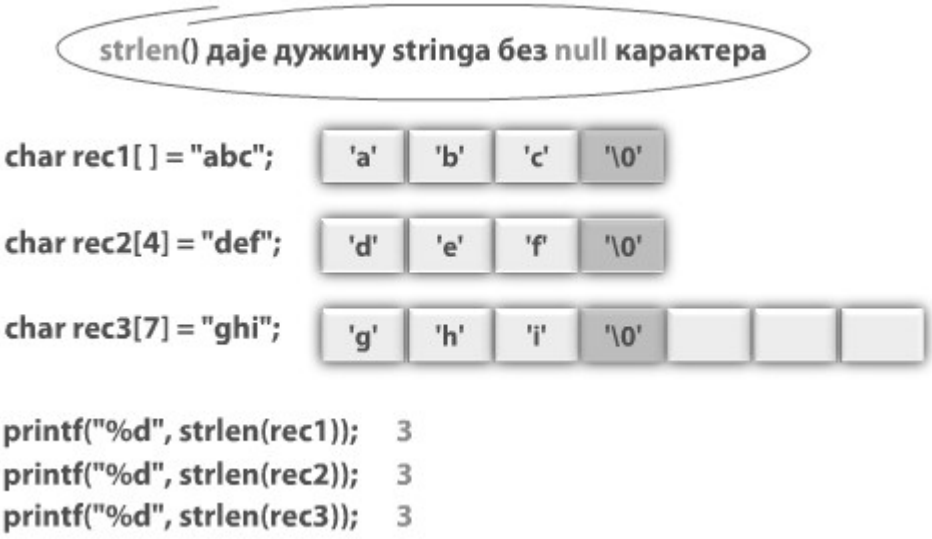
У библиотеци <*string.h*> постоји велики број функција за обраду садржаја *stringova*. То су функције за: одређивање дужине *stringa (strlen()),* поређење комплетних или делова садржаја *stringova (strcmp()* и *strncmp())*, копирање комплетног или дела садржаја *stringa* (*strcpy()* и *strncpy()*), надовезивање комплетног или дела садржаја *stringa (strcat()* и *strncat()*) ... слика 7.17. Поменуте функције користе и показиваче на *stringove* које обрађују, а то могу бити имена *stringova* или унапред дефинисани знаковни низови између наводника.



Слика 7.17 Кратак опис функција које се најчешће користе код *stringova*

Функција за одређивање дужине *stringa*

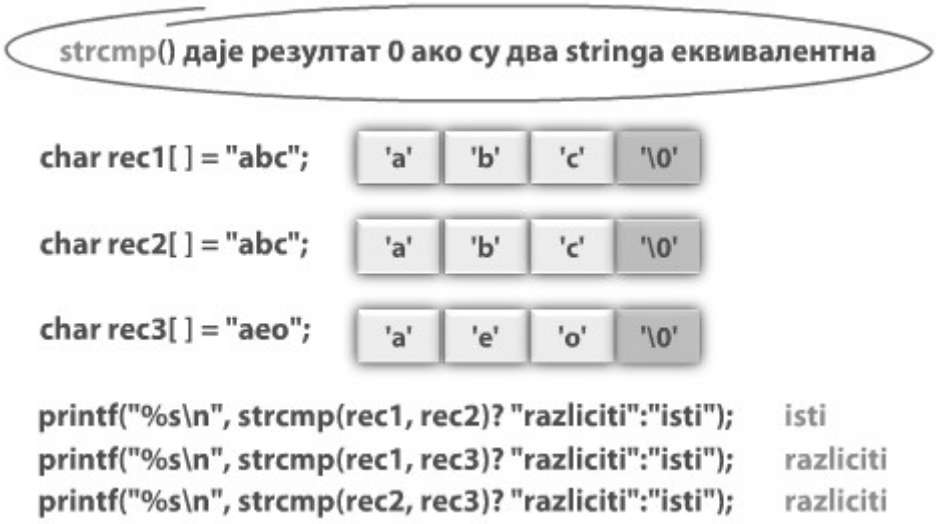
Језик *C* не ограничава дужину једног *stringa.* Функција *strlen()* даје као резултат (типа *int*) дужину *stringa*, ако у аргументу добије показивач на овај *string* (што може бити његово име). При томе броји елементе *stringa* без последњег (знака *null*) тако да се добија само број "корисних" елемената *stringa*, слика 7.18. Ова функција може се користити ако је потребно програму да учита са неког уређаја или обради на неки начин *string*, али карактер по карактер. Онда се предвиди циклус *for* коме је број итерација управо ова дужина *stringa*.



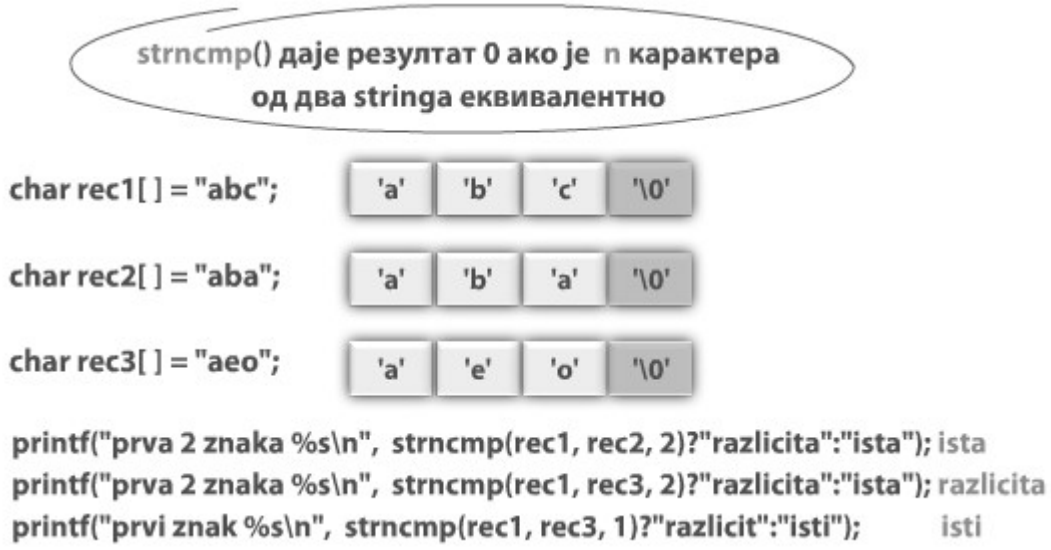
Слика 7.18 Функција *strlen()* одређује дужину *stringa*

Функције за поређење садржаја *stringova*

За поређење садржаја *stringova* користе се функције *strcmp()* и *strncmp()*. Функција *strcmp()* пореди садржаје *stringova* чије показиваче (имена) добија у аргументима и даје као резултат: вредност 0 ако су *stringови* еквивалентних садржаја, или вредност различиту од 0 ако то није случај, слика 7.19. Функција *strncmp()* ради на исти начин, само што пореди првих *n* елемената од сваког *stringa*, при чему је потребно да поред показивача на *stringove* у последњем аргументу добије овај број *n*, слика 7.20.



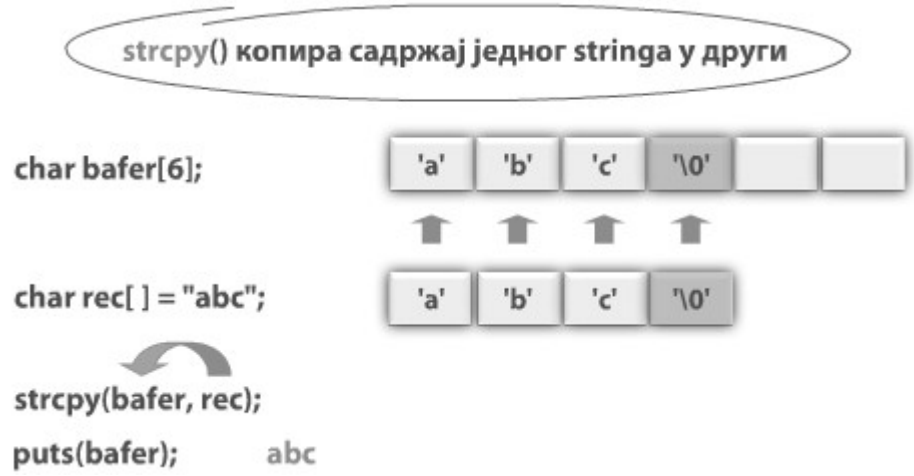
Слика 7.19 Функција *strcmp()* пореди комплетне садржаје *stringova*



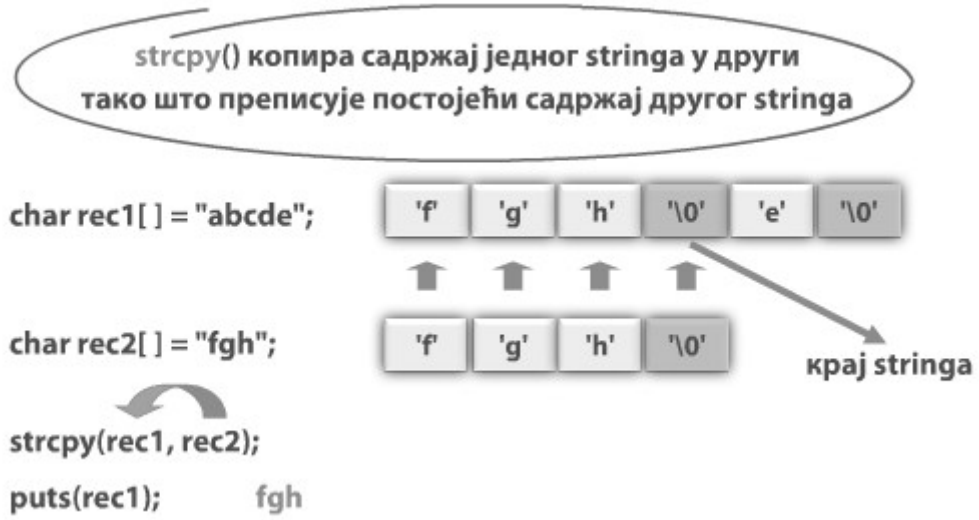
Слика 7.20 Функција *strncmp()* пореди делове *stringova*

Функције за копирање садржаја *stringa*

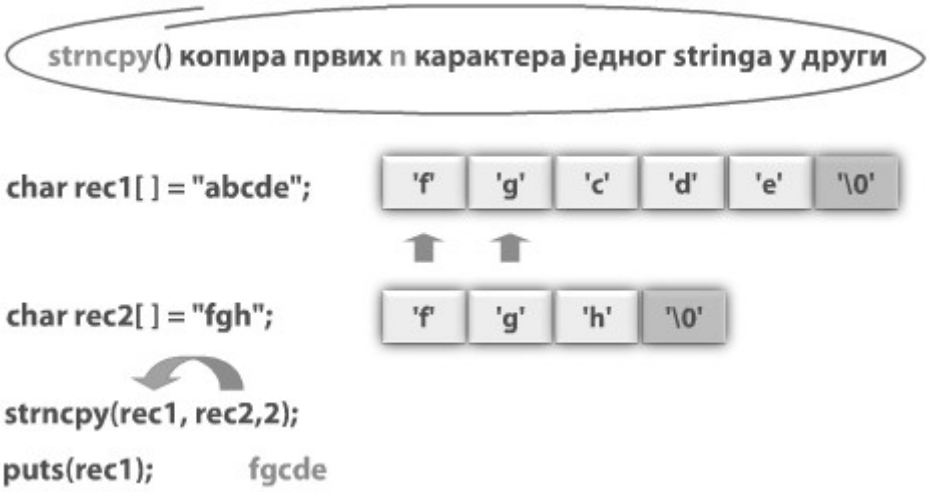
Да се не би увек писао приступ *stringu* карактер по карактер користе се функције *strcpy()* и *strncpy()*. Функција *strcpy()* копира садржај *stringa* од адресе коју је добила у 2. аргументу у *string* на адреси коју је добила у 1. аргументу и то од почетка, слика 7.21. Резултат ове функције је показивач на *string* у који је ископиран садржај. Резултујући *string* добија знак за крај од *stringa* који је копиран, слика 7.22. Функција *strncpy()* ради на исти начин као *strcpy()*, само што копира првих *n* елемената (у трећем аргументу добија број *n*), слика 7.23.



Слика 7.21 Функција *strcpy()* копира комплетан садржај *stringa* у резултујући *string*



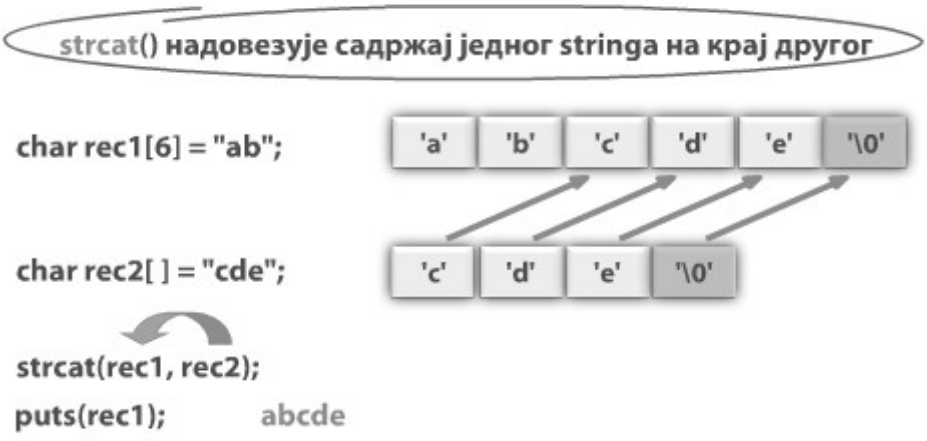
Слика 7.22 Начин рада функције *strcpy()* ако већ постоји садржај у резултујућем *stringu*



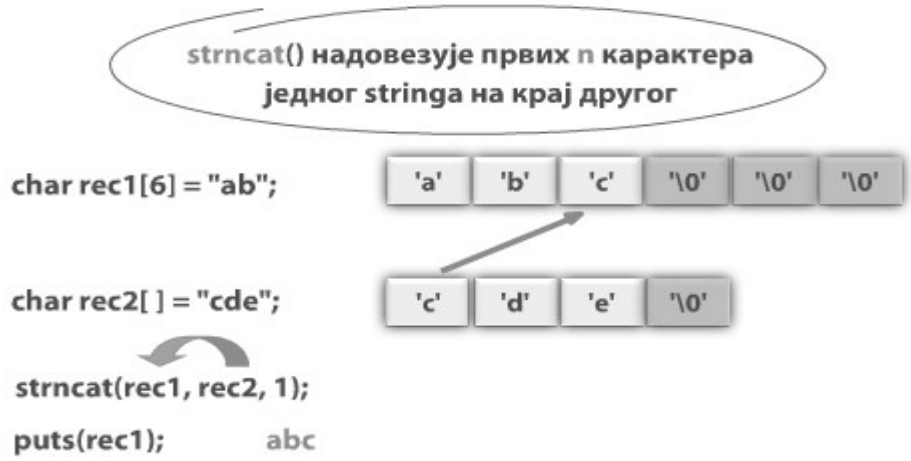
Слика 7.23 Функција *strncpy()* копира део *stringa* у резултујући *string*

Функције за надовезивање садржаја *stringova*

Функције *strcat()* и *strncat()* омогућавају повезивање садржаја *stringova*. Функција *strcat()* копира садржај *stringa* од адресе коју је добила у 2. аргументу у *string* на адреси коју је добила у 1. аргументу, али у продужетку постојећег садржаја. Знак *null* скраја првобитног садржаја у резултујућем *stringu* преписује се првим знаком од додатог *stringa*, слика 7.24. Функција *strncat()* ради на исти начин као *strcat()*, само што копира првих *n* елемената (у трећем аргументу добија број *n*), слика 7.25.



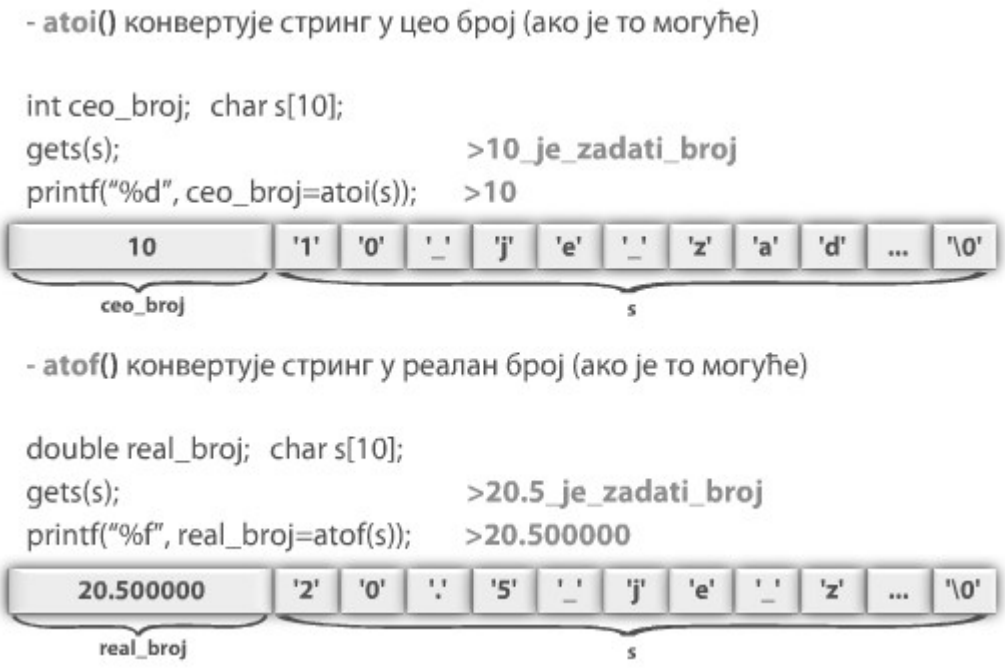
Слика 7.24 Функција *strcat()* копира садржај stringa у продужетку садржаја резултујућег *stringa*



Слика 7.25 Функција *strncat()* копира део *stringa* у продужетку садржаја резултујућег *stringa*

Конвертовање *stringova* у бројеве

Библиотека <*stdlib.h*> садржи више група сродних функција које се често користе у *C* програмима. Једна од њих је група функција за конверзију *stringova*. Биће објашњене само неке од ових функција које су највише заступљене у програмима, то су: функција за конверзију *stringa* у цео број *atoi()* и функција за конверзију *stringa* у реалан број *atof()*. Ове функције омогућавају да програм прочита податак једноставно као *string* а онда препознаје број у њему, конвертује део *stringa* у број и онда га даље користи, слика 7.26.



Слика 7.26 Функције *atoi()* и *atof()* за конверзију *stringova*

Функције за конвертовање *stringova*

Функција *atoi()* препознаје цео број типа *int* (предзнак и децималне цифре) у *stringu* чији је показивач добила у аргументу и даје као резултат овај цео број, слика 7.27. Функција *atof()* препознаје у *stringu* реалан број типа *double* (предзнак, децималне цифре, децималну тачку или ознаку за експонент) и као резултат даје овај реалан број, слика 7.28. Поменуте функције не узимају у обзир беле знакове и нуле на почетку *stringa*, раде конверзију до првог знака који не препознају као део броја, а ако нема конверзије у број дају резултат 0.


```
char tekst1[ ] = "20 i neki tekst";
char tekst2[ ] = "30.15";
char tekst3[ ] = "neki tekst 40";

printf("\nceo broj: %d", atoi(tekst1));   20
printf("\nceo broj: %d", atoi(tekst2));   30
printf("\nceo broj: %d", atoi(tekst3));   0
```

Слика 7.27 Резултат примене функције за конверзију *stringa* у цео број


```
char tekst1[ ] = "10.25 i neki tekst";
char tekst2[ ] = "5e+2";
char tekst3[ ] = "neki tekst 200.30";

printf("\nbroj: %f", atof(tekst1));     10.250000
printf("\nbroj: %f", atof(tekst2));    500.000000
printf("\nbroj: %f", atof(tekst3));      0.000000
```

Слика 7.28 Резултат примене функције за конверзију *stringa* у реалан број

7.4. Питања

1. Шта представља *string* у *C* програму?
2. Да ли је *string* у *C* програму један од основних типова?
3. Да ли постоји ограничење у дужини *stringa*?
4. Како се означава крај сваког *stringa*?
5. Који начини постоје за декларацију и иницијализацију *stringa*?
6. Како се може декларисати и иницијализовати правоугаони низ *stringova*?
7. Помоћу којих функција се *string* може учитати са тастатуре а помоћу којих исписати на екрану?
8. Која функција се користи за израчунавање дужине *stringa*?
9. Шта функција *strcmp()* пореди код *stringova*?
10. Која је разлика између функција *strcmp()* и *strncmp()*?
11. Како ради функција *strcpy()*, а како функција *strcat()*?
12. Шта је потребно обезбедити код *stringa*, за копирање садржаја у овај *string* помоћу функције *strcpy()*?
13. Шта је потребно обезбедити код *stringa*, за копирање садржаја у овај *string* помоћу функције *strcat()*?
14. Које се функције могу користи за конверзију *stringa* у број?
15. Како ради функција за конверзију *stringa* у цео број, а како функција за конверзију *stringa* у реалан број?

8. Сортирање и претраживање низова

Кратак преглед лекције

Низ било које димензије и типа једноставније се користи ако се његови елементи сортирају. Развијен је велики број алгоритама за сортирање низова у било који од могућих поредака: растући, опадајући, нерастући или неопадајући поредак. Следе описи алгоритама за сортирање који су међу најједноставнијим и њихова имплементација на језику *C*. Биће поменути и алгоритми за претраживање низова који су такође доста заступљени и који су поједностављени ако је низ сортиран на неки начин.

8.1. Приступ елементима низа: појединачно и у циклусу

Алгоритми за сортирање и претраживање низова користе приступ елементима низова различитих типова, појединачно (помоћу индекса) или у циклусу (помоћу наредбе *for*). Код *stringa* је на почетку потребно одредити корисну дужину *stringa* (помоћу функције *strlen()*) без знака *null*, који при извршавању програма треба да остане на свом месту. Код низа *stringova* сваком елементу (једном *stringu*), као и делу низа (*stringovima* у циклусу), може се доделити вредност помоћу функције *strcpy()*.

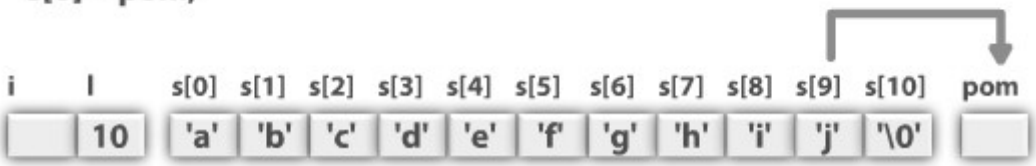
Померање вредности у елементима низа

У алгоритмима за сортирање предвиђено је померање вредности у елементима низа за одређени број места и у одређеном смеру. Ово је описано на примерима цикличног померања вредности елемената следећих низова: једнодимензионалног низа типа *int*, слика 8.1, једног *stringa*, слика 8.2 и низа *stringova*, слика 8.3. Циклично померање значи да нема губљења вредности елемената после помераја, већ само њихово померање спочетка на крај низа или у обрнутом смеру.

```
int i, niz[] = {1, 2, 3, 4}, pom;
pom = niz[0];
for(i=0; i<3; i++)
  niz[i]=niz[i+1];
niz[3] = pom;
```

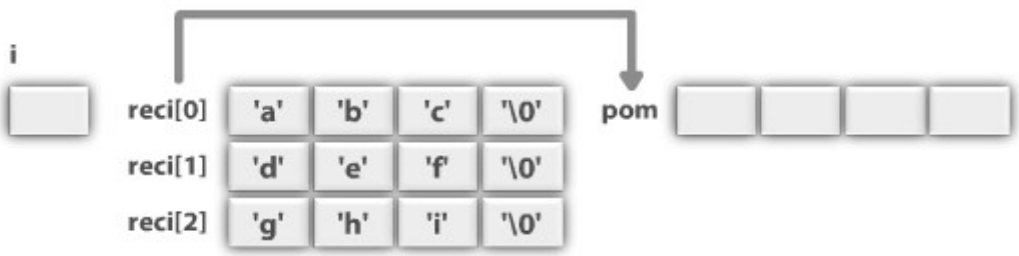
Слика 8.1 Циклично померање елемената у низу целих бројева за једно место улево

```
int i, l;
char s[] = "abcdefghij", pom;
l=strlen(s);
pom = s[l-1];
for(i=l-1; i>0; i--)
   s[i]=s[i-1];
s[0] = pom;
```



Слика 8.2 Циклично померање знакова у *stringu* за једно место удесно

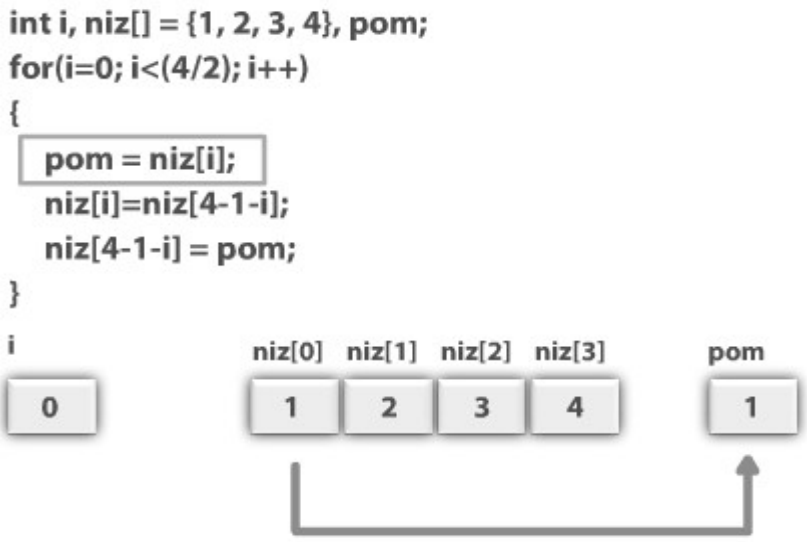
```
int i;
char reci[][4] = {"abc", "def", "ghi"}, pom[4];
strcpy(pom, reci[0]);
for(i=0; i<2; i++)
   strcpy(reci[i], reci[i+1]);
strcpy(reci[2], pom);
```



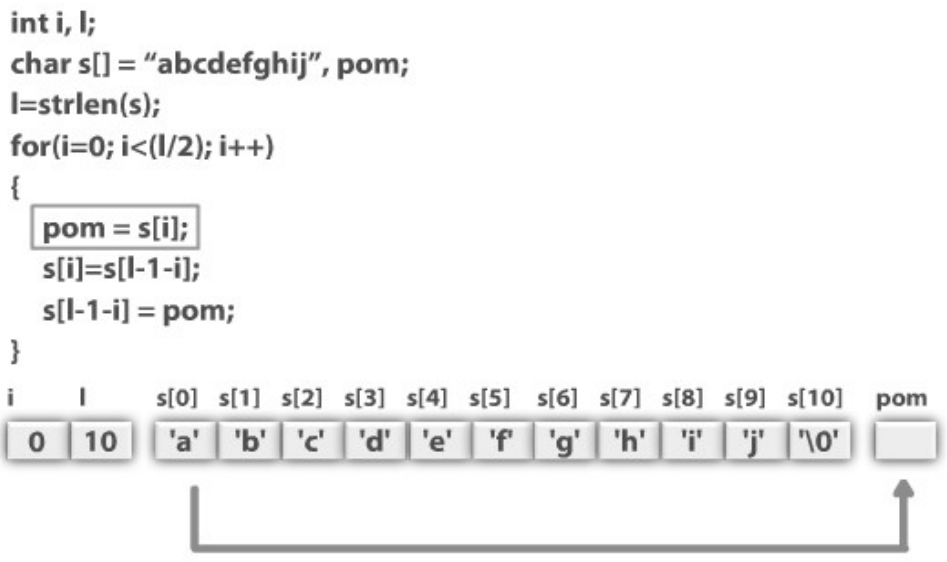
Слика 8.3 Циклично померање *stringova* у низу за једно место улево

Замена вредности у елементима низа

У сваком алгоритму за сортирање предвиђена је и евентуална замена вредности у елементима низа, уз коришћење помоћне променљиве или помоћног низа променљивих. Замена вредности елементима низа описана је на примерима инвертовања код следећих низова: једнодимензионалног низа типа *int*, слика 8.4, једног *stringa*, слика 8.5 и дводимензионалног низа типа *char* (низа *stringova*), слика 8.6. Инвертовање места елементима у низу значи замену места елементима: првом и последњем, другом и претпоследњем ...

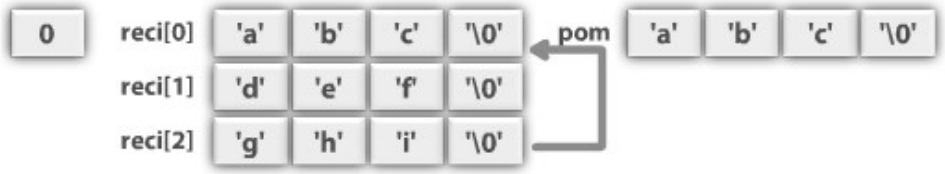


Слика 8.4 Инвертовање места елементима у низу целих бројева



Слика 8.5 Инвертовање места знаковима у *stringu*

```
int i;
char reci[][4] = {“abc”, “def”, “ghi”}, pom[4];
for(i=0; i<(3/2); i++)
{
   strcpy(pom, reci[i]);
   strcpy(reci[i], reci[3-1-i]);
   strcpy(reci[3-1-i], pom);
}
```



Слика 8.6 Инвертовање места *stringovima* у низу

8.2. Сортирање низова

Низове који садрже само међусобно различите елементе могуће је уредити у растући или у опадајући поредак (релација између суседних елемената: „мање“ или „веће“). Код осталих низова поредак може бити неопадајући или нерастући (релација између суседних елемената: «мање или једнако» или «веће или једнако»), слика 8.7. Ако је неопходан растући или опадајући поредак поновљене вредности укидају се формирањем новог низа (биће овде објашњено) или сажимањем (неопходна употреба показивача).

- растући поредак

$$niz[0] < niz[1] < \ldots < niz[n-2] < niz[n-1]$$

- неопадајући поредак

$$niz[0] \leq niz[1] \leq \ldots \leq niz[n-2] \leq niz[n-1]$$

- опадајући поредак

$$niz[0] > niz[1] > \ldots > niz[n-2] > niz[n-1]$$

- нерастући поредак

$$niz[0] \geq niz[1] \geq \ldots \geq niz[n-2] \geq niz[n-1]$$

Слика 8.7 Могућа уређења низова

Алгоритам за сортирање методом избора елемента

Ако користи методу избора програм бира редом сваки пар елемената једног низа (први и други, први и трећи, ... први и последњи, други и трећи, други и четврти, ... други и последњи, ... претпоследњи и последњи), и пореди им

вредности. Ако релација између елемената у пару није очекивана замењује им места, у супротном оставља их како јесу, слика 8.8. После поређења сваког елемента (и евентуалне замене места) са свим елементима даље у низу, на месту овог елемента је одговарајућа вредност.

Слика 8.8 Алгоритам за сортирање методом избора елемента,
у неопадајући поредак

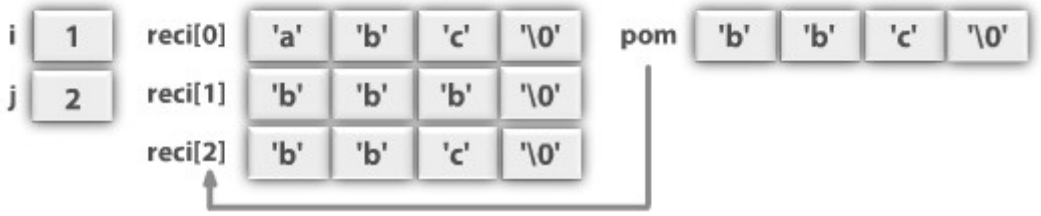
Сортирање бројева у низу методом избора елемента

Имплементација на језику C алгоритма за сортирање методом избора показана је на примеру низа типа *int*, слика 8.9, као и низа *stringova*, слика 8.10. Код низа бројева потребне су две наредбе циклуса: једна спољна за избор сваког елемента редом, од првог до претпоследњег и друга унутрашња за поређење овог елемента са сваким даље у низу. Код низа *stringova* алгоритам се реализује на исти начин, само што се за поређење вредности *stringova* користи функција *strcmp()*, а за доделу вредности *stringu* функција *strcpy()*.

```
int i, j, niz[] = {5, 2, 9, 4}, pom;
for(i=0; i<3; i++)
   for(j=i+1; j<4; j++)
     if(niz[i]>niz[j])
      {
         pom = niz[i];
         niz[i]=niz[j];
         niz[j] = pom;
      }
```

Слика 8.9 Пример сортирања методом избора на низу бројева,
у неопадајући поредак

```
int i, j;
char reci[][4] = {"bbb", "bbc", "abc"}, pom[4];
for(i=0; i<2; i++)
   for(j=i+1; j<3; j++)
      if(strcmp(reci[i], reci[j])>0)
      {  strcpy(pom, reci[i]);
         strcpy(reci[i], reci[j]);
         strcpy(reci[j], pom);
      }
```



Слика 8.10 Пример сортирања методом избора на низу *stringova*, у неопадајући поредак

Бројеви поређења и замена код методе избора

Код методе избора број поређења елемената је у ствари број парова елемената. Први елемент се пореди са осталих *n-1*, други са осталих *n-2*, трећи са осталих *n-3*... и претпоследњи само са последњим. То значи да број поређења представља суму реда: $n_{por} = S = 1 + 2 + \ldots + (n-2) + (n-1)$, $n_{por} = n*(n-1)/2$. За разлику од броја поређења, који је код ове методе увек исти, без обзира на уређеност низа, број потребних замена елемената зависи од уређености и може бити у опсегу од 0 до броја поређења ($n_{zam} = 0 \div n_{por}$).

Алгоритам за сортирање методом уметања елемента

Ако користи методу уметања програм у помоћној променљивој памти редом сваки елемент (од другог до последњег) и испитује релацију у којој је овај елемент са претходним елементима. Све док релација није очекивана помера претходне елементе удесно, кад то више није случај умеће упамћену вредност на своје место, слика 8.11. После поређења сваког елемента са елементима претходницима у низу (и евентуалних померања), сви елементи укључујући и овај елемент поређани су у одговарајући поредак.

Слика 8.11 Алгоритам за сортирање методом уметања елемента

Сортирање бројева у низу методом уметања елемента

Имплементација алгоритма за сортирање методом уметања показана је на примеру низа типа *int*, слика 8.12, као и низа *stringova*, слика 8.13. Код низа бројева и овде су потребне две наредбе циклуса, једна спољна за избор сваког елемента редом, од другог до последњег, и друга унутрашња за поређење са претходним елеменатима и њихово померање у низу. Код низа *stringova* користе се функције *strcmp()* и *strcpy()* за операције поређења и доделе које су потребне алгоритму за сортирање.

```
int i, j, niz[] = {5, 2, 9, 4}, pom;
for(i=1; i<4; i++)
{  pom = niz[i];
   for(j=i-1; j>=0; j--)
      if(niz[j]>pom)
         niz[j+1]= niz[j];
      else
         break;
   niz[j+1] = pom;
}
```

Слика 8.12 Пример сортирања методом уметања на низу бројева,
у неопадајући поредак

```
int i, j;
char reci[][4] = {“bbb”, “bbc”, “abc”}, pom[4];
for(i=1; i<3; i++)
{   strcpy(pom, niz[i]);
    for(j=i-1; j>=0; j--)
       if(strcmp(niz[j], pom)>0)
          strcpy(niz[j+1], niz[j]);
       else
          break;
    strcpy(niz[j+1], pom);
}
```

Слика 8.13 Пример сортирања методом уметања на низу *stringova*, у неопадајући поредак

Бројеви поређења и замена код методе уметања

Код методе уметања број поређења елемената зависи од уређености првобитног низа. У најгорем случају број поређења је исти као и код методе избора $n_{por} = n*(n-1)/2$. У најбољем случају у сваком пролазу кроз низ реализује се једно поређење (за $i=1$, $i=2$,… $i=n-1$), што значи да их је укупно $n_{por} = n-1$. Код ове методе свака уређеност низа смањује број поређења, што није био случај код методе избора. Код једне и друге методе број замена може бити у опсегу од 0 до броја поређења ($n_{zam} = 0 \div n_{por}$), слика 8.14.

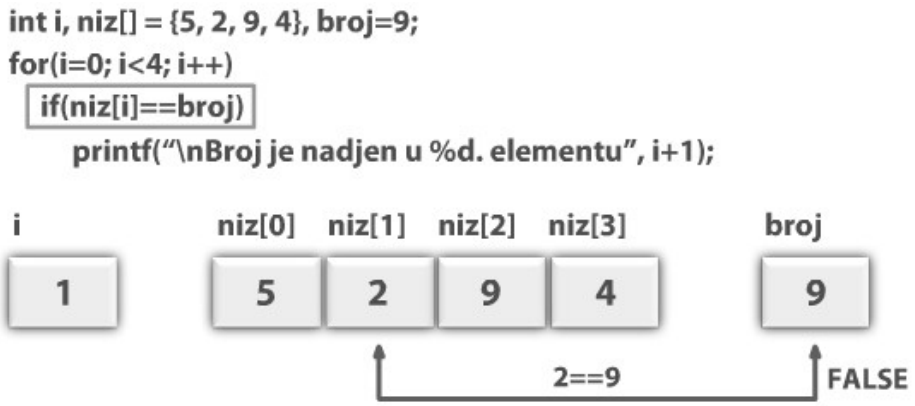
Метода	Број поређења (најбољи случај)	Број поређења (најгори случај)	Број замена (најбољи случај)	Број замена (најгори случај)
Избора елемента	n*(n-1)/2	n*(n-1)/2	0	n*(n-1)/2
Уметања елемента	n-1	n*(n-1)/2	0	n*(n-1)/2

Слика 8.14 Табела са бројевима поређења и замена места елементима низа, код методе избора и методе уметања елемента у низу

8.3. Претраживање низова

Линеарно претраживање неуређеног низа

Када низ у коме је потребно наћи неку вредност (или вредности) није већ уређен (сортиран на неки од могућих начина) за претраживање се може користити линеарна метода као најједноставнија. То значи да се у једној наредби циклуса предвиди испитивање вредности сваког елемента низа редом, слика 8.15. Када се примењује ова метода за претраживање број итерација циклуса уједно је и број елемената низа, тако да није погодна код низова велике дужине.



Слика 8.15 Линеарно претраживање неуређеног низа

Бинарно претраживање уређеног низа

Бинарно претраживање погодно је код дугих али претходно сортираних низова. Следи опис овог алгоритма на примеру растућег низа. Испитује се да ли је тражена вредност у елементу на средини низа. Ако то није случај, и ако је ова вредност мања од елемента на средини, претражује се само прва половина низа, а ако је већа од елемента на средини претражује се друга половина низа. Сваки од ових низова претражује се тако што се испитује прво елемент на средини и дели се на два, све док се не пронађе вредност, слика 8.16.

```
int niz[] = {1, 2, 3, 4, 5, 6, 7, 8}, broj=6;
int imin=0, imax=7, isr;
while(imin<=imax)
{  isr=(imin+imax)/2;
   if(broj==niz[isr])
   {  printf("\nBroj je nadjen u %d. elementu", isr+1);
      break;
   }
   else if(broj<niz[isr])
       imax=isr-1;
   else
       imin = isr+1;
}
```

```
Broj je nadjen u  6. elementu
```

Слика 8.16 Бинарно претраживање уређеног низа

8.4. Питања

1. Да ли се елементма низа може приступити и појединачно сваком?
2. Шта значи циклично померање елемената у низу?
3. Како се остварује циклично померање елемената у низу улево / удесно?
4. Шта значи инвертовање елемената у низу?
5. Како се остварује инвертовање елемената у једнодимензионалном низу?
6. Како се остварује инвертовање елемената у вишедимензионалном низу?
7. Која су могућа уређења сортираних низова?
8. Како ради метода избора код сортирања једнодимензионалног низа?
9. Како ради метода избора код сортирања вишедимензионалног низа?
10. Како ради метода уметања код сортирања једнодимензионалног низа?
11. Како ради метода уметања код сортирања вишедимензионалног низа?
12. Која од метода избора / уметања узима у обзир почетно уређење низа?
13. Које су предности методе избора, а које методе уметања код сортирања?
14. Како ради метода линеарног претраживања низа?
15. Како ради метода бинарног претраживања низа?

9. Показивачи и низови

Кратак преглед лекције

Постоје програмски задаци који на језику *C* имају само решења помоћу показивача, као и неки који имају много ефикасније решење ако се примене показивачи. Показивач је променљива у којој може бити сачувана адреса променљиве основног или изведеног типа или било које функције програма. Када се примењује код низа, као изведеног типа, онда се користи адресна аритметика која дефинише скуп могућих операција над показивачима. На почетку биће објашњени показивачи на променљиве.

9.1. Индиректан приступ меморији

Један начин да се у самим наредбама програма напише приступ променљивој је онај који је у претходним примерима коришћен: помоћу имена променљиве или помоћу имена и индекса елемента ако је о низу реч. То је један од могућих приступа променљивој, директан приступ. Ако се декларише и иницијализује одговарајући показивач може се издвојити и користити у програму адреса од које је резервисан простор у меморији за променљиву и могућ је приступ променљивој помоћу њеног показивача, индиректан приступ.

Појам показивача

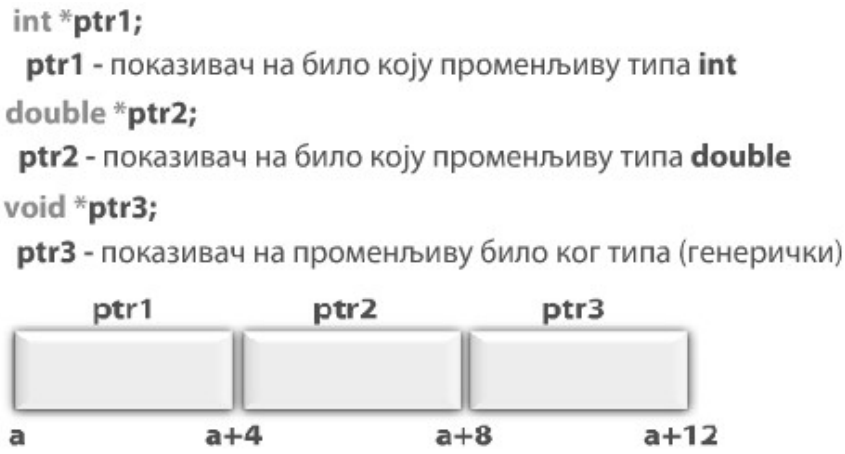
Показивач *(pointer)* је променљива изведеног типа која се декларише у програму за чување адресе било које друге променљиве или функције у програму. Декларација показивача значи у ствари декларацију показивачке променљиве целобројног типа. Тип показивачке променљиве зависи од адресног опсега расположиве меморије и не везује се унапред за неки тип (*int* или *long*). Показивач се иницијализује тако што му се додељује адреса. Следи објашњење показивача на променљиве.

Декларација показивача

Декларација показивача разликује се од декларације променљиве основног типа по томе што на почетку има тип променљиве чију ће адресу чувати (на коју ће

показивати) а не тип показивача (одређен опсегом расположиве меморије). Да се ради о показивачу у декларацији се види по знаку звездица (*) који се пише непосредно испред имена промењиве и представља симбол за оператор индиректног проступа, слика 9.1. Ако уместо типа на почетку пише *void* показивач је генерички и може чувати адресу било ког типа, слика 9.2.

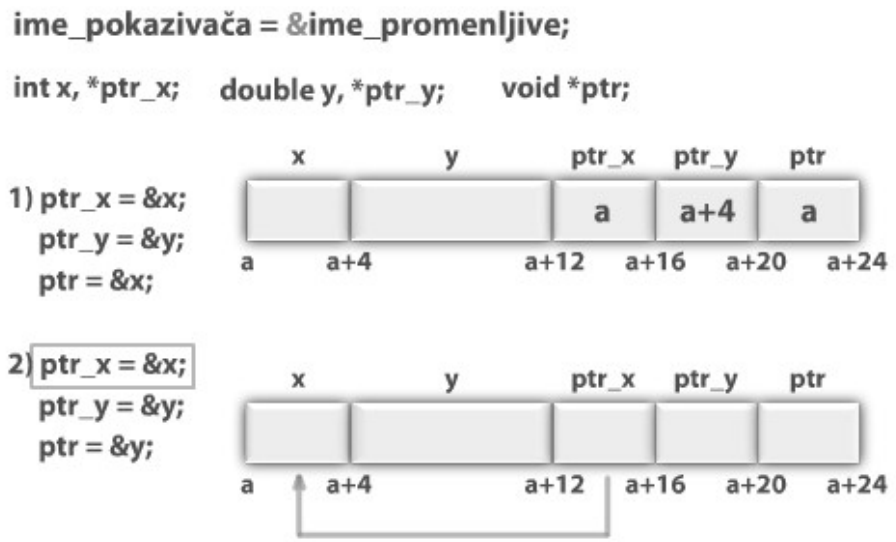
Слика 9.1 Декларација показивача



Слика 9.2 Примери декларације показивача

Иницијализација показивача

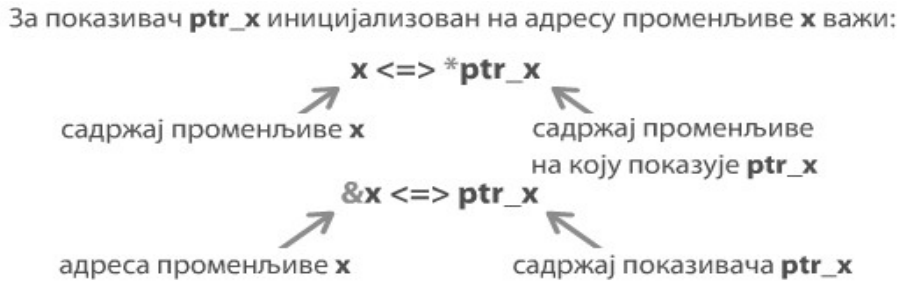
Да би показивач био повезан са неком промењивом (показивао на неку промењиву) треба да буде иницијализован на њену адресу. За доделу адресе користи се адресни оператор *&*, који написан испред имена промењиве даје адресу њеног првог бајта, слика 9.3. Ако је један показивач декларисан да показује на одређен тип, онда исти показивач може бити искоришћен за адресу неке друге промењиве истог типа, али не и за адресу неког другог типа. Једино је код генеричког показивача могућа додела адресе било ког типа.



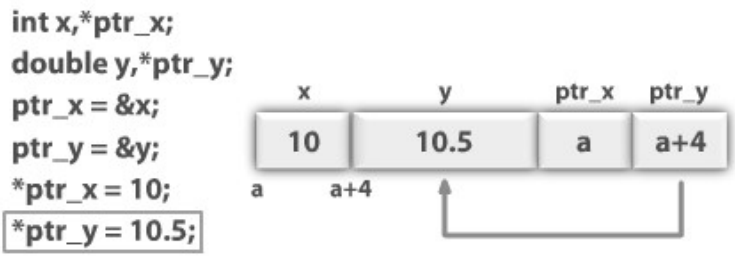
Слика 9.3 Иницијализација показивача

Приступ промењљивим помоћу показивача

Ако је показивач декларисан и иницијализован помоћу њега се може индиректно приступити промењљивој. За приступ садржају промењљиве у наредбама програма пише се оператор индиректног приступа, непосредно испред имена показивача на ову промењљиву (чита се: садржај на који показује). За адресу промењљиве пише се име показивача на исту промењљиву (јер је његов садржај адреса), слика 9.4. Без обзира на други начин приступа промењљивим сви оператори користи се на познат начин, слика 9.5.



Слика 9.4 Индиректан приступ промењљивама



Слика 9.5 Примери индиректног приступа промењљивама

9.2. Адресна аритметика

За индиректан приступ свим редом елементима низа користе се показивач иницијализован на адресу почетка низа и операције адресне аритметике. Додела вредности једног показивача другом користи се кад показивачи треба да добију исту адресу. Диференцирање је потребно за одређивање броја елемената низа од једне до друге адресе. Компарација се користи за проверу релације између адреса у низу. Инкремент или декремент обавезно се користе да би се на основу адресе једног приступило осталим елементима низа.

Додела вредности једног показивача другом

За доделу вредности једног показивача другом користи се оператор доделе. Ако два показивача садрже адресе променљивих међусобно истог типа онда се користи само овај оператор. У супротном, примењује се и конверзија у тип на који показују, слика 9.6. Ако се вредност показивача додељује генеричком показивачу нема потребе за експлицитном конверзијом.

ptr1 = ptr2 - ако су **ptr1** и **ptr2** показивачи на исти тип или је један од њих генерички показивач

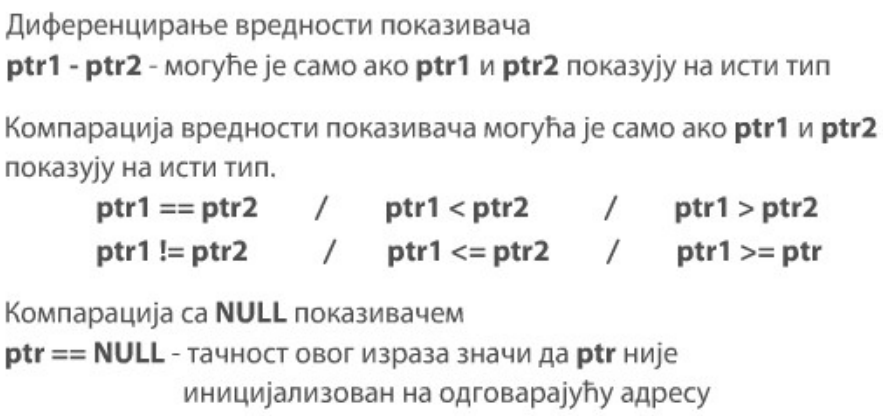
ptr1 = (tip *)ptr2 - ако се разликују типови на које показују **ptr1** и **ptr2**, користи се конверзија типа (наводи се тип на који показује **ptr1**)

ptr1 = a; - додела константе адресе **a** показивачу **ptr1**

Слика 9.6 Додела вредности једног показивача другом

Диференцирање и компарација показивача

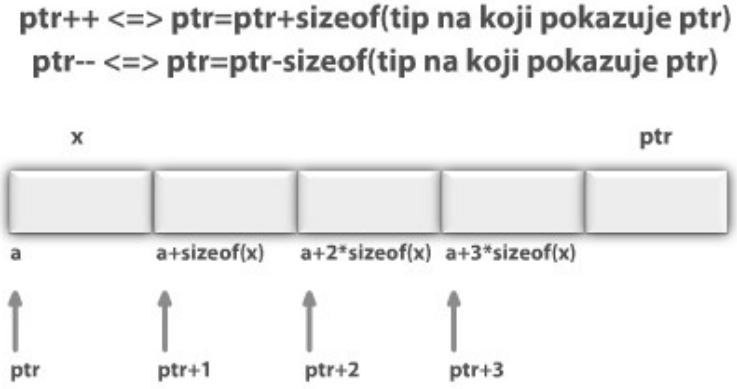
Од основних аритметичких операција код показивача има смисла само одузимање, односно диференцирање. Сабирање нема велику примену а множење, дељење и остатак дељења нису дозвољени. Релацијски оператори користе се за компарацију адреса. Да ли је показивач иницијализован, проверава се компарацијом са *NULL* показивачем (ако показивач има вредност еквивалентну *NULL* онда није иницијализован). Диференцирање и компарација примењују се код показивача на међусобно исти тип, слика 9.7.



Слика 9.7 Диференцирање и компарација показивача

Инкремент и декремент показивача

Адреса једног низа је адреса његовог првог елемента. Адреса сваког даље елемента померена је у односу на адресу претходног за величину типа елемента (број бајтова једног елемента). Због овог помераја код показивача је уведена специфична примена оператора инкремент и декремент. Инкремент показивача (преинкремент или постинкремент) увећава његову вредност за величину типа на који показује, а декремент (предекремент или постдекремент) умањује вредност показивача за величину типа на који показује, слика 9.8.



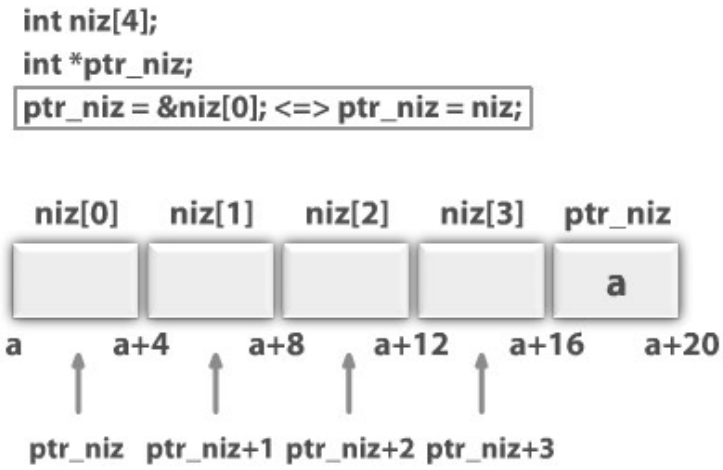
Слика 9.8 Инкремент и декремент показивача

9.3. Примена показивача код низова

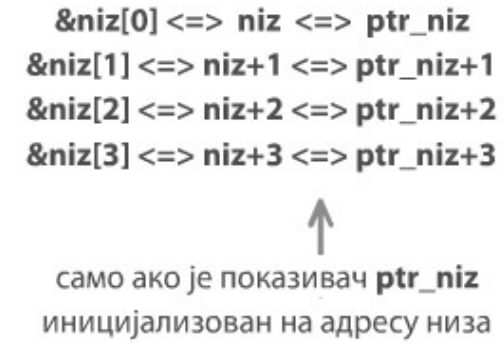
Примена показивача код једнодимензионалних низова

Показивач који је декларисан на променљиву одређеног типа може бити искоришћен и за чување адресе низа истог типа (првог елемента низа). У овом случају показивач се иницијализује тако што му се додели адреса првог

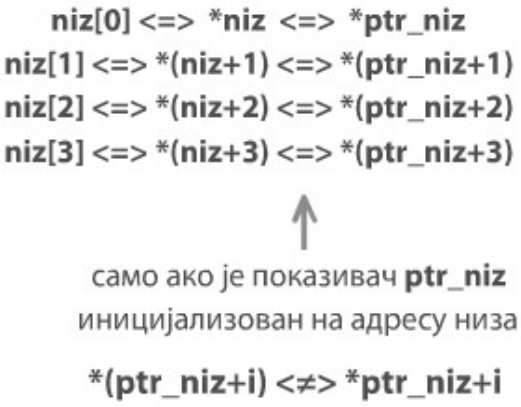
елемента (помоћу адресног оператора), или једноставније само име низа (као вредност константног показивача на низ), слика 9.9. После ове иницијализације, помоћу показивача на низ могућ је индиректан приступ адресама, слика 9.10 и садржајима свих елемената, слика 9.11.



Слика 9.9 Примена показивача код једнодимензионалних низова



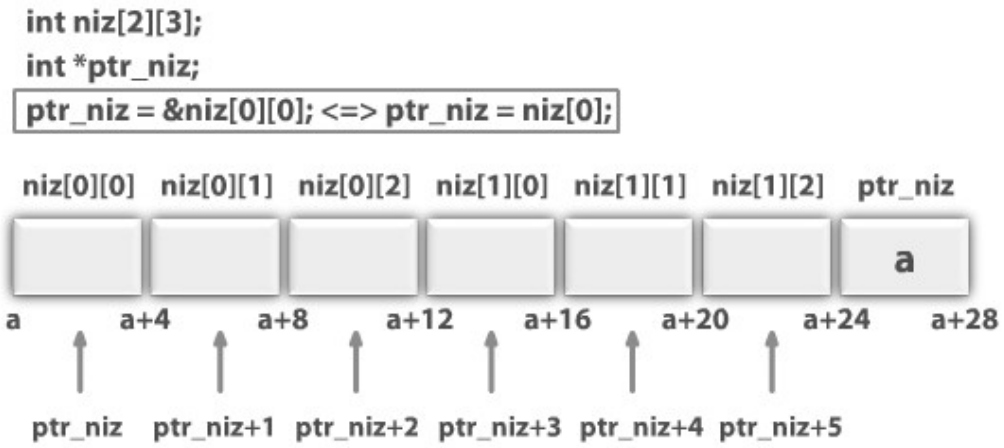
Слика 9.10 Индиректан приступ адресама елемената једнодимензионалног низа



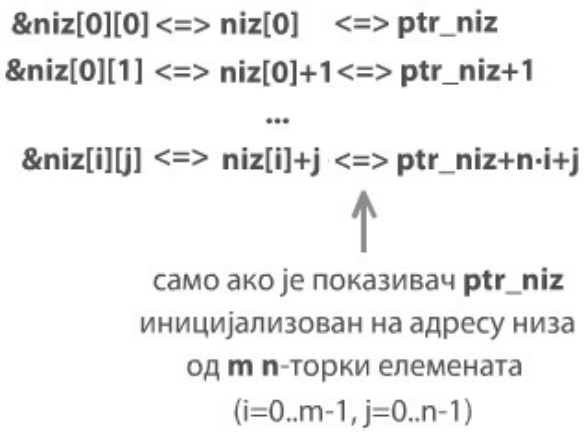
Слика 9.11 Индиректан приступ садржајима елемената једнодимензионалног низа

Примена показивача код дводимензионалних низова

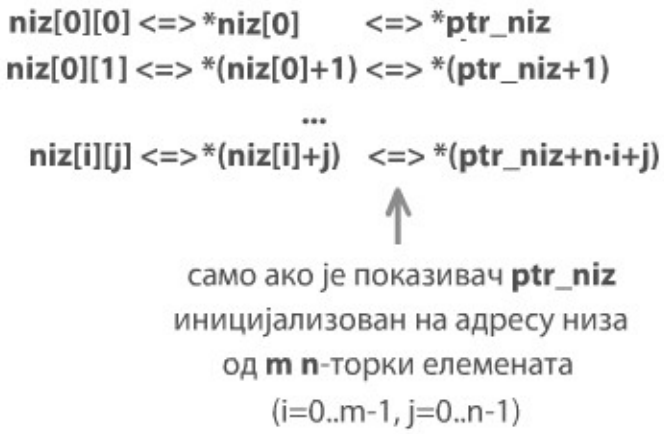
Показивач на променљиву одређеног типа може бити искоришћен за чување адресе вишедимензионалног низа истог типа. Код дводимензионалног низа показивач се иницијализује тако што му се додели адреса првог елемента од првог низа (помоћу адресног оператора), или само име првог од низова (константни показивач на дводимензионалан низ), слика 9.12. Помоћу показивача на низ могућ је индиректан приступ свим адресама, слика 9.13 и свим садржајима елемената у низу, слика 9.14.



Слика 9.12 Примена показивача код дводимензионалних низова



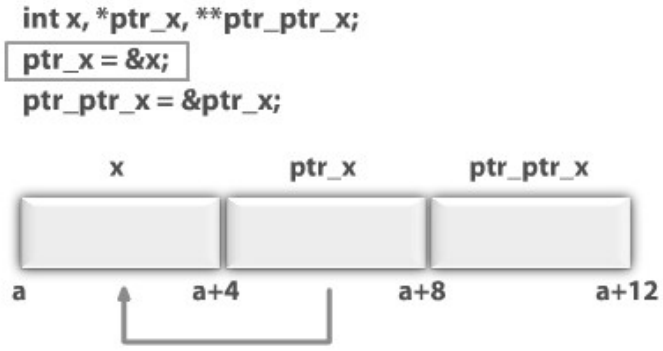
Слика 9.13 Индиректан приступ адресама елемената дводимензионалног низа



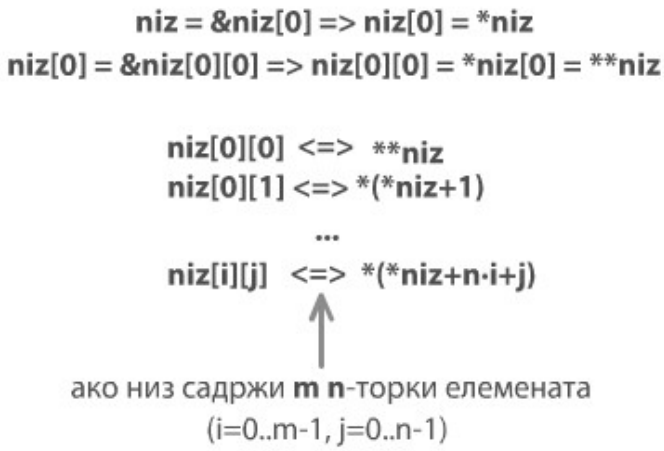
Слика 9.14 Индиректан приступ садржајима елемената дводимензионалног низа

Показивач на показивач

Показивач на показивач *(pointer* на *pointer)* је променљива изведеног типа која се декларише у програму за чување адресе неког другог показивача. Показивач на показивач декларише се такође као показивачка променљива целобројног типа (*int* или *long* у зависности од расположиве меморије), а иницијализује тако што му се додељује адреса другог показивача и онда се може рећи да показује на тај други показивач, слика 9.15. Приступ елементима дводимензионалног низа могућ је помоћу показивача на показивач на исти низ, слика 9.16.



Слика 9.15 Показивач на показивач на низ



Слика 9.16 Приступ елементима низа помоћу показивача на показивач

Низови показивача

Елементи низа декларисаног у *C* програму могу бити и показивачи. Једна примена низа показивача је код креирања неправоугаоних дводимензионалних низова, који на ефикаснији начин користе меморију у односу на правоугаоне низове, слика 9.17. На почетку декларације низа показивача је тип на који показује сваки од њих, онда име низа (симбол * испред имена, јер се ради о показивачима). Са иницијализацијом у продужетку нема непотребног заузећа меморије, већ се одваја неопходан простор за сваки од низова, слика 9.18.

```
char reci[][8]={"Prva","Druga","Treca","Cetvrta"};
```

'P'	'r'	'v'	'a'	'\0'	'\0'	'\0'	'\0'
'D'	'r'	'u'	'g'	'a'	'\0'	'\0'	'\0'
'T'	'r'	'e'	'c'	'a'	'\0'	'\0'	'\0'
'C'	'e'	't'	'v'	'r'	't'	'a'	'\0'

Слика 9.17 Правоугаони дводимензионални низ

Слика 9.18 Низ показивача на неправоугаони дводимензионални низ

9.4. Питања

1. Шта значи директан а шта индиректан приступ променљивама у меморији?
2. Шта је показивач и како се декларише у програму?
3. Како се показивач иницијализује на адресу променљиве?
4. Како се декларише и иницијализује генерички показивач?
5. Које операције адресне ариметике су најзаступљеније у програму?
6. Како раде оператори инкремент / декремент код показивача?
7. На који начин се вредност једног показивача може доделити другом?
8. На који начин се код показивача може применити диференцирање, а на који компарација?
9. Како се име низа у програму може користити за приступ адресама и садржајима појединих елемената тог низа?
10. Како се показивач може иницијализовати на адресу низа?
11. Да ли се помоћу посебно декларисаног и иницијализованиог показивача на низ може приступити сваком посебно елементу тог низа?
12. Како се посебно декларисани и иницијализовани показивач на низ може користи за приступ адресама и садржајима појединих елемената тог низа?
13. Шта је показивач на показивач и како се декларише у програму?
14. Како се показивач иницијализује на адресу другог показивача?
15. Како се декларише и користи у програму низ показивача?

10. Функције

Кратак преглед лекције

У *C* програму једини могући потпрограм је функција. Сваки *C* програм има бар једну функцију, то је главна функција *main()*. Да би обавила свој задатак главна функција позива друге функције, функције из *C* библиотека и функције које пише програмер за потребе конкретног програма. За функцију из *C* библиотеке довољан је само позив (ако је укључена њена библиотека). За остале функције различите од главне поред позива потребни су: декларација (прототип функције) и дефиниција (саме наредбе функције).

10.1. Појам и основни делови функције

Функција је независан сегмент кода *C* програма, који има своје име, обавља самостално свој програмски задатак, опционо користи податке са места у програму са ког је позвана и опционо враћа вредност на исто место у програму. Свака функција у програму различита од главне позива се из главне функције, непосредно или посредно (из било које друге функције која се опет позива из главне) и свака од њих даје решење свог програмског подзадатка, које се уклапа у решење главног програмског задатка.

Размена података између функција

Размена података између појединих функција у програму може бити реализована механизмом позива/повратка функције, помоћу аргумената и повратне вредности. Аргументи су подаци које функција може добити од функције која је позива и може их бити: 0, 1 или више. Функција може вратити само једну вредност неког свог резултата функцији из које је позвана и то је повратна вредност. Ако функција не враћа вредност, већ резултате шаље само на излазне уређаје, она има само "бочне ефекте".

Предности писања програма са функцијама

Писање изворног кода *C* програма једноставније је када се уоче делови програмског задатка и сваки од њих реши помоћу једне функције, у односу на писање целог програма у једној функцији. Ако неки од програмских задатака треба решити на више места у програму функција се дефинише само једном а позива потребан број пута, тако да је у овом случају и меморија ефикасније искоришћена. Тестирање је такође поједностављено јер је лако издвојити функцију у којој су евентуалне грешке.

Декларација (прототип) функције

За сваку функцију коју ће користити програму је потребна декларација која пружа основне информације о једној функцији: име, број и типови аргумената, као и тип повратне вредности, слика 10.1. Декларација функције може бити написана на почетку програма, ван сваке функције, да би могла бити позвана из било које од њих (када програм садржи више датотека декларације свих функција могу бити издвојене у посебној датотеци), мада је дозвољено и писање декларације унутар неке друге функције, ако је само ова позива.

Слика 10.1 Декларација (прототип) функције

Тип вредности коју враћа функција

У зависности од типа повратне вредности постоје две категорије функција: функције које враћају вредност и друге које имају само "бочне ефекте". Тип повратне вредности функције може бити основни (*char, int, float*…) или изведени *C* тип (показивач на тип *char, int, float…)*. Ако функција нема

повратну вредност на месту њеног типа пише *void*, слика 10.2. Испред имена функције *main()* није обавезно писање ознаке типа јер ова функција враћа статус оперативном систему типа *int*, који је подразумевани тип у језику *C*.

```
char f1(...);
  f1 враћа вредност типа char
int f2(...);
  f2 враћа вредност типа int
float f3(...);
  f3 враћа вредност типа float
double f4(...);
  f4 враћа вредност типа double
void f5(...);
  f5 не враћа вредност
```

10.2 Тип повратне вредности функције

Име функције

Име функције додељује се по правилима за идентификаторе (исто као и за доделу имена променљивим, низовима...) и треба да опише, колико је то могуће, задатак који решава функција. Име сваке функције у једном програму је јединствено (у једном програму не може бити више функција са међусобно истим именом, већ је обавезна разлика бар у неком знаку). Име функције пише се у декларацији после типа повратне вредности, а после имена следи листа формалних аргумената.

Листа формалних аргумената функције

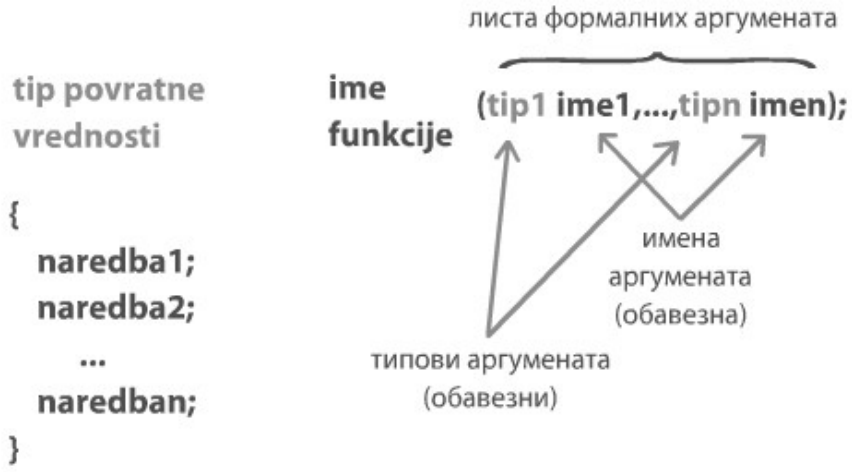
Декларација функције има листу формалних аргумената, која обавезно садржи типове свих аргумента и евентуално имена аргумената (имена нису битна у декларацији). Листа формалних аргумената може бити потпуна и непотпуна. У првом случају листа аргумената садржи типове за све аргументе. Непотпуна листа аргумената обично има тип првог аргумента (који чува информацију о броју осталих) и три тачке после тога, што значи да ће остали аргументи бити предвиђени дефиницијом функције и познати при њеном позиву, слика 10.3.

```
int f1(int x, int y);
  f1 користи два податка, сваки типа int и враћа вредност истог
  типа
double f2(int x, int y);
  f2 користи два податка, сваки типа int и враћа вредност типа
  double
double f3(int x, long y, double z);
  f3 користи три податка (типови су int, long и double) и враћа
  вредност типа double
int f4(void);
  f4 не користи податке од остатка програма и враћа вредност
  типа int
void f5(int x, long y);
  f5 користи два податка (типови су int и long) и не враћа вредност
```

10.3 Листа формалних аргумената функције

Дефиниција функције

Дефиниција функције садржи саме наредбе функције и пише се ван сваке друге функције у програму. Прва линија дефиниције је заглавље функције са листом формалних аргумената из декларације, само су овде обавезна имена аргумената (јер су потребна у наредбама функције). То значи да прва линија дефиниције може бити истог облика као декларација, само без знака тачка-зарез на крају. Тело функције садржи наредбе које се пишу унутар витичастих заграда а извршавају се након позива ове функције, слика 10.4.



10.4 Дефиниција функције

Наредбе функције

Функција може садржати наредбе: декларације, извршне и наредбе повратка слика 10.5. Наредба декларације промењиве у функцији чини ову промењиву видљивом само у овој функцији, то је њена локална промењива. Извршне наредбе садрже операторе, између осталих и позиве других функција. Наредба повратка, *return*, омогућава враћање вредности резултујећег израза слика 10.6. Ако функција не враћа вредност у наредби повратка нема израза или једноставно и нема наредбе повратка.

```
int proizvod(int x, int y)
{
 int z;       /*naredba deklaracije*/
 z = x*y;     /*izvršna naredba*/
 return z;  /*naredba povratka*/
}
        ^
        ||
        v
int proizvod(int x, int y)
{
 return (x*y);  /*naredba povratka*/
}
```

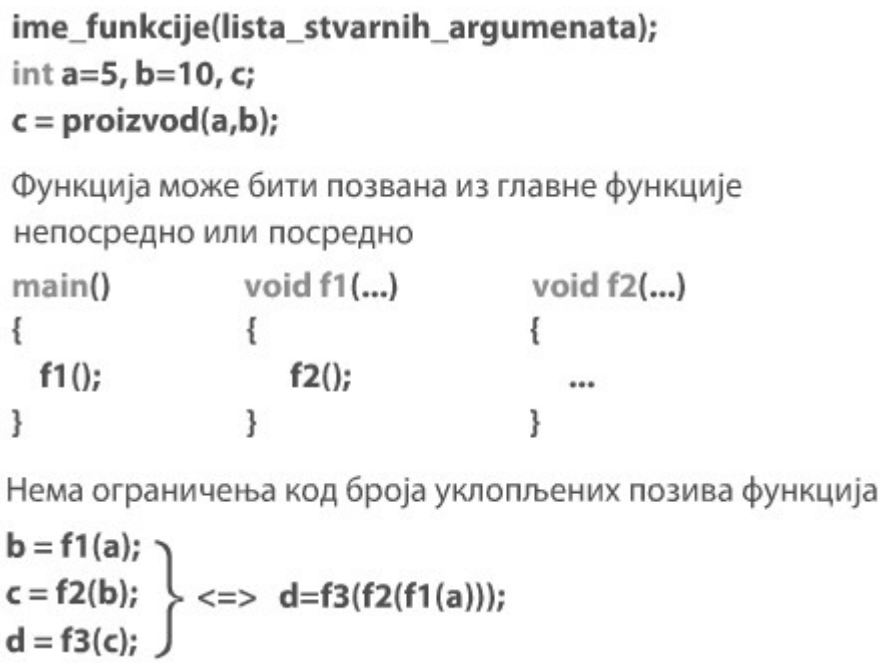
10.5 Наредбе у једној функцији

```
return vrednost; <=> return(vrednost);

return x;
return(x*y);
return((x>=y) ? x : y);

Функција може имати више return наредби,
али не може вратити више од једне вредности

if(uslov)
   return(vrednost1);
else
   return(vrednost2);
```

10.6 Наредба повратка

Позив функције

Позив функције је бинарни C оператор чији је један операнд име функције, а други је листа стварних аргумената (то су имена која се користе за конкретне вредности у том програму, одвојена знаком зарез), слика 10.7. Позив једне функције може бити написан у наредби било које функције програма. Ако функција враћа резултујућу вредност онда се ова вредност користи у наредби којој је враћена (додељује се променљивој одговарајућег типа, користи на месту аргумента у позиву неке друге функције...).



10.7 Позив функције

Механизам позива функције и повратка из функције

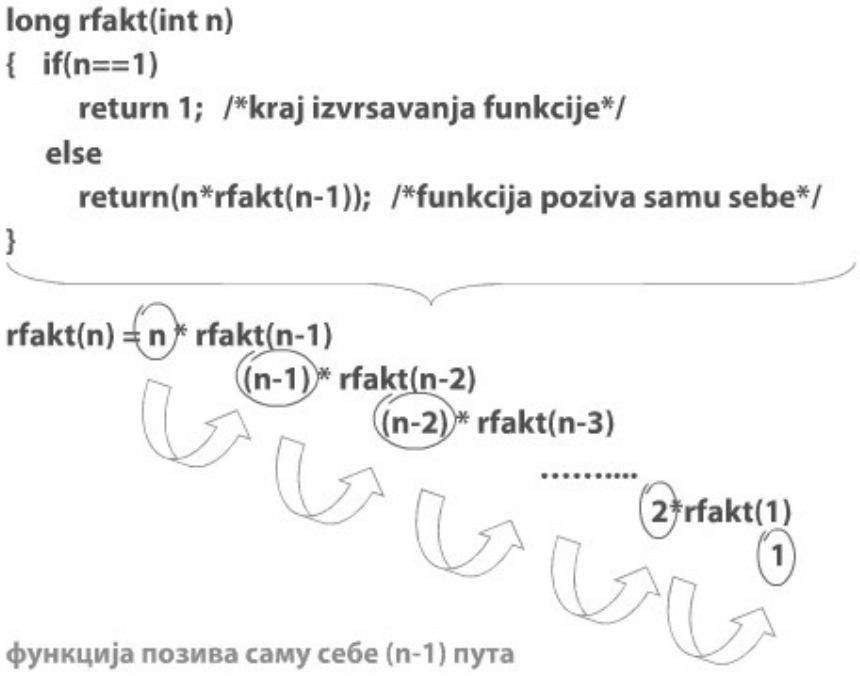
Када се функција позове ток извршавања програма преноси се овој функцији. Све информације неопходне за повратак памте се у одвојеном простору у меморији (то је стек). Следи скок на почетак позване функције, при чему се формални аргументи функције иницијализују на вредности прослеђених аргумената. Када се изврше наредбе позване функције повратна вредност (ако је има) додељује се, помоћу наредбе повратка и сачуваних информација на стеку, на место позива. Ток извршавања враћа се функцији из које је био позив.

Предаја аргумената функцији по вредности

У *C* програму могућа су два начинапредаје аргумената функцији: по вредности и по референци. Предаја аргумента по вредности значи да функција у аргументу добија вредност променљиве и може да је користи али не може да је промени (јер добија копију садржаја променљиве). Предаја аргумента по референци значи да функција у аргументу добија вредност показивача на променљиву. У овом другом случају функција добија адресу променљиве и онда може променити њен садржај.

10.2. Рекурзивна функција

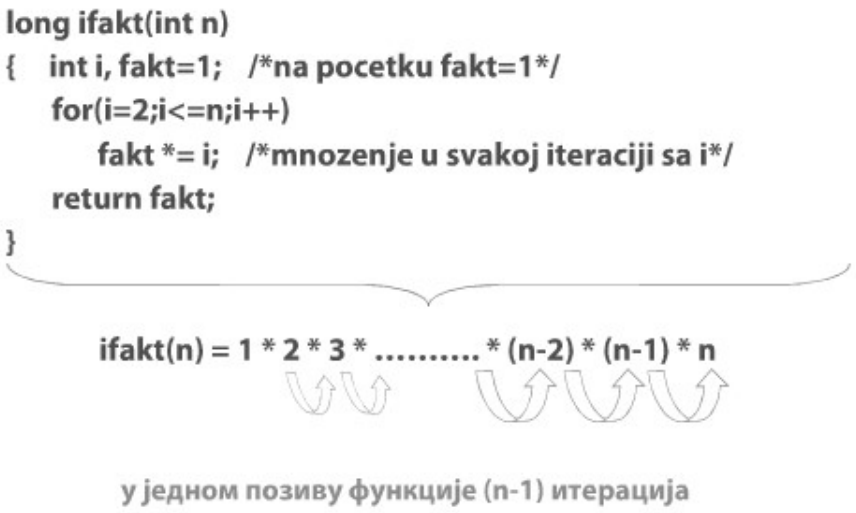
Рекурзивна функција је функција која позива саму себе, коначан број пута и сваки пут са измењеним аргументима. Пример може бити функција која решава факторијел броја *n*, слика 10.8. У првом позиву функција добија у аргументу вредност *n* и множи је са резултатом позива исте функције за вредност аргумента *n-1*. Иста функција из другог позива множи вредност *n-1* са резултатом позива исте функције за вредност аргумента *n-2*... . Вредност 1 у аргументу значи крај позива и резултат факторијел *n!=n*(n-1)*(n-2)*...*3*2*1*.



10.8 Рекурзивни алгоритам функције која решава факторијел броја

Поређење рекурзивног и итеративног алгоритма

Свака рекурзија може бити решена итерацијом, слика 10.9. Предност рекурзивног поступка у односу на итеративни је у једноставнијем алгоритму. Рекурзија међутим значи велику учестаност позива једне функције и при сваком њеном позиву покретање описаног механизма позива и повратка. У случајевима у којима време које се троши на често покретање овог механизма не условљава проблеме при извршавању програма рекурзија се користи, а у супротном замењује се итеративним алгоритмом.



10.9 Итеративни алгоритам функције која решава факторијел броја

10.3. Главна и друге специфичне функције

Могући облици главне функције

У дефиницији главне функције није обавезно писати тип повратне вредности испред имена, јер је то подразумевани тип *int*. Главна функција треба да врати статус оперативном систему под којим се извршава, помоћу наредбе *return* са назначеном вредношћу 0 (или написаним симболом *EXIT_SUCCESS*) на самом крају функције, слика 10.10. Прослеђивање овог статуса оперативном систему значи да није било превременог краја извршавања програма. Ова наредба је у примерима до сада била изостављена због једноставнијег кода.

```
main()                  main()
{                       {
   ...          <=>        ...
   return 0;                  return EXIT_SUCCESS;
}                       }
```

10.10 Главна функција враћа статус оперативном систему

Крај извршавања програма из било које његове функције

У програму треба предвидети ситуације после којих није дозвољен наставак његовог извршавања. У библиотеци <*stdlib.h*> постоји функција *exit()* чијим се позивом може обезбедити крај извршавања једног програма, из било које његове функције. Функција *exit()* треба да добије у аргументу вредност типа *int* коју прослеђује оперативном систему. Ако је ова вредност 0 (или симбол *EXIT_SUCCESS*) статус је "крај програма, нема грешке", а ако је 1 (или симбол *EXIT_FAILURE*) оперативни систем добија статус "крај програма услед грешке". Функција *exit()* нема повратну вредност, слика 10.11.

Функција **exit()** из библиотеке **<stdlib.h>**

```
void exit(int status);
```

- када је потребан превремени крај извршавања програма и нема грешке

```
if(uslov)              if(uslov)
  exit(0);     <=>       exit(EXIT_SUCCESS);
```

- када је потребан превремени крај извршавања програма услед грешке

```
if(uslov)              if(uslov)
  exit(1);     <=>       exit(EXIT_FAILURE);
```

10.11 Превремени крај извршавања програма

Функција за прекид (скретање) тока извршавања програма

У библиотеци <*stdlib.h*> постоји функција *system()* чијим се позивом може прекинути (скренути) ток извршавања једног програма и пребацити неком

другом програму, под истим оперативним системом, а по завршетку другог програма наставити прекинути програм. Функција *system()* треба да добије у аргументу показивач на *string* који чува име другог програма (може бити и само име програма између наводника). Функција враћа вредност 0 ако је добила аргумент, или -1 ако није, слика 10.12.

```
Функција system() из библиотеке <stdlib.h>
int system(const char *ime_programa);
(враћа 0 ако добије у аргументу име програма, у супротном -1)

system("cls");
system("dir");
system("ime_drugog_programa");
```

10.12 Позив другог програма из било које функције једног програма

10.4. Питања

1. Која врста потпрограма је могућа у *C* програму?
2. Шта је функција у *C* програму?
3. Које су предности распоређивања изворног кода програма по функцијама?
4. Које делове садржи свака функција у програму?
5. Шта су аргументи а шта повратна вредност функције?
6. Шта је декларација функције и шта све она садржи?
7. Које све делове садржи дефиниција једне функције?
8. Како се пише заглавље функције?
9. По чему се разликују декларација и заглавље дефиниције функције?
10. Које врсте наредби може садржати једна функција?
11. Шта је и где у програму може бити написан позив функције?
12. Како се реализује механизам позива / повратка из функције?
13. Шта је рекурзивна функција?
14. Помоћу које функције се може прекинути извршавање било које функције програма?
15. Који су све могући облици главне функције у C програму?

Индекс појмова

Литература

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

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

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

4. С. Ђенић, Основи програмирања 1, електронски уџбеник, ВИШЕР, Београд, 2014.

5. С. Ђенић, Ј. Митић, С. Штрбац, Основи програмирања на језику C, збирка примера и задатака, ВИШЕР, Београд, 2012.