ლევან შოშიაშვილი

პროგრამირების ენა C

თბილისი

2016

სარჩევი

წინასიტყვაობა **v**

შესავალი **vii**

C და *C* + + . . . vii

C++ უპირატესობანი . . . vii

ობიექტზე ორენტირებულობა და მონაცემებზე ორიენტირებულობა . . . vii

პროგრამული კოდი **ix**

1 *C* – ლექსიკური ელემენტები, ცვლადები და ფუნქციები **1**

1.1 ცვლადები და ტიპები . . . 1

1.2 ცვლადების დეკლარაცია და ინიციალიზაცია . . . 1

1.3 ფუნქცია printf ეკრანზე ბეჭდვა . . . 2

1.4 printf ფუნქციის ფორმატი . . . 3

1.4.1 sizeof ცვლადის ზომა . . . 4

1.5 *C* მათემატიკური ოპერაციები . . . 5

1.5.1 ზრდის (increment)შემცირების(decrement) ოპერატორები . . . 5

1.6 ფუნქცია scanf წაკითხვა ეკრანიდან . . . 6

1.7 პრეპროცესორის დირექტივები . . . 10

1.7.1 დირექტივა #define . . . 10

2 გადაწყვეტილების მიღება და ციკლები **13**

2.1 შედარების(პირობითი) ოპერატორები (<,>, <=, >=, ==, !=) . . . 13

2.2 '**if**' და '**if ... else**' ბრძანება . . . 13

2.3 ბრძანება switch ... case . . . 18

2.4 ციკლები . . . 22

2.4.1 for ციკლი . . . 22

2.4.2 ბრძანება break . . . 24

2.4.3 while ციკლი . . . 25

2.4.4 ბრძანება continue . . . 27

2.5 ჩამონათვალი(დანომრილი) ტიპი enum . . . 28

3 მნიშვნელობა, მისამართი, მითითება, ფუნქციები **31**

3.1 მნიშვნელობა და მისამართი . . . 31

3.2 მიმთითებელი 32
3.2.1 NULL მიმთითებელი 32
3.3 ფუნქციები 34
3.4 მონაცემების გადაცემა მიმთითებლით 40
3.5 დეკლარაცია იმპლემენტაცია სხვადასხვა ფაილებში 42
3.6 extern "C" 46
3.7 *C* პროექტის შექმნა 46
3.8 *C* ენის სტანდარტები და იმპლემენტაცია 47

4 მასივი **51**
4.1 ერთგანზომილებიანი მასივი 51
4.1.1 ერთგანზომილებიანი მასივის ინიციალიზაცია 51
4.1.2 C99 სპეციფიური ინიციალიზაცია 52
4.1.3 მასივის ელემენტებზე მიმართვა 52
4.1.4 მიმთითებელი და მასივი 53
4.1.5 მაგალითი: მთვლელების მასივი 56
4.1.6 მაგალითი: მთელი მონაცემების ჰისტოგრამა 57
4.1.7 მაგალითი: შემთხვევითი რიცხვები გენერატორი 57
4.1.8 მაგალითი: ფიბონაჩის რიცხვები 58
4.1.9 მაგალითი: რიცხვი სხვადასხვა სიტემაში 59
4.2 ასო ნიშანთა მასივი 60
4.3 მასივის გადაცემა ფუნქციაში 62
4.3.1 მაგალითი: ერთგანზომილებიანი მასივის გადაცემა 63
4.3.2 მაგალითი:მასივის დალაგება 65
4.4 მრავალგანზომილებიანი მასივები 66
4.4.1 მრავალგანზომილებიანი მასივის ინიციალიზაცია 66
4.4.2 მაგალითი: ორგანზომილებიანი მასივების შეკრება 67
4.4.3 2D მასივის გადაცემა ფუნქციაში 68

5 ცვლადები და მეხსიერება **71**
5.1 ავტომატური ცვლადები და მხედველობის არე 71
5.2 გლობალური და ლოკალური ცვლადები 72
5.3 გლობალური ცვლადების მხედველობის არე. **static** ცვლადები 73
5.4 ცვლადის კვალიფიკატორი volatile 76
5.5 ტიპის დაყვანა — „კასტირება“ 77
5.6 void მიმთითებელი 77
5.7 დინამიური ობიექტები 79
5.7.1 malloc() ფუნქცია 79
5.7.2 free() ფუნქცია 80
5.7.3 realloc() ფუნქცია 80
5.7.4 calloc() ფუნქცია 81

6 შედგენლი ტიპეპი: struct და union **83**
6.1 სტრუქტურები – struct . . . 83
6.2 სტრუქტურის ცვლადების ინიციალიზაცია . . . 86
6.3 მაგალითი: სტრუქტურების მასივის დალაგება . . . 86
6.4 ჩადგმული სტრუქტურები . . . 88
6.5 სტრუქტურების გამოყენება ფუნქციებში . . . 89
6.6 სტრუქტურაზე მიმთითებელი . . . 91
6.7 typedef და struct . . . 92
6.8 გაერთიანება – union . . . 93

7 ბიტური ოპერაციები **97**
7.1 AND, OR, XOR და NOT . . . 97
7.2 მარჯვნივ და მარცხნივ წანაცვლების ოპერაციები . . . 98
7.3 ბიტური ოპერაციების თანმიმდევრობა . . . 99
7.4 ბიტური ოპერაციების გამოყენება . . . 99
7.5 ბიტ ველები . . . 101

8 ფაილში ჩაწერა და წაკითხვა **103**
8.1 ტექსტური და ბინარული მოდა . . . 103
8.1.1 ტექსტური ფაილის წაკითხვა/ჩაწერა . . . 104
8.1.2 `getc()` და `putc()` . . . 107
8.1.3 ფაილის დასასრული . . . 107
8.1.4 `fscanf()` და `fprintf()` . . . 108

წინასიტყვაობა

C პროგრამირების ენის ცოდნა აუცილებელია ელექტრული და ელექტრონული ინჟინერიის სპეციალობის სტუდენტებისათვის, რომლებიც სწავლობენ მიკროკონტროლიორების და ჩაშენებული სისტემების პროგრამირებას.

სამწუხაროდ, ქართველი სტუდენტები არ არიან განებივრებული სასწავლო-სამეცნიერო ლიტერატურით ქართულ ენაზე. ეს განსაკუთრებით ეხება პროგრამირების თანამედროვე ენებს.

თუკი C++ ენისათვის შეიძლება მოიძებნოს რამდენიმე სახელმძღვანელო თუ ლექციათა კურსი ქართულ ენაზე, C პროგრამირების ენისათვის ასეთი რამ არ არსებობს.

წინამდებარე დოკუმენტი წარმოადგენს ლექციათა კურსს წაკითხულს ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტში ზუსტ და საბუნებისმეტყველო ფაკულტეტზე ელექტრული და ელექტრონული ინჟინერიის დეპარტამენტის სტუდენტებისათვის.

უნდა აღინიშნოს, რომ ესაა ლექციათა კურსი და არა წიგნი C ენაში. რაც ნიშნავს, რომ საკითხები რაც განხილვის მიღმა არის დატოვებული განიხილება პრაქტიკულ მეცადინეობებზე. პროგრამირების ენის სწავლება განუყოფელია პროგრამული კოდის წერის პროცესთან.

ამ სალექციო კურსში მოყვანილი მასალა საკმარისია დამწყებთათვის, რომ დამოუკიდებლად გაეცნონ C ენაში არსებულ წარმოდგენებს და პროგრამირების ტექნიკას, დამოუკიდებლად ამოხსნან სალექციო მასალაში მოყვანილი ამოცანები[1].

მოცემული კურსის შესწავლა საკმარისია, რომ შემდგომში გაუადვილდეთ „მიკროკონტროლიორების დაპროგრამების" და „მოდელირება და ვიზუალიზაცია" კურსების შესწავლა და ჰქონდეთ მყარი საფუძველი დამოუკიდებლად გაიღრმავონ ცოდნა მოცემულ სფეროში.

აღსანიშნავია, რომ ქართული ტერმინოლოგია მაღალ ტექნოლოგიურ დარგებში ნაკლულია. ზოგიერთი ტერმინის ქართული შესატყვისი ან არ გამოიყენება, ან საერთოდ არ არსებობს. ჩვენ ვცდილობდით ტერმინოლოგია ყოფილიყო როგორც ქართულად გამართული, ასევე გასაგები[2].

ლევან შოშიაშვილი
2016წ.

[1] დოკუმენტის განახლებული ვერსია, პროგრამული კოდები და კურსის ვიდეომასალა შეგიძლიათ იხილოთ საიტზე http://lshoshia.science.tsu.ge. მოსაზრებები, შენიშვნები შეგიძლიათ გაგვიზიაროთ shoshia at hotmail dot com

[2] დაიხ, სამწუხაროდ ზოგიერთ ავტორთან გვხვდება ისეთი ქართული ტერმინებიც რომელთა მნიშვნელობაც გაუგებარია.

შესავალი

C და *C*++

Unix სისტემის შექმნასთან ერთად აუცილებელი შეიქმნა ახალი პროგრამული ენის შექმნა, რომელიც სამუშაოდ და ასათვისებლად უნდა ყოფილიყო გაცილებით იოლი ვიდრე ასემბლერი. ნაკლებად უნდა ყოფილიყო დამოკიდებული მანქანის მოწყობილობებზე. ამ ენაზე შესაძლებელი უნდა ყოფილიყო სისტემური პროგრამირება. დაწერილი პროგრამული კოდი უნდა ყოფილიყო გადატანადი ერთი სისტემიდან მეორეზე. ასე შეიქმნა პროგრამირების ენა *C*.

C++ განვითარდა 80-იან წლებში AT&T ბელლ ლაბორატორიაში. თავდაპირველად. საჭირო იყო პრეკომპილატორი, რომელსაც *C*++ კოდი გადაჰყავდა *C* კოდში, ხოლო შემდეგ ხდებოდა ამ კოდის კომპილაცია. *C*++ კოდი, რომელსაც კითხულობდა პრეკომპილატორი მოთავსებული იყო ფალილებში .cpp, .C ან .cc გაფართოებით. დღეს , თანამედორვე *C*++ კომპილატორებს გადაჰყავთ *C*++ კოდი მანქანურ კოდებში პრეკომპილაციის გარეშე. ვინაიდან *C*++ წარმოადგენს *C* განვითარებას, *C*++ კომპილატორებს შეუძლიათ *C* კოდის კომპილირებაც. თანამედროვე *C* უკვე აღარ წარმოადგენს *C*++-ის ქვესიმრავლეს. ეს ორი ენა ვითარდება ცალ-ცალკე. ცალ-ცალკეა *C* და *C*++ ენების სტანდარტიზაციის კომისიები. რაც არის *C* სტანდარტი, მაგალითად gcc კომპილატორისთვის ის არის *C*++ ნაწილი მიკროსოფტის კომპილატორისთვის.

C++ უპირატესობანი

ხშირად საუბრობენ *C*++ უპირატესობაზე მოჰყავთ რა სხვდასხვა არგუმენტები, მაგალითად ის, რომ *C* ძირითადად სისტემური პროგრამირების ენაა ხოლო *C*++ უფრო ზოგადი პროგრამირების ენა, რაც ობიექტური შეფასება არაა. *C* ზეც შეიძლება დაიწეროს ზოგადი მოხმარების პროგრამები, თუ ბიბლიოთეკები. ამის მაგალითია gtk პროგრამირების ბიბლიოთეკა, Gnome დესკტოპ გარემო, Gimp – ფოტოშოპის მსგავსი გრაფიკული რედაქტორი, სხვადასხვა დონის მათემატიკური ბიბლიოთეკები: gsl– გნუ სამცნიერო ბიბლიოთეკა, gts – ტრიანგულაციის ბიბლიოთეკა.

C ზე კარგად დაწერილი პროგრამა თითქმის არ ჩამოუვარდება FORTRAN პროგრამებს, თუმცა FORTRAN კვლავ რჩება სამეცნიერო გამოთვლებისთვის ყველაზე სწრაფ ენად. *C*++ მთავარი უპირატესობაა მონაცემების „დამალვის“ შესაძლებლობა, ახალი ტიპების და ობიექტების უფრო ადვილად შექმნა, შაბლონური პროგრამირება, პოლიმორფიზმი და ობიექტორიენტირებულობა. ყველა სხვა „უპირატესობა“ რასაც მიაწერენ *C*++ შესაძლებელია *C* ფარგლებშიც.

ობიექტზე ორენტირებულობა და მონაცემებზე ორიენტირებულობა

ისტორიულად *C*++ წარმოადგენს „*C* + ობიექტზე ორიენტირებულიბის კონცეფცია“. ობიექტის იდეა *C* შეტანილ იქნე იმ დროისათვის ცნობილი ობიექტზე ორიენტირებული ენებიდან SmallTalk და Simula . პროცედურული ენების შემთხვევაში, როგორებიცაა, მაგალითად Pascal,

C, ხდება პრობლემის იდენტიფიცირება. შემდეგ ეს პრობლემა იყოფა რამდენიმე ქვეპრობლემად. შემოგვყავს თითოეული ქვეპრობლემისათვის დამახასიათებელი ცვლადები და ფუნქციები, რომლებიც შეგვიძლია გავაერთიანოთ ბიბლიოთეკაში შემდგომი გამოყენებისათვის.

ობიექტზე ორიენტირებული მიდგომისას კი, ჩვენ გამოვყოფთ ცალკეულ ობიექტებს. აღვწერთ მათ თვისებებს. ასვე აღვწერთ ამ ობიექტთა შორის ურთიერთმიმართებას. თითოეული ობიექტი არის სრულყოფილი იმ გაგებით, რომ მან იცის ყველაფერი თავის შესახებ. ასევე მან იცის რა დამოკიდებულებაში არიან მასთან სხვა ობიექები.

როცა გვაქვს მონაცემთა დიდი რაოდენობა, აღიწეროს თითოეული ერთეული როგორც ობიექტი არაეფექტურია როგორც მეხსიერების ასევე პროგრამის სწრაფი შესრულების თვალსაზრისით. ობიექტზე ორიენტირებული ენა, მაგალითად როგორიცაა C++ თუ მაღალ ეფექტურია მაგალითად მომხმარებლის გრაფიკული ინტერფეისის შესაქმნელად, გამოთვლითი ამოცანებისთვის გაცილებით ეფექტურია პროცედურული მიდგომა.

პროგრამული კოდი

1.1 printf მაგალითი . . . 2
1.2 ცვლადების ზომები. sizeof მაგალითი . . . 4
1.3 printf და scanf. შეკრების მაგალითი . . . 6
1.4 scanf წაკითხვა . . . 6
1.5 ცვლადების სპეციფიკატორები . . . 6
1.6 float და double ცვლადები სხვადასხვა სიზუსტით . . . 7
2.1 printf მაგალითი და შედარების ოპერატორები . . . 17
2.2 „ელემენტარული კალკულატორი“ . . . 19
2.3 switch case მაგალითი . . . 20
2.4 კვადრატული განტოლება . . . 21
2.5 „n რიცხვის ჯამი“ . . . 22
2.6 „გამრავლების ცხრილი“ . . . 22
2.7 „რიცხვის გამყოფი“ . . . 23
2.8 „საერთო გამყოფი“ . . . 23
2.9 „ორმაგი ციკლი“ . . . 24
3.1 ერთი მიმთითებელის მრავალჯერადი გამოყენება . . . 33
3.2 ფუნქცია არგუმენტის გადაცემა მნიშვნელობით . . . 35
3.3 ფუნქცია არგუმენტის გადაცემა მისამართით . . . 37
3.4 არგუმენტის გადაცემა const მისამართით . . . 38
3.5 არგუმენტის გადაცემა მიმთითებლით . . . 40
3.6 myheader.h დეკლარაციის ფაილი . . . 42
3.7 myheader.c ფუნქციის იმპლემენტაცია . . . 44
3.8 main.cpp პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას. . . . 45
5.1 testglobal.cpp პროექტის ფაილი გლობალური ცვლადებით. . . . 73
5.2 util.h პროექტის ფაილი გლობალური ცვლადებით. . . . 74
5.3 util.h პროექტის ფაილი გლობალური ცვლადებით. c კოდის ჩართვაა c++ პროექტში 74
5.4 util.c პროექტის ფაილი გლობალური ცვლადებით. . . . 74
5.5 „void მიმთითებელი და void ტიპის მასივი“ . . . 78
8.1 მასივი . . . 107
8.2 მასივის ჩაწერა/წაკითხვა . . . 110

თავი 1

C – ლექსიკური ელემენტები, ცვლადები და ფუნქციები

1.1 ცვლადები და ტიპები

C ენაში გამოიყენება ლათინური ალფავიტის ასო ნიშნები – A,B ... Z

რიცხვები – 0,1,2...9

ოპერატორები +,-,=...<

პუნქტუაციის ნიშნები ;,...}

C ცვლადებს აქვთ შემდეგი ატრიბუტები: სახელი, მნიშვნელობა, ტიპი, მისამართი.

C ანსხვავებს დიდ და პატარა ასოებს. მაგ. Object და object არის სხვადასხვა ცვლადი.

ცვლადის სახელში შეიძლება იყოს რიცხვები, მაგრამ ცვლადის სახელი არ უნდა იწყებოდეს რიცხვით. მაგალითად:

`int x=4;` სწორია

`int x2=4;` სწორია

`int 2x=4;` შეცდომაა

ბაზური მონაცემთა ტიპებია: ლოგიკური,მთელი, მცოცავი მძიმით და ასო ნიშნები.

ყოველ ტიპს აქვს ზომა, რაც გვიჩვენებს თუ რამდენი მეხსიერება(ბაიტებში) გამოიყოფა თითოეული ცვლადისათვის. ეს ზომა, სხვადასხვა შეიძლება იყოს სხვადასხვა ოპერატიულ სისტემაზე. ასევე შეიძლება იყოს დამოკიდებული, კომპილატორზე. თუ მთელი ტიპის ცვლადის ზომაა N ბაიტი, მაშინ ამ ცვლადის დიაპაზონია $\{-2^N/2, 2^N/2-1\}$ ე.წ. უნიშნო (`unsigned`) ცვლადებისათვის კი — $\{0, 2^N-1\}$

1.2 ცვლადების დეკლარაცია და ინიციალიზაცია

იმისათვის, რომ შეიქმნას რაიმე ცვლადი საჭიროა მისი დეკლარირება[1]. მაგალითად:

`bool a;` ლოგიკური ცვლადი

`int i;`მთელი ცვლადი

`unsigned int j;`უნიშნო (ანუ იღებს მხოლოდ დადებით მნიშვნელობებს) მთელი ცვლადი[2].

`float x;`ცვლადი მცოცავი მძიმით

`double t;`ორმაგი სიზუსტის ცვლადი მცოცავი მძიმით

`char character;`ასო ნიშანი

როგორც ვხედავთ აქ გამოცხადებულია მხოლოდ ცვლადები, მაგრამ არაა განსაზღვრული

[1]აქ საუბარია სტატიკურ ცვლადებზე, ანუ ცვლადებზე, რომლებიც ავტომატურად იქმნება და ნადგურდება. „სტატიკური“ ამ შემთხვევაში არ ნიშნავს `static` ცვლადს. როგორ იქმნება ცვლადები მეხსიერებაში და ე.წ. დინამიურ ცვლადებს შემდგომში განვიხილავთ თავში „ცვლადები და მეხსიერება“ გვ.71.

[2]ასეთ ცვლადზე უარყოფითი მნიშვნელობის მინიჭება შეცდომაა

ცვლადების მნიშვნელობა. თუ ვეცდებით ასეთი ცვლადის დაბეჭდვას ან მისი მნიშვნელობის სხვა ცვლადზე მინიჭებას შეიძლება კომპილატორმა, პროგრამის კომპილაციისას მოგცეს შეცდომა, ან მოგვცეს გაფრთხილება.

თუ გაფრთხილებას ყურადღებას არ მივაქცევთ, პროგრამა აეწყობა მაგრამ გაშვებისას მოგვცემს შეცდომას.

ცვლადების დეკლარირების გარდა საჭიროა ცვლადების ინიციალიზაცია, ანუ რაღაც მნიშვნელობის მინიჭება.

პროგრამირების სწორი პრაქტიკაა მოხდეს ცვლადის ინიციალიზაცია მის გამოცხადებისთანავე. ზემოთ მოყვანილი შემთხვევისათვის გვექნება:

`bool a=true;` ლოგიკური ცვლადი

`int i=0;`მთელი ცვლადი

`float x=0.0;`ცვლადი მცოცავი მძიმით

`double t=0.0;`ორმაგი სიზუსტის ცვლადი მცოცავი მძიმით

`char character='\0';`ასო ნიშანი. ესაა ე.წ. ნულლ ასო.

1.3 ფუნქცია printf ეკრანზე ბეჭდვა

პროგრამის კოდი 1.1 printf მაგალითი

```
1 // Fig. 2.1: fig02_01.c
2 // A first program in C
3 #include <stdio.h>
4 // function main begins program execution
5 int main( void )
6 {
7    printf( "Welcome to C!\n" );
8 } // end function main
```

```
\t ჰორიზონტალური ტაბულაცია.
\a განგაშის ხმა.
\\ Backslash.
\" Double quote. ორმაგი ბრჭყალი
\' (single quote) ერთმაგი ბრჭყალი.
\? კითხვის ნიშანი.
\b (backspace) 1 პოზიციით მარცხნივ გადანაცვლება მოცემულ ხაზზე.
\f ახალ ლოგიკურ გვერდზე გადასვლა
\n ახალი ხაზის დასაწყისში გადასვლა
\r მოცემული ხაზის დასაწყისში გადასვლა
\v ვერტიკალური ტაბულაცია.
\xdd სიმბოლოს თექვსმეტობითი კოდი.
```

`printf()` ფუნქციაში შეიძლება იყოს რამდენიმე ფორმატირების პარამეტრი ან ერთი პარამეტრი რამდენჯერმე. განვიხილოთ შემდეგი კოდი:

```
printf("Testing...\n..1\n...2\n....3\n");
```

```
printf("char \'A\'=\t teqvbsmetobiti \\x41 \x41 \n");
printf("char \'a\'=\t teqvbsmetobiti \\x61 \x61 \n");
```

ეკრანზე გამოვა წერტილები და რიცხვები ახალ ხაზზე. შემდეგ გამოვა 'A=' შესაბამისი თექვსმეტობითი კოდი და ასო ნიშნანი "A რომლებიც წანაცვლებული იქნება 'A='-სგან ტაბულაციის ნიშნით.

ერთმაგი ბრჭყალი დავაბეჭდინეთ "\'" ბრძანებით. რადგან "\"არის სპეციალური ასო ნიშნანი, დავაბეჭდინეთ "\\" ბრძანებით ამის შემდეგ "x41"-ს ფუნქცია აღიქვამს როგორც ასო ნიშნებს. თექვსმეტობითი კოდის შესაბამისი სიმბოლოს მოცემა ხდება "\x41" ბრძანებით.

printf საშუალებით ასევე შესაძლებელია ეკრანზე გამოვიტანოთ ცვლადების მნიშვნელობები.

1.4 printf ფუნქციის ფორმატი

განვიხილოთ შემდეგი მაგალითი:

```
1  #include <stdio.h>
2   int main (void)
3   {
4    int value1, value2, sum;//სამი int ცვლადი
5
6    value1 = 50;//მნიშვნელობის მინიჭება
7    value2 = 25;
8    sum = value1 + value2;//შეკრება
9    //ეკრანზე ბეჭდვა
10   //output
11    printf ("%d+%d= %d\n", value1, value2, sum);
12
13      return 0;
14   }
```

ამ შემთხვევაში სიახლე არის %d. ასევე გაიზარდა ფუნქციის არგუმენტების რაოდენობა. აღსანიშნავია, რომ რამდენი % გვაქვს იმდენივე დამატებითი არგუმენტი უნდა მიეწოდოს printf ფუნქციას. ამ შემთხვევაში printf მუშაობს შემდეგი წესით:

ა) ბეჭდავს ასო ნიშნებს, ტექსტს

ბ) როგორც კი შეხვდება %, იმ ადგილას სადაც შეხვდა, უნდა დაბეჭდოს ცვლადი, რომელიც შეესაბამება მოცემულ %. პირველ %-ს შეესაბამება პირველი არგუმენტი "-ის დახურვის შემდეგ, მეორეს—მეორე და ა.შ.

გ) შემდეგ ისევ აგრძელებს ტექსტის და ტაბულაციის(თუ ასეთი არის რა თქმა უნდა) ბეჭდვას, რომელიც არის პირველ არგუმენტში ბრჭყალებს შორის, მანამ სანამ არ შეხვდება შემდეგი %.

დ) პროცესი მთავრდება როცა ფუნქცია "*გაივლის* " პირველ არგუმენტს (ანუ დაიბეჭდება სრულად ზემოთ მოყვანილი წესით რაც არის ბრჭყალებში — პირველ არგუმენტში)

"%" შემდეგ ასო ნიშანი, ამ შემთხვევაში "d" , არის ე.წ. ცვლადის ტიპის სპეციფიკატორი. ყოველ ცვლადს აქვს თავის სპეციფიკატორი. ამ შემთხვევაში d არის int ტიპის სპეციფიკატორი. სპეციფიკატორი უნდა იყოს ცვლადის იმ ტიპის შესაბამისი რა ცვლადსაც ვაბეჭდინებთ. %d ნაცვლად შეგვიძლია გამოვიყენოთ %i printf ფუნქციისათვის განსხვავება არაა, მაგრამ scanf ფუნქციაში:

ა) %d კითხულობს მთელ რიცხვებს მხოლოდ ათობითში.მაგრამ არა თექვსმეტობითში და რვაობითში

ბ) %i ასევე კითხულობს მთელ რიცხვებს ათობითში, მაგრამ ასევე კითხულობს თექვსმეტობითში თუ რიცხვის წინ შეხვდა "0x"და რვაობითში თუ რიცხვის წინ შეხვდა "0".

მაგალითად "033" %i წაიკითხავს როგორც "27", ხოლო %d როგორც "33".

1.4.1 sizeof ცვლადის ზომა

განვიხილოთ ქვემოთ მოყვანილი კოდი

პროგრამის კოდი 1.2 ცვლადების ზომები. sizeof მაგალითი

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  int a=0;
  float  b=0;
  long float  lb=0;
  double   c=10;
  long int li=0;
  unsigned int ui =0;
  long long int lli=0;
  unsigned long int uli=0;
  unsigned long long int ulli=0;
  long double lc=0.0;
  //long long double llc=0.0;
  char ch[]="a";
  unsigned char uch[]="a";
  printf("sizeof(int) %d\n", sizeof(a) );
  printf("sizeof(float) %d\n", sizeof(b) );
  printf("sizeof(long float) %d\n", sizeof(lb) );
  printf("sizeof(long int) %d\n", sizeof(li) );
  printf("sizeof(unsigned int) %d\n", sizeof(ui) );
  printf("sizeof(long long int) %d\n", sizeof(lli) );
  printf("sizeof(unsigned long  int) %d\n", sizeof(uli) );
  printf("sizeof(unsigned long long int) %d\n", sizeof(ulli) );
  printf("sizeof(double) %d\n", sizeof(c) );
    printf("sizeof(long double) %d\n", sizeof(lc) );

//  printf("sizeof(long long double) %d\n", sizeof(llc) );
  printf("sizeof(char) %d\n", sizeof(ch) );
```

```
30   printf("sizeof(unsigned char) %d\n", sizeof(uch) );
31
32
33   return 0;
34 }
```

[1-16] ხაზზე ხდება ცვლადების შემოტანა და ინიციალიზაცია.14-ე ხაზზე კოდი მოცემულია ე.წ. კომენტარებში. კომენტარს თუ მოვუხსნით, კომპილაციისას იქნება შეცდომა. მართლაც ასეთი ტიპი განმარტებული არაა და ასეთ ცვლადს ვერ განვმარტავთ. [17-30] ხაზზე კი ხდება ცვლადის ზომის ბეჭდვა. sizeof ფუნქცია იღებს ცვლადს და აბრუნებს ცვლადის ზომას ბაიტებში. სიტყვებს long , long long , short , unsigned , signed [1] ტიპის სპეციფიკატორები ეწოდებათ.

1.5 *C* მათემატიკური ოპერაციები

ოთხი ცნობილი ოპერაცია: +,-,*,/ ამას გარდა არის ე.წ. ნაშთის ოპერაცია— %. კერძოდ, "c=a%b;"გამოსახულებაში c არის ნაშთითი რომელიც მიიღება a/b. არის ასევე ბინარული ოპერატორები, რომლებსაც მოგვიანებით განვიხილავთ. ოპერაციის შესრულებისას მნიშვნელობა აქვს ტიპს. მაგ. მთელ ტიპს თუ ვყოფთ მთელ ტიპზე მიიღება ისევ მთელი ტიპი[2] (6/4=1). ოპერაციების შესრულების თანმიმდევრობა იგივეა რაც ვიცით ყოველდღიური ცხოვრებიდან: ჯერ სრულდება გამრავლება/გაყოფა, შემდეგ მიმატება გამოკლება. თუ გამოსახულება ან მისი ნაწილი მოთავსებულია ()-ფრჩხილებში, მაშინ სრულდება () მოთავსებული გამოსახულება ხოლო შემდეგ დანარჩენი. მაგალითად 1+3+4-3-7+4 იგივეა რაც ((((1+3)+4)-3)-7)+4 ვინაიდან + და - ოპერატორებისათვის შესრულების რიგი ერთი და იგივეა.

1.5.1 ზრდის (increment)შემცირების(decrement) ოპერატორები

ზემოთ მოყვანილი ოპერაციების გარდა არის ასევე ზრდის (++) და შემცირების (- -) ოპერატორები. მაგალითად int a=3;int q=4; ++a;--q; a იზრდება ერთი ერთეულით,b მცირდება ერთი ერთეულით. მნიშვნელობა აქვს ეს ოპერატორები წერია ცვლადის წინ თუ ცვლადის შემდეგ. თუ წერია ცვლადის წინ მაშინ სრულდება ზრდის/შემცირების ოპერატორი შემდეგ შემდეგი ოპერაცია, თუ წერია ცვლადის შემდეგ ჯერ სრულდება ოპერაცია რომელიც ტარდება ხოლო შემდეგ ზრდის/შემცირების ოპერაცია მოცემულ რიცხვზე.

ამოცანა:

```
int a, b, c =0;
a=++c;
b=c++;
printf( %d, %d, %d,a,b,++c);
```

რა დაიწერება ეკრანზე?

[1] ამ უკანასკნელს ხშირად არ იყენებენ, ვინაიდან იგულისხმება
[2] ცვლადების გარდაქმნას მოგვიანებით განვიხილავთ.

1.6 ფუნქცია scanf წაკითხვა ეკრანიდან

პროგრამის კოდი 1.3 printf და scanf. შეკრების მაგალითი

```
// Fig. 2.5: fig02_05.c
// Addition program
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int integer1; // first number to be entered by user
   int integer2; // second number to be entered by user
   int sum; // variable in which sum will be stored

   printf( "Enter first integer\n" ); // prompt
   scanf( "%d", &integer1 ); // read an integer

   printf( "Enter second integer\n" ); // prompt
   scanf( "%d", &integer2 ); // read an integer
   //scanf_s
   sum = integer1 + integer2; // assign total to sum

   printf( "Sum is %d\n", sum ); // print sum
} // end function main
```

ზემოთ განხილულ მაგალითში "%d"ნიშნავს, რომ წაკითხულ უნდა იქნას მთელი ტიპის ცვლადი, თუ გვსურს, რომ წაკითხულ იქნას float ტიპის ცვლადი მაშინ უნდა გამოვიყენოთ "%f".

ქვემოთ მოყვანილ კოდში მოყვანილია სხვადასხვა ტიპის ცვლადების ეკრანზე გამოტანა.მიაქციეთ ყურადღება რომ ერთი ტიპის ცვლადს შეიძლება ჰქონდეს სხვადასხვა სპეციფიკატორი.

პროგრამის კოდი 1.4 scanf წაკითხვა

```

scanf("%d,%d,%d\n",&value1,&value2,&value3);

```

პროგრამის კოდი 1.5 ცვლადების სპეციფიკატორები

```
#include <stdio.h>
int main (void)
{
  int integerVar = 100;
  float floatingVar = 331.79;
  double doubleVar = 8.44e+11;
```

```
7    char charVar = 'W';
8    bool boolVar = 0;
9    printf("integerVar = %i\n", integerVar);
10   printf("floatingVar = %f\n", floatingVar);
11   printf("doubleVar = %e\n", doubleVar);
12   printf("doubleVar = %g\n", doubleVar);
13   printf("charVar = %c\n", charVar);
14   printf("boolVar = %i\n", boolVar);
15   return 0;
16 }
```

%g და %e წარმოადგენენ double ტიპის ცვლადის სპეციფიკატორებს. ხშირად საჭიროა float და double ტიპის ცვლადების ჩაწერა სხვადასხვა სიზუსტით. [1] განვიხილოთ მაგალითი

პროგრამის კოდი 1.6 float და double ცვლადები სხვადასხვა სიზუსტით

```
1  #include <stdio.h>
2  int main(int argc, char **argv)
3  {
4    float  fx=23.34567812345;
5    double dx=23.34567812345;
6
7    printf ("fx=%f\n", fx);
8    printf ("fx=%.1f\n", fx);
9    printf ("fx=%.2f\n", fx);
10   printf ("fx=%.3f\n", fx);
11   printf ("fx=%.4f\n", fx);
12   printf ("fx=%.5f\n", fx);
13   printf ("fx=%.6f\n", fx);
14   printf ("shecdoma fx=%.7f\n", fx);
15   printf ("shecdoma fx=%.8f\n", fx);
16   printf ("\n-------------------------\n");
17
18   printf ("dx=%e\n", dx);
19   printf ("dx=%.1e\n", dx);
20   printf ("dx=%.2e\n", dx);
21   printf ("dx=%.3e\n", dx);
22   printf ("dx=%.6e\n", dx);
23   printf ("dx=%.12e\n", dx);
24   printf ("dx=%.13e\n", dx);
25   printf ("\n-------------------------\n");
26
27   printf ("dx=%g\n", dx);
28   printf ("dx=%.1g\n", dx);
29   printf ("dx=%.2g\n", dx);
```

[1] ამ შემთხვევაში მხოლოდ ეკრანზე გამოტანას განვიხილავთ, მაგრამ რაც ითქვა აქ სპეციფიკატორების შესახებ სამართლიანია ფაილში ჩაწერის შემთხვევაშიც.

```
30    printf ("dx=%.3g\n", dx);
31    printf ("dx=%.6g\n", dx);
32    printf ("dx=%.12g\n", dx);
33    printf ("dx=%.13g\n", dx);
34    printf ("\n------------------------\n");
35
36    printf ("dx=%lg\n", dx);
37    printf ("shecdoma dx=%Lg\n", dx);
38    printf ("\n------------------------\n");
39    return 0;
40  }
```

ცხრილი 1.1: printf და ტიპების სპეციფიკატორები

ტიპი	მაგალითი	სპეციფიკატორი
char	'a' '\n'	%c
char მასივი[1]	"this is C string"	%s
bool	0,1	%i, %u
short int		%hi %hx %ho
unsigned short int		%hu,%hx, %ho
int	-123,234,0xFFAD, 0123	%i,%x, %o
unsigned int	12u,123U,0xafu	%u, %x, %o
long int	12L,-2000,0xaffdL	%li, %lx, %lo
unsigned long int	12UL,100ul,0xaffdUL	%lu, %lx, %lo
long long int	12LL,100ll,0xaffdLL	%lli, %llo, %llx
unsigned long long int	12ULL,100ull,0xaffdULL	%llu, %llo, %llx
float	12.345f,3.4e-7f,0x1.5p10,0x1P-1	%f, %e, %g ,%a
double	12.345,3.4e-7,0x1.5p10,0x1P-1	%f, %e, %g ,%a
long double	12.345,3.4e-71	%Lf, %Le, %Lg

ამას გარდა printf ნაცვლად შეძლება გამოყენებულ იქნას puts ტექსტური ინფორმაციის ეკრანზე გამოსატანად. განსხვავებით printf საგან ამ შემთხვევაში არაა ახალ ხაზზე გადას-

ვლის ბრძანების მითითება- puts გამოაქვს ტექსტი ახალ ხაზზე. ანუ

```
printf("Hello.\n");
```

იგივეა, რაც

```
puts("Hello.");
```

ამოცანები

1. დაწერეთ პროგრამა, რომელიც ეკრანზე დაბეჭდავს ქვემოთ მოყვანილ ტექსტს:
   C ენაში მნიშვნელოვანია პატარა ასოები.
   main არის ფუნქცია, რომელიც ასრულებს პროგრამას.
   გახსნილ და დახურულ ფრჩხილებში '', '' მოთავსებული ბრძანებები ქმნიან ბრძანებების ბლოკს.
   ყველა ბრძანება უნდა მთავრდებოდეს წერტილმძიმით ';'.
2. რას აკეთებს ქვემოთ მოყვანილი პროგრამა?

```
#include <stdio.h>
int main (void)
{ printf("Testing...");
  printf("....1");
  printf("...2");
  printf("\n");
  return 0;
}
```

3. დაწერეთ პროგრამა, რომელიც 123 აკლებს 23 და ეკრანზე გამოაქვს ოპერაცია და მისი შედეგი.
4. იპოვნეთ შეცდომები და გაასწორეთ ქვემოთ მოყვანილი კოდი

```
#include <stdio.h>
main(Void)
{Chapter 3 Compiling and Running Your First Program
  INT sum;
  sum = 25 + 37 - 19
  printf ("The answer is %i\n" sum);
  return 0;
}
```

[1] უნდა მთავრდებოდეს "\0"-ით

5. რა შედეგს მოგვცემს ქვემოთ მოყვანილი პროგრამა?

```
#include <stdio.h>
int main (void)
{
  int answer, result;
  answer=100;
  result= answer - 10;
  printf("shedegi aris %d\n",result+5);
  return 0;
}
```

6. დაწერეთ პროგრამა, რომელიც დაბეჭდავს სხვადასხვა ტიპის ცვლადებს სხვადასხვა სპეციფიკატორების გამოყენებით(იხ.ცხრილი 1.1)

არის კიდევ ერთი ფორმატი "%g"რაც გამოიყენება float და double ტიპის ცვლადების ჩაწერისას. ამ შემთხვევაში "%e"და "%f"არჩეულ იქნება ის, რომლის ჩანაწერიც უფრო მოკლეა.

1.7 პრეპროცესორის დირექტივები

ჩვენ შეგვხვდა გამოსახულება #include <stdio.h> ეს არის ბრძანება კომპილატორისათვის ჩართოს კოდში ფაილი stdio.h და ეს ფაილი უნდა მოძებნოს სისტემურ დირექტორიებში(ამას აღნიშნავს <>). ხშირად დაგვჭირდება შემდეგი #include "filename.h". ესეც ნიშნავს, რომ კომპილაციისას ჩართული უნდა იყოს filename.h, მაგრამ ამ შემთხვევაში ამ ფაილის ძებნა წარმოებს მიმდინარე (ანუ სადაც წერია ფაილი, რომელშიც ხდება ჩართვა) დირექტრორიაში. ჩართვის ბრძანებას შეიძლება ჰქონდეს სხვა სახეც #include "../filename.h" ან #include "dirname/filename.h". პირველი ნიშნავს, რომ ფაილი აღებულ უნდა იყოს ერთი დირექტორიით მაღლა დირექტორიაში, მეორე კი მიმდინარე დირექტორიაში არსებულ 'dirname' დირექტორიიდან.

1.7.1 დირექტივა #define

დირექტივა #define და სხვა დირქტივები, როგორც წესი იწერება ფაილის დასაწყისში #include დირექტივების შემდეგ. დირექტივა #define განსაზღვრავს სიმბოლურ მუდმივას. მაგალითად

#define R 100

#define PI 3.14159

აქ შემოტანილია ორი რიცხვი (და არა ცვლადი) და კომპილაციის დროს ეს რიცხვები „ჩაჯდება“ იმ სიმბოლოების ადგილას, რომლებიც განმარტებულია ამ რიცხვებით. ზოგადად, # განმარტებული ბრძანებები სრულდება კოდის კომპილაციისას. ზემოთ მოტანილი მაგალითებისათვის PI ცვლადის შექმნა კი არ ხდება არამედ კომპილაციისას სადაც წერია PI გამოყენებულ იქნება მისი მნიშვნელობა. #define დირექტივა ხშირად გამოიყენება ე.წ. საჭირო ბიბლიოთეკების ჩასართავად, ასევე სხვადასხვა ტიპების განსამარტავად, შესამოწმებლად თუ რა ოპერატიული სისტემა ან რა კომპილატორია გამოყენებული და ა.შ. განვიხილოთ ქვემოთ მოყვანილი კოდი:

```
1  #include <stdio.h>
2  #define PI 3.141592653589793
3  int main (void)
4  {
5   double radius=0.0;
6   printf("\n %s\n\n%s ",
7        "creciris fartobis gamotvla.",
8        "shemoitanet radiusi:  ");
9   scanf( "%lf", &radius);
10  printf("\n %s\n %s %.2f %s %.2f %s %.2f\n %s %.5f\n\n",
11         "fartobi = PI * radius * radius",
12         "         = ",PI,"*",radius,"*",radius,
13         "         = ",PI*radius*radius);
14  return 0;
15 }
```

ეკრანზე გამოვა შემდეგი:

```
 creciris fartobis gamotvla.

 shemoitanet radiusi:   2

  fartobi = PI * radius * radius
          =  3.14 * 2.00 * 2.00
          =  12.56637
  Hit ENTER to continue...
```

მაგრამ კალკულატორით თუ გამოვთვლით იგივეს პასუხი იქნება 12.5663706144 ეს იმიტომ, რომ პროგრამაში გამოყენებულია დამრგვალება— შეტანისას მძიმის შემდეგ ორი ციფრით %.2f , და გამოტანისას—მძიმის შემდეგ ხუთი ციფრით %.5f . თუ შევიტანთ რადიუსის სხვა მნიშვნელობას მაგ. "3.119455" განსხვავება კიდე უფრო დიდი იქნება.

თავი 2

გადაწყვეტილების მიღება და ციკლები

ხშირად სხვადასხვა შემთხვევებისას საჭიროა, რომ პროგრამამ მიიღოს სხვადასხვა გადაწყვეტილება/შეასრულოს სხვადასხვა ბრძანება. მაგალითად როცა ერთი რიცხვი მეტია მეორეზე მაშინ სრულდება ერთი ბრძანება, თუ არა და – მეორე. ან თაგუნას მარჯვენა ღილაკის დაჭერისას ერთი, მარცხენა დაჭერისას სხვა; ან "როცა დაჭერილია Shift კლავიატურის ღილაკი და დაჭერილ მდგომარეობაშია თაგუნას მარცხენა ღილაკი მაშინ თაგუნას მოძრაობისას კეთდება....."და ა.შ. ასეთმა ბრძანებებმა, რომ იმუშაოს საჭიროა რაღაც პირობებზე შემოწმება და შემდეგ, პასუხიდან გამომდინარე საჭირო ბრძანების შესრულება. ასეთი ბრძანებებია if,if else, switch.

იმისათვის, რომ ამ ბრძანებებმა იმუშაოს საჭიროა ე.წ. შედარების ოპერატორები.

2.1 შედარების(პირობითი) ოპერატორები (<,>, <=, >=, ==, !=)

<,>, <=, >= ამ ოპერატორების აზრი გასაგებია.

== ამოწმებს მის მარჯვნივ და მარცხნივ მდგომი სიდიდეების ტოლობას.

!= ამოწმებს მის მარჯვნივ და მარცხნივ მდგომი სიდიდეების უტოლობას.

სხვაგვარად, რომ ვთქვათ != მოგვცა მნიშვნელობა "მცდარია"ეს ნიშნავს, რომ ტოლი ყოფილა.

იმის მიხედვით თუ რა გვსურს ან რომელი მოგვწონს შეგვიძლია გამოვიყენოთ == ან !=

2.2 'if' და 'if ... else' ბრძანება

if შეიძლება გამოყენებულ იქნას სხვადასხვა ფორმით, იმის მიხედვით თუ რამდენად კომპლექსური პირობებია შესამოწმებელი. მარტივი if პირობა

```
1 if('გამოსახულება') 'ბრძანება1';
```

ამ შემთხვევაში თუ 'გამოსახულება' ჭეშმარიტია სრულდება 'ბრძანება1'; და შემდეგ გრძელდება პროგრამის შესრულება.

```
1 if('გამოსახულება')
2   {
3     'ბრძანება1';
4     'ბრძანება2';
5     'ბრძანებაN';
6   }
```

ამ შემთხვევაშიც მხოლოდ მაშინ როცა 'გამოსახულება' ჭეშმარიტია სრულდება '{ }' ბლოკში მოთავსებული ბრძანებები. და გრძელდება პროგრამის შესრულება.

```
if('გამოსახულება')
{
  'ბრძანება1';
  'ბრძანება2';
  ....
  'ბრძანებაN';
}
else
{
  'ბრძანება11';
  'ბრძანება12';
  ....
  'ბრძანება1N';
}
```

ამ შემთხვევაშიც მხოლოდ მაშინ როცა 'გამოსახულება' ჭეშმარიტია სრულდება:

```
{
'ბრძანება1';
'ბრძანება2';
....
'ბრძანებაN';
}
```

ხოლო თუ 'გამოსახულება' მცდარია სრულდება

```
{
'ბრძანება11';
'ბრძანება12';
....
'ბრძანება1N';
}
```

ბლოკში მოთავსებული ბრძანებები. და გრძელდება პროგრამის შესრულება:

```
#include< stdio.h>
int  main( void)
{   int x,y;
    x=15;
    y=13;
    if (x > y )
    {
      printf("x > y");
    }
  return 0;
}
```

if ... else პირობების მაგალითი:

```
#include< stdio.h>
int  main( void)
{
  int x,y;
  x=15;
  y=18;
  if (x > y )
  {
    printf("x > y");
  }
  else
  {
    printf("y > x");
  }
  return 0;
}
```

შესაძლებელია გვქონდეს ერთმანეთში ჩადგმული რამდენიმე if ... else

```
if( გამოსახულება )
{
if( გამოსახულება1 )
{
ბრძანებების_ბლოკი1;
}
else
{
ბრძანებების_ბლოკი2;
}
}
else
{
ბრძანებების_ბლოკი3;
}
```

მაგალითი:

```
#include< stdio.h>
int main( void )
{
  int a,b,c;
  printf("enter 3 number");
  scanf("%d%d%d",&a,&b,&c);
  if(a>b)
  {
```

```
9       if( a > c)
10      {
11        printf("a is greatest");
12      }
13      else
14      {
15        printf("c is greatest");
16      }
17    }
18    else
19    {
20      if( b> c)
21      {
22        printf("b is greatest");
23      }
24      else
25      {
26        printf("c is greatest");
27      }
28    }
29  return 0;
30  }
```

if .. else if კონსტრუქციის მაგალითი:

```
if(piroba 1)
{
statement-block1;
}
else if(piroba 2)
{
statement-block2;
}
else if(piroba 3 )
{
statement-block3;
}
else
default-statement;
```

ასეთ კონსტრუქციაში სრულდება მხოლოდ ის ბლოკი `'piroba n'` ჭეშმარიტია. თუ არცერთი არაა ჭეშმარიტი სრულდება `'else'` ანუ `'default-statement'`. if შეიძლება იყოს რამდენიმე პირობა:

```
1  #include< stdio.h>
2  int main( void )
```

```
3  {
4    int a;
5    printf("shemoitane ricxvi");
6    scanf("%d",&a);
7    if( a%5==0 && a%8==0)
8    {
9      printf("iyofa 5 ze da 8-ze");
10   }
11   else if( a%8==0 )
12   {
13     printf("iyofa 8-ze");
14   }
15   else if(a%5==0)
16   {
17     printf("iyofa 5-ze");
18   }
19   else
20   {
21     printf("ar iyofa arc 5-ze arc 8-ze");
22   }
23   return 0;
24 }
```

პროგრამის კოდი 2.1 printf მაგალითი და შედარების ოპერატორები

```
1  // Fig. 2.13: fig02_13.c
2  // Using if statements, relational
3  // operators, and equality operators
4  #include <stdio.h>
5
6  // function main begins program execution
7  int main( void )
8  {
9     int num1; // first number to be read from user
10    int num2; // second number to be read from user
11
12    printf( "Enter two integers, and I will tell you\n" );
13    printf( "the relationships they satisfy: " );
14
15    scanf( "%d%d", &num1, &num2 ); // read two integers
16
17    if ( num1 == num2 ) {
18       printf( "%d is equal to %d\n", num1, num2 );
19    } // end if
20
21    if ( num1 != num2 ) {
```

```
22        printf( "%d is not equal to %d\n", num1, num2 );
23     } // end if
24
25     if ( num1 < num2 ) {
26        printf( "%d is less than %d\n", num1, num2 );
27     } // end if
28
29     if ( num1 > num2 ) {
30        printf( "%d is greater than %d\n", num1, num2 );
31     } // end if
32
33     if ( num1 <= num2 ) {
34        printf( "%d is less than or equal to %d\n", num1, num2 );
35     } // end if
36
37     if ( num1 >= num2 ) {
38        printf( "%d is greater than or equal to %d\n", num1, num2 );
39     } // end if
40  } // end function main
```

2.3 ბრძანება switch ... case

'if' ოპერატორებით შესაძლებელია სხვადასხვა ბრძანებების ბლოკის გაშვება სხვადასხვა შემთხვევისათვის, მაგრამ კომპლექსური პირობებისთვის შეიძლება ბევრმა ერთმანეთში ჩადგმულმა პირობებმა პროგრამული კოდი გახადოს რთულად გასაგები.

არის ამოცანები, 'თუ/თუ არა' კონსტრუქციის ნაცვლად რაიმე ცვლადის ან ცვლადების მდგომარეობა შეიძლება აღიწეროს ერთი მთელი ტიპის ცვლადით და იმის მიხედვით თუ რისი ტოლია ამ ცვლადის მნიშვნელობა შესრულდეს სხვადასხვა ბრძანება(ან ბრძანებების ბლოკი). ასეთი კონსტრუქცს საშუალებას იძლევა switch... case. ბრძანების სინტაქსი:

```
1   switch (n) {
2     case constant1:
3     brdazneba romelic sruldeba roca n=constant1;
4     break;
5     case constant2:
6       brdazneba romelic sruldeba roca n=constant2;
7     break;
8     .
9     .
10    .
11    default:
12      brdazneba romelic sruldeba roca n ar akmayofilebs
13      arc ert zemot agceril pirobas;
14  }
```

n შეიძლება იყოს ან ასო ნიშანი ან მთელი ტიპის ცვლადი. როცა n ტოლია ერთერთი 'constant' მაშინ სრულდება შესაბამისი ბრძანება (ან ბრძანებები). ბრძანების შემდეგ წერია break, რაც

ნიშნავს, რომ უნდა შეწყდეს switch ბრძანება. თუ რომელიმე ან რამდენიმე case არაა მითითებული break მაშინ case მუშაობს შემდეგი სახით: სრულდება ის შემთხვევა როცა n=constant. ამის შემდეგ შესრულდება ყველა დანარჩენი case სანამ არ შეხვდება ბრძანება break.

განვიხილოთ მაგალითი

```
#include <stdio.h>
int main(int argc, char* argv[])
{
  int n=3;
  switch (n)
  {
  case 1:
    printf("n=1\n");
    break;
  case 2:
    printf("n=2\n");
  case 3:
    printf("n=3\n");
  case 4:
    printf("n=4\n");
  case 5:
    printf("n=5\n");
    break;
  default:
    printf("\n n is not defined");
  }
  return 0;
}
```

დავალება: შექმენით პროექტი და გაარჩიეთ ქვემოთ მოყვანილი კოდი:

პროგრამის კოდი 2.2 „ელემენტარული კალკულატორი“

```
# include <stdio.h>
int main(void) {
    char o;
    float num1,num2;
    printf("Select an operator either + or - or * or / \n");
    scanf("%c",&o);
    printf("Enter two operands: ");
    scanf("%f%f",&num1,&num2);
    switch(o) {
        case '+':
            printf("%.1f + %.1f = %.1f",num1, num2, num1+num2);
            break;
        case '-':
            printf("%.1f - %.1f = %.1f",num1, num2, num1-num2);
            break;
```

```
16          case '*':
17              printf("%.1f * %.1f = %.1f",num1, num2, num1*num2);
18              break;
19          case '/':
20              printf("%.1f / %.1f = %.1f",num1, num2, num1/num2);
21              break;
22          default:
23              /* If operator is other than +, -, * or /, */
24              /*error message is shown */
25              printf("Error! operator is not correct");
26              break;
27      }
28      return 0;
29  }
```

ქვემოთ მოყვანილია მაგალითი switch ბრძანების გამოყენებით.

პროგრამის კოდი 2.3 switch case მაგალითი

```
1   // case_test.cpp : Defines the entry point for the console application.
2   //
3   //switch case test programm case_test.c
4
5   #include <stdio.h>
6
7
8   int main(int argc, char* argv[])
9   {
10    float T=0;
11    int N=-1;
12    float Tgrad=0;
13    float Tfaren=0;
14    puts("shemoitanet 1 tu temratura aris gradusebshi");
15    puts(" da 0  Temperatura aris farenheitebshi");
16    scanf("%d",&N);
17    printf("N= %d\n",N);
18    puts("shemoitanet Temperatura");
19    scanf("%f",&T);
20    printf("T= %f\n",T);
21    switch (N)
22  {
23  case 0:
24    puts(" N=0 temperatura shemotanil iqna farenheitebshi");
25    puts(" gadamyavs gradusebshi");
26    Tgrad=(T-32)/1.8;
27    printf("Temperatura gradusebshi = %f",Tgrad);
28    break;
29  default:
```

```
30   puts(" N=1 temperatura shemotanil iqna gradusebshi");
31   puts(" gadamyavs farenheitebshi");
32   float Tfaren=T*1.8+32;
33   printf("Temperatura farenheitebshi = %f",Tgrad);
34
35   }
36   return 0;
37 }
```

კითხვა:

რა პრობლემები აქვს ზემოთ მოყვანილ კოდს?

დავალება:

1.შეასწორეთ ზემოთ მოყვანილი კოდი.

2.დაუმატეთ კელვინის ერთეულებში შეყვანა.

3.შეამოწმეთ უარყოფით ტემპერატურაზე ცელსიუსის და ტემპერატურის შემთხვევაში (ნაკლები არ უნდა იყოს 273 ზე)

4. დაუმატეთ შემოწმებები ფარენჰეიტის და კელვინის შემთხვევაში.

ქვემოთ მოყვანილია კვადრატული განტოლების ამოხსნა:

პროგრამის კოდი 2.4 კვადრატული განტოლება

```
1  #include <stdio.h>
2  #include <math.h>
3  int main(int argc, char* argv[])
4  {
5  /*ა*/
6    double x1=0;
7    double x2=0;
8    double a=1;
9    double b=2;
10   double c=4;
11   double d=b*b-4*a*c;//დეტერმინანტი
12   if(d<0){
13     printf("D<0 araa amoxsna\n");
14     return 0;
15      }
16   x1=(-b+sqrt(d))/(2*a);
17   x2=(-b-sqrt(d))/(2*a);
18   printf("aris 2 fesvi:\n");
19   printf("x1=%g:\n x2=%g\n",x1,X2);
20   return 0;
21 }
```

2.4კოდის 11 ხაზზე ითვლება დეტერმინანტი

დავალება:

შეცვალეთ ზემოთ მოყვანილი კოდი ისე, რომ შესაძლებელი იყოს კვადრატული განტოლების კოეფი-

ციენტების შეტანა კონსოლიდან.

ამოცანა:

დაწერეთ პროგრამა, რომელიც კონსოლიდან მიიღებს მთელი ტიპის დადებით რიცხვს. ეს იქნება წამები. გადაიყვანეთ მირებული რიცხვი წუთებში, საათებში, დღეებში. დაბეჭდინეთ ეკრანზე შემდეგი ფორმატით:

shemotanilia X cami.

X cami = A dge, B saati, C suti, D cami

2.4 ციკლები

არის შემთხვევები, როცა ბრძანება ან ბრძანებების სიმრავლე უნდა შესრულდეს რამდენჯერმე. შეიძლება ერთი და იგივე ცვლადზე შეიძლება რამდენიმე ცვლადზე. უფრო მეტიც შეიძლება გვჭირდებოდეს, რომ რაიმე ოპერაციები(მაგალითად რაღაცის გამოთვლა: ტემპერატურა, კოორდინატი, ელექტრული ველი, დენი, ძაბვა...) სრულდებოდეს მანამ სანამ არ შესრულდება რაღაც პირობა. ასეთი ამოცანებისთვის გამოიყენება ციკლები.

2.4.1 for ციკლი

for ციკლს აქვს შემდეგი სინტაქსი

```
for(ინიციალიზაცია; პირობის შემოწმება;განახლება)
{
        შესრულებადი კოდი;
}
```

ინიციალიზაციის ბრძანება სრულდება მხოლოდ ერთხელ. შემდეგ ხდება პირობის შემოწმება თუ პირობა ჭეშმარიტია სრულდება 'შესრულებადი კოდი;' შემდეგ ხდება 'განახლება' ნაწილის შესრულება და კვლავ ხდება პირობის შემოწმება. თუ პირობა არ სრულდება წყდება ციკლის შესრულება და პროგრამა შეასრულებს შემდეგ ბრძანებებს.

ქვემოთ მოყვანილი კოდი ითვლის n რიცხვის ჯამს

პროგრამის კოდი 2.5 „n რიცხვის ჯამი"

```
1  #include <stdio.h>
2  int main(void){
3      int n, count, sum=0;
4      printf("shemoitanet  n mnishvneloba.\n");
5      scanf("%d",&n);
6      for(count=1;count<=n;++count) //for cikli ckdeba rocaf count>n
7      {
8          sum+=count;     /* es igivea rac sum=sum+count */
9      }
10     printf(" %d ricxvis jami=%d",n,sum);
11     return 0;
12 }
```

ქვემოთ მოყვანილია გამრავლების ცხრილი:

პროგრამის კოდი 2.6 „გამრავლების ცხრილი"

```
/* C program to find multiplication table up to 10. */
#include <stdio.h>
int main()
{
    int n, i;
    printf("shemoitanet raime mTeli ricxvi: ");
    scanf("%d",&n);
    for(i=1;i<=10;++i)
    {
        printf("%d * %d = %d\n", n, i, n*i);
    }
    return 0;
}
```

ამოცანა:

დაწერეთ პროგრამა, რომელიც კონსოლზე გამოიტანს გამრავლების ცხრილს $(1-9)\bigotimes(1-9)$შენიშვნა: გამოიყენეთ printf, for ციკლი და ტაბულაციის ელემენტები '\t', '\n'

მაგალითი: რიცხვის გამყოფი

ქვემოთ მოყვანილია პროგრამის კოდი, რომელიც ითვლის რიცხვის გამყოფებს:

პროგრამის კოდი 2.7 „რიცხვის გამყოფი“

```
#include <stdio.h>
int main()
{
  int n,i;
  printf("shemoitanet mteli dadebiti ricxvi: ");
  scanf("%d",&n);
  printf("%d -s gamyofebia: ", n);
  for(i=1;i<=n;++i)
  {
      if(n%i==0)
         printf("%d ",i);
  }
  return 0;
}
```

მაგალითი: ორი რიცხვის გამყოფი

ქვემოთ მოყვანილია პროგრამის კოდი, რომელიც ითვლის ორი რიცხვის საერთო გამყოფს:

პროგრამის კოდი 2.8 „საერთო გამყოფი“

```
#include <stdio.h>
int main()
{
  int num1, num2, min,i;
```

```
5     printf("Enter two integers: ");
6     scanf("%d %d", &num1, &num2);
7     min=(num1>num2)?num2:num1; /* min inaxeba num1 */
8     /*da num2 -s shoris minimaluri mnishvneloba */
9     for(i=min;i>=1;--i)
10    {
11        if(num1%i==0 && num2%i==0)
12        {
13            printf("%d da %d -s saerto gamyofia %d", num1, num2,i);
14             break;
15        }
16    }
17    return 0;
18 }
```

ისევე როგორც if შემთხვევაში ციკლების შემთხვევაშიც შეიძლება გვქონდეს ე.წ. „ჩადგმული ციკლები"

პროგრამის კოდი 2.9 „ორმაგი ციკლი"

```
1  #include <stdio.h>
2  int main()
3  {  int i,j;
4    for(i=1;i<5;i++)
5    {   printf("\n");
6        for(j=i;j>0;j--)
7        {
8              printf(" %d",j);
9        }
10   }
11   return 0;
12 }
```

for ციკლში თუ არ გვაქვს ციკლის შეწყვეტის პირობა ან პირობა, რომელიც არასდროს არ შესრულდება, მაშინ ციკლი იქნება უსასრულო. იმისათვის, რომ ციკლი შეწყდეს საჭიროა ბრძანება break.

2.4.2 ბრძანება break

ბრძანება break უკვე შევხვდით switch განხილვისას. ეს ბრძანება ასევე ხშირად გამოიყენება if, switch კონსტრუქციებში და ციკლებში. break ნიშნავს, რომ მოცემულ კონსტრუქციის(if, switch, for, while) შესრულება წყდება და პროგრამა გადადის შემდეგი მრძანების შესრულებაზე. გაარჩიეთ ქვემოთ მოყვანილი კოდი

```
1  /* C program to demonstrate the working of break */
2  /*statement by terminating a loop, if user inputs negative number*/
3  # include <stdio.h>
4  int main(){
5     float num,average,sum;
```

```
6     int i,n;
7     printf("Maximum no. of inputs\n");
8     scanf("%d",&n);
9     for(i=1;i<=n;++i){
10        printf("Enter n%d: ",i);
11        scanf("%f",&num);
12        if(num<0.0)
13        break;                    //for loop breaks if num<0.0
14        sum=sum+num;
15 }
16   average=sum/(i-1);
17   printf("Average=%.2f",average);
18   return 0;
19 }
```

ზემოთ მოყვანილ კოდს აქვს პრობლემა როცა i=1.

ამოცანა:

შეასწორეთ ზემოთ მოყვანილი კოდი $i = 1$ შემთხვევისათვის

2.4.3 while ციკლი

while ციკლს აქვს შემდეგი სინტაქსი

```
while( პირობის შემოწმება)
{
        შესრულებადი კოდი;
}
```

ანუ სანამ 'პირობის შემოწმება' არის ჭეშმარიტი სრულდება 'შესრულებადი კოდი'. თუ პირობა ყოველთვის ჭეშმარიტია მაშინ გვაქვს უსასრულო while ციკლი. ციკლი შეიძლება შევწყვიტოთ break ბრძანების გამოყენებით ისევე როგორც for ციკლის შემთხვევაში. ამ მეთოდის გარდა ასევე გამოიყენება მთვლელი ციკლის კონტროლისათვის. განვიხილოთ მაგალითები:

```
1  // Fig. 4.1: fig04_01.c
2  // Counter-controlled repetition
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     unsigned int counter = 1; // initialization
9
10    while ( counter <= 10 ) { // repetition condition
11       printf ( "%u\n", counter ); // display counter
12       ++counter; // increment
13    } // end while
14 } // end function main
```

პროგრამა მარტივია: სანამ $counter <= 10$ დაბეჭდე $counter$ ცვლადის მნიშვნელობა და შემდეგ გაზარდე ერთი ერთეულით. ამ შემთხვევაში შეგვეძლო გამოგვეყენებინა for ციკლიც ვინაიდან ვიცით ზედა საზღვარი(ამ შემთხვევაში 10).

მაგრამ არის შემთხვევები როცა წინასწარ არ ვიცით როდის შეიძლება შეწყდეს ციკლი და ციკლი შეწყვეტის პირობა გამოითვლება მათემატკურად (მაგალითად სანამ რაღაც სიდიდის ცდომილება არის 0.001 ზე მეტი უნდა დავითვალოთ ეს სიდიდე) ან დამოკიდებულია მომხმარებლის მიერ შეტანილ რაიმე მნიშვნელობაზე.

ქვემოთ მოყვანილ მაგალითში ციკლი წყდება როცა მომხმარებელი აკრეფს $Ctrl - Z$. ამ მაგალითში მომხმარებლებს შეჰყავს ნიშნები (A-F) პროგრამა ითვლის და ბეჭდავს თითოეული ნიშნის რაოდენობას.

```
1  // Fig. 4.7: fig04_07.c
2  // Counting letter grades with switch
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     int grade; // one grade
9     unsigned int aCount = 0; // number of As
10    unsigned int bCount = 0; // number of Bs
11    unsigned int cCount = 0; // number of Cs
12    unsigned int dCount = 0; // number of Ds
13    unsigned int fCount = 0; // number of Fs
14
15    puts( "Enter the letter grades."  );
16    puts( "Enter the EOF character to end input."  );
17
18    // loop until user types end-of-file key sequence
19    while ( ( grade = getchar() ) != EOF ) {
20
21       // determine which grade was input
22       switch ( grade ) { // switch nested in while
23
24          case 'A': // grade was uppercase A
25          case 'a': // or lowercase a
26             ++aCount; // increment aCount
27             break; // necessary to exit switch
28
29          case 'B': // grade was uppercase B
30          case 'b': // or lowercase b
31             ++bCount; // increment bCount
32             break; // exit switch
33
34          case 'C': // grade was uppercase C
35          case 'c': // or lowercase c
```

```
36              ++cCount; // increment cCount
37              break; // exit switch
38
39           case 'D': // grade was uppercase D
40           case 'd': // or lowercase d
41              ++dCount; // increment dCount
42              break; // exit switch
43
44           case 'F': // grade was uppercase F
45           case 'f': // or lowercase f
46              ++fCount; // increment fCount
47              break; // exit switch
48
49           case '\n': // ignore newlines,
50           case '\t': // tabs,
51           case ' ': // and spaces in input
52              break; // exit switch
53
54           default: // catch all other characters
55              printf( "%s", "Incorrect letter grade entered." );
56              puts( " Enter a new grade." );
57              break; // optional; will exit switch anyway
58        } // end switch
59     } // end while
60
61     // output summary of results
62     puts( "\nTotals for each letter grade are:" );
63     printf( "A: %u\n", aCount ); // display number of A grades
64     printf( "B: %u\n", bCount ); // display number of B grades
65     printf( "C: %u\n", cCount ); // display number of C grades
66     printf( "D: %u\n", dCount ); // display number of D grades
67     printf( "F: %u\n", fCount ); // display number of F grades
68  } // end function main
```

while ციკლის მსგავსია ციკლი do..while,რომელსაც აქვს შემდეგი სინტაქსი:

```
do
{
  შესრულებადი კოდი;
}while( პირობის შემოწმება)
```

როგორც ვხედავთ აქ ერთხელ მაინც სრულდება რაღაც ოპერაციები და შემდეგ ხდება პირობის შემოწმება.

2.4.4 ბრძანება continue

ციკლებთან მუშაობისას ასევე ხშირად გამოიყენება ბრძანება `continue`. `break` ბრძანება თუ წყვეტდა ციკლს `continue` შემთხვევაში ხდება ციკლის დასაწყისში გადასვლა.

2.5 ჩამონათვალი(დანომრილი) ტიპი enum

ვთქვათ ამოცანა არის ასეთი: გვჭირდება ისეთი მონაცემტა ტიპი რომლის მოცემის შემდეგ მოცემული ტიპის ცვლადმა შეიძლება მიიღოს მხოლოდ რაღაც წინასწარ ჩვენთვის სასურველი მნიშვნელობები. მაგალითად გვინდა, რომ გვქონდეს ცვლადი myColor, რომელმაც შეიძლება მიიღოს მხოლოდ შემდეგი მნიშვნელობები: red, green, blue ასეთი რამ შესაძლებელია ჩამონათვალი ტიპის(ან დანომრილი ტიპის – enumerated type) შემოტანით. ამ ტიპის სპეციალური სიტყვა ენაში არის enum. ზემოთ მოყვანილი მაგალითისთვის შეგვიძლია შემოვიტანოთ შემდეგი ჩამონათვალი ტიპი:

```
enum MainColors {red, green, blue};
```

ეს ბრძანება განმარტავს ტიპს MainColors და ამ ტიპის ცვლადმა შეიძლება მიიღოს მხოლოდ მნიშვნელობები red, green ან blue. როცა გვსურს ცვლადის შემოტანა ისევ ვიყენებთ სიტყვებს enum MainColors და ცვლადების სახელები. მაგალითად

```
enum MainColors color1, color2, myColor;
```

ხოლო პროგრამაში შეგვიძლია შევასრულოთ შემდეგი:

```
color1=green; color2=red; myColor=color2;
```

ამ შემთხვევაში ჩვენ შემოვიყვანეთ MainColors ტიპის სამი ცვლადი. ჩამონათვალი ტიპის გამოყენება ბუნებრივია როცა ვმუშაობთ თარიღებთან. მაგალითად:

```
enum Month{Jan,Feb,Mar,Apr,May,Jun...};
enum WeekDay {Sun,Mon,Wed,Thr...};
enum Direction {Left,Right,Up,Down};
```

[1]

როგორ მუშაობს ასეთი ცვლადი? კომპილატორი ჩამონათვალის ტიპის განმარტებაში არსებულ იდენტიფიკატორებს (Jan,Feb,Mar...) აღიქვამს როგორც მთელი ტიპის რიცხვებს, რომელთაგან პირველის მნიშვნელობაა 0, მეორისა -1, მესამისა -2 და ა.შ. მაგალითად თუ კოდში გვაქვს

```
1 enum  WeekDay day;
2 day=Mon;
```

ცვლად day მნიშვნელობა იქნება 1;

თუ გვსურს, რომ მნიშვნელობები იყოს რაღაც სხვა შეგვიძლია გავაკეთოთ შემდეგი:

```
enum Direction {Left,Right,Up=11,Down};
```

ამ შემთხვევაში :Left=0,Right=1,Up=11,Down=12. განვიხილოთ ქვემოთ მოყვანილი კოდი

```
1 #include <stdio.h>
2 int main (void)
3 {
4 enum  month  { january = 1,february,march,april, may,june,
5 july, august, september, october, november, december };
6 enum  month  aMonth;
7 int          days;
8 printf "(Enter month number: ");
```

[1] ცხადია '...' მხოლოდ სიმოკლისთვისაა და არა ცვლადის სახელი.

```
9  scanf ‘‘("%i, &aMonth);
10 switch (aMonth ) {
11 case january:
12 case march:
13 case may:
14 case july:
15 case august:
16 case october:
17 case december:
18 days = 31;
19 break;
20 case april:
21 case june:
22 case september:
23 case november:
24 days = 30;
25 break;
26 case february:
27 days = 28;
28 break;
29 default:
30 printf ‘‘(bad month number\"n);
31 days = 0;
32 break;
33 if ( days != 0 )
34 printf ‘‘(Number of days is %i\"n, days);
35 if ( amonth  == february )
36 printf ‘‘(...or 29 if ’its a leap year\"n);
37 return 0;
38 }
```

მე-4 ხაზზე შემგვყავს ტიპი month ხოლო მე-6 ხაზზე ცხადდება month ტიპის ცვლადი amonth. month ტიპის ცვლადის განმარტებისას პირველი ელემენტი january ინიციალიზდება როგორც 1 (january = 1), ამიტომ შემდეგი ელემენტის მნიშვნელობა იქნება 2 და ა.შ. მე-6 ხაზზე ხდება amonth ცვლადის წაკითხვა. იმის მიხედვით თუ რა რიცხვი იქნა შეტანილი მომხმარებლის მიერ ხდება თვეში არსებული დღეების რაოდენობის ბეჭდვა.

დავალება: შეასწორეთ ზემოთმოყვანილი კოდი ისე, რომ:

1.შესაძლებელი იყოს წლის შეყვანა. თუ წელი ნაკიანია დაიბეჭდოს თებერვალში დღეების რაოდენობა 29.

2.შეცვალეთ კოდი ისე, რომ პროგრამა ამთავრებდეს მუშაობას როცა მომხმარებელი შეიყვანს '-1'

3. რა შეცდომაა ზემოთ მოყვანილ კოდში?

enum ტიპის ცვლადებს შეიძლება ჰქონდეთ ერთი და იგივე მნიშვნელობაც:

```
enum switch { no=0, off=0, yes=1, on=1 };
```

შეიძლება ასევე შემოტანილ იქნას ჩამონათვალი ტიპის ცვლადი ისე, რომ არ იყოს მითითებული ცვლადის ტიპი `enum { east, west, south, north } direction;`

თავი 3

მნიშვნელობა, მისამართი, მითითება, ფუნქციები

მისამართის აღების ოპერატორია "&". ეს ჩვენ უკვე შეგვხვდა scanf ფუნქციაში. scanf ფუნქციას გადაეცემა ცვლადის მისამართი, რომელსაც უნდა მიენიჭოს წაკითხული მნიშვნელობა.

3.1 მნიშვნელობა და მისამართი

განვიხილოთ მაგალითი:

```
int main(int argc, char* argv[])
{
  double x=1;

  printf("x mnishvneloba cakitxvamde=%lf\n",x );
  printf("x misamarti=%d atobitshi\n",&x );
  printf("x misamarti=%x teqvsmetobitshi\n",&x );


  scanf("%lf",&x);
  printf("x axali mnishvneloba =%lf\n",x );
  printf("x misamarti=%d atobitshi\n",&x );
  printf("x misamarti=%x teqvsmetobitshi\n",&x );

  return 0;

}
```

ამ პროგრამის გაშვების შემდეგ ნახავთ, რომ რა ახალი მნიშვნელობაც არ უნდა მივანიჭოთ x მისი მისამართი იგივე იქნება. ცხადია ყოველი ახალი გაშვებისას მისამართი დიდი ალბათობით იქნება სხვა, ვინადან ალოკაცია(ცვლადის შექმნა მეხსიერებაში) ხდება პირველსავე შემთხვევით ადგილას თავისუფალ მეხსიერებაში(ავტომატური ცვლადებისთვის, როგორიცაა ზემოთ მოყვანილი ცვლადი ე.წ. სტეკ მეხსირებაში) და წინასწარ ვერ ვიტყვით სად მოხდება ეს. მაგრამ ერთი რამ შეიძლება დარწმუნებით ვთქვათ რომ მას შემდეგ რაც შეიქმნება ცვლადი მეხსიერებაში მისი მისამართი რჩება მუდმივი ამ ცვლადის არსებობის განმავლობაში (ცვლადების დინამიურად შექმნას და „მოკვლას“ შემდეგში განვიხილავთ. ჯერჯერობით საუბარია ე.წ. ავტომატურად(სტატიკურად) შექმნილ ცვლადებზე(სტატიკური ტიპის ცვლადი სხვაა ამასაც მოგვიანებით განვიხილავთ)).

3.2 მიმთითებელი

მიმთითებელი არის ობიექტი, რომელიც უთითებს რაიმე ობიექტზე. მიმთითებლიდან შეიძლება „ავიღოთ“ ცვლადის მნიშვნელობა რომელზეც ის უთითებს,ასევე მისამართი იმ ცვლადზე, რომელზეც ის უთითებს. მისამართისგან განსხვავება ისაა, რომ ერთი მიმთითებელი სხვადასხვა შემთხვევაში შეიძლება უთითებდეს სხვადასხვა ცვლადზე(რომელთა მისამართი ცხადია სხვადასხვაა) ანუ მიმთითებლიდან აღებული მისამართი შეიძლება იყოს სხვადასხვა, მაშინ როცა მოცემული ცვლად-ობიექტის მისამართი უცვლელია. განვიხილოთ მაგალითი:

```
1  int main(int argc, char* argv[])
2  {
3    double x=1;// cvladebis gamocxadeba da sheqmna inicializacia
4    double y=1;
5    double *px; //mimtiteblis gamocxadeba
6    printf("x mnishvneloba=%lf\n",x );
7    printf("x misamarti=%x teqvsmetobitshi\n",&x );
8    scanf("%lf",&x);//vkitxulobt x cvladis mnisvnelobas
9    printf("x axali mnishvneloba =%lf\n",x );
10   printf("x misamarti=%x teqvsmetobitshi\n",&x );
11   px=&x;//mimtitebeli utitebs x cvladze
12   printf("mnishv. romelzec is utitebs px =%lf\n",(*px) );
13   printf("misam. romelzec is utitebs px=%x \n",&(*px) );
14   printf("y  mnishvneloba =%lf\n",y );
15   printf("y misamarti=%x teqvsmetobitshi\n",&y );
16   px=&y;//igive mimtitebeli exla utitebs y cvldze
17   printf("mnishvneloba romelzec utitebs px =%lf\n",(*px) );
18   printf("misam. romelzec  utitebs px=%x \n",&(*px) );
19   return 0;
20 }
```

$(*px)$ იძლევა ცვლადის მნიშვნელობას, რომელზეც უთითებს px . $\&(*px)$ იძლევა ცვლადის მისამართს, რომელზეც უთითებს px . ცხადია გამოცხადებული მიმთითებელი უნდა იყოს იგივე ტიპის რა ტიპზეც ის უთითებს[1]. ზემოთ მოყვანილ კოდში გამოცხადებული მიმთითებელი არაფერზე არ უთითებს და ასეთ შემთხვევაში მისამართის ან მნიშვნელობის აღებას თუ მოვინდომებთ იქნება შეცდომა.

3.2.1 NULL მიმთითებელი

ხშირად მიმთითებლის გამოცხადებისას ინიციალიზაციას უკეთებენ ე.წ NULL ზე. ზოგ სისტემაში NULL განმარტებულია როგორც 0. მაგრამ ეს არ ნიშნავს, რომ მითითებლის ან ცვლადის, რომელსაც ის უთითებს მნიშვნელობაა ნული. ესაა სპეციალური ე.წ. ნულ მიმთითებელი და საჭიროა იმ შემთხვევებში როცა გვინდა შევამოწმოთ უთითებს თუ არა მიმთითებელი რაიმეზე:

```
1  int main(void)
2  {
3  int *p = NULL;
```

[1] არის ასევე void ტიპის მიმთითებელიც, როემლიც შეიძლება უთითებდეს ნებისმიერი ტიპის ცვლადზე. მას მოგვიანებით განვიხილავთ.

```
if (p == NULL) {
printf("p is uninitialized!\n");
} else {
printf("p points to %d\n", *p);
}
return 0;
}
```

არსანიშნავია, რომ მიმთითებლის შემოტანისას მიმთითებელი ავტომატურად არ ინიციალიზდება როგორც NULL.

ცხადია შესაძლებელია, რომ ერთი და იგივე მიმთითებელი გამოყენებულ იქნას სხვადასხვა ცვლადზე მისათითებლად:

პროგრამის კოდი 3.1 ერთი მიმთითებელის მრავალჯერადი გამოყენება

```
int main(int argc, char* argv[])
{
  double x=1;// cvladebis gamocxadeba
  double y=1;//da sheqmna inicializacia
  double *px=NULL; //mimtiteblis gamocxadeba
    if(px==NULL)
      printf("px araferze ar utitebs\n" );

  double &adr_x=x; //misamartis ageba;
  // adr_x da x arian
  // absoliturad erti da igive cvladebi

  printf("x mnishvneloba=%lf\n",x );
  printf("x mnishvneloba=%lf\n",x );
  printf("x misamarti=%x teqvsmetobitshi\n",&x );
  printf("adr_x misamarti=%x teqvsmetobitshi\n",&adr_x );
  adr_x=17;
  printf("x mnishvneloba=%lf\n",x );
  printf("x misamarti &x=%x \n",&x );
  printf("adr_xmnishvneloba adr_x=%lf\n",adr_x );
  printf("adr_x misamarti &adr_x=%x \n",&adr_x );
  scanf("%lf",&x);//vkitxulobt x cvladis mnisvnelobas
  printf("x axali mnishvneloba =%lf\n",x );
  printf("x misamarti=%x teqvsmetobitshi\n",&x );
  px=&x;//mimtitebeli utitebs x cvladze
  if(px!=NULL)
    printf("px araa NULL e.i. utitebs \n" );
  printf("mnishv. romelzec is utitebs px =%lf\n",(*px) );
  printf("misam. romelzec is utitebs px=%x \n",&(*px) );
  printf("y  mnishvneloba =%lf\n",y );
  printf("y misamarti=%x teqvsmetobitshi\n",&y );
  px=&y;//igive mimtitebeli exla utitebs y cvldze
```

```
33    if(px!=NULL)
34      printf("px araa NULL e.i. utitebs \n" );
35    printf("mnishvneloba romelzec utitebs px=%lf\n",(*px));
36    printf("misam. romelzec  utitebs px=%x \n",&(*px) );
37    return 0;
38  }
```

ზემოთ შემოტანილია როგორც მიმთითებელი ასევე მისამართი double &adr_x. adr_x იგივეა რაც x არ აქვს მნიშვნელობა x ცვლდს მივანიჭებთ, წავიკითხავთ თუ adr_x. adr_x არის უბრალოდ x ცვლადის ახალი სახელი.

3.3 ფუნქციები

ჩვენ უკვე შეგვხვდა რამდენიმე ფუნქცია: main, scanf, printf. კვადრატული ფესვის ამოღების ფუნქცია. ესენია ე.წ. სისტემური ფუნქცია (main) და დანარჩენი C ბიბლიოთეკების ფუნქციები. ცხადია მომხმარებელს შეუძლია გამოიყენოს სხვადასხვა ბიბლიოთეკები და ფუნქციები. ისევე როგორც სხვა პროგრამულ ენებში C შეიძლება განმარტებულ იქნას ფუნქციები. ფუნქცია შეიძლება იღებდეს რაღაც ცვლადებს როგორც არგუმენტებს და შეიძლება აბრუნებდეს "რაღაცას". ეს "რაღაც"შეიძლება იყოს არაფერი, შეიძლება ენაში არსებული ტიპის ცვლადი ან მომხმარებლის მიერ შემოტანილი შედგენილი ტიპის ცვლადის მნიშვნელობა. ან ცვლადის მისამართი ან მითითება ცვლადზე.

გავიხსენოთ კვადრატული განტოლება კოდი:2.4 გვ.21 და შევცვალოთ იგი ფუნქციის გამოყენებით.

შემოვიტანოთ ფუნქცია:

```
int solve_sqeq(double _a, double _b, double _c);
```

რომელიც ხსნის გვადრატულ განტოლებას და პროგრამაში ხდება ამ ფუნქციის გამოძახება. ფუნქცია აბრუნებს მთელ მნიშვნელობას (თუ არაფერს არ აბრუნებს მაშინ გამოვიყენებდით `void` , რატომ ავიღეთ `int` ამას მოგვიანებით ვნახავთ) და იღებს სამ ცვლადს(კვადრატული განტოლების კოეფიციენტებს) არგუმენტებად. მიაქციეთ ყურადღება, რომ ფუნქცია გამოცხადებული და აღწერილია ე.წ. `main` ფუნქციამდე. ჩვენ შეგვეძლო გაგვეკეთებინა სხვაგვარადაც, კერძოდ `main` ფუნქციის წინ გაგვეკეთებინა გამოცხადება შემდეგი სახით[1] :

```
int solve_sqeq(double _a, double _b, double _c);
```

ამის შემდეგ `main` ფუნქცია როგორც ტექსტშია და `main` ფუნქციის შემდეგ `int solve_sqeq(double _a, double _b, double _c)`[2] ფუნქცია როგორც ზემოთ ტექსტშია.

აქ მთავარი ისაა, რომ `main` ფუნქციამ შესრულებაზე გაშვებისას უნდა იცოდეს ის ფუნქციები (უფრო სწორედ სახელი, დაბრუნების ტიპი და მისაწოდებელი ცვლადები), რომლებსაც ის იძახებს.

```
1  #include <stdio.h>
2  #include <math.h>
```

[1] ყურადღება მიაქციეთ ';' სახელის ბოლოს
[2] აქ კი ';' სახელის ბოლოს არაა საჭირო

```
3  int solve_sqeq(double _a, double _b, double _c)
4  {
5    double d=_b*_b-4*_a*_c;
6    if(d<0){
7      printf("D<0 araa amoxsna\n");
8      return 0;
9    }
10   double x1=(-_b+sqrt(d))/(2*_a);
11   double x2=(-_b-sqrt(d))/(2*_a);
12   printf("aris 2 fesvi:\n");
13   printf("x1=%g:\n x2=%g\n",x1,x2);
14   return 1;
15 }
16 int main(int argc, char* argv[])
17 {
18   double a=1;
19   double b=2;
20   double c=4;
21   puts("shemoitanet kv. gant. koeficientebi: a, b, c");
22   scanf("%lf%lf%lf",&a,&b,&c);
23   solve_sqeq(a,b,c);
24   return 0;
25 }
```

ეს ახალი კოდი აკეთებს ზუსტად იგივეს რასაც თავდაპირველი პროგრამა. თითქოს ეს კოდი უკეთესია, მაგრამ რჩება პრობლემები: რა ვქნათ, თუ კვადრატული განტოლების ამოხსნა გვჭირდება რამდენიმე ადგილას და ამას გარდა გვჭირდება ამონახსენებიც? როგორ გავიგოთ პროგრამის ფარგლებში ერთი ამონახსენი აქვს განტოლებას, ორი თუ არცერთი?

მეორე კითხვაზე პასუხი ადვილია რადგან ფუნქცია დავწერეთ ისე, რომ აბრუნებს მთელ ტიპს, შეგვიძლია დავაბრუნებინოთ ინფორმაცია ამონახსენების შესახებ. თუ აქვს ერთი ამონახსენი დააბრუნოს 1, თუ ორი–2, თუ არა აქვს ამონახსენი –0.

რაც შეეხება ინფორმაციას ამონახსენების მნიშვნელობის შესახებ ჩვენ გვჭირდება ორი ცვლადი სადაც შეინახება პასუხები და შემდეგ ეს ცვლადები შეიძლება გამოყენებული იქნას პროგრამაში. თითქოს ქვემოთ მოყვანილი კოდი უნდა აკეთებდეს რაც გვსურს:

პროგრამის კოდი 3.2 ფუნქცია არგუმენტის გადაცემა მნიშვნელობით

```
1  #include <stdio.h>
2  #include <math.h>
3  int solve_sqeq(double _a, double _b,
4  double _c, double _x1, double _x2)
5  {
6  double d=_b*_b-4*_a*_c;
7  if(d<0){
8    printf("D<0 araa amoxsna\n");
9    return 0;
```

```
}
else if(d==0)
{
  printf("D=0 aqvs 1 amonaxseni\n");
  _x1=-_b/(2*_a);
  _x2=_x1;
  printf("x1=x2=%g\n",_x1);
  return 1;
}
printf("aris 2 fesvi:\n");
  _x1=(-_b-sqrt(d))/(2*_a);
  _x2=(-_b+sqrt(d))/(2*_a);
  printf("x1=%g:\n x2=%g\n",_x1,_x2);
  return 2;
}
int main(int argc, char* argv[])
{
double a=1;
double b=2;
double c=4;
double x1=0;
double x2=0;
int result=-1;
puts("shemoitanet kv. gant. koeficientebi: a, b, c");
scanf("%lf%lf%lf",&a,&b,&c);
result=solve_sqeq(a,b,c, x1,x2);
return 0;
}
```

თითქოს ზემოთ მოყვანილი კოდი უნდა მუშაობეს, მაგრამ არ იმუშავებს. ფუნქცია ამოხსნის კვადრატულ განტოლებას მაგრამ ამონახსნები არ იქნება ხელმისაწვდომი ფუნქციის გარეთ. ასე იმიტომ მოხდა, რომ ფუნქციას მიეწოდება არა ცვლადები სადაც უნდა ჩაიწეროს მიღებული მნიშვნელობები არამედ თავად ცვლადების მნიშვნელობები. ამ შემთხვევაში ამბობენ, რომ ადგილი აქვს არგუმენტის გადაცემას მნიშვნელობით. და ჩვენს შემთხვევაში x1,x2 გადაცემას არავითაი აზრი არ აქვს, ვინაიდან ფუნქცია აკეთებს ამ ცვლადების ასლებს და ამ ცვლადების მნიშვნელობებს იღებენ ასლები, თავად x1,x2 ფუნქციის შეიგნით მიუწვდომელია. შესაბამისად მიუწვდომელია მიღებული ამონახსნები ფუნქციის გარეთ.

განვიხილოთ კოდი 3.3 გვ. 37 იმისათვის, რომ ფუნქციამ იმუშაოს საჭიროა გადავცეთ არა ცვლადების მნიშვნელობები არამედ ცვლადების მისამართები. a,b,c ცვლადები კვლავ გადაეცემა როგორც მნიშვნელობები და ეს შესაძლებელია ვინაიდან ამ ცვლადების მნიშვნელობები არ იცვლება ფუნქციის შიგნით.

პროგრამის კოდი 3.3 ფუნქცია არგუმენტის გადაცემა მისამართით

```
#include <stdio.h>
#include <math.h>
int solve_sqeq(double _a, double _b, double _c,
double& _x1, double& _x2)
{
double d=_b*_b-4*_a*_c;
if(d<0){
printf("D<0 araa amoxsna\n");
return 0;
}
else if(d==0)
{
printf("D=0 aqvs 1 amonaxseni\n");
_x1=-_b/(2*_a);
_x2=_x1;
printf("x1=x2=%g\n",_x1);
return 1;
}
printf("aris 2 fesvi:\n");
_x1=(-_b-sqrt(d))/(2*_a);
_x2=(-_b+sqrt(d))/(2*_a);
printf("x1=%g:\n x2=%g\n",_x1,_x2);
return 2;
}
int main(int argc, char* argv[])
{
double a=1;
double b=2;
double c=4;
double x1=0;
double x2=0;
int result=-1;
puts("shemoitanet kv. gant. koeficientebi: a, b, c");
scanf("%lf%lf%lf",&a,&b,&c);
result=solve_sqeq(a,b,c, x1,x2);
return 0;
}
```

ცხადია შესაძლებელია ეს ცვლადებიც გადაეცეს მისამართების სახით. ამ შემთხვევაში ამ ცვლადების შეცვლა შესაძლებელი იქნება ფუნქციის შიგნით. მაგრამ ეს განხილულ მაგალითში არ შეიძლება. როგორც წესი მარტივი ტიპის ცვლადებს, თუ ეს ცვლადი არ იცვლება ფუნქციის შიგნით გადასცემენ მნიშვნელობით, მაგრამ რთული(შედგენილი) ტიპის ცვლადებს გადასცემენ მისამართით, რომ არ მოხდეს მონაცემების ასლის შექმნა, რამაც შეიძლება წაიღოს დიდი დრო

(წარმოიდგინეთ, რომ ფუნქციას იძახებთ ციკლის შიგნით).

იმისათვის, რომ შემთხვევით არ მოხდეს ცვლადის მნიშვნელობის არასასურველი შეცვლა ფუნქციის შიგნით არგუმენტებში შეიძლება გამოყენებულ იქნას const. ასევე კომპილატორი ცდილობს ოპტიმიზირება გაუკეთოს კოდს და const გამოყენებით ხშირ შემთხვევაში კოდი იქნება ოპტიმიზირებული (ანუ შესრულდება სწრაფად. const ტიპის ცვლადები ისევე როგორც ბრძანებები ინახება 'მხოლოდ წაკითხვად'(read only) მეხსიერებაში). ზემოთმოყვანილი კოდი მისამართების ენაზე გადაიწერება შემდეგი სახით:

პროგრამის კოდი 3.4 არგუმენტის გადაცემა const მისამართით

```
1  #include <stdio.h>
2  #include <math.h>
3  int solve_sqeq(const double& _a, const double& _b,const  double& _c,
4  double& _x1, double& _x2)
5  {
6  double d=_b*_b-4*_a*_c;
7  if(d<0){
8    printf("D<0 araa amoxsna\n");
9    return 0;
10 }
11 else if(d==0)
12 { printf("D=0 aqvs 1 amonaxseni\n");
13   _x1=-_b/(2*_a);
14   _x2=_x1;
15   printf("x1=x2=%g\n",_x1);
16   return 1;
17 }
18   printf("aris 2 fesvi:\n");
19   _x1=(-_b-sqrt(d))/(2*_a);
20   _x2=(-_b+sqrt(d))/(2*_a);
21   printf("x1=%g:\n x2=%g\n",_x1,_x2);
22   return 2;
23 }
24 int main(int argc, char* argv[])
25 { double a=1;
26   double b=2;
27   double c=4;
28   double x1=0;
29   double x2=0;
30   int result=-1;
31   puts("shemoitanet kv. gant. koeficientebi: a, b, c");
32   scanf("%lf%lf%lf",&a,&b,&c);
33   result=solve_sqeq(a,b,c, x1,x2);
34   return 0;
35 }
```

ზემოთ მოყვანილი კოდი მუშაობს როგორც საჭიროა:

ა) ითვლის კვადრატული განტოლების ფესვებს x1,x2.

ბ) x1,x2 გამოყენება შეიძლება ფუნქციის გარეთ.

გ) result ცვლადში გვაქვს ინფორმაცია თუ რამდენი ფესვი აქვს განტოლებას.

დ) a,b,c ცვლადები დაცულია ფუნქციის შიგნით შემთხვევითი ცვლილებისაგან.

ქვემოთ მოყვანილ კოდში გვაქვს რამდენიმე გაუმჯობესება.

```
1  #include <stdio.h>
2  #include <math.h>
3  #define EPS 0.0000001
4  int solve_sqeq(const double& _a, const double& _b,const  double& _c,
5  double& _x1, double& _x2)
6  {
7    double d=_b*_b-4*_a*_c;
8    if(d<0){
9    #ifdef _DEBUG
10     printf("D<0 ar aqvs amoxsna\n");
11   #endif
12   return 0;
13   }
14   else if(d<EPS)
15   {
16
17     _x1=-_b/(2*_a);
18     _x2=_x1;
19     #ifdef _DEBUG
20     printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
21     printf("x1=x2=%g\n",_x1);
22     #endif
23     return 1;
24   }
25
26 _x1=(-_b-sqrt(d))/(2*_a);
27 _x2=(-_b+sqrt(d))/(2*_a);
28 #ifdef _DEBUG
29 printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
30 printf("x1=%g:\n x2=%g\n",_x1,_x2);
31 #endif
32
33 return 2;
34 }
35 int main(int argc, char* argv[])
36 {
37   double a=1;
38   double b=2;
39   double c=4;
```

```
40     double x1=0;
41     double x2=0;
42     int result=-1;
43     puts("shemoitanet kv. gant. koeficientebi: a, b, c");
44     scanf("%lf%lf%lf",&a,&b,&c);
45     result=solve_sqeq(a,b,c, x1,x2);
46  switch(result)
47  {
48     case 0:
49     printf("D<0 ar aqvs amoxsna\n");
50     break;
51     case 1:
52     printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
53     printf("x1=%g:\n x2=%g\n", x1, x2);
54     break;
55     case 2:
56     printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
57     printf("x1=%g:\n x2=%g\n", x1, x2);
58     break;
59     default:
60     break;
61  }
62
63  return 0;
64  }
```

დავალება:

1. შეცვალეთ კოდი 2.2 გვ.19 ფუნქციებით. დაამატეთ ხარისხში აყვანის ფუნქცია `pow` რაიმე ღილაკზე.
2. შეტანა შეცვალეთ შემდეგი თანმიმდევრობით: რიცხვი–ოპერაცია – რიცხვი
3. შეამოწმეთ ნულზე გაყოფა

ორმაგი და მცოცავი მძიმის სიზუსტის ცვლადების ტოლობაზე შემოწმება არც სწორია არც სასურველი, ვინაიდან დროის მიხედვით სჭირდება დიდი დრო (ვიდრე მაგ. ლოგიკურ ცვლადის შემოწმებას ან ნაკლებობაზე შემოწმებას) და ამავე დროს ვინაიდან გამოთვლებისას ყოველთვის გვაქვს ე.წ. მანქანური ცდომილება, შეიძლება ტოლობა არც სრულდებოდეს. ამიტომ შემოგვაქვს EPS ცდომილება.

ამას გარდა ზოგიერთი შეტყობინება (შეტყობინებები ფუნქციიდან) იბეჭდება ეკრანზე მხოლოდ პროგრამის DEBUG ვერსიაში. ასევე დამატებული გვაქვს `switch` სადემონსტრაციოდ თუ როგორ შეიძლება გამოვიყენოთ ფუნქციის მიერ დაბრუნებული მნიშვნელობა....

3.4 მონაცემების გადაცემა მიმთითებლით

მონაცემები შეიძლება გადაცემულ იქნას მიმთითებლის საშუალებით. ქვემოთ მოყვანილ კოდში გარდა იმისა, რომ გამოყენებულია მიმთითებლები, წინა კოდისგან განსხვავებით ფუნქცია დეკლარირებულია, შემდეგ არის ე.წ. `main` ფუნქცია, ხოლო შემდეგ აღწერილია დეკლარირებული ფუნქცია.

პროგრამის კოდი 3.5 არგუმენტის გადაცემა მიმთითებლით

```
1  #include <stdio.h>
2  #include <math.h>
3  #define EPS 0.000001
4  int solve_sqeq(const double*_a, const double* _b,
5  const  double* _c, double* _x1, double* _x2);
6  int main(int argc, char* argv[])
7  {
8   double a=1;
9   double b=2;
10  double c=4;
11  double x1=0;
12  double x2=0;
13  int result=-1;
14  puts("shemoitanet kv. gant. koeficientebi: a, b, c");
15  scanf("%lf%lf%lf",&a,&b,&c);
16  result=solve_sqeq(&a,&b,&c,&x1,&x2);
17  return 0;
18 }
19 int solve_sqeq(const double*_a, const double* _b,
20 const  double* _c, double* _x1, double* _x2)
21 {
22   double d=(*_b)*(*_b)-4*(*_a)*(*_c);
23   if(d<0){
24      printf("D<0 araa amoxsna\n");
25     return 0;
26   }
27   else if(d<EPS)
28   {
29     printf("D=0 aqvs 1 amonaxseni\n");
30     (*_x1)=-(*_b)/(2*(*_a));
31    _x2=_x1;
32    printf("x1=x2=%g\n",(*_x1));
33    return 1;
34   }
35   printf("aris 2 fesvi:\n");
36   (*_x1)=(-(*_b)-sqrt(d))/(2*(*_a));
37   (*_x2)=(-(*_b)+sqrt(d))/(2*(*_a));
38   printf("x1=%g:\n x2=%g\n",(*_x1),(*_x2));
39   return 2;
40 }
```

მონაცემების ასეთი სახით გადაცემას უწოდებენ *C* სტილით მონაცემების გადაცემას მისამართით. განსხვავებით 3.4 მაგალითში გადაცემისგან, რომელსაც უწოდებენ მონაცემების გადაცემა მისამართით *C++* სტილით . მართლაც მონაცემების გადაცემა მნიშვნელობით (კოდი 3.2), მონაცემების გადაცემა მისამართით (კოდი 3.4) და მონაცემების გადაცემა მიმთითებ-

ლით (კოდი 3.5) მუშაობს *C*++-ში, ხოლო *C*-ში ძალაშია მხოლოდ გადაცემა მნიშვნელობით და გადაცემა მიმთითებლით. უფრო სწორედ *C*-ში გვაქვს მონაცემთა გადაცემა მნიშვნელობით. ერთ შემთხვევაში გადაეცემა ცვლადის მნიშვნელობა (მაგალითად რაიმე რიცხვი), ხოლო მეორე შემთხვევაში გადაეცემა მიმთითებლის მნიშვნელობა(რომელიც არის სწორედ მისმართი და შემდეგ აიღება იმ ცვლადის მნიშვნელობა რომელზეც უთითებს მიმთითებელი). *C*++-ში კი მისამართით გადაცემისას ფუნქციის შიგნით მიეწოდება მისამართი. 3.1 მოყვანილი კოდი (გვ. 33) სამართლიანია *C*++-თვის. *C* კომპილატორისთვის გაუგებარი იქნება 9, 25, 32 ხაზზე არსებული კოდი და მოგვცემს შეცდომას.

3.5 დეკლარაცია იმპლემენტაცია სხვადასხვა ფაილებში

როგორც ზემოთ ვთქვით სანამ ფუნქცია გამოყენებულ იქნება სადმე (მაგ. main ფუნქციაში) საჭიროა, რომ პროგრამამ იცოდეს ამ ფუნქციის შესახებ. *C*/*C*++ ში შესაძლებელია ფუნქციის დეკლარაცია და იმპლემენტაცია (ცხადი სახე) იყოს სხვადასხვა ფაილებში. როგორც წესი ".c, .cpp, .cc, .cxx"გაფართოების ფაილები აღნიშნავენ იმპლემენტაციის ფაილებს, ხოლო ".h, .hpp, .hh"ფაილები დეკლარაციის ფაილებს[1]. სათაო ფაილის მაგალითებია იგივე "stdio.h"და "math.h"რაც უკვე შეგვხვდა. სათაო ფაილები მნიშვნელოვანია. შეიძლება არ გვქონდეს კოდის წყარო ფაილები (.c, .cpp) და გვქონდეს კომპილირებული ბიბლიოთეკა. იმისათვის, რომ გამოვიყენოთ ეს ბიბლიოთეკა საჭიროა .h ფაილი, რომელსაც ჩავრთავთ კოდში და კომპილატორს "ვეტყვით "გამოიყენოს ეს ბიბლიოთეკა[2]. ".h"ფაილი შეიცავს ფუნქციების, ცვლადების, კლასების დეკლარაციებს.

დავუბრუნდეთ ზემოთ მოყვანილ მაგალითს კოდი 3.5 და შევცვალოთ კოდები შემდეგი სახით [3] პროექტში ვამატებთ ორ ფაილს `myheader.c`, `myheader.h` შემდეგი კოდით:

პროგრამის კოდი 3.6 myheader.h დეკლარაციის ფაილი

```
//file myheader.h
#ifndef MYHEADER___H
#define MYHEADER___H
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus
extern "C" {
#endif
int solve_sqeq(const double* _a, const  double* _b,
const double* _c, double* _x1, double* _x2);
#ifdef __cplusplus
}
#endif
#endif
};
```

[1] ".h"სიტყვისგან "header"- სათავო(თავი) ფაილი
[2] ცხადია კომპილატორმა უნდა იცოდეს ფაილის მდებარეობაც
[3] კოდები და ვიდეო ფაილები იხილეთ კურსის ვებსაიტზე: http://lshoshia.science.tsu.ge.

როდესაც პროექტს ვაკეთებდით *MS Visual Studio 2010* ყურადღებას არ ვაქცევდით არც გაფართოებას (იყო .cpp) არც კომპილაციის ოპციებს, ვინაიდან აქამდე განხილული ცნებები სამართლიანი იყო როგორც *C* ასევე *C++* პროგრამული ენებისთვის. დავამატეთ ფაილი გაფართოებით `.c` ხოლო კომპილაციის პარამეტრებში მითითებული არ გვაქვს, რომ კოდი დაკომპილირდეს როგორც *C* კოდი. ასევე ფაილის გაფართოება სადაც არის `main` ფუნქცია კვლავ არის `.cpp`. რაც ნიშნავს შემდეგს:

ა) თუ სპეციალურად მითითებული არაა *MS Visual Studio* აკომპილირებს კოდს როგორც *C++* კოდს.

ბ) '`.c`' ფაილები კომპილატორს ესმის როგორც *C* ფაილები

გ) აქედან გამომდინარე გვაქვს ე.წ. შერეული კოდი *C* კოდი/ფუნქციები გამოყენებულია *C++* პროგრამაში

ასეთი შემთხვევა პრაქტიკაში ხშირია. მაგალითად, თუ გვაქვს რაიმე ბიბლიოთეკა დაწერილი *C*-ზე (არ აქვს მნიშვნელობა ამ ბიბლიოთეკის '`.c`' ფაილებს ვიყენებთ თუ ბინარულ (უკვე დაკომპილირებულ) '`.lib`' ან '`.a`' ფაილს[1]) და გვსურს მისი გამოყენება(ანუ ამ ბიბლიოთეკაში არსებული ცვლადების, ობიექტების, ფუნქციების) *C++* პროგრამულ კოდში. ამისათვის საჭიროა გვქონდეს ამ ბიბლიოთეკისs '`.h`' ფაილები. ზემოთ მოყვანილ ფალში 7-12 ხაზზე მოყვანილი კონსტრუქცია უზრუნველყოფს *C* ენაზე დაწერილი კოდის (ამ შემთხვევაში ფუნქციის) გამოყენებას *C++* კოდში. `extern` ზოგადად ნიშნავს, რომ დეკლარირებული სიდიდე (ეს შეიძლება იყოს ცვლადი, ობიექტი, ფუნქცია) აღწერილია სადღაც სხვაგან, მოცემული ფაილის გარეთ და მისი ცხადი სახე ჩართული იქნება კოდში მიკავშირების ("linking") ეტაპზე. ასეთ შემთხვევაში ამ ცვლადს არ ამოწმებს კომპილატორი კომპილაციის ეტაპზე. თუ მიკავშირების დროს ვერ მოიძებნა ეს სიდიდე მაშინ გვექნება შეცდომის შეტყობინება, რომ "ლინკერმა"ვერ იპოვნა ეს სიმბოლო და გვაქვს განუსაზღვრელი გარე სიმბოლო ("unresolved external symbol")[2]. 2 და 3 ხაზზე არსებული ბრძანებები უზრუნველყოფს, რომ როცა `myheader.c` დაკომპილირდება და შეიქმნება დროებითი ობიექტი `myheader.obj` (ან `myheader.o` - დამოკიდებულია კომპილატორზე) და ამის შემდეგ საჭირო იქნება ამ ობიექტის მიკავშირება მთავარ გამშვებ პროგრამასთან ეს ობექტი პროგრამაში ჩართული იქნება მხოლოდ ერთხელ. გავარჩიოთ ეს შემთხვევა დაწვრილებით.

ვთქვთ კვადრატული განტოლების ფუნქცია გვჭირდება სხვადასხვა რამდენიმე ფაილში. მაშინ ყოველ ფაილში უნდა გვქონდეს `#include "myheader.h"` . ვთქვათ მიკავშირების ეტაპზე შეხვდა "ლინკერს"ასეთი პირველი ფაილი. მაშინ ლინკერი ნახავს, რომ `MYHEADER_H` ჯერ არაა განმარტებული. შესრულდება მე-3 ხაზი ანუ განიმარტება `MYHEADER_H` (გახდება ერთი) და `myheader.obj` ჩაირთვება პროგრამაში. გრძელდება მიკავშირების პროცესი. ვთქვათ შეხვდა სხვა ფაილი სადაც ასევე არის `#include "myheader.h"`, მაშინ "ლინკერი"კვლავ ეცდება ჩართოს `myheader.obj` პროგარამაში, მაგრამ ისევ მე-2 ხაზზე შემოწმებისას ნახავს, რომ `MYHEADER_H` უკვე განმარტებულია (ტოლია 1) ამიტომ აღარ ჩართავს ამ ობიექტს პროგრამაში. 2-3 ხაზზე ბრძანებები, რომ არ ყოფილიყო მაშინ "ლინკერი"ეცდებოდა რამდენჯერმე ჩაერთო `myheader.obj`

[1] Windows პლატფორმაზე ბიბლიოთეკებს აქვთ '`.lib`' და Linux Unix სისტემებში '`.a`' გაფართოება

[2] `extern "C"` მოგვიანებით განვიხილავთ.

პროგრამაში და ეს იქნებოდა შეცდომა. ასეთ შემთხვევაში არის შეცდომური შეტყობინება მრავალჯერადად განმარტებული სიმბოლოების შესახებ (error: "Multiple defined symbols"). 12-13 ხაზზე პირველი `#endif` ეკუთვნის მე-2 ხაზზე არსებულ `#ifndef` („If not defined“– „თუ არაა განმარტებული“), ხოლო მეორე –მე-7 ხაზს.

ქვემოთ მოყვანილია ფაილი `myheader.c`

პროგრამის კოდი 3.7 myheader.c ფუნქციის იმპლემენტაცია

```
1  //file myheader.c
2  #include <math.h>
3  #include <stdio.h>
4  #define EPS 0.0000001
5  #include "myheader.h"
6  int solve_sqeq(const double* _a, const  double* _b,
7  const double* _c, double* _x1, double* _x2)
8  {
9  const double & bb=_c;//es mushaobs C++ magram ara C-shi
10 //C-shi ar gvaqvs reference--misamarTis agebis operacia.
11 //misamarti shenaxulia pointershi--mimtitebelshi
12 double d=(*_b)*(*_b)-4*(*_a)*(*_c);
13 if(d<0){
14 #ifdef _DEBUG
15 printf("D<0 ar aqvs amoxsna\n");
16 #endif
17 return 0;
18 }
19 else if(d<EPS)
20 {
21 (*_x1)=-(*_b)/(2*(*_a));
22 _x2=_x1;
23 #ifdef _DEBUG
24 printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
25 printf("x1=x2=%g\n",*_x1);
26 #endif
27 return 1;
28 }
29 *_x1=(-(*_b)-sqrt(d))/(2*(*_a));
30 *_x2=(-(*_b)+sqrt(d))/(2*(*_a));
31 #ifdef _DEBUG
32 printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
33 printf("x1=%g:\n x2=%g\n",*_x1,*_x2);
34 #endif
35 return 2;
36 };
```

როგორც ვხედავთ თავად ფუნქციის კოდში არავითარი ცვლილება არაა. მთავარი პროგრამის ფაილიდან კოპირებულ იქნა ფუნქცია და ფაილის დასაწყისში მითითებული აქვს

#include myheader.h.

შემდეგი ფაილი არის ის ფაილი სადაც ხდება ფუნქციის გამოძახება. ჩვენს შემთხვევაში ესაა ფაილი სადაც გვაქვს main ფუნქცია

პროგრამის კოდი 3.8 main.cpp პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას.

```
//file main.cpp
#include "myheader.h"
#define EPS 0.0000001
int main(int argc, char* argv[])
{
double a=1;
double b=2;
double c=4;
double x1=0;
double x2=0;
int result=-1;
puts("shemoitanet kv. gant. koeficientebi: a, b, c");
scanf("%lf%lf%lf",&a,&b,&c);
result=solve_sqeq(&a,&b,&c,&x1,&x2);
switch(result)
{
case 0:
printf("D<0 ar aqvs amoxsna\n");
break;
case 1:
printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
printf("x1=%g:\n x2=%g\n", x1, x2);
break;
case 2:
printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
printf("x1=%g:\n x2=%g\n", x1, x2);
break;
default:
break;
}
return 0;
}
```

აქაც როგორც ვხედავთ ცვლილება არის მხოლოდ ის, რომ არ გვაქვს ფუნქციის დეკლარაცია (გადატანილ იქნა myheader.h ფაილში) და არც ფუნქციის ცხადი სახე (გადატანილ იქნა myheader.c ფაილში). ფაილის დასაწყისში მხოლოდ მითითებულია #include myheader.h. სხვა #include ბრძანებებიც აღარ არის აქ საჭირო ვინაიდან შესაბამისი ფუნციები ჩაერთვებიან პროგრამის კოდში, რადგან ეს ბრძანებები უკვე გვაქვს myheader.h ფაილში. ამ პროექტს თუ დავაკომპილირებთ ჩვეულებრივ იმუშავებს.

3.6 extern "C"

მიკროსოფტმა შემოიტანა ე.წ. "name mangling" , რაც საჭიროა იმისათვის რომ ყოველ ფუნქციას,ცვლადს(თუ ის არის ე.წ. namespace ნაწილი, რაც იძლევა საშუალებას განასხვავოს სხვადასხვა namespace არსებული ერთსახელიანი ცვლადები), ტიპს ჰქონდეს უნიკალური იდენტიფიკატორი ბინარულ დონეზე. მაგ. როცა გვაქვს ე.წ. ფუნქციის გადატვირთვა *C*++-ში. ეს მექანიზმი გაზიარებულია სხვა კომპილატორებშიც.

კომპილატორი გენერაციას უკეთებს ფუნქციის სახელს რომელშიც არის ინფორმაცია ფუქციაზე გადაცემული არგუმენტების ტიპების შესახებ. *C*++-ში არ არსებობს ერთიანი სტანდარტი თუ რა წესით ხდება ეს გენერირება და გენერირებული სახელები დამოკიდებულია კომპილატორზე. *C*++ კომპილატორი ასევე ცვლის *C*-ს ცვლადების ფუნქციების სახელებსაც, რომ იდენტიფიცირება გაუკეთოს namespace-ს სადაც ეს ცვლადები არსებობს.

გენერირებული სახელები დამოკიდებულია როგორც კომპილატორზე ასევე ობიექტების მოდელზე. ერთი და იგივე კლასისათვისაც კი იმის მიხედვით თუ რა მოდელია გამოყენებული გენერირებული სახელები იქნება სხვადასხვა. რა მოდელია გამოყენებული ეს დამოკიდებულია კომპილაციის ოპციებზე.

"mangling"არაა სასურველი C მოდულების ან ობექტების *C*++ ობიექტებთან მიკავშირებისას. *C*++ კომპილატორმა, რომ არ მოახდინოს ფუნქციების/ცვლადების სახელების შეცვლა ამისათვის საჭიროა extern "C" სპეციფიკატორი დეკლარირებისას

```
extern "C" {
int f1(int);
int f2(int);
int f3(int);
};
```

ეს ეუბნება კომპილატორს, რომ f1, f2, f3 სახელების გენერირება არ უნდა მოხდეს.

extern "C" შეიძლება გამოყენებულ იქნას *C*++ კოდშიც თუ გვსურს *C*++ ფუნქციების გამოძახება *C*-ში.

```
  extern "C" {
    void p(int){
      /* not mangled */
      }
      };
```

დაწვრილებით: http://en.wikipedia.org/wiki/Name_mangling

3.7 *C* პროექტის შექმნა

C პროექტის შექმნა აღწერილია ვიდეო ფაილში. .cpp გაფართოების ნაცვლად ფაილებს უნდა ჰქონდეს .c გაფართოება და კომპილატორს უნდა ვუთხრათ, რომ პროექტი დააკომპილიროს როგორც *C* კოდი. GNU gcc შემთხვევაში თუ ვაკომპილირებთ როგორც *C* კოდს ვიყენებთ gcc როგორც კომპილატორს, ხოლო თუ ვაკომპილირებთ როგორც *C*++ მაშინ — g++.

ამას გარდა როცა ვაკომპილირებთ როგორც *C* მაშინ myheader.h ფაილში შეგვიძლია მოვუხსნათ #ifdef __cplusplus extern "C" კონსტრუქცია.

მაგრამ სასურველია დავტოვოთ ორი მიზეზის გამო: ა) ეს არაფერს არ უშლის ხელს *C* კოდის კომპილაციისას და ბ) თუ გვსურს *C* კოდის ბიბლიოთეკის გამოყენება *C++* პროგრამაში ეს კონსტრუქცია საჭიროა.

C კომპილატორით კომპილირებისას არის კიდევ ერთი პატარა ნიუანსი `.h` ფაილების ბოლო `#endif` შემდეგ უნდა იყოს ახალი ხაზი წინააღმდეგ შემთხვევაში კომპიოლატორი მოგცემთ შეცდომას ფაილის მოულოდნელი დასასრულის შესახებ („error: unxepected end of file"). ბოლო `#endif` შემდეგ, როგორც იტყვიან, *„Just hit Enter".*

3.8 *C* ენის სტანდარტები და იმპლემენტაცია

როცა ამბობენ, რომ *C++* არის *C* გაფართოება ეს გამართლებული არაა. მით უმეტეს, რომ სხვადასხვა კომპანია ორგანიზაციებს ჰქონდათ/აქვთ *C++* განსხვავებული იმპლემენტაცია და *C* ცალკე როგორც *C++* ნაწილი შესული იყო სხვადასხვა სტანდარტების სახით (ეს განსაკუთრებით ეხება მიკროსოფტის კომპილატორებს *C++* სტანდარტული ბიბლიოთეკა 2003 წლამდე ნაკლული იყო და პროგრამისტები ძირითადად იყენებდნენ http://stlport.sourceforge.net/, რაც შეეხება *C* 2013 წლამდე მიკროსოფტის *C* იმპლემენტაცია იყო C89. *Microsoft VS 2013* არის C99 სტანდარტი.) თანამედროვე `GNU`კომპილატორით (რომელსაც იყენებს მაგალითად `AVRStudio` Amtel მიკროპორცესორების პროგრამირებისათვის) შესაძლეებლია დაკომპილირდეს კოდი როგორც C89 ასევე C99 სტანდარტით) .

სტანდარტამდე არსებულ *C* უწოდებენდ "K&R C"(„კერნიგამის და რიჩის C"), როცა ამბობდნენ *C* 1972-1989 წლებში იგულისხმებოდა კერნიგამის და რიჩის წიგნში აღწერილი ენა. *C* ენის გამოქვეყნების შემდეგ სხვადასხვა უნივერსიტეეტბში და კომპანიებში ამ ენის საფუძველზე შეიქმნა სხვადასხვა ვარიაციები/დამატებები.

1983 წელს ამერიკის ეროვნული სტანდარტების ინსტიტუტმა (*ANSI*) შექმნა *C* სტანდარტიზაციის კომისია. 1989 წელს მოხდა *C* სტანდარტის რატიფიცირება როგორც ANSI X3.159-1989 "Programming Language C". ამ სტანდარტს შემოკლებით უწოდებენ C89 ან ANSI-C.

1990 წელს სტანდარტების საერთაშორისო ორგანიზაციის მიერ ANSI-C მიღებულ იქნა *C* ენის საერტაშორისო სტანდარტად (ISO 9899:1990 შემოკლებით – C90) ტექნიკური მხრივ C90 და C89 შორის არავითარი განსხვავება არაა. 1990-1999 წლებში *C* ენა ესაა C90.

1999 წელს მოხდა *C* სტანდარტის გადახედვა (ISO 9899:1999 შემოკლებით C99). 1999-2011 წლებში პროგრამირების ენა *C* ესაა — C99. კომპილატორების უმრავლესობა დღესაც იყენებს ამ სტანდარტს (მიკროსოფტის კომპილატორი VS2013 ვერსიიდან)

2011 წელს კვლავ მოხდა სტანდარტის შეცვლა (შეცვლა არ გულისხმობს ცხადია, რომ „ძველი" კოდი არ დაკომპიოლირდება და არ იმუშავებს)–ISO 9899:2011. ამ სტანდარტს უწოდებენ C11. http://en.wikipedia.org/wiki/C11_(C_standard_revision)

იმისათვის, რომ უზრუნველყონ კოდის პორტაბელურობა(კომპილირებადობა კოდში ცვლილებების შეტანის გარეშე სხვადასხვა სისტემებზე და სხვადასხვა კომპილატორით) *C* ენაზე დაწერილი კოდის დიდი ნაწილი არის C89 და "K&R C"სტანდარითაც კი.

იმისათვის, რომ დაკომპილირებულიყო C99 კოდი *Microsoft VS* გარემოში საჭირო იყო მისი კომპილაცია როგორც *C++* კოდისა. თუ გავითვალისწინებთ, რომ *Microsoft VS* 2013 წლამდე ვერსიები დღესაც ფართოდაა გავრცელებული მდგომარეობა დიდად არ შეცვლილა. რაც ნიშნავს, რომ შეიძლება *C* ზე დაწერილი კოდი(C99 სტანდარტით), რომელიც კომპილირდება `GNU gcc` კომპილატორით არ დაკომპილირდეს *Microsoft VS C* კომპილატორით, არამედ

კომპილატორს უნდა ვუთხრათ, რომ დააკომპილიროს როგორც *C++* კოდი. ამიტომ საჭიროა ვიცოდეთ ის მინიმალური განსხვავებები და საერთო ნიშნები მაინც რაც არის *C* ენის სხვადასხვა სტანდარტებსა და *C++* შორის.

C89 და C99

// კომენტარების ნიშანი გვაქვს C99 .

C89 კომპილატორი ანსხვავებდა პირველ 31 ასო ნიშანს. C99 — 63.

C99 არის ახალი სიტყვები:inline, restrict,_Bool, _Imaginery, _Complex

C89 main ფუნქცია არ აბრუნებდა არაფერს და ოპერატიული სისტემისათვის გაუგებარი იყო წარმატებით დამთავრდა თუ არა პროცესი. C99 main ფუნქცია აბრუნებს int ტიპს და 0 დაბრუნება ნიშნავს პროცესის წარმატებით დასრულებას.

C99 არის შემდეგი ბაზური ტიპები რაც არ არის C89:

long long int და unsigned long long int ასევე ამ ტიპების const ვარიანტები. C99 შესაძლებელია const float თექვსმეტობითში ჩაწერა.

სხვა განსხვავებებს ავღნიშნავთ შესაბამის ნაწილებში. აქ მხოლოდ მოვიყვანთ კიდევ რამდენიმე განსხვავებას, რომელიც შეეხება ცვლადების დეკლარირებას და ფუნქციებს.

ცვლადების დეკლარირება: როგორც ვთქვით პროგრამა შედგება ინსტრუქციებისაგან (ბრძანებები). ყოველი ბრძანება მთავრდება წერტილმძიმით ';'. ბრძანებები შეიძლება გაერთიანებული იყვნენ ბლოკში. ბლოკი იწყება ფიგურული ფრჩხილით '{' და მთავრდება ფიგურული ფრჩხილით '}'. C89 სტანდარტის მიხედვით ცვლადი დეკლარირებული უნდა იყოს ბრძანებათა ბლოკის დასაწყისში ანუ '{' შემდეგ. C99 (და *C++*) კი ცვლადი შეიძლება დეკლარირებულ იქნას კოდის ნებისმიერ ადგილას. მაგალითად:

```
{
  int x=1;
   printf("Hello\n");
}
```

დაკომპილირდება როგორც *C++* ასევე *C* კომპილატორით (C89,C90) ხოლო კოდი ქვემოთმოყვანილი კოდი მოგვცემს შეცდომას C89 ოპციით კომპილაციისას:

```
{
    printf("Hello\n");
    int x=1;
}
```

C99 (და *C++*) შესაძლებელია for ციკლში პირველი გამოსახულება იყოს დეკლარაცია, მათ შორის ციკლის საკუთარი ცვლადების დეკლარაცია:

```
int n=10;
for(int i=0;i<n;i++)
{
  .....
}
```

კომპილირდება C99 (და C++) , მაგრამ არა C89. ამ უკანასკნელისთვის გვაქვს:

```
int n=10;
int i;
for( i=0;i<n;i++)
{
.....
}
```

C89 შეიძლება არ იქნას მითითებული თუ რა ტიპს აბრუნებს ფუნქცია. თუ არაა მითითებული იგულისხმება, რომ აბრუნებს `int`. C99(და C++) დაბრუნების ტიპი ცხადად უნდა იყოს მითითებული.
C99(და C++) მოითხოვს, რომ ფუნქცია დეკლარირებულ ან განმარტებული უნდა იყოს მის გამოძახებამდე. ასეთი მოთხოვნა არაა C89 . თუ ფუნქცია გამოყენებულია მის დეკლარაციამდე ან განმარტებამდე მაშინ C89 კომპილატორი უშვებს, რომ ფუნქცია აბრუნებს `int` მნიშვნელობას.

აქვე ავღნიშნოთ რამდენიმე განსხვავება K&R C და C89 შორის:
K&R C არ იყო `enum` ტიპი, `unsigned` შეიძლებოდა ყოფილიყო მხოლოდ `int`, არ იყო `long long`,`const`,`void` და ა.შ. C89-ში დასაშვებია ფუნქციის დეკლარაციისას გადაცემულ პარამეტრებში მითითებული იყოს ცვლადები.

```
double square(double x);//scoria C89
double square(double);//scoria C89 da K&R
```

ფუნქციის იმპლემენტაციისას K&R C მოითხოვს, რომ ცვლადები აღწერილი იყოს ცალკე ჩამონათვალში.ესაა C89:

```
  double square(double x)
  {
  return x*x;
  }
```

ესაა K&R:

```
  double square(x)
  double x;
  {
  return x*x;
  }
```

C სტანდარტიზაციის პროცესის მიმართულებები არის შემდეგი:

- ტიპიზაციის გამკაცრება — რაც არ იყო მკაცრი მოთხოვნა K&R C ენაში მკაცრად მოთხოვნადია ახალ სტანდარტებში.
- ახალი ფუნქციების, ბიბლიოთეკების დამატება.

ISO C90 (ANSI C 89) არის 32 სიტყვა:
ტიპები: int, double, long, char, float, short, unsigned, signed, typedef, sizeof (10).

კონტროლი: if, else, switch, case, break, default, for, do, while, continue, goto (11).
ფუნქცია: return, void (2)
მონაცემთა სტრუქტურა: struct, enum, union (3)
მეხსიერება: auto, register, extern, const, volatile, static (6).
ISO C99 დამტებულია 5 სიტყვა, სულ 37:
_Bool, _Complex, _Imaginary, inline, restrict (5).
ISO C11 დამატებულია კიდევ 7 სიტყვა, სულ of 44:
_Alignas, _Alignof, _Atomic,_Generic,_Noreturn, _Static_assert, _Thread_local (7).

დასკვნა: თუ ვწერთ პროგრამას *C*-ზე უმჯობესია გამოვიყენოტ C89 სტანდარტი. თუ გვსურს გამოვიყენოთ C89 გაფართოებები რაც არის მაგ. *C*++ მაშინ უნდა დავაკომპილიროთ კოდი როგორც *C*++. შეიძლება ეს გაფართოებები არ იყოს *C*++ ნაწილიც (ეს დამოკიდებულია *C*++ კომპილატორზე), მაშინ უნდა დავაკომპილიროთ როგორც C99. მაგალითად `GNU gcc` შემთხვევაში C99 სტანდარტის ნაწილი შესულია როგორც C89 გაფართოება და არა როგორც *C*++-ის ნაწილი[1]. `Visual Studio 2013` C99 შესულია როგორც C++11 ნაწილი[2].ასევე იხ. ეს[3]

C99 თითქმის სრული მხარდაჭერა აქვს ინტელის კომპილატორში. C99 სტატუსის შესახებ იხ. http://en.wikipedia.org/wiki/C99

[1] https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/C-Extensions.html
[2] http://www.drdobbs.com/cpp/interview-with-herb-sutter/231900562
[3] http://blogs.msdn.com/b/vcblog/archive/2013/07/19/c99- library-support-in-visual-studio-2013.aspx

თავი 4

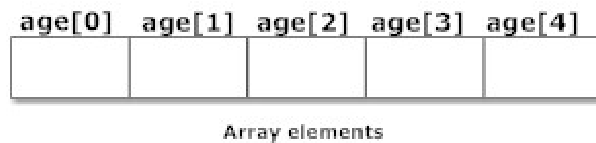
მასივი

4.1 ერთგანზომილებიანი მასივი

C-ში შეიძლება განმარტებულ იქნას ნებისმიერი ტიპის მასივი . სინტაქსი არის შემდეგი:

```
type name[dim];
```

C-ში მასივი იწყება ნულოვანი პოზიციიდან(ანუ ელემენტების ინდექსირება იწყება ნულიდან). მასივის ელემენტები განლაგებულია მეხსიერებაში მიმდევრობით..მაგალითად თუ გვაქვს `int` ელემენტების მასივი და რომელიმე `i` ელემენტის მისამართია `adr_i`, `i+1` ელემენტის მისამართი იქნება `adr_i2=adr_i+sizeof(int)`. მაგალითად გვქვს `int age[5];`, მაშინ ეს მასივი მეხსიერებაში იქნება შემდეგი სახით:



Array elements

მასივის ზომა იქნება $5 * sizeof(int)$. $age[0]$ იქნება პირველი ელემენტი, age[1] მეორე და ა.შ. ვთქვათ $age[0]$ მისამართია 2120d და int ზომაა 4 ბაიტი. მაშინ შემდეგი ელემენტის (a[1]) მისამართი იქნება 2124d, a[2]– 2128d და ა.შ.

4.1.1 ერთგანზომილებიანი მასივის ინიციალიზაცია

მასივი შეიძლება ინიციალიზირებულ იქნას დეკლარირებისთანავე

```
int age[5]={2,4,34,3,4};
```

არაა აუცილებელი მასივის ზომის მითითება თუ ვაკეთებთ ინიციალიზაციას

```
int myArray[] = { 5, 5, 5, 5, 5, 5, 5, 5, 5, 5 };
```
სწორია

ელემენტები, რომელიც არა მითითებული ინიციალიზდება როგორც 0:

```
int myArray[10] = { 1, 2 }; // 1,2,0,0,0...
```

ყველა ელემენტი ინიციალიზდება როგორც 0:

```
int myArray[10] = { 0 }; // ყველა ელემენტი ნულია.
```

მუშაობს C++, მაგრამ არაა დაშვებული C-ში:

```
int myArray[10] = {}; // ყველა ელემენტი ნულია( C++ )
```

სტატიკური ობიექტები ინიციალიზდება როგორც ნული `static int myArray[10]; //` ყველა ელემენტი ნულია `int age[]={2,4,34,3,4};` ამ შემთხვევაში კომპილატორი თავად გამოთვლის მასივის ზომას. `char` მასივის ინიციალიზაციის მაგალითი: `char string[5] = "Hello";` ან `char letters[5] = { 'H','e', 'l', 'l', 'o'};`

დავალება:

1. შექმენით *C* პროექტი და ნახეთ რა განსხვავებაა ქვემოთ მოყვანილ ცვლადებს შორის(დებაგირებისას 'Watch' ფანჯარაში ნახეთ ქვემოთ მოყვანილი ცვლადების მნიშვნელობები)

```
1  #include <stdio.h>
2  int main()
3  {
4  char string[5] = "Hello";
5  char string[6] = "Hello";
6  char  letters[5] = { 'H','e', 'l', 'l', 'o' };
7
8  return 0;
9  }
```

მასივის ინიციალიზაციისას ინიცილიზირებული ელემენტების რაოდენობა არ უნდა აღემატებოდეს მასივის ზომას. თუ მასივის ზომაზე ნაკლებია მაშინ დანარჩენი ელემენტები ინიცილიზირდება როგორც 0.

4.1.2 C99 სპეციფიური ინიციალიზაცია

აქ მოყვანილი მაგალითები არ კომპილირდება ვიზუალ სტუდიო 2010 (არც როგორც *C* არც როგორც *C++*), მაგრამ მუშაობს სტუდიო 2013 სადაც არის C99 სტანდარტის მხარდაჭერა და GNU gcc კომპილატორში. C99 შეიძლება გვქონდეს დანიშნული ინიცილიზატორები მასივისთვის, სტრუქტურისთვის, გაერთიანებისთვის. მაგალითად:

`int a[6] = { [4] = 29, [2] = 15 };` იგივეა რაც `int a[6] = { 0, 0, 15, 0, 29, 0 };`(ეს კომპილირდება ცხადია) დაშვებულია ასეთი ინიციალიზაციაც: `int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };`(არც ეს კომპილირდება) სტრუქტურის ინიციალიზაციისას მითითებულ უნდა იქნას .ველის სახელი= ელემენტის მნიშვნელობამდე. მაგალითად:

`struct point { int x, y; };`

ქვემოთ მოყვანილი ინიციალიზაცია

`struct point p = { .y = yvalue, .x = xvalue };`(არ კომპილირდება ვიზუალ სტუდიო 2010 არც როგორც *C* არც *C++*. კომპილირდება/მუშაობს ვიზუალ სტუდიო 2013)

იგივეა რაც

`struct point p = { xvalue, yvalue };`

როცა გვაქვს სტუქტურის მასივი მაგალითად ზემოთ მოყვანილი სტრუქტურისათვის, შესაძლებელია შემდეგი ინიციალიზაცია:

`struct point ptarray[10] = { [2].y = yv2, [2].x = xv2, [0].x = xv0 };`

რაც ნიშნავს: ათ ელეემნტიანი მასივის ელემენტის, რომლის ინდექსია 2, y ველს მიანიჭე მნიშვნელობა yv2, ხოლო x—xv2, ნულოვანი ინდექსის ელემენტის x ველს კი —xv0. ასეთი კონსტრუქცია მუშაობს ვიზუალ სტუდიო 2013.

4.1.3 მასივის ელემენტებზე მიმართვა

მასივის ელემენტი განიხილება როგორც ნებისმიერი ჩვეულებრივი ცვლადი. მაგალითად: ცვლადის წაკითხვა და მინიჭება მასივის მესამე ელემენტზე: `scanf("%d",&age[2]);`

ცვლადის წაკითხვა და მინიჭება მასივის i+1 ელემენტზე(ვინაიდან მასივის პირველი ელემენ-

ტი(ნულოვანი ინდექსით) არის age[0]): scanf("%d",&age[i]);
პირველი ელემენტის ბეჭდვა:printf("%d",age[0]);
i+1 ელემენტის ბეჭდვა:printf("%d",age[i]);
განვიხილოთ მაგალითი:

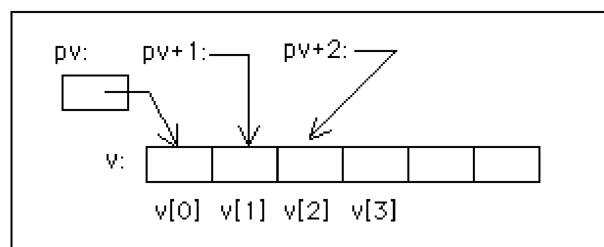
```
#include <stdio.h>
int main()
{
 int marks[10],i,n,sum=0;
 printf("shemoitanet studentebis raodenoba: ");
 scanf("%d",&n);//cakitxva
 for(i=0;i<n;++i)
  {
   printf("shemoitanet nishnebi %d: ",i+1);
   scanf("%d",&marks[i]);//cakitxva da masivis
                        //elementze minicheba
   sum+=marks[i];//shekreba
  }
 printf("Jami= %d",sum);
 return 0;
}
```

ქვემოთ მოყვანილ კოდში პროგრამა კითხულობს მომხმარებლის მიერ შეყვანილ ხაზს და ასამ-დე სიმბოლოს ინახავს სიმბოლოების მასივში char line[100];

```
main( ) {
int n, c;
char line[100];
n = 0;
while( (c=getchar( )) != '\n' ) {
if( n < 100 )
line[n] = c;
n++;
}
printf("length = %d\n", n);
}
```

4.1.4 მიმთითებელი და მასივი

C აღიქვამს მასივის სახელს ისე, როგორც მასივის პირველი ელემენტის მიმთითებელს, ეს მნიშვნელოვანია მასივებში მიმთითებლების არითმეტიკის სწორი გაგებისათვის.



ანუ თუ v არის მასივი, *v იგივეა რაც v[0], *(v+1) იგივეა რაც v[1], და ა.შ. განვიხილოთ მიმთითებლის მასივში გამოყენების მაგალითი

```
#define SIZE 3

 int main(int argc, char ** argv)
{
  float x[SIZE];
  float *p;
  int   i;
  /* initialize the array x          */
  /* use a "cast" to force i         */
  /* into the equivalent float       */
for (i = 0; i < SIZE; i++)
  x[i] = 0.5*(float)i;
/* print x                         */
for (i = 0; i < SIZE; i++)
  printf("  %d  %f \n", i, x[i]);
  /* make fp point to array x        */
  fp = x;
  /* print via pointer arithmetic    */
  /* members of x are adjacent to    */
  /* each other in memory            */
  /* *(fp+i) refers to content of    */
  /* memory location (fp+i) or x[i] */
for (i = 0; i < SIZE; i++)
  printf("  %d  %f \n", i, *(fp+i));
}
```

ვინაიდან x[i] ნიშნავს x მასივის i-ელემენტს , და fp = x უთითებს x მასივის დასაწყისზე, მაშინ *(fp+i) არის იმ ელემენტის მნიშვნელობა, რომელიც i მანძილითაა დაშორებული ელემენტს, რომელზეც უთითებს fp, ეს ელემენტი კი არის x[i].

მნიშვნელოვანია გვახსოვდეს, რომ მასივის ელემენტებზე მიმართვა უნდა მოხდეს მასივის ზომის ფარგლებში. მაგალითად თუ გვაქვს 10 ელემენტიანი მასივი `arr[10]`. შეგვიძლია მივმართოთ მასივის ელემენტებს `arr[0]` დან `arr[9]` ჩათვლით. მაგრამ მიმართვა `arr[10]`, `arr[13]` და ა.შ. იქნება შეცდომა. კომპილატორმა შეიძლება არ აჩვენოს შეცდომა კოდის კომპილირებისას, მაგრამ პროგრამის გაშვებისას პროგრამა არ იმუშავებს.

მასივ არგუმენტად შეიძლება მიეწოდოს გამოსახულება, რომელიც აბრუნებს unsigned int ტიპის მნიშვნელობას

```
#include <stdio.h>
#define SIZE 10
int main(int argc, char ** argv){
double x[SIZE]={0};
int n=7;
double xv=0.0;
```

```
7   int i;
8   x[n/2]=10.0;
9   x[2]=4.0;
10  x[7]=3.5;
11  x[8]=x[2]+x[7];
12  for(i=0;i<SIZE;i++)
13  {
14   printf("x[%d]= %lf\n",i,x[i]);
15  }
16  return 0;
17  }
```

unsigned int განმარტებულია *C* ბიბლიოთეკის ფაილში <stddef.h>(ეს ფაილი როგორც წესი ჩართულია სხვა ფაილებში და საჭირო არაა მისი ცალკე მითითება) როგორც size_t (ან unsigned __int64 64 ბიტ ვინდოუსის შემთხვევაში) რადგან მასივის ზომა და ინდექსი არ შეიძლება იყოს < 0 სწორი პრაქტიკაა გამოყენებულ იქნას ტიპი size_t, მით უმეტეს თუ გვსურს კოდმა იმუშაოს 64 ბიტ პლატფორმაზე. ასევე როცა მასივის ზომა მუდმივია და ცნობილია კოდში რიცხვის წერის ნაცვლად უმჯობესია განმარტებულ იქნას ზომა #define ბრძანების საშუალებით.

```
1  #include <stdio.h>
2  #define  SIZE 40
3  // function main begins program execution
4  int main(int argc, char ** argv){
5  double n[ SIZE ]; // n is an array of 10 integers
6  size_t i; // counter
7  double sum=0.0;
8  // initialize elements of array n to 0
9  for ( i = 0; i <SIZE; ++i ) {
10 n[ i ] = i*i*10/2.47; // set element at location i to 0
11 sum+=n[i];
12 } // end for
13 printf( "%s%13s\n", "Element", "Value" );
14 
15 // output contents of array n in tabular format
16 for ( i = 0; i < SIZE; ++i ) {
17 printf( "%7u%13f\n", i, n[ i ] );
18 } // end for
19 printf( "%20s\n","Array Average");
20 printf( "%7u%13f\n",SIZE,sum/SIZE );
21 return 0;// end main
22 }
```

#define SIZE 10 განმარტავს სიმბოლურ მუდმივას SIZE რრომლის მნიშვნელობა არის 10. ზემოთ მოყვანილ კოდში არის კიდევ ერთი სიახლე–ე.წ. ფორმატირებული ბეჭდვა %13s მე-10 ხაზზე ნიშნავს რომ გამოყოფილია 13 პოზიცია (ითვლება მარჯვნიდან მარცხნივ). შესაბამისად %7u%13f ნიშნავს, რომ გამოყოფილია 7 პოზიცია უნიშნო ცვლადისთვის და 13 პოზიცია მცოცავი

მძიმით ცვლადისთვის.

4.1.5 მაგალითი: მთვლელების მასივი

ვთქვათ ვატარებთ გამოკითხვას მოცემული ათი შოუდან , რომელი უფრო მოსწონს მაყურებელს. ამ ამოცანის მარტივად გადაწყვეტა შეიძლება შემდეგი გზით: შემოგვაქვს ე.წ. მთვლელების 10 ელემენტიანი მასივი. მასივის ინდექსი შეესაბამება შოუს ინდექსს. რომელი შოუც მოსწონს მაყურებელს შესაბამის მასივის ელემენტს ვზრდით ერთი ერთეულით.

```
1    int main(int argc, char ** argv){
2     int  ratingCounters[11], i, response;
3     for ( i = 1;  i <= 10;  ++i )
4     ratingCounters[i] = 0;
5     printf ("shemoitanet shous nomeri\n");
6     for ( i = 1;  i <= 20;  ++i ) {
7       scanf ("%i", &response);
8      if ( response < 1 || response > 10 )
9      printf ("aseti nomeri ar arsebobs: %i\n", response);
10     else
11     ++ratingCounters[response];
12    }
13    printf ("\n\nRating   gamokitxulta ricxvi\n");
14    printf ("------ -------------------\n");
15    for ( i = 1;  i <= 10;  ++i )
16    printf ("%4i%14i\n", i, ratingCounters[i]);
17    return 0;
18   }
```

ყურადღება მიაქციეთ შემდეგს:

1. გამოკითხულთა რაოდენობაა 20.
2. მიუხედავად იმისა, რომ გვაქვს 10 შოუ შემოტანილ იქნა 11 განზომილებიანი მასივი.
3. მე-3 და მე-4 ხაზზე ხდება მასივის 1–10 ელემენტების ინიციალიზაცია.
4. 6-12 ხაზზე 20 ჯერ ხდება შოუს ნომრის წაკითხვა კონსოლიდან.
5. მე-8 ხაზზე მოწმდება შოუს ნომერი. თუ ის არის საზღვრებში [1-10] იზრდება შესაბამისი მთვლელის მნიშვნელობა ერთი ერთეულით (მე-11 ხაზი) თუ არა გამოდის შეტყობინება შეცდომის შესახებ (მე-9 ხაზი)
6. ხაზებზე [13-16] ხდება მიღებული შედეგების ბეჭდვა აქ აღსანიშნავია ის, რომ არ ვიყენებთ მასივის ნულოვან ელემენტს. ეს ელემენტ არც კი ყოფილა ინიციალიზებული.

დავალება:

შეცვალეთ ზემოთ მოყვანილი კოდი ისე, რომ მასივში იყოს ზუსტად 10 ელემენტი და ინდექსირება იწყებოდეს ნულიდან, მაგრამ ისე, რომ ე.წ. შოუს ნომრები კვლავ დარჩეს 1,2,3. . . , ანუ მომხმარებელს კვლავ შეჰყავს 1,2,3. . . და ა.შ. მაგრამ მასივში ჩალაგება ხდება 0 ინდექსიდან.

4.1.6 მაგალითი: მთელი მონაცემების ჰისტოგრამა

კონსოლ პროგრამებში ხშირად ნახავთ '*' ან სხვა ფსევდოგრაფიკულ სიმბოლოებს.

```
1  // Fig. 6.8: fig06_08.c
2  // Displaying a histogram.
3  #include <stdio.h>
4  #define SIZE 10
5  // function main begins program execution
6   int main(int argc, char ** argv)
7  {
8  // use initializer list to initialize array n
9  int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
10 size_t i; // outer for counter for array elements
11 int j; // inner for counter counts *s in each histogram bar
12 printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
13 // for each element of array n, output a bar of the histogram
14 for ( i = 0; i < SIZE; ++i ) {
15 printf( "%7u%13d        ", i, n[ i ]) ;
16 for ( j = 1; j <= n[ i ]; ++j ) { // print one bar
17 printf( "%c", '*' );
18 } // end inner for
19 puts( "" ); // end a histogram bar
20 } // end outer for
21 } // end main
```

ზემოთ მოყვანილ კოდში 14-20 ხაზზე გვაქვს ორმაგი ციკლი. პირველი ციკლში გავდივართ მასივის ელემენტებს. მეორე ციკლში კი ყოველი ელემენტისათვის ვბეჭდავთ იმდენ '*' სიმბოლოს რა მნიშვნელობაც აქვს მასივის მოცემულ ელემენტს.

დავალება:

„შოუს რეიტინგების" კოდში ჩაამატეთ რეიტინგების ჰისტოგრამის ბეჭდვა

4.1.7 მაგალითი: შემთხვევითი რიცხვები გენერატორი

ქვემოთ მოყვანილი კოდით ვამოწმებთ რამდენად სწორად მუშაობს შემთხვევითი რიცხვების გენერატორი. ამოცანა არის ასეთი. წარმოვიდგინოთ, რომ ვაგორებთ კამათელს. კამათელს აქვს 6 წახნაგი. კამათელს ვაგორებთ 6,000,000 ჯერ. ყველა წახნაგის მოსვლის ალბათობა ერთი და იგივეა ამიტომ შემთხვევითი რიცხვების გენერატორმა უნდა მოგვცეს დაახლოებით ერთი და იგივე რიცხვი.

```
// Fig. 6.9: fig06_09.c
// Roll a six-sided die 6,000,000 times
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7
// function main begins program execution
int main( void )
{
size_t face; // random die value 1 - 6
unsigned int roll; // roll counter 1-6,000,000
// gagorebis mtvleli
unsigned int frequency[ SIZE ] = { 0 }; // clear counts
// am masivshi inaxeba mosvlis raodenoba
srand( time( NULL ) ); // seed random number generator
// gagoreba xdeba  6,000,000jer
for ( roll = 1; roll <= 6000000; ++roll ) {
face = 1 + rand() % 6;
++frequency[ face ]; // vzrdit mocemuli caxnagis mosvlis raodenobas
} // end for
printf( "%s%17s\n", "Face", "Frequency" );//bechdva
// output frequency elements 1-6 in tabular format
for ( face = 1; face < SIZE; ++face ) {
printf( "%4d%17d\n", face, frequency[ face ] );
} // end for
} // end main
```

4.1.8 მაგალითი: ფიბონაჩის რიცხვები

ვიღებთ საწყის ორ რიცხვს და ვამბობთ ყომ ყოველი შემდეგი რიცხვი მიიღება წინა ორის შეკრებით:$F_i = F_{i-1} + F_{i-2}$. ქვემოთ მოყვანილი კოდი შეესაბამება ამოცანას საწყისი რიცხვებით 0 და 1.

```
 #include <stdio.h>
 #define SIZE 15
  int main(int argc, char ** argv){
  int  Fibonacci[SIZE], i;
  Fibonacci[0] = 0;    // by definition
  Fibonacci[1] = 1;    // ditto
  for ( i = 2;  i < SIZE;  ++i )
   Fibonacci[i] = Fibonacci[i-2] + Fibonacci[i-1];
   for ( i = 0;  i < SIZE;  ++i )
  printf ("%i\n", Fibonacci[i]);
  return 0;
 }
```

ზემოთ მოყვანილი კოდი შეესაბამება შემდეგ ამოცანას: ვთქვათ გვაქვს კურდღლების ერთი წყვილი. ყოველი წყვილი თვის ბოლოს წარმოშობს ახალ წყვილს. კურდღლები არ იხოცებიან.

დავალება:

რამდენი წყვილი კურდღელი იქნება მეათე თვის ბოლოს? მეთორმეტე თვის ბოლოს?

4.1.9 მაგალითი: რიცხვი სხვადასხვა სიტემაში

ქვემოთ მოყვანილ პროგრამას ათობითში შეყვანილი რიცხვი გადაჰყავს სხვა თვლის სისტემაში (ორობითი, რვაობითი,ტექვსმეტობითი)

```
1  #include <stdio.h>
2  int main (void)
3  {
4    const char baseDigits[16] = {
5      '0', '1', '2', '3', '4', '5', '6', '7',
6      '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };
7    int      convertedNumber[64];
8    long int numberToConvert;
9    int      nextDigit, base, index = 0;
10   // get the number and the base
11   printf ("Number to be converted? ");
12   scanf  ("%ld", &numberToConvert);
13   printf ("Base? ");
14   scanf  ("%i", &base);
15   // convert to the indicated base
16   do {
17     convertedNumber[index] = numberToConvert % base;
18     ++index;
19     numberToConvert = numberToConvert / base;
20 
21   }
22   while  ( numberToConvert != 0 );
23   // display the results in reverse order
24   printf ("Converted number = ");
25   for (--index;  index >= 0;  --index ) {
26     nextDigit = convertedNumber[index];
27     printf  ("%c", baseDigits[nextDigit]);
28   }
29   printf ("\n");
30   return 0;
31 }
```

აქ ისევ შეგვხვდა სიტყვა `const`, რაც ნიშნავს, რომ გვაქვს მუდმივების მასივი. ანუ ამ მასივის ელემენტების მნიშვნელობის შეცვლა პროგრამის მიერ აკრძალულია (თუმცა `const` ჯერ კიდევ არ ნიშნავს, რომ ასეთი ტიპის მნიშვნელობის შეცვლა შეუძლებელია. ეს შეუძლებელია პროგრამის მიერ მაგრამ შეიძლება შეიცვალოს „გარედან“ რაიმე მოწყობილობის მიერ. მაგალითად მოწყობილობა , რომელიც ზომავს ძაბვას ან ტემპერატურას და პროგრამას „უგზავნის“ გაზომილ მნიშვნელობებს)

დავალება:
გაარჩიეთ ზემოთ მოყვანილი კოდი. აღწერეთ პროგრამის ყოველი ნაბიჯი თუ რა კეთდება ყოველ ხაზზე და უპასუხეთ შემდეგ კითხვებს: რა ინახება მასივში convertedNumber და როგორ? რას ნიშნავს 64 convertedNumber[64], რატომ იქნა შემოტანილი ტიპი long int? რა ხდება 25-ე ხაზზე? რას ნიშნავს 25-ე ხაზზე --index? რატომ გახდა ეს საჭირო? შეიძლება თუ არა პროგრამამ იმუშაოს nextDigit ცვლადის გარეშეც? რატომ და როგორ? შეცვლათ პროგარმა ისე, რომ შემოწმდეს თუ მომხმარებელმა შეიტანა ბაზად რიცხვი > 16-ზე და მისცეს მომხმარებელს ახალი ბაზის შეყვანის შესაძლებლობა.

4.2 ასო ნიშანთა მასივი

ასო ნიშნებს ვიყენებდით printf ფუნქციაში. ზემოთ რამდენჯერმე შეგვხვდა ასო ნიშანთა მასივი. ასონიშანთა მიმდევრობისათვის *C*++ სტანდარტულ ბიბლიოთეკაში არის კლასი std::string. *C*-ში ცალკე ასეთი ობიექტი არაა. „სიტყვა“(„string“) არის ასო ნიშანთა სპეციალური მასივი. აღსანიშნავია, რომ ასო ნიშანთა მასივი ჯერ კიდევ არ ნიშნავს, რომ გვაქვს „*C* string“. მაგალითად
char string1[] = "first"; ეს ბრძანება ქმნის მასივს, რომლის ელემენტებია ასო ნიშნები + ე.წ. ნულლ ასო ნიშანი '0'.
char string1[] = { 'f', 'i', 'r', 's', 't', '\0'};
ასე, რომ მასივი char string2[] = { 'f', 'i', 'r', 's', 't'}; არ არის იგივე რაც string1. „*C* string“ არის სწორედ string1 და არა string2. '0' ასო ნიშანი საჭიროა იმისათვის, რომ *C*-ს ფუნქციებმა, რომლებიც ასრულებენ ოპერაციებს სიტყვებზე იცოდნენ სად თავდება სიტყვა. *C* ში სიტყვებზე მანიპულაცია შეიძლება მხოლოდ როგორც მასივის ელემენტებზე ან მიმთითებლების საშუალებით ან სპეციალური ფუნქციების საშუალებიტ რაც არის *C*-ს სტანდარტულ ბიბლიოთეკაში string.h. შეგვიძლია სიტყვა წავიკითხოთ scanf ფუნქციის საშუალებით: char string2[20];
scanf("%19s", string2); ზემოთ მოყვანილ კოდში არის ერთი სიახლე რაც არ გვიხსენებია scanf ფუნქციის განხილვისას. ესაა %19. რაც ნიშნავს, რომ წაკითხულ უნდა იქნას 19 სიმბოლო. ეს საჭიროა მასივის წასაკითხად. ყოველ მასივს აქვს ფიქსირებული ზომა იმისათვის, რომ არ მოხდეს მასივის გადავსება–საზღვრებს გარეთ გასვლა, უნდა მიეთითოს თუ რამდენი სიმბოლო უნდა იქნას წაკითხული. ზემოთ რადგან გამოცხადებულ იქნა 20 ასო ნიშნიანი მასივი, წაკითხულ უნდა იქნას 19 სიმბოლო, ვინაიდან მე-20 უნდა იყოს ე.წ. '\0' სიმბოლო. ასევე განსხვავებით სხვა ცვლადების წაკითხვისაგან მასივის წაკითხვისას არაა საჭირო '&' სიმბოლოს მითითება, ვინაიდან მასივის სახელი *C*-ში თავად არის მასივის მისამართი. scanf წაიკითხავს string2-ში შეიტანს მაქსიმუმ 19 ასო ნიშანს მანამ სანამ არ შეხდება ხარე, ტაბულაცია, ახალი ხაზის ნიშანი ან ფაილის დასასრულის სიმბოლო. scanf არ ამოწმებს მასივის ზომას. პროგრამისტმა უნდა უზრუნველყოს რომ მასივის წაკითხვისას არ გადის მასივის ზომის გარეთ. string შეიძლება გამოტანილ იქნას ეკრანზე: printf("%s\n", string2); ქვემოთ მოყვანილ პროგრამაში გამოცხადებულია ორი სიტყვა/string მეორე ცვლადის ინიციალიზაცია ხდება სიტვა ასო ნიშნებით. პირველ მასივში იკითხება 19 ასო ნიშანი. მე-20 ხაზზე იბეჭდება string1 მასივი ასო ნიშნებს შორის ცარიელი ადგილით (ქართულად ეწოდება „ხარე“)" %c ". დაბეჭდვის პირობა არის ორი: ან სანამ არაა მიღწეული მასივის ბოლო ან სანამ არ შეხდება '\0' ასო ნიშანი. ყურადღება მაიქციეთ, რომ 16-17 ხაზებზე გვაქვს ორი წინადადების კომბინაცია, რაც მოხერხებუ-

ლია კოდის ადვილად წაკითხვისათვის. კომპილატორს ასეთი გამოსახულებების კომბინირება შეუძლია.

```
1  // Fig. 6.10: fig06_10.c
2  // Treating character arrays as strings.
3  #include <stdio.h>
4  #define SIZE 20
5  // function main begins program execution
6   int main(int argc, char ** argv)
7  {
8  char string1[ SIZE ]; // reserves 20 characters
9  char string2[] = "string literal"; // reserves 15 characters
10 size_t i; // counter
11 // read string from user into array string1
12 printf( "%s", "Enter a string (no longer than 19 characters): " );
13 scanf( "%19s", string1 ); // input no more than 19 characters
14
15 // output strings
16 printf( "string1 is: %s\nstring2 is: %s\n"
17 "string1 with spaces between characters is:\n",
18 string1, string2 );
19 // output characters until null character is reached
20 for ( i = 0; i < SIZE && string1[ i ] != '\0'; ++i ) {
21 printf( "%c ", string1[ i ] );
22 } // end for
23 puts( "" );
24 } // end main
```

მიმთითებლების გამოყენება ადვილია, ვინაიდან როგორც ყველა მასივისათვის მასივის სახელი უთითებს მასივის პირველ ელემენტზე (ნულოვანი ინდექსი). განვიხილოთ შემდეგი კოდი

```
1  int main(int argc, char ** argv)
2  {
3  char text_1[100], text_2[100], text_3[100];
4  char *ta, *tb;
5  int i;
6  char text_x[]="test";
7  /* set message to be an arrray  */
8  /* of characters; initialize it */
9  /* to the constant string "..." */
10 /* let the compiler decide on   */
11 /* its size by using []         */
12 char message[] = "Hello, I am a string; what are you?";
13
14 printf("Original message: %s\n", message);
15
16 /* copy the message to text_1   */
17 /* the hard way                 */
```

```
i=0;
while ( (text_1[i] = message[i]) != '\0' )
i++;
printf("Text_1: %s\n", text_1);

/* use explicit pointer arithmetic */
ta=message;
tb=text_2;
while ( ( *tb++ = *ta++ ) != '\0' ) ;
printf("Text_2: %s\n", text_2);
ta=text_x;
tb=text_3;
while ( ( *tb++ = *ta++ ) != '\0' ) ;
printf("Text_3: %s\n", text_3);
return 0;
}
```

4.3 მასივის გადაცემა ფუნქციაში

როგორც ითქვა მასივის სახელი იგივეა რაც მასივის მისამართი და იგივე რაც მასივის პირველი ელემენტის(ნულოვანი ინდექსით) მისამართი. ვნახოთ ეს მაგალითზე:

```
// Fig. 6.12: fig06_12.c
// Array name is the same as the address of the 'arrays
// first element.
#include <stdio.h>
// function main begins program execution
 int main(int argc, char ** argv)
{
  char array[5]; // define an array of size 5
  printf( "     array = %p\n&array[0] = %p\n
  &array = %p\n", array, &array[0], &array );
} // end main
```

ფუნქციას შეიძლება მიეწოდოს მასივი როგორც არგუმენტი, ან მასივის ელემენტი.

მასივის ელემენტის მიწოდების მაგალითი:

```
#include <stdio.h>
void display(int a)
{
  printf("%d",a);
}
int main(){
  int c[]={2,3,4};
  display(c[2]);  //gadaecema mxolod elementi  c[2].
  return 0;
}
```

ანუ, მასივის ელემენტი მიეწოდება ისევე როგორც სხვა ჩვეულებრივი ცვლადი.

ფუნქციას შეიძლება გადაეცეს მასივიც. უფრო სწორედ მასივის მისამართი. რადგან მასივის სახელი ინახავს მასივის მისამართსაც(ანუ ამავე დროს არის მიმთითებელიც სადაც შენახულია მასივის მისამართი), ფუნქციას მიეწოდება მასივის სახელი

4.3.1 მაგალითი: ერთგანზომილებიანი მასივის გადაცემა

ფუნქციას გადაეცემა ერთგანზომილებიანი მასივი და ითვლის ამ მასივის ელემენტების საშუალო მნიშვნელობას

```
1  #include <stdio.h>
2  float average(float a[]);
3   int main(int argc, char ** argv)
4   {
5    float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
6    avg=average(c);   /* Only name of array is passed as argument. */
7    printf("Average age=%.2f",avg);
8    return 0;
9  }
10 float average(float a[]){
11   int i;
12   float avg, sum=0.0;
13   for(i=0;i<6;++i){
14     sum+=a[i];
15   }
16   avg =(sum/6);
17   return avg;
18 }
```

როგორც ვხედავთ მასივის გადაცემისას არაა აუცილებელი გადაეცეს მასივის ზომაც. ოღონდ ცხადია უნდა ვიცოდეთ რა ზომის მასივს გადავცემთ. ზემოთ მოყვანილ მაგალითში ესაა 6. თუ მასივის ზომა კომპილაციის მომენტში არ ვიცით და საჭიროა, რომ დათვლილ იქნას სხვადასხვა ზომის მასივების საშუალო, მაშინ ფუნქციას უნდა გადავცეთ მასივის ზომაც `float average(float a[], int N);` და ციკლი გაივლის არა 6 არამედ `N` ელემენტს.

მაგალითად ზემოთ როცა ვარჩევდით კვადრატული განტოლების ამოხსნას ვიცით, რომ ფუნქციას მიეწოდება სამი ცვლადი როგორც კვადრატული განტოლების კოეფიციენტები და ორი ცვლადი როგორც კვადრატული განტოლების ამონახსნები. რაც ნიშნავს, რომ შეგვიძლია ფუნქციას მივაწოდოთ ორი, სამ და ორ განზომილებიანი მასივები და აუცილებელი არ იქნება მათი ზომების მიწოდება. მაგრამ თუ ფუნქციამ არ იცის მასივის ზომა მაშინ იმისათვის, რომ ოპერაციები შეასრულოს მასივის ელემენტებზე საჭიროა მასივის ზომის მიწოდებაც.

```
1 // Fig. 6.13: fig06_13.c
2 // Passing arrays and individual array elements to functions.
3 #include <stdio.h>
4 #define SIZE 5
5 // function prototypes
6 void modifyArray( int b[], size_t size );
7 void modifyElement( int e );
```

```
8   int main(int argc, char ** argv)
9  {
10   int a[ SIZE ] = { 0, 1, 2, 3, 4 }; // initialize array a
11   size_t i; // counter
12   puts( "Effects of passing entire array by reference:\n\nThe "
13   "values of the original array are:" );
14   // output original array
15   for ( i = 0; i < SIZE; ++i ) {
16     printf( "%3d", a[ i ] );
17   } // end for
18   puts( "" );
19   // pass array a to modifyArray by reference
20   modifyArray( a, SIZE );
21
22   puts( "The values of the modified array are:" );
23   // output modified array
24   for ( i = 0; i < SIZE; ++i ) {
25     printf( "%3d", a[ i ] );
26   } // end for
27     // output value of a[ 3 ]
28   printf( "\n\n\nEffects of passing array element "
29   "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );
30   modifyElement( a[ 3 ] ); // pass array element a[ 3 ] by value
31
32
33   // output value of a[ 3 ]
34   printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
35 } // end main
36 // in function modifyArray, "b" points to the original array "a"
37 // in memory
38 void modifyArray( int b[], size_t size )
39 {
40   size_t j; // counter
41
42   // multiply each array element by 2
43   for ( j = 0; j < size; ++j ) {
44     b[ j ] *= 2; // actually modifies original array
45   } // end for
46 } // end function modifyArray
47 // in function modifyElement, "e" is a local copy of array element
48 // a[ 3 ] passed from main
49 void modifyElement( int e )
50 {
51   // multiply parameter by 2
52   printf( "Value in modifyElement is %d\n", e *= 2 );
53 } // end function modifyElement
```

მასივის გადაცემისას void arraytest(int a[]) და void arraytest(int *a) არის ერთი დაიგივე. C მასივი გადაეცემა არა მნიშვნელობით არამედ მისამართით, ანუ მასივის მნიშვნელობები ლოკალურად არ კოპირდება. მასივების შემთხვევაშიც შეიძლება გამოყენებულ იქნას const კვალიფიკატორი, რაც ნიშნანი იქნება იმისა, რომ არის მუდმივ ელემენტებიანი მასივი და მისი მნიშვნელობების შეცვლა აკრძალულია პროგრამის მიერ.

დავალება:

შეცვალეთ ზემოთ გარჩეული კვადრატული განტოლების ამოხსნის პროექტის უკანასკნელი ვერსია ისე, რომ კოეფიციენტების და ფესვების ცვლადების ნაცვლად გამოყენებულ იქნას მასივები. გაითვალისწინეთ, რომ მასივის სახელი იგივეა რაც მიმითითებელი.

4.3.2 მაგალითი:მასივის დალაგება

ქვემოთ მოყვანილია მასივის დალაგების ე.წ. „Buble Sort“ ალგორითმი.

```
1  // Fig. 6.15: fig06_15.c
2  // Sorting an array's values into ascending order.
3  #include <stdio.h>
4  #define SIZE 10
5  // function main begins program execution
6  int main(int argc, char ** argv)
7  {
8  // initialize a
9  int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
10 int pass; // passes counter
11 size_t i; // comparisons counter
12 int hold; // temporary location used to swap array elements
13 puts( "Data items in original order" );
14 // output original array
15 for ( i = 0; i < SIZE; ++i ) {
16   printf( "%4d", a[ i ] );
17   } // end for
18
19   // bubble sort
20   // loop to control number of passes
21   for ( pass = 1; pass < SIZE; ++pass ) {
22
23 // loop to control number of comparisons per pass
24 for ( i = 0; i < SIZE - 1; ++i ) {
25
26   // compare adjacent elements and swap them if first
27   // element is greater than second element
28   if ( a[ i ] > a[ i + 1 ] ) {
29     hold = a[ i ];
30     a[ i ] = a[ i + 1 ];
31     a[ i + 1 ]=hold;
32     } // end if
33     } // end inner for
34     } // end outer for
```

```
35      puts( "\nData items in ascending order" );
36      // output sorted array
37      for ( i = 0; i < SIZE; ++i ) {
38    printf( "%4d", a[ i ] );
39    } // end for
40          puts( "" );
41  } // end main
```

დავალება:

1. შეცვალეთ მასივის დალაგების პროგრამა მასივის დალაგების ფუნქციად, რომელსაც გადაეცემა მასივი და მასივის ზომა.
2. შემოიტანეთ ფუნქცია, რომელიც ბეჭდავს მასივს.
3. დაწერეთ ფუნქცია, რომელიც წაიკითხავს მასივს.
4. დაწერეთ პროგრამა და გამოიყენეთ ზემოთ დაწერილი ფუნქციები:
   წააკითხეთ 10 ელემენტიანი მასივი, გამოიტანეთ ეკრანზე,დაალაგეთ ზრდადობით, შედეგი გამოიტანეთ ეკრანზე

დავალება:

იგივე რაც ზემოთ ოღონდ შემოიტანეთ დალაგების ფუნქცია, რომელიც მასივს დაალაგებს კლებადობით.

4.4 მრავალგანზომილებიანი მასივები

C შესაძლებელია განმარტებულ იქნას მრავალგანზომილებიანი მასივები ანუ მასივის მასივი. მაგალითად: `float a[2][6];` a არის ორგანზომილებიანი მასივი. ეს მასივი გრაფიკულად შეგვიძლია წარმოვიდგინოთ ასეთი სახით:

	სვ. 1	სვ. 2	სვ. 3	სვ. 4	სვ. 5	სვ. 6
სტრ. 1	$a[0][0]$	$a[0][1]$	$a[0][2]$	$a[0][3]$	$a[0][4]$	$a[0][5]$
სტრ. 2	$a[1][0]$	$a[1][1]$	$a[1][2]$	$a[1][3]$	$a[1][4]$	$a[1][5]$

მანქანის მეხსიერებაში მასივი სწორედ ასეა განლაგებული, ანუ – სტრიქონების მიხედვით, განსხვავებით ფორტრანის მასივებისგან. ფორტრან მასივები მეხსიერებაში განლაგდებიან სვეტების მიხედვით. მიაქციეთ ყურადღება, რომ შიდა ინდექსი გარბის სვეტებს და გარე – სტრიქონებს

4.4.1 მრავალგანზომილებიანი მასივის ინიციალიზაცია

C ენაში მრავალგანზომილებიანი მასივების ინიციალიზაცია შეიძლება სხვადასხვა გზით:
`int c[2][3]={{1,3,0}, {-1,5,9}};`— ცხადად მიეთითება ორივე განზომილება(2,3) და თითოეული ერთგანზომილებიანი მასივი(3 ელემენტიანი) ინიციალიზდება ცალ-ცალკე.
`int c[][3]={{1,3,0}, {-1,5,9}};`— შეიძლება მიეთითოს მხოლოდ ერთ განზომილებიანი მასივის განზომილება (ეს აუცილებელია, ანუ მასივების რაოდენობა შეიძლება არ მიეთითოს).
`int c[2][3]={1,3,0,-1,5,9};`— შეიძლება ინიციალიზდეს როგორც ერთგანზომილებიანი მასივი.
`int myPoints[][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9} };`—შეცდომაა
`int myPoints[9][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9}};`—შეცდომაა

სამგანზომილებიანი მასივის ინიციალიზაციის მაგალითი:

```
double cprogram[3][2][4]={
 {{-0.1, 0.22, 0.3, 4.3}, {2.3, 4.7, -0.9, 2}},
 {{0.9, 3.6, 4.5, 4}, {1.2, 2.4, 0.22, -1}},
 {{8.2, 3.12, 34.2, 0.1}, {2.1, 3.2, 4.3, -2.0}}
};
```

ეს სამგანზომილებიანი მასივი შეიძლება გავიგოთ ასე: გვაქვს ორ სვეტიანი და სამ სტრიქონიანი მასივი, რომლის ელემენტებია ოთხელემენტიანი მასივი.

თუ გვაქვს მაგ. arr[i][j][k][m] მასივი მაშინ ამ მასივის ელემენტების რაოდენობაა i*j*k*m.

4.4.2 მაგალითი: ორგანზომილებიანი მასივების შეკრება

ამოცანა ქვემოთ მოყვანილ კოდში მომხმარებელს შეჰყავს კონსოლიდან ორი ორორაგნზომილებიანი მასივი, რომელთა ელემენტები იკრიბება და მიენიჭება მესამე ორგანზომილებიანი მასივის ელემენტებს.

```
#include <stdio.h>
int main(){
  float a[2][2], b[2][2], c[2][2];
  int i,j;
  printf("shemoitanet pirveli matricis elementebi\n");
  /* vkiTxulobT elementebs.*/
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
    printf("shemoitane a%d%d: ",i+1,j+1);
    scanf("%f",&a[i][j]);
  }
  printf("shemoitanet meore matricis elementebi\n");
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
    printf("shemoitane b%d%d: ",i+1,j+1);
    scanf("%f",&b[i][j]);
  }
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
    /* vkribavt ori matricis elementebs da vwert mesameshi. */
    c[i][j]=a[i][j]+b[i][j];
  }
  printf("\nmatricebis jamia:");
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
    printf("%.1f\t",c[i][j]);
    if(j==1)              /* gadavdivarT axal xazze. */
    printf("\n");
  }
  return 0;
}
```

4.4.3 2D მასივის გადაცემა ფუნქციაში

ორგანზომილებიანი მასივის გადაცემისას მიეწოდება ამ მასივის დასაწყისის მისამართი, მსგავსად ერთგანზომილებიანი მასივის შემთხვევისა

```
void Function(int c[2][2]);
int main(){
  int c[2][2],i,j;
  printf("shemoitane 4 ricxvi:\n");
  for(i=0;i<2;++i)
  for(j=0;j<2;++j){
    scanf("%d",&c[i][j]);
  }
  Function(c);   /* 2d masivis gadacema funqciashi*/
  return 0;
}
void Function(int c[2][2]){
  /* sheidzleba aseve gadacemul iqnas, rogorc
  void Function(int c[][2]){.....}*/
    int i,j;
    printf("Displaying:\n");
    for(i=0;i<2;++i)
    for(j=0;j<2;++j)
    printf("%d\n",c[i][j]);
  }
```

ასევე შეისაძლებელია და სწორია შემდეგი გადაცემაც `void Function(int c[][2]){.....}` ანუ არაა აუცილებელი გარე განზომილების(პირველი განზომილების ანუ სტრიქონთა რაოდენობის) აღწერა/გადაცემა. მსგავსი შემთხვევა გვქონდა ერთგანზომილებიან შემთხვევაშიც. მაგრამ აუცილებელია დანარჩენი განზომილებების აღწერა/გადაცემა. ზემოთ მოყვანილ შემთხვევაში ეს ნიშნავს, რომ ფუნქციას გადაეცემა ორელემენტიანი მასივები, მაგრამ რამდენი ორლელემენტიანი მასივი გადაეცემა ამის მითითება არაა აუცილებელი. ცხადია ამ მასივებთან სამუშაოდ უნდა ვიცოდეთ რამდენი მასივი გადაეცემა. ეს კი ან უნდა იყოს თავიდანვე ჩადებული პროგრამაში კომპილაციის დროს ან ფუნქციას უნდა გადავცეთ მასივების რაოდენობაც, ისე როგორც ეს გვქონდა ერთგანზომილებიან შემთხვევაში.

მეორე და დანარჩენი განზომილებების გადაცემა კი აუცილებელია.

1) ვთქვათ მეორე განზომილება ხელმისაწვდომია გლობალურად(ანუ მოცემულია ან მაკროსის სახით, ან ცვლადის სახით, ან ცხადი სახით მითითებულია კოდში კომპილაციის მომენტში).

```
#include <stdio.h>
const int n = 3;
void print(int arr[][n], int m)
{
  int i, j;
  for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
  printf("%d ", arr[i][j]);
```

```
9 }
10
11 int main()
12 {
13   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
14   print(arr, 3);
15   return 0;
16 }
```

ზემოთ მოყვანილი მეთოდი მუშაობს თუ მეორე განზომილება ფიქსირებულია (ანუ ცნობილია კოდის კომპილაციისას) და არაა მომხმარებელზე დამოკიდებული. ქვემოთ მოყვანილი მეთოდი მუშაობს იმ შემთხვევაშიც როცა მეორე განზომილება არაა ცნობილი:
1) თუ კოდი არის C99 თავსებადი
C99 ვერსიიდან, *C* ენაში შესაძლებელია ცვლადი ზომის მასივების გადაცემაც ზომების გადაცემის საშუალებით. ქვემოთ მოყვანილი მაგალითი მუშაობს მხოლოდ C99 თავსებადი კომპილატორისათვის.

```
1 #include <stdio.h>
2
3 // n must be passed before the 2D array
4 void print(int m, int n, int arr[][n])
5 {
6   int i, j;
7   for (i = 0; i < m; i++)
8   for (j = 0; j < n; j++)
9   printf("%d ", arr[i][j]);
10 }
11 int main()
12 {
13   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
14   int m = 3, n = 3;
15   print(m, n, arr);
16   return 0;
17 }
```

თუ კომპილატორი არაა C99 თავსებადი მაშინ შეიძლება გამოყენებულ იქნას ქვემოთ მოყვანილი მეთოდებიდან ერთ-ერთი. 2) გადაცემულ იქნას მიმთითებელი.
ამ მეთოდის გამოყენებისას მრავალგანზომილებიან მასივს ვიხილავთ როგორც ერთგანზომილებიანს სადაც „ჩალაგებულია“ მრავალგანზომილებიანი მასივის ელემენტები. ამ მეთოდის გამიყენებისას საჭიროა ე.წ. „ტიპის კასტირება“/„ტიპის დაყვანა“. მაგ. ქვემოთ printf ფუნქციაში ცხადის სახით ვეუბნებით, რომ arr არის int ტიპის მასივი:

```
1 #include <stdio.h>
2 void print(int *arr, int m, int n)
3 {
4   int i, j;
5   for (i = 0; i < m; i++)
```

```
6    for (j = 0; j < n; j++)
7    printf("%d ", *((arr+i*n) + j));
8  }
9
10 int main()
11 {
12   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
13   int m = 3, n = 3;
14   print((int *)arr, m, n);
15   return 0;
16 }
```

და მეორე მეთოდია

3) მიმთითებლების მასივის გამოყენება ანუ ორმაგი მიმთითებელი

ამ მეთოდის გამოყენებისას ასევე აუცილებელია ტიპის დაყვანა.

```
1  #include <stdio.h>
2  // იგივეარაც  "void print(int **arr, int m, int n)"
3  void print(int *arr[], int m, int n)
4  {
5    int i, j;
6    for (i = 0; i < m; i++)
7    for (j = 0; j < n; j++)
8    printf("%d ", *((arr+i*n) + j));
9  }
10 int main()
11 {
12   int arr[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
13   int m = 3;
14   int n = 3;
15   print((int **)arr, m, n);
16   return 0;
17 }
```

უნდა გავსოვდეს, რომ *C* ენაში მასივის პარამეტრები განიხილება როგორც მიმთითებლები, რაც ნიშნავს, რომ მიმთითებლების მასივი და ორმაგი მიმთითებელი ერთი და იგივეა.

დინამიურად ალოკაციისას[1] გვაქვს ორი მეთოდი მიმთითებლების მასივი და ორმაგი მიმთითებელი. 2) და 3) მეთოდები გამოდგება ასევე C99 თავსებადი კომპილატორის შემთხვევაშიც როცა საჭიროა 2D მასივის დინამიური ალოკაცია `malloc()` ფუნქციის საშუალებით.

[1]განხილულ იქნება შემდეგ ლექციაზე

თავი 5

ცვლადები და მეხსიერება

5.1 ავტომატური ცვლადები და მხედველობის არე

როგორც ადრე ავღნიშნეთ int a=3; ტიპის ბრძანება ნიშნავს, რომ იქმნება int ტიპის ცვლადი a , რომელსაც ენიჭება მნიშვნელობა 3. ასეთი გზით გამოცხადებულ/შექმნილ ცვლადებს ავტომატური ცვლადები ეწოდებათ, ვინაიდან ისინი იქმნებიან მეხსიერებაში ავტომატურად. ავტომატური ცვლადებისათვის შეიძლება გამოყენებულ იქნას სპეციფიკატორი auto, მაგრამ ეს იგულისხმება და შეიძლება გამოტოვებულ იქნას. მაგრამ როდის „ქრებიან“? ანუ როდის თავისუფლდება მეხსიერება ასეთი ცვლადებისგან? ერთი პასუხი შეიძლება იყოს „პროგრამის მუშაობის დამთავრებისას“, რაც ყოველთვის სწორი არაა.

ადრე ავღნიშნეთ, რომ {...} მოთავსებული ბრძანებები ქმნიან ერთ ბლოკს. ზემოთმოყვანილი გზით შექმნილი ცვლადები არსებობენ მხოლოდ იმ ბრძანებათა ბლოკში სადაც ისინი შეიქმნენ. როგორც კი პროგრამა შესრულებისას გადის ბრძანებათა ბლოკიდან მეხსიერება ავტომატურად თავისუფლედება ზემოთმოყვანილი გზით შექმნილი ცვლადებისგან. მაგალითად:

```
1  #include<stdio.h>
2  int main()
3  {
4    int a=3;
5    int b=4;
6    printf("\na*b=%d * %d =%d",a,b,a*b);
7    {
8      int a=5;
9      int b=6;
10     printf("\nqveblokshi a*b=%d * %d =%d",a,b,a*b);
11     int c=8;
12     printf("\nqveblokshi c=%d",c);
13   }
14   printf("\nisev gare bloki a*b=%d * %d =%d",a,b,a*b);
15   puts("\n");
16   return 0;
17 }
```

ასეთ ცვლადებს სხვაგვარად ლოკალურ ცვლადებსაც უწოდებენ, ვინაიდან ისინი იქმნებიან ავტომატურად და არსებობენ მხოლოდ ბრძანებების ერთ ბლოკში და წაიშლებიან ბრძანებების ბლოკიდან გასვლისას. პირველად განმარტებული a და b არსებობენ პროგრამის მუშაობის განმავლობაში, ხოლო ქვებლოკში განმარტებული a და b მხოლოდ ამ ბლოკში. ანუ ბლოკის გარეთ განმარტებული ცვლადები ხელმისაწვდომია ბლოკში, ხოლო ბლოკში განმარტებული ცვლადები – მხოლოდ ბლოკში. აქ ასევე გასათვალისწინებელია, რომ „ხელმისაწვდომია“ ნიშნავს

განმარტების(უფრო ზუსტად ინიციალიზაციის) შემდეგ. არა ინიცირიზირებული ცვლადის გამოყენება შეცდომაა. შემდეგ. მაგალითად თუ ზემოთ მოყვანილ კოდში მე-9 ხაზის შემდეგ დავწერთ c=7; ეს იქნება შეცდომა ვინაიდან ასეთი ცვლადი ჯერ არ შექმნილა.

მიუხედავად იმისა, რომ a და b შექმნილია main()ში და არსებობენ პროგრამის მუშაობის განმავლობაში ეს ცვლადებიც ლოკალურია, იმ გაგებით, რომ შეუძლებელია მათი გამოყენება სხვა, ამ ფაილის გარეთ განმარტებულ ფუნქციებში ცხადი სახით, ანუ არგუმენტად გადაცემის გარეშე.

5.2 გლობალური და ლოკალური ცვლადები

შეიძლება თუ არა ცვლადები შემოტანილ იქნას main() ფუნქციამდე? შეიძლება და სხვაგვარად a და b main() ფუნქციამდე განმარტებულ ცვლადებს გლობალური ცვლადები ეწოდებათ, ხოლო სხვა {...} ბლოკში (მათ შორის ფუნქციებშიც) განმარტებულ ცვლადებს – ლოკალური. განვიხილოთ კოდი:

```
#include<stdio.h>
int a=3;
int b=4;
int mult_and_add(int _x, int _y)
{
  return _x*_y+a;
}

int main()
{

  printf("\na*b=%d * %d =%d",a,b,a*b);
  {
    int tmpa=5;
    int tmpb=6;
    printf("\nqveblokshi tmpa*tmpb=%d * %d =%d",tmpa,tmpb,tmpa*tmpb);
    int c=8;
    printf("\nqveblokshi c=%d",c);
  }
printf("\nisev gare bloki a*b=%d * %d =%d",a,b,a*b);
puts("\n");
printf("\n%d\n",mult_and_add(3,5));
return 0;
}
```

ეკრანზე დაიბეჭდება '18'. ამ შემთხვევაში a და b გლობალური ცვლადებია. ისინი არსებობენ პროგრამის მუშაობის განმავლობაში და ხელმისაწვდომია ყველგან ყველა ფუნქციაში. გლობალური ცვლადები მიეკუთვნებიან *სტატიკურად* შენახვის ხანგძლივობის კლასს (*static* storage duration) მაშინ როცა ლოკალური ცვლადები *ავტომატურად* შენახვის ხანგძლივობის კლასს (*automatic* storage duration). გლობალური ცვლადები ყოველთვის ინიციალიზირდება როგორც ნული.

5.3 გლობალური ცვლადების მხედველობის არე. static ცვლადები

static ნიშნავს, რომ ცვლადი განაგრძობს არსებობას ბრძანებათა ბლოკიდან პროგრამის გასვლის შემდეგაც. ანუ ცვლადი არ ნადგურდება და არსებობს პროგრამის მუშაობის განმავლობაში. იმის მიხედვით თუ სად იყო ეს ცვლადი დეკლარირებული ის შეიძლება იყოს როგორც გლობალური ასევე ლოკალურიც, მაგრამ „გლუბალურობა" აქ შეზღუდულია კომპილაციის ერთეულში(ობიექტში) ფაილით სადაც ცვლადი იყო გამოცხადებული. ამას გარდა თუკი პროგრამის აწყობისას (მიკავშირება „linking") პროგრამას შეხვდა ერთხელ უფრო მეტ შემთხვევაში ბლობალური ცვლადი მოგვცემს შეცდომას, რომ ეს სიმბოლო უკვე განმარტებულია. უნდა განვასხვავოთ ერთმანეთისგან ცვლადის სიცოცხლის განგძლივობა და მხედველობის არე. როგორც ავღნიშნეთ ავტომატური ცვლადები არსებობენ მხოლოდ მათ მხედველობის არეში. რაც შეეხება გლობალურ ცვლადებს ისინი არსებობენ პროგრამის მუშაობის დამთავრებამდე, მაგრამ შეიძლება ჰქონდეთ შეზღუდული მხედველობის არე. იმისათვის, რომ ერთ ფაილში(კომპილაციის ერთეულში) არსებული გლობალური ცვლადი ხელმისაწვდომი (დანახვადი) გავხადოთ სხვა ერთეულში გამოიყენება სიტყვა extern. ეს ეხება გლობალურ ცვლადებს static სპეციფიკატორის გარეშე. განვიხილოთ მაგალითი:

პროგრამის კოდი 5.1 testglobal.cpp პროექტის ფაილი გლობალური ცვლადებით.

```
int d=5;
static int e=10;
#include<stdio.h>
#include "util.h"
extern int b;
int main()
{
  b=8;
  a=7;
  //printf("\na*b=%d * %d =%d",a,b,a*b);
  {
    int a=5;
    int b=6;
    int c=8;
    printf("\nqveblokshi a*b=%d * %d =%d",a,b,a*b);
    printf("\nqveblokshi c=%d",c);
  }
  //printf("\nisev gare bloki b=%d",b);
  puts("\n");
  printf("\n%d\n",mult_and_add(3,5));
  puts("\n");
  printf("\n%d\n",mult_and_add(3,5));
  return 0;
}
```

და გავქვს ფაილები util.h და util.c სადაც ასევე განმარტებულია გლობალური ცვლადები.

პროგრამის კოდი 5.2 util.h პროექტის ფაილი გლობალური ცვლადებით.

```
#ifndef TEST_UTIL_H
#define TEST_UTIL_H
static int a=3;
int mult_and_add(int _x, int _y);
#endif
```

შეგახსენებთ, რომ თუ თქვენ გაქვთ C++ პროექტი(აკომპილირებთ როგორც C++) მაშინ ზემოთ მოყვანილ ფაილს უნდა ჰქონდეს სახე:

პროგრამის კოდი 5.3 util.h პროექტის ფაილი გლობალური ცვლადებით. C კოდის ჩართვაა C++ პროექტში

```
#ifndef TEST_UTIL_H
#define TEST_UTIL_H
static int a=3;
#ifdef __cplusplus
extern "C" {
#endif

  int mult_and_add(int _x, int _y);

#ifdef __cplusplus
}
#endif
#endif
```

და შესაბამისი იმპლემენტაციის ფაილი:

პროგრამის კოდი 5.4 util.c პროექტის ფაილი გლობალური ცვლადებით.

```
int b=4;
#include "util.h"
extern int d;
int mult_and_add(int _x, int _y)
{
  static int lb=0;
  ++lb;
  d=8;
  b=14;
  a+=1;
  return _x*_y+a+lb;
}
```

გავარჩიოთ თითოეულ ფაილში მოყვანილი კოდი. მთავარ ფაილში `testglobal.c` გვაქვს ორი გლობალური ცვლადი `int d=5;` და `static int e=10;` მათ შორის განსხვავებაა ისაა, რომ d

ცვლადის გამოყენება შეიძლება სხვა ფაილებშიც (პროგრამის ერთეულებში), ხოლო e ცვლადი-სა არა, თუმცა ორივე არსებობს პროგრამის მუშაობის განმავლობაში. მე-5 ხაზზე `extern int b;` ნიშნავს, რომ სადღაც სხვაგან დეკლარირებული და განმარტებულია `int b` გლობალური ცვლა-დი, რომლის გამოყენებაც გვსურს ამ ფაილში. `extern` ნიშნავს, რომ ეს ცვლადი არის ამ ფაილის გარეთ სადღაც სხვაგან. თუ ეს ცვლადი არ არსებობს ბინარული ფაილის აწყობისას კომპილა-ტორი მოგვცემს შეცდომას.

`main()` ფუნქციაში როგორც ვხედავთ იცვლება `b` მნიშვნელობა (მე-8 ხაზი). ასევე შეიძლება თავისუფლად გამოვიყენოთ ცვლადი `e` ნებისმიერ ადგილას. მაგრამ მე-9 ხაზზე იცვლება რა-ღაც `a` ცვლადის მნიშვნელობაც. საიდან მოვიდა ეს ცვლადი? ესა არაა ცვლადი `a` , რომელიც დეკლარირებულია მე-12 ხაზზე, ვინაიდან: ა) 11-17 ხაზზე მოთავსებული კოდი არის ფიგურულ ფრჩხილებში და მასში დეკლარირებული ცვლადები არის ლოკალური, ავტომატური და არსე-ბობენ მხოლოდ ამ ნაწილში. ბ) ფრჩხილები კიდეც რომ არ იყოს მე-9 ხაზზე გამოყენებულია `a` მის დეკლარირებამდე. ის არსად დეკლარირებული არ ყოფილა, მაგრამ კოდი კომპილირდება და „მუშაობს“. შეგვიძლია გავაკეთოთ დასკვნა, რომ `a` არის სადღაც დეკლარირებული და გან-მარტებული გლობალური ცვლადი და ეს მართლაც ასეა. მაგრამ რატომ არაა დეკლარირებული ეს ცვლადი `extern int b;` მსგავსად?

ვნახოთ ფაილი `util.h`. აქ დეკლარირებულია ფუნქცია `int mult_and_add(int _x, int _y);` და ასევე გლობალური ცვლადი `a` `static` სპეციფიკატორით. რატომ დაგვჭირდა ეს სპეციფი-კატორი? ვნახოთ ფაილი`util.c`. ამ ფაილის მე-9 ხაზზე ვხედავთ, რომ იცვლება `a` ცვლადის მნიშვნელობა. რადგან `a` ცვლადი გამოყენებულია როგორც ამ ფაილში ასევე მთავარ ფაილ-ში კომპილატორი ცდილობს, რომ შექმნას ეს ობიექტი ორჯერ კომპილაციისას. და ვინაიდან ერთი და იგივე ობიექტი არ შეიძლება იყოს დუბლირებული, თუ არ ექნება ამ ცვლადს სპეცი-ფიკატორი `static` კომპილატორი მოგვცემს შეცდომას. სპეციფიკატორი `static` ამ შემთხვევა-ში უზრუნველყოფს, რომ კომპილაციისას იქმნება მხოლოდ ერთი ობიექტი და გლობალურობა უზრუნველყოფს, რომ შეიძლება მისი გამოყენება ყველგან სადაც კი ჩართული იქნება `util.h` ფაილი.

`util.c` ფაილში ასევე ვხედავთ `int b=4;` ეს ის ცვლადია, რომელიც დეკლარირებულია რო-გორც გარე `extern int b;` მთავარ ფაილში. აქ არის ასევე დეკლარირებული გარე ცვლადი `extern int d;` ეს ცვლადი დეკლარირებული და განმარტებულ იყო მთავარ ფაილში და ზემოთ მოყვანილი დეკლარაცია უზრუნველყოფს მის გამოყენების შესაძლებლობას `util.c` ფაილში.

როგორც ვხედავთ ამავე ფაილში ფუნქციის შიგნით მე-6 ხაზზე გვაქვს კიდევ ერთი სტატიკუ-რი ცვლადი `static int lb=0;` ეს ცვლადი იქმნება მხოლოდ ერთხელ, განაგრძობს არსებობას როცა გავდივართ ფუნქციიდან და არსებობს პროგრამის მუშაობის განმავლობაში, მაგრამ ჩანს და ხელმისაწვდომია, მხოლოდ ამ ფუნქციაში, ანუ შეზღუდულია მისი მხედველობის არე და არა სიცოცხლის ხანგძლივობა.

სტატიკური და გლობალური ცვლადები ერთმანეთისგან განსხვავდებიან სიციცხლის ხანგძლივობით და მხედველობის არით. განსხვავება მხედველობის არესა და სიციცხლის ხანგძლივობას შორის არის ის, რომ სიცოცხლის ხანგძლივობა განსაზღვრავს არის თუ არა ობიექტი მეხსიერებაში, მაშინ როცა მხედველობის არე განსაზღვრავს ხელმისაწვდომია თუ არა ობიექტი(ანუ შეგვიძლია თუ არა მივმართოთ მას სახელით კოდის მოცემულ ადგილას). შესაძლებელია, რომ ობიექტი არსებობდეს, მაგრამ უხილავია კოდის მოცემულ არეში.

სტატიკური ცვლადები ლოკალურია იმ მოდულში სადაც მოხდა მათი განმარტება (მაგ. ფაილი). ასე მაგალითად სტატიკური ცვლადი, რომელიც შემოტანილ იქნა ფუნქციის შიგნით უხილავია ფუნქციის გარეთ, მაგრამ ის არსებობს პროგრამის მუშაობის განმავლობაში (არსებობს მეხსიერებაში). როცა ხდება ფუნქციაში შესვლა გამოიყენება იგივე ცვლადი (სახელი და მისამართი), რომელიც იყო შექმნილი ფუნქციაში შესვლამდე და არ ხდება მეხსიერების ხელახალი გამოყოფა განსხვავებით ე.წ. ავტომატური ცვლადებისაგან, რომლებიც იქმნება და იშლება მეხსიერებიდან ფუნქციაში ყოველი შესვლისას.

სტატიკური ცვლადი თუ შემოტანილ იქნა გლობალურად (ფაილის დასაწყისში, როგორც ეს გვქონდა ზემოთ მაგალითში), მაშინ ეს ცვლადი ხელმისაწვდომია მხოლოდ ამ ფაილში (და თუ შემოტანილ იქნა ე.წ. სათაო ფაილის დასაწყისში ხელმისაწვდომი იქნება ყველა ფაილში სადაც ეს სათაო ფაილი იქნება ჩართული)

მეორე მხრივ გლობალური ცვლადები განმარტებულ უნდა იქნან გლობალურად. არსებობენ პროგრამის მუშაობის განმავლობაში და ხელმისაწვდომია ყველგან პროგრამაში (შეიძლება `extern` გამოყენებით, რაც არაა შესაძლებელი სტატიკური თუნდაც გლობალური მოცემულ ფაილში ცვლადებისთვის)

ეს მექანიზმი საშუალებას იძლევა მაგალითად, თუ წერთ რაიმე ბიბლიოთეკას, სადაც განმარტებულია გლობალური ცვლადები და არ გსურთ, რომ ეს ცვლადები გამოძახებულ იქნას მომხმარებლის მიერ თავის პროგრამაში ამ ბიბლიოთეკის გამოყენებისას მაშინ დეკლარირებული უნდა გქონდეთ როგორც `static`

ცხადია თუ ზემოთ მოყვანილ გლობალურ ცვლადებს თუ არ ვიყენებთ სხვა ფაილებში მაშინ მათი `extern` დეკლარაციები არაა საჭირო და ამ ცვლადების გამოყენების(მხედველობის) არე იქნება მხოლოდ ის ფაილები სადაც ისინი შეიქმნა.

`extern` , `static` შეიძლება იყოს არა მარტო მარტივი ენაში არსებული ბაზური ცვლადების ტიპის ობიექტები არამედ რთული შედგენილი ობიექტებიც და ასევე ფუნქციებიც. მათი დეკლარირება და გამოყენება სხვადასხვა მხედველობის არით კეთდება ზუსტად ისევე როგორც ზემოთ მოყვანილ მაგალითში.

ავტომატურ ცვლადებს განეკუთვნება ასევე ცვლადები კვალიფიკატორით `register`. მაგალითად '`register int c;`'. `register` ეუბნება კომპილატორს, რომ ცვლადი უნდა განთავსებულ იქნას პროცესორის მეხსიერების რეგისტრებში. რეგისტრ მეხსიერებაში განთავსებულ ცვლადებს პროცესორი მიმართავს უფრო სწრაფად. თანამედროვე კომპილატორებს შეუძლიათ ავტომატური ოპტიმიზაცია — განათავსონ თუ არა დროებითი ცვლადი რეგისტრებში და ხშირად `register` ინსტრუქცია არ გამოიყენება.

5.4 ცვლადის კვალიფიკატორი volatile

კვალიფიკატორ `volatile` საშუალებით აღიწერება ცვლადი, რომლის მნიშვნელობა შეიძლება შეიცვალოს არაპროგრამულად ნებისმიერ შემთხვევაში/მომენტში. მაგალითად იმ შემთხვევაშიც კი, როცა ეს ცვლადი არის სტრუქტურის წევრი[1] და სტრუქტურა გადაეცემა ფუნქციას როგორც მუდმივი (`const`). ასეთი შემთხვევები გვაქვს მაგალითად მრავალნაკადიან პროგრამირებისას და მიკროპროცესორებში, როცა ცვლადი შეიძლება შეიცვალოს „გარედან“ ან სხვა ნაკადის მიერ ან მოწყობილობის რაიმე ღილაკზე დაჭერით.

[1] სტრუქტურებს მოგვიანებით განვიხილავთ

5.5 ტიპის დაყვანა — „კასტირება“

ხშირად საჭიროა ერთი ტიპის ცლადის გადაყვანა სხვა ტიპში. განვიხილოთ მაგალითი:

```
1  int main(void)
2  {
3    int a = 5;
4    int b = 10;
5    float f;
6    f = a / b;  /* calculate 5 divided by 10 */
7    printf("%.2f\n", f);
8    return 0;
9  }
```

ეკრანზე გამოვა 0; ეს შედეგი მივიღეთ იმიტომ, რომ `int` ტიპის ცვლადების გაყოფა გვაქვს და შედეგიც არის მთელი ტიპის. ანუ იგნორირებულია წილადი ნაწილი. თუ ჩვენი ამოცანა სწორედ ესაა მაშინ შედეგი სწორია, მაგრამ თუ გვსურს წილადი ნაწილის პოვნა? ამისათვის საჭიროა ერთი, ან ორივე, მთელი ტიპის ცვლადი გადაყვანილ იქნას `float` ან `double` ტიპის ცვლადში. ამისათვის მე-6 ხაზზე გაყოფის ბრძანებას უნდა ჰქონდეს შემდეგი სახე:

```
f =(float) a / b;
```

5.6 `void` მიმთითებელი

სანამ განვიხილავდეთ მეხსიერების დინამიურ ალოკაციას (მოცემული ტიპის და ზომის მეხსიერების გამოყოფა მონაცემთა შესანახად) განვიხილოთ სპეციალური ტიპის მიმთითებელი `void`. `void` მიმთითებელი არ არის იგივე რაც `void` მნიშვნელობა, რომელსაც აბრუნებს ფუნქცია, ან `void` არგუმენტების ჩამონათვალი. `void` მნიშვნელობა ნიშნავს, რომ ფუნქცია არაფერს არ აბრუნებს, მაშინ როცა `void` მიმთითებელი ნიშნავს, რომ მიმთითებელი შეიძლება მიუთითებდეს ნებისმიერ ტიპზე. რაც იგივეა, მისამართზე რომელზეც უთითებს `void` მიმთითებელი შეიძება იყოს ნებისმიერი ტიპის ცვლადი. მაგრამ იმისათვის, რომ მივმართოთ ცვლადს/ შევცვალოთ იგი და ა.შ. აუცილებელია ცვლადის ტიპის ცოდნა. როგორ შეიძლება გამოვიყენოთ `void` მიმთითებელი? იმისათვის, რომ გამოვიყენოთ `void` მიმთითებელი, უფრო სწორედ ვიმუშაოთ ცვლადზე, რომელზეც ის უთითებს საჭიროა ტიპის დაყვანა (type casting). ამის შემდეგ შესაძლებელია ცვლადის/ობიექტის გამოყენება. რა შემთხვევებში გვჭირდება `void` მიმთითებელი? მაგალითად ბმულ სიაში (linked list). ან ვთვათ მასივში ვინახავთ მიმთითებლებს ისე, რომ არ გვაინტერესებს ობიექტის ტიპი. ანუ, გვსურს მასივში შევინახოთ *ნებისმიერი ტიპის* მიმთითებელი. და ვთქვათ ასევე გვსურს მივმართოთ ამ მიმთითებელთა მასივის ელემენტს. ამისათვის თუ დავწერთ მასივში დამატების და მასივის ელემენტზე მიმართვის ფუნქციებს ამ ფუნქციებმა არ უნდა იცოდნენ მასივის ელემენტის ტიპი. უფრო სწორედ, ვინაიდან მასივის ელემენტების ტიპი შეიძლება იყოს ნებისმიერი, ფუნქციებს უნდა შეეძლოთ ნებისმიერ ტიპთან მუშაობა. ასევე, ცხადია თავად მასივიც უნდა იყოს ისეთი, რომ ნებისმიერი ტიპის მიმთითებლის შენახვა უნდა შეეძლოს.

განვიხილოთ კოდი „`void` მიმთითებელი“(კოდი 5.5 გვ.78). მე-2 ხაზზე შემოტანილია `void` ტიპის 10 ელემენტიანი გლობალური მასივი. ასევე გლობალური ცვლადი `index` და ორი გლო-

ბალური ფუნქცია: ერთი მათგანი ამატებს მიმთითებელს მასივში, მეორე კი აბრუნებს მოცემულ პოზიციაზე მასივში არსებულ მიმთითებელს.

პროგრამის კოდი 5.5 „void მიმთითებელი და void ტიპის მასივი“

```
#include <stdio.h>
void *pointer_array[10]; /* we can hold up to 10 void-pointers */
int index=0;
void append_pointer(void *p)
{
   pointer_array[index++] = p;
}
void *get_pointer(int i)
{
  return pointer_array[i];
}

int main(void)
{
  char *s = "some data!";  /* s points to a constant */
  int a = 10;
  int *b;
  char *s2;  /* when we call get_pointer(), we'll store*/
  int *b2;
  b = &a; /* b is a pointer to a */
  /* now let's store them both, even though they're*/
  append_pointer(s);
  append_pointer(b);
  /* they're stored!  let's get them back! */
  s2 =(char*) get_pointer(0); /* this was at index 0 */
  b2 =(int*) get_pointer(1); /* this was at index 1 */
  printf("\n b2 arsebuli nishvneloba *b2=%d",*b2);
  printf("\n s2 masivia s2[]=%s\n",s2);
  return 0;
}
```

ყურადღება მიაქციეთ მე-6 ხაზზე index++ რაც ნიშნავს, რომ ჯერ ხდება ელემნტის დამატება index პოზიციაზე ხოლო შემდეგ index ცვლადის გაზრდა ერთი ერთეულით 25-ე და მე-2 ხაზზე შემოტანილია void ტიპის 10 ელემენტიანი გლობალური მასივი. ასევე გლობალური ცვლადი index და ორი გლობალური ფუნქცია: ერთი მათგანი ამატებს მიმთითებელს მასივში, მეორე კი აბრუნებს მოცემულ პოზიციაზე მასივში არსებულ მიმთითებელს. ყურადღება მიაქციეთ მე-6 ხაზზე index++ რაც ნიშნავს, რომ ჯერ ხდება ელემნტის დამატება index პოზიციაზე ხოლო შემდეგ index ცვლადის გაზრდა ერთი ერთეულით 25-ე და 26-ე ხაზებზე ჯერ ხდება ცვლადის დაყვანა(ამ შემთხვევაში მიმთითებლის), ხოლო შემდეგ მინიჭება.

5.7 დინამიური ობიექტები

ობიექტების დინამიურად შექმნისათვის საჭიროა მეხსიერების დინამიურად ალოკაცია (გამოყოფა) მოცემული ობიექტისათვის — პროგრამის შესაძლებლობა გამოიყენოს მეტი მეხსიერება ვიდრე მას ჰქონდა შესრულებაზე გაშვების მომენტში და გაანთავისფლოს ეს მეხსიერება გამოყენების შემდეგ. ეს ამოცანა დგას თითქმის ყველა პროგრამაში, რომელიც აკეთებს რაიმე გამოთვლებს და არ ვიცით წინასწარ (კოდის კომპილაციისას) რა და რამდენი ობიექტი იქნება საჭირო (მაგალითად, შეიძლება პროგრამა კითხულობდეს მასივის ზომებს ფაილიდან, რომელიც შეიძლება იყოს სხვადსხვა.).

მეხსიერების დინამიურად ალოკაციისათვის *C* აქვს სამი ფუნქცია: `malloc()` — გამოყოფს მეხსიერებას მოხმარებისთვის. `free()` — ათავისფულებს `malloc()` მიერ გამოყოფილ მეხსიერებას. `realloc()`— ცვლის ადრე გამოყოფილი მეხსიერების ზომას და `calloc()` — `malloc()` მსგავსია ოღონდ გამოყოფილ მეხსიერება ივსება ნულებით.

5.7.1 malloc() ფუნქცია

`malloc()` გამოყოფს მეხსიერებაში არეს და აბრუნებს მიმთითებელს ამ არეზე. თუ მეხსიერების გამოყოფა ვერ მოხერხდა აბრუნებს `NULL`. დაბრუნებული ტიპი არის `void*` რაც შესაძლებლობას იძლევა დაყვანილ იქნას ნებისმიერი ტიპის მიმითითებელზე.

ვინაიდან `malloc()` მოქმედებს მეხსიერების ბაიტებით, ხოლო მომხმარებელი კი რაღაც ცვლადებით, რომლებსაც აქვს ტიპი (მაგ. „გამოყავი მეხსიერება 12 `int` ცვლადისთვის“), ხშირად გამოიყენება ფუნქცია `sizeof()` იმისათვის, რომ ზუსტად იქნას მითითებული თუ რამდენი ბაიტ მეხსიერების გამოყოფაა საჭირო. მაგალითად:

```
1    int *p;
2    p = malloc(sizeof(int) * 12); // gamoyavi mexsiereba 12 int_tvis
```

მაგრამ რა მოხდება თუ მეხსიერების გამოყოფა ვერ მოხერხდა? ამიტომ უმჯობესია მიმთითებელი ყოველთვის შევამოწმოთ—განსხვავდება თუ არა `NULL`-ისაგან

```
1 int *p;
2 p = malloc(sizeof(float) * 6400);
3 if (p == NULL) {
4 printf("ver moxerxda mexsierebis allokacia!\n");
5 exit(1);
6 }
```

ზემოთ მოყვანილი კოდი შეიძლება ასეც დაიწეროს:

```
1 int *p;
2
3 if (p = malloc(sizeof(float) * 6400) == NULL) {
4 printf("ver moxerxda mexsierebis allokacia!\n");
5 exit(1);
6 }
```

დავალება: გააკეთეთ *C* პროექტი და გაუშვით ზემოთ მოყვანილი კოდი. რა შეცდომაა კოდში და როგორ აისახება ის პროგრამაში?

თუ თქვენ ხშირად იყენებთ malloc() და არ ათავისუფლებთ მეხსიერებას შეიძლება მიიღოთ მეხსიერების გადავსება. მეხსიერების გასათავისუფლებლად გამოიყენება ფუნქცია free()

5.7.2 free() ფუნქცია

ფუნქცია არგუმენტად იღებს მიმთითებელს, რომელიც გამოყენებულ იყო malloc() ან calloc() ფუნქციაში

```
int *p;
p = malloc(sizeof(int) * 37); // 37 ints!
free(p);  // on second thought, never mind!
```

malloc() და free() გამოძახებებს შორის შესაძლებელია გამოყოფილი მეხსიერებით სარგებლობა (ამ შემთხვევაში გვაქვს int ტიპის 37 ელემენტიანი მასივი)

პროგრამირების სწორი პრაქტიკაა გაანთავისფლოთ მეხსიერება იმ პროგრამულ ერთეულში (მაგ. ფაილში, ფუნქციაში) სადაც მოხდა მისი გამოყოფა.

5.7.3 realloc() ფუნქცია

ეს ფუნქცია იყენებს მეხსიერებას, რომელიც გამოიყო malloc() ან calloc() ფუნქციის საშუალებით და ცვლის გამოყოფილი მეხსიერების ზომას. მაგალითად თქვენ გამოყავით მეხსიერება 100 მთელი ტიპისთვის, მაგრამ საჭიროა მისი ზომის შეცვლა 200 მდე. ამ ფუნქციის გამოყენებისას ყურადღებით უნდა ვიყოთ, ვინაიდან realloc() ფუნქციამ *შეიძლება გადაიტანოთ თქვენი მონაცემები მეხსიერების სხვა ადგილას*, თუ მას რატომღაც არ შეეძლება შეცვალოს გამოყოფილი ბლოკის ზომა.

განვიხილოთ მაგალითი:

```
#include <stdlib.h>
#define INITIAL_SIZE 10
#define BUCKET_SIZE 5
static int data_count; // cvladebis raodenoba mexsierebashi
static int data_size;  // ramdeni cvladis shenaxva shegvidzlia
static int *data;      // monacemta bloki
void add_data(int new_data);//axali monacemebis damateba blokshi
int main(void)
{
  int i;
  // inicializacia:
  data_count = 0;
  data_size = INITIAL_SIZE;
  data = malloc(data_size * sizeof(int)); // alokacia
  // axali monacemebis damateba
  for(i = 0; i < 23; i++) {
    add_data(i);
  }
  return 0;
}
void add_data(int new_data)
{
```

```
23 // tu data_count == data_size, bloki savsea
24 // da sachiroebs  realloc()'s sanam davamatebt axal monacems:
25 if (data_count == data_size) {
26 // bloki savsea. vcvlit zomas da
27 // realloc tavidan uketebs alokacias
28 //cxadia dzveli monacemebi ar ikargeba
29 data_size += BUCKET_SIZE;
30 data = realloc(data, data_size * sizeof(int));
31 }
32 // vumatebt axal monacemebs
33 *(data+data_count) = new_data;
34 // ^^^ faqtiurad gvaqvs int masivi
35 // zemot moyvanili kodi igivea rac:
36 //  data[data_count] = new_data;
37 data_count++;
38 }
```

ზემოთ მოყვანილ კოდში საწყისი ზომაა 5, მაგრამ ვცდილობთ შევიტანოთ მასივში 23 ელემენტი. ამიტომ ყოველი ახალი ელემენტის დამატებისას მოწმდება ელემენტების რაოდენობა მასივში. თუ მასივი სავსეა მაშინ ვაკეთებთ 5 ელემენტით მეტი მასივის ალოკაციას.

გასათვალისწინებელია, რომ არც `malloc()` არც `realloc()` ფუნქციით გამოყოფილი მასივები (მეხსიერების ბლოკებში) არაა ინიციალიზირებული (არ აქვთ მონაცემებს მინიჭებული საწყისი, რაიმე მნიშვნელობა). ეს პროგრამისტის ამოცანაა.

მაქსიმალურად უნდა ვერიდოთ `realloc()` ფუნქციის არ გამოყენებას. თავიდანვე რაიმე გზით უნდა ვიცოდეთ თუ რა მონაცემების მასივი (მასივები) გჭირდება. თუ არა მაშინ ხელახალი ალოკაცია უნდა გავაკეთოთ ისე, რომ არ დაგვჭირდეს კიდევ ერთხელ ალოკაცია. ზემოთ მოცემულ მაგალითში ყოველი ალოკაცია საშუალებას იძლევა დამატებულ იქნას 5 ახალი ელემენტი. შეგვეძლო აგვეღო 20 მაშინ ერთზე მეტი რეალოკაცია არ იქნებოდა საჭირო.

5.7.4 calloc() ფუნქცია

`calloc()` ფუნქცია მსგავსია `realloc()` ფუნქციისა ორი განსხვავებით:

1. იღებს ორ მნიშვნელობას ცვლადების რაოდენობას და ცვლადის ზომას
2. ასუფთავებს გამოყოფილ მეხსიერებას

ეს:

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

იგივეა რაც:

```
p = calloc(10, sizeof(int));
```

განსხვავება ისაა, რომ მეორე შემთხვევაში მასივი ინიციალიზირებულია ნულებით.

თავი 6

შედგენილი ტიპები: struct და union

C საშუალებას იძლევა შემოტანილ იქნას ახალი ტიპები: სტრუქტურები struct და გაერთიანებები union

6.1 სტრუქტურები – struct

C არსებული ტიპები საშუალებას გვაძლევს გვქონდეს მოცემული ტიპის ობიექტი. როცა გვაქვს მოცემული ტიპის ბევრი ობიექტი, *C* საშუალებას იძლევა გაერთიენდეს ობიექტები ერთ ობიექტში–მასივში. ანუ მასივი შედგება ერთი და იგივე ტიპის ობიექტებისაგან (გამონაკლისად შეიძლება ჩაითვალოს ადრე განხილული void მიმთითებლების მასივი, მაგრამ ასეთ მასივთან მუშაობა რთულია). მაგრამ ხშირად ამოცანებში გვაქვს ობიეტები, რომლებიც რამდენიმე თვისებით ხასიათდებიან. თუ ასეთი ობიექტების რიცხვი მცირეა მათზე მუშაობა არაა რთული, მაგრამ თუ ბევრია? ვთქვათ 100 ან 1000? ერთ-ერთი გზა შეიძლება იყოს ობიექტის თითოეული თვისებისთვის მასივის შემოტანა. რაც ნიშნავს, რომ თუ ობიეტს ვახასიათებთ მაგალითად 6 თვისებით, უნდა გვქონდეს 6 მასივი. ახალი ობიექტის დამატება ნიშნავს 6 მასივში ახალი ობიექტის დამატებას, წაშლა — 6 მასივში წაშლას და ა.შ., რაც ასევე მოუხერხებელია.

ასეთი ამოცანებისათვის *C* ენაში გათვალისწინებულია struct სტრუქტურა. ბაზური ტიპის ცვლადები შეიძლება გაერთიანებულ იქნან ერთ ობიექტში—სტრუქტურაში.

```
1 struct Employee
2 {
3   char m_name[50];
4   int m_birthDateDay;
5   int m_birthDateMonth;
6   int m_birthDateYear;
7   int m_Salary;
8 };
```

შემოტანილია სტრუქტურა Employee(დაქირავებული), რომელიც შედგება ერთი char ტიპის 50 ელემენტიანი მასივისგან name და ოთხი int ტიპის ობიექტისგან (დაბადების დღე, თვე, წელი და შემოსავალი)

struct სიტყვა არის ინსტრუქცია, რომელიც ამბობს, რომ '{' და '}' მოთავსებული ობიექტები არიან შემოტანილი ტიპის Employee წევრები.

სტრუქტურა არის კომპლექსური მონაცემთა ტიპი. ის შეიძლება შედგებოდეს როგორც ბაზური ტიპების ობიექტებისგან ასევე მასივებისგან. ეს ობექტები შეიძლება იყოს const ტიპისა ან ცვლადები. სტრუქტურაში შეიძლება გვქონდეს მიმთითებლებიც და ასევე სხვა სტრუქტურებიც. სტრუქტურის სახელი უნდა იყოს უნიკალური. ეს სახელი გამოიყენება როგორც სტრუქტურის ტიპის სახელი.

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

struct SEmployee
{
  char m_Name[50];
  int m_BirthDateDay;
  int m_BirthDateMonth;
  int m_BirthDateYear;
  int m_Salary;
};


int main(int argc, char ** argv)
{
   struct SEmployee X;

   strcpy(X.m_Name, "George Saakadze");
   X.m_BirthDateYear = 1980;
   X.m_BirthDateMonth = 12;
   X.m_BirthDateDay = 22;
   X.m_Salary = 50;

   printf("X Sedgeba(%s, %d/%d/%d, %d)\r\n", X.m_Name,
   X.m_BirthDateDay, X.m_BirthDateMonth,
   X.m_BirthDateYear, X.m_Salary);
   return 0;
}
```

ზემოთმოყვანილ მაგალითში X განმარტებულია, როგორც Employee ტიპის ელემენტი. X ცვლადი შედგება შემდეგი წევრებისაგან: `m_Name`, `m_BirthDateYear`, `m_BirthDateMonth`, `m_BirthDateDay`, `m_Salary`. ამ მაგალითში სტრუქტურის ყოველ წევრს ენიჭება მნიშვნელობა და შემდეგ იბეჭდება ეკრანზე.

სტრუქტურის ყოველი წევრი ცვლადი შეიძლება გამოყენებულ იქნას ერთმანეთისგან დამოუკიდებლად. იმისათვის, რომ მივამართოთ სტრუქტურის წევრ ცვლადს. სტრუქტურის წევრ ცლადზე მიმართვისათვის საჭიროა ჯერ მივმართოთ სტრუქტურა ცვლადს: `X.m_Salary` ნიშნავს, რომ მივმართავთ `X` `Employee` ტიპის ცვლადის შიგნით არსებულ წევრ ცვლადს `m_Salary`, რომელსაც შეგვიძლია მივანიჭოთ რაიმე მნიშვნელობა, გამოვიტანოთ ეკრანზე, ჩავწეროთ ფაილში და ა.შ.

შესაძლებელია როგორც სტრუქტურის ცალკეული წევრის გამოყენება ასევე მთლიანად — სტრუქტურა ცვლადით ოპერირება:

```
#include "stdio.h"
#include "string.h"
struct Employee
```

```
4  {
5  char m_Name[50];
6  int m_BirthDateDay;
7  int m_BirthDateMonth; int m_BirthDateYear;
8  int m_Salary;
9  };
10
11 void main()
12 {
13 struct Employee X, Y, Z;
14 strcpy(X.m_Name, "Ahmed Said");
15 X.m_BirthDateDay = 22;
16 X.m_BirthDateMonth = 12;
17 X.m_BirthDateYear = 1980;
18 X.m_Salary = 50;
19 printf("X contains(%s, %d/%d/%d, %d)\r\n", X.m_Name,
20 X.m_BirthDateDay, X.m_BirthDateMonth,
21 X.m_BirthDateYear, X.m_Salary);
22 strcpy(Y.m_Name, X.m_Name);
23 Y.m_BirthDateDay = X.m_BirthDateDay;
24 Y.m_BirthDateMonth = X.m_BirthDateMonth;
25 Y.m_BirthDateYear = X.m_BirthDateYear;
26 Y.m_Salary = X.m_Salary;
27 printf("Y contains(%s, %d/%d/%d, %d)\r\n", Y.m_Name,
28 Y.m_BirthDateDay, Y.m_BirthDateMonth,
29 Y.m_BirthDateYear, Y.m_Salary);
30
31 Z = X;
32
33 printf("Z contains(%s, %d/%d/%d, %d)\r\n", Z.m_Name,
34 Z.m_BirthDateDay, Z.m_BirthDateMonth,
35 Z.m_BirthDateYear, Z.m_Salary);
36 }
```

ფაილის დასაწყისში დეკლარირებულია სტრუქტურა `struct Employee`. 14-ე ხაზზე შემოტანილია `struct Employee` ტიპის სამი ცვლადი. 15-ე ხაზზე გამოყენებულია ფუნქცია `strcpy`. ესაა ე.წ. `string`(სიტყვა) ცვლადზე გამოყენებადი ფუნქცია.

ფუნქცია strcpy

ფუნქცია `char *strcpy(char *dest, const char *src)` აკოპირებს სიტყვას, რომელზეც უთითებს `src` სიტყვაში, რომელსაც უთითებს `dest` და აბრუნებს მიმთითებელს სიტყვაზე `dest`.

16-19 ხაზზე ხდება სტრუქტურის დანარჩენ წევრ ცვლადებზე მნიშვნელობების მინიჭება. 20-ე ხაზზე ხდება `X` სტრუქტურის წევრ ცვლადების ბეჭდვა. 24-27 ხაზზე Y-ს ენიჭება X წევრ-წევრად მინიჭებით: `Y` ცვლად სტრუქტურის წევრებს ენიჭებათ `X` ცვლად სტრუქტურის შესაბამისი ცვლადების მნიშვნელობები. 28-ე ხაზზე ხდება ცვლად `Y` სტრუქტურის ბეჭდვა. ცხადია მისი

მნიშვნელობები იგივეა რაც X ცვლადისა. 32-ე ხაზზე ხდება ერთი სტრუქტურის ტიპის ცვლადის მინიჭება მეორე იგივე ტიპის ცვლადზე— Z ენიჭება X პირდაპირი კოპირებით '=' ოპერატორის საშუალებით. ცხადია ეს მეთოდი უფრო ადვილი და მოსახერხებელია, ვიდრე წევრ-წევრად მინიჭება. ეს მეთოდი ასევე ამცირებს კოდის ზომას. 34-ე ხაზზე ხდება Z ბეჭდვა. საბოლოოდ გავქვს სამი struct Employee ტიპის ცვლადი ერთი და იგივე მნიშვნელობებით.

6.2 სტრუქტურის ცვლადების ინიციალიზაცია

ზემოთ მოყვანილ მაგალითებში სტრუქტურის წევრებზე რაღაც საწყისი მნიშვნელობების მინიჭება არის სტრუქტურის ინიციალიზაცია. ქვემოთ მოყვანილი კოდი ასევე წარმოადგენს სტრუქტურის ცვლადების და შესაბამისად სტრუქტურა ცვლადის ინიციალიზაციას

```
1 void main()
2 {
3 struct Employee X = {"Ahmed Said", 22, 12, 1980, 500};
4 printf("X contains(%s, %d/%d/%d, %d)\r\n", X.m_Name,
5 X.m_BirthDateDay, X.m_BirthDateMonth,
6 X.m_BirthDateYear, X.m_Salary);
7 }
```

6.3 მაგალითი: სტრუქტურების მასივის დალაგება

ქვემოთ მოყვანილ კოდში შემოტანილია struct Employee ტიპის 100 ელემენტიანი მასივი. და ხდება მონაცემთა დამატება. მონაცემები ლაგდება ზრდადობით დაქირავებულის დაბადების თარიღის მიხედვით:

```
1 #include "stdio.h"
2 #include "string.h"
3 
4 struct Employee
5 {
6 char m_Name[50];
7 int m_BirthDateDay;
8 int m_BirthDateMonth;
9 int m_BirthDateYear;
10 int m_Salary;
11 };
12 int main()
13 {
14 struct Employee employees[100], tempEmployee;
15 int count = 0, i, j;
16 char firstName[50], secondName[50];
17 
18 do
19 {
20 printf("Enter Employee First Name:");
21 scanf("%s", firstName);
22 
23 printf("Enter Employee Second Name:");
```

```
24 scanf("%s", secondName);
25 strcpy(employees[count].m_Name, firstName);
26 strcat(employees[count].m_Name, " ");
27 strcat(employees[count].m_Name, secondName);
28
29 printf("Enter Employee Birth Date (day/month/year)
30 example(23/3/1970):");
31 scanf("%d/%d/%d",
32 &employees[count]. m_BirthDateDay,
33 &employees[count].m_BirthDateMonth,
34 &employees[count].m_BirthDateYear);
35 printf("Enter Employee Salary:");
36 scanf("%d", &employees[count].m_Salary);
37
38 count++; if(count==100)break;
39 printf("Do you want to add more, press 'y' to
40 continue?\r\n");
41 }
42 while(getch()=='y');
43
44 for(i=0;i<count-1;i++)
45 {
46 for(j=i+1;j<count;j++)
47 {
48 if(
49 employees[i].m_BirthDateYear>employees[j].m_BirthDateYear ||
50 (employees[i].m_BirthDateYear==employees[j].m_BirthDateYear &&
51 employees[i].m_BirthDateMonth>employees[j].m_BirthDateMonth)||
52 (employees[i].m_BirthDateYear==employees[j].m_BirthDateYear &&
53 employees[i].m_BirthDateMonth==employees[j].m_BirthDateMonth
54 && employees[i].m_BirthDateDay>employees[j].m_BirthDateDay))
55 {
56 tempEmployee = employees[i];
57 employees[i] = employees[j];
58 employees[j] = tempEmployee;
59 }
60 }
61 }
62
63 for(i=0;i<count;i++)
64 {
65 printf("%s\t%d/%d/%d\t%d\r\n",
66 employees[i].m_Name, employees[i].m_BirthDateDay,
67 employees[i].m_BirthDateMonth,
68 employees[i].m_BirthDateYear,
69 employees[i].m_Salary);
70 }
```

```
71 return 0;
72 }
```

employees არის 100 ელემენტიანი მასივი. ისევე როგორც ყველა მასივში აქაც მიმართვა ელემენტზე ხდება კვადრატული ფრჩხილებით და მასივის ინდექსის გამოყენებით. რადგან მასივის ელემენტები წარმოადგენენ სტრუქტურას, i ელემენტის წევრებზე მიმართვა ხდება სტრუქტურის ელემენტზე მიმართვის წესის თანახმად—წერტილი და შემდეგ წევრის სახელი. 57-59-ე ხაზზე ხდება ადგილის გაცვლა სტრუქტურის ორ ლემენტს შორის. როგორც ვხედავთ მონაცემების მინიჭება ხდება '=' ოპერატორის საშუალებით(j სტრუქტურის მონაცემები ენიჭება i სტრუქტურის მონაცემებს).

ზემოთ კოდში შეგვხვდა ახალი ფუნქცია strcat

ფუნქცია strcat

ფუნქცია `char *strcat(char *dest, const char *src)` აკოპირებს სიტყვას, რომელზეც უთითებს src და უმატებს მას სიტყვის ბოლოს, რომელსაც უთითებს dest და აბრუნებს მიმთითებელს სიტყვაზე dest. dest მასივი უნდა იყოს საკმარისი სიგრძის, რომ ჩაეტიოს src მასივი.

6.4 ჩადგმული სტრუქტურები

როგორც ადრე ავღნიშნეთ სტრუქტურის წევრი შეიძლება იყოს ნებისმიერი ტიპისა, მათ შორის სხვა სტრუქტურაც.

ზემოთ მოყვანილ კოდში გვაქვს დაბადების წელი, თვე და რიცხვი. ბუნებრივია, რომ ეს სამი სიდიდე გაერთიანდეს ერთ ტიპში TDate — თარიღი. რაც საშუალებას მოგვცემს ადვილად დამატებულ იქნას დამქირავებლის სტრუქტურაში სხვადასხვა თარიღები გარდა დაბადების თარიღისა, მაგალითად, სამსახურში მიღების თარიღი, სასწავლებლის დამთავრების თარიღი და ა.შ. თუ შემოვიტანთ სტრუქტურას:

```
1 struct TDate
2 { int m_Day;
3   int m_Month;
4   int m_Year;
5 };
```

მაშინ struct Employee იქნება ასეთი:

```
1 struct Employee
2 { char m_Name[50];
3   struct TDate m_BirthDate;
4   struct TDate m_GraduationDate;
5   int m_Salary;
6 };
```

ეხლა სტრუქტურა უფრო ადვილად კითხვადია. ამ სტრუქტურის ინიციალიზაციას და ეკრანზე მნიშვნელობის გამოტანას ექნება შემდეგი სახე

```
1 Employee X = {"Ahmed Said", {22, 12, 1990},
```

```
2  {2, 7, 1970}, 5000};
3  printf("X contains(%s, %d/%d/%d, %d/%d/%d, %d)\r\n",
4  X.m_Name, X.m_BirthDate.m_Day,
5  X.m_BirthDate.m_Month, X.m_BirthDate.m_Year,
6  X.m_GraduationDate.m_Day,
7  X.m_GraduationDate.m_Month,
8  X.m_GraduationDate.m_Year, X.m_Salary);
```

ჩანაწერი `X.m_GraduationDate.m_Year` ნიშნავს, რომ მივმართავთ `X` ცვლადის წევრის `m_GraduationDate` წევრს `m_Year`-ს

დავალება:

შეცვალეთ დალაგების კოდი ჩადგმული სტრუქტურების შემთხვევისათვის

6.5 სტრუქტურების გამოყენება ფუნქციებში

სტრუქტურები შეიძლება გამოყენებულ იქნან ფუნქციებში, როგორც სხვა ცვლადები: გადაცემა ფუნქციაში, სტრუქტურის ობიექტზე მიმთითებლის გადაცემა, სტრუქტურის დაბრუნება, სტრუქტურაზე მიმთითებლის დაბრუნება.

```
1  #include "stdio.h"
2  #include "string.h"
3
4  struct TDate
5  {
6    int m_Day;
7    int m_Month;
8    int m_Year;
9  };
10
11 struct Employee
12 {
13   char m_Name[50];
14   struct TDate m_GraduationDate;
15   struct TDate m_BirthDate;
16   int m_Salary;
17 };
18 struct Date ReadDate(char dateName[])
19 {
20   struct TDate date;
21   printf("Enter %s (day/month/year) example(23/3/1970):",
22   dateName);
23  scanf("%d/%d/%d", &date.m_Day, &date.m_Month,
24  &date.m_Year);
25  return date;
26 }
27 struct Employee ReadEmploee()
28 {
29   struct Employee employee;
```

```
  char firstName[50], secondName[50];

  printf("Enter Employee First Name:");
  scanf("%s", firstName);

  printf("Enter Employee Second Name:");
  scanf("%s", secondName);

  strcpy(employee.m_Name, firstName);
  strcat(employee.m_Name, " ");
  strcat(employee.m_Name, secondName);

  employee.m_BirthDate = ReadDate("Employee Birth Date");
  employee.m_GraduationDate =
  ReadDate("Employee Graduation Date");

  printf("Enter Employee Salary:");
  scanf("%d", &employee.m_Salary);

  return employee;
}
void PrintEmploee(struct Employee emploee)
{
  printf("%s\t%d/%d/%d\t%d/%d/%d\t%d\r\n",
  emploee.m_Name,
  emploee.m_BirthDate.m_Year,
  emploee.m_BirthDate.m_Month,
  emploee.m_BirthDate.m_Day,
  emploee.m_GraduationDate.m_Year,
  emploee.m_GraduationDate.m_Month,
  emploee.m_GraduationDate.m_Day,
  emploee.m_Salary);
}
void main()
{
  struct Employee X = ReadEmploee();
  PrintEmploee(X);
}
```

შემოტანილია დახმარე ფუნქცია `struct Date ReadDate(char dateName[])`,რომელიც კითხულობს და აბრუნებს თარიღს. `struct Employee ReadEmploee()` კითხულობს დაქირავებულის მონაცემებს და აბრუნებს სტრუქტურა ცვლადს, ხოლო `void PrintEmploee(struct Employee emploee)` იღებს სტრუქტურა ცვლადის მნიშვნელობას და ბეჭდავს ეკრანზე.

შეგვიძლია ასევე შემოვიტანოთ ორი თარიღის შედარების ფუნქცია:

```
int CompareDates(struct Date A, struct Date B)
```

```
{
  if( A.m_Year == B.m_Year &&
  A.m_Month == B.m_Month &&
  A.m_Day == B.m_Day)
  return 0;

  if(  A.m_Year > B.m_Year ||
  (A.m_Year == B.m_Year && A.m_Month > B.m_Month) ||
  (A.m_Year == B.m_Year && A.m_Month == B.m_Month &&
  A.m_Day > B.m_Day))
  return 1;

  return -1;
}

void main()
{
  struct Date X = ReadDate("X Date");
  struct Date Y = ReadDate("Y Date");
  switch(CompareDates(X, Y))
  {
    case -1:printf("X date < Y date\r\n");break;
    case  0:printf("X date = Y date\r\n");break;
    case  1:printf("X date > Y date\r\n");break;
  }
}
```

დავალება:

გამოიყენეთ შემოტანილი ფუნქციები და შეცვალეთ დალაგების კოდი ჩადგმული სტრუქტურების შემთხვევისათვის.

დავალება:

დაქირავებულის სტრუქტურაში დამატეთ სქესი როგორც enum GENDER ტიპის ცვლადი. შედარების ფუნქცია შეცვალეთ ისე, რომ ჯერ იყოს მდედრობითი ხოლო შემდეგ მამრობითი სქესის დაქირავებულები.

დავალება:

დაწერეთ სტრუქტურა კომპლექსური რიცხვისათვის. შემოიტანეთ კომპლექსური რიცხვის წაკითხვის, ბეჭდვის,შეკრება-გამოკლების, გაყოფის და შედარების ფუნქციები. დაწერეთ ფუნქციები, რომელბიც აბრუნებენ კომპლექსური რიცხვის მოდულს და ფაზას და მოდული და ფაზის დაშუალებით აბრუნებენ კომპლექსურ რიცხვს.

6.6 სტრუქტურაზე მიმთითებელი

ზემოთმოყვანილი კოდი ცხადია მუშაობს, მაგრამ აქვს ერთი ნაკლი—სტრუქტურები ფუნქციებში გადაეცემა მნიშვნელობით. ანუ იქმნება დროებითი ცვლად სტრუქტურა, რომელსაც ენიჭება გადაცემული სტრუქტურის მონაცემები და ეს გამოიყენება ფუნქციაში.

შეგვიძლია შემოვიტანოთ მიმთითებელი სტრუქტურაზე და საჭირო ფუნქციებში გადავცეთ

არა მთლიანად სტურქტურა არამედ მითითებელი. ასევე შესაძლებელია სტრუქტურის მიმთითებლიდან მივმართოთ იმ ცვლადების მნიშვნელობებს, რომლებიც წარმოადგენენ სტრუქტურის წევრებს განვიხილოთ მაგალითი:

```
void PrintEmploee(struct Employee* pEmploee)
{
  printf("%s, %d/%d/%d, %d/%d/%d, %d\r\n",
  pEmploee->m_Name,
  pEmploee->m_GraduationDate.m_Day,
  pEmploee->m_GraduationDate.m_Month,
  pEmploee->m_GraduationDate.m_Year,
  pEmploee->m_BirthDate.m_Day,
  pEmploee->m_BirthDate.m_Month,
  pEmploee->m_BirthDate.m_Year,
  pEmploee->m_Salary);
}

void main()
{
  struct Employee X = {"Ahmed Said", {22, 12, 1990},
    {2, 7, 1970}, 5000};
  PrintEmploee(&X);
}
```

როგორც ვხედავთ მიმთითებელი ფუნქციის არგუმენტად შემოტანილია ზუსტად ისევე როგორც ეს ადრე გვქონდა განხილული. ფუნქციის გამოყენებისას მას გადაეცემა სტრუქტურის მისამართი. მისამართის აღებაც ზუსტად მუშაობს ისევე როგორც ეს ჩვეულებრივი ცვლადების შემთხვევაში გვქონდა.

ერთი სიახლე არის სტრუქტურის მიმთითებლიდან სტრუქტურის წევრებზე მიმართვა. თუ სტრუქტურა ცვლადისათვის გამოიყენებოდა წერტილი, სტრუქტურა მიმთითებლის შემთხვევაში გამოიყენება '->'

მიმთითებლის გარეშე მაგალითში, როცა ფუნქციას გადაეცემოდა სტრუქტურა მნიშვნელობით `PrintEmploee(X)` ფუნქცია იღებს ცვლადის მნიშვნელობას რაც მოცემულ შემთხვევაში არის $50 + 3 \cdot 4 + 3 \cdot 4 + 4 = 78$ ბაიტი.

ბოლო მაგალითში კი (`PrintEmploee(&X)`) ფუნქცია იღებს მხოლოდ ცვლადის მისამართს, როგორც არგუმენტს, რაც ნიშნავს, რომ გადაეცემა მხოლოდ 4 ბაიტი და პროგრამა სრულდება უფრო სწრაფად.

6.7 typedef და struct

სტრუქტურას შესაძლებელია ჰქონდეს გრძელი სახელი. typedef შესაძლებლობას იძლევა შემოტანილ იქნას ცვლადის ახალი სახელი. ზემოთ მოყვანილი მაგალითისთვის შეგვიძლია დავწეროთ შემდეგი:

```
typedef struct Employee t_Employee;
```

და გამოვიყენოთ კოდში როგორც

```
void main()
{
  t_Employee X = {"Ahmed Said", {22, 12, 1990},
    {2, 7, 1970}, 5000};
  PrintEmploee(&X);
}
```

typedef შესაძლებელია გამოყენებულ იქნას სტრუქტურის შექმნისას

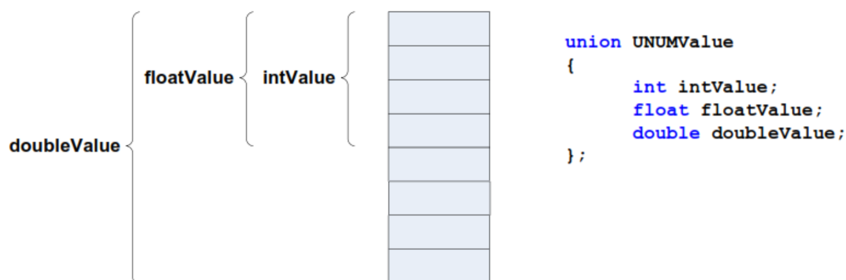
```
typedef struct {
    int a;
    float b;
} someData;
```

და შემდეგ გამოყენებულ იქნას როგორც ჩვეულებრივი ცვლადი someData X;

6.8 გაერთიანება – union

uninon არის მონაცემთა სპეციალური ტიპი სადაც ადგილი აქვს სხვადასხვა ტიპების გადაფარვას. ეს ნიშნავს შემდეგს. მაგალითად გვაქვს სტრუქტურა, რომელიც შედგება ერთი char და ერთი double ტიპის ცვლადისგან. ამ ცვლადებს შეიძლება ორივეს მიენიჭოს მნიშვნელობა. ასეთი სტრუქტურა მეხსიერებაში დაიკავებს ადგილის sizeof(char)+sizeof(double).

მაგრამ თუ გვექნება გაერთიანება, რომელიც შედგება იგივე ტიპის ცვლადებისაგან ის მეხსიერებაში დაიკავებს მხოლოდ sizeof(double) ზომას. ანუ გაერთიანება ცვლადი მხესიერებაში იკავებს უდიდესი წევრის შესაბამის მეხსიერებას.



სურ 6.1: გაერთიანება

რაც ნიშნავს, რომ გაერთიანების წევრი ცვლადები ეს სხვადასხვა ცვლადი კი არაა არამედ წარმოადგენენ ერთ — გაერთიანება ცვლადს, რომელიც შეიძლება იყოს იმ ტიპისა რა ტიპის წევრებიცაა გაერთიანებაში და იკავებს გაერთიანების წევრის მაქსიმალურ ზომას. გაერთიანების წევრებიდან მხოლოდ ერთს შეიძლება მიენიჭოს რაიმე მნიშვნელობა. თუკი მანამდე მნიშვნელობა მინიჭებული ჰქონდა სხვა წევრ ცვლადს მაშინ უკანასკნელი მინიჭების შემდეგ წინა წევრ ცვლადის გამოყენება იქნება შეცდომა. გაერთიანება საშუალებას იძლევა ერთი და იგივე ცვლადი(გაერთიანება ცვლადი) წარმოდგენილი იყოს სხვადასხვა სახით.

მაგალითად ზემოთმოყვანილ სურათზე UNUMValue ცვლადის ზომაა sizeof(double), ხოლო მნიშვნელობა შეიძლება იყოს(უფრო სწორედ მნიშვნელობა შეიძლება ჰქონდეს) წევრ ცვლადებიდან ერთ-ერთი.

განვიხილოთ მაგალითი:

```
1  #include <stdio.h>
2
3  enum NUMType{INT, FLOAT, DOUBLE};
4  union UNUMValue
5  {
6    int u_intValue;
7    float u_floatValue;
8    double u_doubleValue;
9  };
10
11 union UNUMValue Add(union UNUMValue value1, union UNUMValue
12 value2, enum NUMType type)
13 {
14   union UNUMValue result;
15
16   switch(type)
17   {
18     case INT: result.u_intValue = value1.u_intValue +
19     value1.u_intValue; break;
20
21     case FLOAT: result.u_floatValue = value1.u_floatValue +
22     value1.u_floatValue; break;
23
24     case DOUBLE: result.u_doubleValue = value1.u_doubleValue
   + value1.u_doubleValue; break;
25   }
26
27   return result;
28 }
29 void main()
30 {
31   union UNUMValue V1, V2, R;
32
33   printf("the size of UNUMValue is %d bytes\n\n",
34   sizeof(union UNUMValue));
35
36   V1.u_intValue = 9898;
37   V2.u_intValue = 8776;
38   R = Add(V1, V2, INT);
39   printf("int: %d + %d = %d\n",
40   V1.u_intValue, V2.u_intValue, R.u_intValue);
41
42   V1.u_floatValue = 86.82;
43   V2.u_floatValue = 83.11;
44   R = Add(V1, V2, FLOAT);
```

```
45  printf("int: %f + %f = %f\n",
46  V1.u_floatValue, V2.u_floatValue, R.u_floatValue);
47
48  V1.u_doubleValue = 821.8;
49  V2.u_doubleValue = 988.2;
50  R = Add(V1, V2, DOUBLE);
51  printf("int: %lf + %lf = %lf\n",
52   V1.u_doubleValue, V2.u_doubleValue, R.u_doubleValue);
53  }
```

დავალება:

აღწერეთ როგორ მუშაობს ზემოთ მოყვანილი კოდი. შეგახსენებთ, რომ მას შემდეგ რაც გაერთიანების double ტიპის ცვლადს მიენიჭა მნიშვნელობა, გაერთიანების int და float ცვლადების გამოყენება იქნება შეცდომა.

დავალება:

დაწერეთ კვადრატული განტოლების ამოხსნის პროგრამა ფუნქციების და სტრუქტურების გამოყენებით. კერძოდ, ფუნქცია იღებს მიმთითებელს სტრუქტურაზე, რომელიც შედგება კოეფიციენტებისაგან და გამოთვლის ჩამონათვალი ტიპის (enum) სტატუსისაგან (არ აქვს ამოხნა, აქვს ერთი ამოხსნა, აქვს ორი ამოხსნა) და აბრუნებს მიმთითებელს სტრუქტურაზე სადაც არის კვადრატული განტოლების ფესვები.

დავალება:

დაწერეთ კვადრატული განტოლების ამოხსნის პროგრამა ფუნქციების და სტრუქტურების გამოყენებით. ისე როგორც ზემოთ მაგალითში, მაგრამ აქ სტრუქტურის ნაცვლად ფუნქციაში გადაეცემოდეს გაერთიანება, რომლის წევრებია მთელი და ორმაგი სიზუსტის წევრების მქონე სტრუქტურები. შესაბამისად ფუნქცია უნდა აბრუნებდეს გაერთიანებას სადაც არის კვადრატული განტოლების ფესვები.

თავი 7

ბიტური ოპერაციები

C ენაში გვაქვს ბიტებზე მოქმედების ექვსი ფუნქცია: და–(AND) &, ან–(OR)|, ექსკლუზიური ან–(exclusive OR ანუ (XOR)) ^, არა–(NOT)~, მარჯვნივ წანაცვლება –(left-shift) <<, და მარცხნივ წანაცვლება –(right-shift) >>. ეს ოპერაციები შეიძლება გამოყენებულ იქნას მხოლოდ მთელი ტიპის ცვლადებზე: `char`, `short`, `int` და `long`, რომლებიც შეიძლება იყოს ნიშნიანი ან უნიშნო(`unsigned`). არ შეიძლება ამ ოპერაციების გამოყენება ცვლადებზე მცოცავი მძიმის ტიპით.

ოპერატორებს:&, |, ^, << და >> აქვთ შესაბამისი მინიჭების ოპერატორები:&=, |=, ^=, <<=,და >>=. ეს ოპერატორები მოქმედებენ ანალოგიურად მათემატიკური მინიჭების ოპერატორებისა.მაგალითად:

```
z&=x|y;
```

იგივეა რაც

```
z=z&(x|y);
```

მნიშვნელოვანია გვახსოვდეს, რომ ბიტური ოპერატორები: &, |, და ~, განსხვავდებიან ლოგიკური ოპერატორებისგან: და (AND) &&,ან (OR)||, არა (NOT) !. განსხვავებებს მათ შორის ქვემოთ ვნახავთ.

7.1 AND, OR, XOR და NOT

ოპერატორები &, ან–(OR)|, ექსკლუზიური ან–(exclusive OR ანუ (XOR)) ^, არა–(NOT)~ საშუალებას იძლევა ბიტებზე ჩატარდეს ლოგიკური ოპერაციები. უარყოფის ოპერატორი ~ უნარულია–ის 0 ცვლის 1-ით და პირიქით. დანარჩენი ოპერატორები ბინარულია. ისინი ადარებენ ორ ბიტს და იძლევიან შედეგს შემდეგი წესის მიხედვით:

ცხრილი 7.1: ბინარული ბიტური ოპერაციები

x	y	x&y	x|y	x^y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

განვიხილოთ 8 ბიტიანი (1 ბაიტი) მაგალითი:

```
unsigned char x = 55; /* 55 (dec) = 0x37 (hex) = 0011 0111 (binary). */
unsigned char y = 25; /* 25 (dec) = 0x19 (hex) = 0001 1001 (binary). */
unsigned char z;
```

და განვიხილოთ z=x&y:

```
0011 0111
0001 1001 &
0001 0001 /* result is 17 (dec) = 0x11 (hex) */
```

ბიტური და & (AND) z ცვლადის ბიტებს აქცევს ერთიანად თუ x და y შესაბამისი ბიტები ტოლია ერთისა. ეს ოპერატორი ხშირად გამოიყენება ზოგიერთი ბიტების განულებისთვის და ზოგიერთი ბიტების შერჩევისთვის შემდგომი შემოწმებისთვის.

ოპერაციის z=x|y შედეგად z იქნება 63:

```
0011 0111
0001 1001 |
0011 1111 /* result is 63 (dec) = 0x3f (hex) */
```

ბიტური ან | (OR) z ცვლადის ბიტებს აქცევს ერთიანად თუ x და y შესაბამისი ერთი ბიტი მაინც ტოლია ერთისა. ეს ოპერატორი ხშირად გამოიყენება შერჩეული ბიტებისთვის ერთიანის მინიჭებისთვის.

მესამე ოპერაციის. „ექსკლუზიური ან“(XOR) z=x^y შედეგად z იქნება 46:

```
0011 0111
0001 1001 ^
0001 1110 /* result is 46 (dec) = 0x2e (hex) */
```

ბიტური ^ (XOR) z ცვლადის ბიტებს აქცევს ერთიანად თუ x და y შესაბამისი ერთი ბიტი მაინც და არა ორივე ერთდროულად ტოლია ერთისა. ეს ოპერატორი ხშირად გამოიყენება შერჩეული ბიტებისთვის მნიშვნელობის შეცვლისთვის(ნულს აქცევს ერთად და ერთიანს ნულად).

მეოთხე ოპერაცია ~(მას ერთამდე დამატებასაც უწოდებენ) ცვლის 1 ბიტს 0-ით და 0-ს 1-ით. ეს ოპერაცია ხშირად გამოიყენება ბიტების შებრუნებისთვის წანაცვლების ოპერაციებთან ერთად. z=~x:

```
0011 0111 ~
1100 1000 /* result is 200 (dec) = 0xc8 (hex) */
```

7.2 მარჯვნივ და მარცხნივ წანაცვლების ოპერაციები

წანაცვლების ოპერატორები << და >> ანაცვლებენ მთელი ცვლადის ბიტებს მარცხნივ ან მარჯვნივ შესაბამისად. მარცხნივ წანაცვლების ოპერატორი x<<n ანაცვლებს x ცვლადის ბიტებს n პოზიციით მარცხნივ. თანრიგის გარეთ მარცხნივ გასული ბიტები იკარგება ხოლო მარჯვნიდან „შემოსული“ ბიტები არის ნულები.

```
1 char x = 0x19; /* Equals 0001 1001 binary (25 decimal). */
2 char y = x << 2; /* Equals 0110 0100 binary (100 decimal). */
```

უარყოფითი სიდიდით წანაცვლება ან წანაცვლება რაოდენობით რომელიც მეტია ცვლადის ბიტების რაოდენობაზე შეცდომაა და არ შეიძლება.

მარჯვნივ წანაცვლების ოპერატორი შედარებით უფრო რთულია. თუ x უნიშნო ცვლადია მაშინ მარჯვნივ წანაცვლების ოპერაცია x>>n მუშაობს მსგავსად მარცხნივ წანაცვლებისა—„გაუსილი“ ბიტები იკარგება და მარცხნიდან „შემოსული“ ბიტების მნიშვნელობა ნულია. თუ x ნიშნიანი ცვლადია მაშინ მარცხნივ წანაცვლება შეიძლება სხვადასხვა იყოს სხვადასხვა. ზოგიერთ მაქნქანაზე კეთდება ე.წ. „ლოგიკური წანაცვლება“ და „შემოსული“ ბიტები ნულებია,

ზოგ მანქანზე კი კეთდება „მათემატიკური წანაცვლება“. რაც ნიშნავს, რომ შემოსული ბიტები ცვლადის ნიშნის შესაბამისია. მაგალითად:

```
signed char x = -75; /* Equals 1011 0101 binary. */
signed char y = x >> 2; /* Equals 0010 1101 if logical shift. */
/* Equals 1110 1101 if arithmetic shift. */
```

ამგვარად ზოგიერთ მანქანაზე ნიშნიანი ცვლადის მარჯვნივ წანაცვლებამ შეიძლება მოგვცეს 45, მეორე შემთხვევაში — -19. ზოგადად, წანაცვლების ოპერატორები უპირატესად გამოიყენება მხოლოდ უნიშნო ცვლადებზე, რათა თავიდან იქნას აცილებული წანაცვლების ოპერატორების არაპორტაბელური ყოფაქცევა.

ადვილია შევნიშნოთ, რომ მარცხნივ წანაცვლება იგივეა რაც 2^n გამრავლებას, ხოლო შესაბამისი მარჯვნივ წანაცვლება — 2^n გაყოფას.

ბიტური ოპერაციები ხშირად გამოიყენება, როგორც მატემატიკური ოპერაციები მთელ უნიშნო რიცხვებზე. მაგალითად, ეკვივალენტურია ქვემოთ მოყვანილი გამოსახულებები:

`x * 2` იგივეა რაც `x << 1`

`x / 16` იგივეა რაც `x >> 4`

`x % 8` იგივეა რაც `x & 7`

ბიტური ოპერაციები უფრო სწარფად სრულდება ვიდრე მათემატიკური ოპერაციები, მაგრამ ზემოთმოყვანილი გამოსახულებების მსგავსი „ოპტიმიზაციები“ ზედმეტია. ვინაიდან თანამედროვე კომპილატორებმა იციან როგორ გააკეთონ 2^n გამრავლება/გაყოფის ოპერაციების ბიტურ დონეზე ოპტიმიზაცია.

7.3 ბიტური ოპერაციების თანმიმდევრობა

ბიტური ოპერაციების თანმიმდევრობა („წონა“) უფრო „დაბალია“ ვიდრე არითმეტიკული ოპერაციებისა, გარდა უნარული უარყოფის `~` ოპერატორისა, როემლსაც იგივე „წონა“(precendence) აქვს (ანუ უფრო ადრე სრულდება) რაც ლოგიკურ უარყოფის ოპერატორს `!`. მარჯვნივ და მარცხნივ წანაცვლების ოპერატორების „წონა“ უფრო მეტია ვიდრე შედარების ოპერატორებისა `<`, `>`, მაგრამ `&`, `|` და `^` „წონები“ უფრო დაბალია. პროცედურა `&` უპირატესია ვიდრე `^`, რომელიც თავის მხრივ უპირატესია ვიდრე `|`. ყველა ბიტურ ოპერაციას აქვს უფრო მეტი „წონა“ ვიდრე `&&` და `||`.

პროგრამირების სწორი სტილია, კოდში გაუგებრობის თავიდან ასაცილებლად, გამოყენებულ იქნას ფრჩხილები, როცა გვაქვს ლოგიკური და ბიტური ოპერაციები გამოსახულებებში.

7.4 ბიტური ოპერაციების გამოყენება

ბიტური ოპერაციები ხშირად გამოიყენება ორი მიზნით:

1. მეხსიერების დაზოგვა და ერთი ცვლადის გამოყენება რამდენიმე ცვლადის ნაცვლად ე.წ. „მდგომარების შესახებ“ ინფორმაციის შენახვისათვის (ე.წ. „ნიშნულები“(flags) სადაც შენახულია 0 ან 1, რითაც შეგვიძლია დავახასიათოთ რაღაც მდგომარეობა პროგრამაში. მაგალითად, თაგუნას რომელი ღილაკია დაჭერილი, თაგუნა მოძრაობს თუ არა და ა.შ.)

2. გამოყენების მეორე არეა ე.წ. მოწყობილობასთან (hardware) ურთიერთობა (მიკროპროცესორები, ჩაშენებული სისტემები), სადაც ბიტების რაღაც ჯგუფი მოცემულ მისამართზე შეესაბამება მოწყობილობის ელექტროდების („პინების“) რაღაც ჯგუფს.

ორივე შემთხვევაში ჩვენ გვჭირდება ბიტებზე ან ცალკეულ ბიტზე ოპერაციების შესრულება.

ქვემოთ მოყვანილ მაგალითში განვიხილავთ ინდივიდუალური ბიტების შეცვლას („ჩართვა/გამორთვა“) და ბიტის შემოწმებას (ნულია თუ ერთი). ამ ოპერაციებს უწოდებენ „masking“(„შენიღბვა“??) ვინაიდან ეს ოპერაციები საშუალებას იძლევა შეიცვალოს ბიტები წინასწარ მოცემული „ბიტ-ნიმუშის“ („ბიტ-ნიღბის“ bit-mask) მიხედვით. „ნიღბის“(mask) შექმნისათვის პირველი ნაბიჯია განვმარტოთ ცვლადები, რომლებიც წარმოადგენენ მთელი ცვლადის ბიტებს[1] (განვიხილავთ მხოლოდ პირველ ოთხ დაბალ ბიტს)

```
1   enum {
2   FIRST = 0x01, /* 0001 binary */
3   SECND = 0x02, /* 0010 binary */
4   THIRD = 0x04, /* 0100 binary */
5   FORTH = 0x08, /* 1000 binary */
6   ALL = 0x0f /* 1111 binary */
7   };
```

თითოეული მუდმივა წარმოადგენს ორის ხარისხს. ანუ მხოლოდ ერთი ბიტი არის ნულისგან განსხვავებული. გამონაკილისია ცვლადი ALL, რომელიც წარმოადგენსს ნიღაბს(ნიმუშს „mask“) ყველა, ოთხივე ბიტის მოსანიშნად.

ზემოთ მოყვანილი მუდმივებს შემოტანის ეკვივალენტური გზაა მარცხნივ წანაცვლების ოპერატორის გამოყენება.

```
1 enum {
2 FIRST = 1 << 0,
3 SECND = 1 << 1,
4 THIRD = 1 << 2,
5 FORTH = 1 << 3,
6 ALL = ~(~0 << 4)
7 };
```

ბოლო ხაზი ზემოთ მოყვანილ კოდში შეიძლება გაუგებარი ჩანდეს ამიტომ განვმარტოთ: ბოლო ხაზი შედგება სამი ოპერაციისაგან. პრიველია ~0, რაც ნიშნავს მოცემული ტიპის ცვლადის ყველა ბიტის შევსებას ერთიანებით. მაგალითად, თუ დავუშვებთ, რომ ცვლადი არის რვა ბიტიანი მაშინ ეს სამი ოპერაცია „გაიშიფრება“ შემდეგნაირად:

1111 1111 /* ~0 ყველა ბიტი არის ერთიანი*/

1111 0000 /* ~0 << 4 მიღებული შედეგი წანაცვლდა მარცხნივ 4 ბიტით*/

0000 1111 /* ~(~0 << 4) რაც მივიღეთ მეორე ნაბიჯზე მოხდა მისი უარყოფა*/

შედეგად მივიღეთ ის რაც გვინდოდა,რომ ყველა დაბალი 4 ბიტი არის ერთიანი.

ბიტების „ნიმუშების“ („ნიღბების“ masks) კომბინაციით შეგვიძლია „ჩავრთოთ/გამოვრთოთ/გადავრთოთ“ ერთი მოცემული ბიტი ან ბიტების ჯგუფი, შევამოწმოთ რაიმე პირობაზე და ა.შ. განვიხილოთ მაგალითი:

```
1 unsigned flags = 0;
2 flags |= SECND | THIRD | FORTH; /*ჩართე(set) ბიტები 2,3,4 (1110).*/
```

[1] enum ნაცვლად შეიძლება გვქონდეს struct ან union-იც კი

```
flags &= ~(FIRST | THIRD); /*გამორთე (reset) ბიტები 1,3 (1010).*/
flags ^= THIRD | FORTH; /*გადართე(toggle) ბიტები  3,4 (0110).*/
if ((flags & (FIRST | FORTH)) == 0) /*TRUE თუ 1 და 4 გამორთულია.*/
flags &= ~ALL; /*გამორთე ყველა ბიტი  (0000).*/
```

ზემოთ მოყვანილ მაგალითში აღსანიშნავია შემდეგი:

1. გამოყენებულია | ოპერატორი „ნიღბის“(„ნიმუშის“, mask) კომბინაციისთვის და უარყოფის ოპერატორი ~ მიღებული შედეგის უარყოფისთვის (ნულების შეცვლა ერთიანებით და პირიქით), ისე, რომ ყველა ბიტი იქნება ერთიანი გარდა იმ ბიტებისა, რომელიც მოცემულია „ნიღაბში“.

2. ყურადღება მიაქციეთ მინიჭების ოპერატორებს, რომლებიც გამოიყენება მარცხენა მხარეს არსებული ცვლადისათვის „ნიღბებთან“ კომბინაციის და შენდეგ მინიჭებისას.

3. |= ოპერატორი გამოიყენება ბიტების ჩასართავად და &= ოპერატორი ბიტების გამოსართავად. ეს ოპერაციები იძლევიან ბიტების მდგომარეობას ცვლადში `flags` მიუხედავად მათი მიმდინარე მნიშვნელობებისა(მდგომარეობისა ანუ ნულია თუ ერთი).

4. მეორე მხრივ, ოპერატორი ^= გამოიყენება წინასწარ შერჩეული ბიტების გადასართავად(თუ ნულია იცვლება ერთით და პირიქით). ანუ ბიტების მდგომარეობა იცვლება საპირისპიროთი.

5. და ბოლოს მიაქციეთ ყურადღება როგორ შეიძლება ბიტური ოპერაციები გამოყენებულ იქნას პირობით ოპერატორებთან ერთად ბიტების მიმდინარე მდგომარეობის შემოწმებისათვის. ინდივიდუალური ბიტები ან ბიტების ჯგუფი შეიძლება მონიშნულ (არჩეულ) იქნას & ოპერატორის გამოყენებით.

7.5 ბიტ ველები

C შესაძლებელია შემოტანილ იქნას ე.წ. *ბიტური ველები*, როგორც ბიტური ოპერაციების ალტერნავივა — რამდენიმე ობიექტი ჩადგებულ იქნას ერთ ე.წ. მანქანურ სიტყვაში. ამ სიდიდეს ეწოდება *ბიტური ველი* და იყენებს სტრუქტურის სინტაქსს რამდენიმე ობიექტის ერთ ცვლადად განმარტებისათვის. ზემოთ მოყვანილი მაგალითი შეძლება წარმოდგენილ იქნას შემდეგი სახითაც ბიტ ველების საშუალებით:

```
struct {
unsigned first : 1;
unsigned secnd : 1;
unsigned third : 1;
unsigned forth : 1;
} flags;
```

ზემოთ მოყვანილი სტრუქტურა განმარტავს ცვლადს `flags`, რომელიც შედგება ოთხი ერთ ბიტიანი ველისგან. რიცხვი როემლიც წერია ორწერილის შემდეგ წარმოადგენს ველის სიგანეს ბიტებში. ცალკეულ ველებზე შესაძლებელია მიმართვა, როგორც სტრუქტურის წევრებზე. ასე, რომ ინდივიდუალურ ბიტებზე შესაძლებელია მიმართვა/შეცვლა შემდეგი სახით:

```
flags.secnd = flags.third = flags.forth = 1;
flags.first = flags.third = 0;
flags.third = !flags.third;
flags.forth = !flags.forth;
if (flags.first == 0 && flags.forth == 0)
flags.first = flags.secnd =
flags.third = flags.forth = 0;
```

ზემოთმოყვანილი ოპერაციები ბიტ ველებზე იგივეა რაც ადრე განხილული ბიტური ოპერაციები.

დამწყები *C* პროგრამისტისთვის შეიძლება ბიტ ველები უფრო ბუნებრივი და ადვილი წარმოდგენა იყოს ვიდრე ბიტური ოპერაციებით ერთ ცვლადზე მოქმედება, მაგრამ ეს მოჩვენებითი უპირატესობაა და ბიტ ველები გამოყენებულ უნდა იქნას დიდი ყურადღებით. პრობლემა ისაა, რომ ბიტ ველები დამოკიდებულია მათ იმპლემენტაციაზე („Implementation dependent“) და მათ გამოყენებამ შესაძლებელია მოგვცეს არა პორტაბელური კოდი. მაგალითად, სხვადასხვა მანქანებს აქვთ სხვადასხვა შეზღუდვა ბიტ ველის სიგრძეზე. ზოგიერთ მანქანაში მაქსიმალური სიგრძე შეიძლება იყოს 16 ბიტ, ზოგშიც — 32. ასევე ბიტური ველები შეიძლება განსხვავებოდნენ სტრუქტურაში ჩალაგების გზით(რამდენი ბიტი უჭირავს თითოეულს სტრუქტურაში). კიდევ ერთი საკითხია ე.წ. ბიტების თანმიმდევრობა(დიდი ბიტი მარჯვნივაა თუ მარცხნივ).

ზოგადად მიღებულია, რომ უმჯობესია გამოყენებულ იქნას ბიტური ოპერაციები ნაცვლად ბიტ ველებისა. ბიტური ოპერაციები საშუალებას იძლევა უშუალოდ ვაკონტროლოთ ბიტების ჩალაგება, ბიტების ჯგუფებზე მოქმედება ბიტ „ნიღბების“ გამოყენებით.

შესაძლებელია შემოტანილ იქნას მაკროსები, როცა საჭიროა ცალკეულ ბიტებზე მუშაობა მაგ. ყვემოთ მოყვანილი მაკროსები[1]

```
#define BitSet(arg,posn) ((arg) | (1L << (posn)))
#define BitClr(arg,posn) ((arg) & ~(1L << (posn)))
#define BitFlp(arg,posn) ((arg) ^ (1L << (posn)))
#define BitTst(arg,posn) ((arg) & (1L << (posn)))
```

ჩართავენ,გამორთავენ,ცვლიან,ამოწმებენ `arg` ცვლადის `posn` პოზიციაზე მყოფ ბიტს. ეს მაკროსები შესაძლებელია გამოყენებულ იქნას მაგალითად, შემდეგი სახით:

```
enum { FIRST, SECND, THIRD, FORTH };
unsigned flags = 0;
flags = BitSet(flags, FIRST); /* Set first bit. */
flags = BitFlp(flags, THIRD); /* Toggle third bit. */
if (BitTst(flags, SECND) == 0) /* Test second bit. */
flags = 0;
```

[1]მაკროსები მოყვანილია საიტიდან http://www.snippets.org

თავი 8

ფაილში ჩაწერა და წაკითხვა

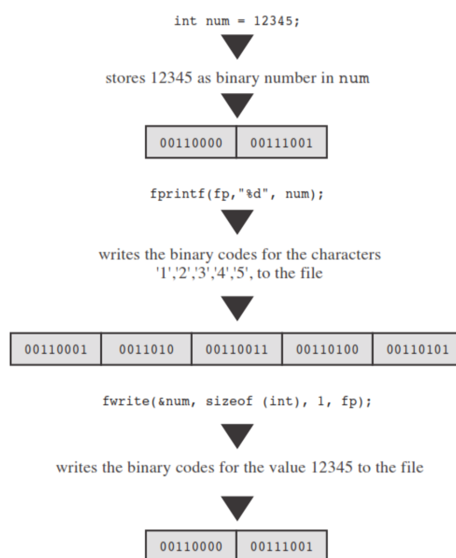
ფაილი არის სექცია დისკზე ან ssd მოწყობილობაზე, რომელსაც აქვს სახელი. ოპერატიული სისტემისთვის ფაილს რთული სტრუქტურა აქვს. ის შეიცავს ინფორმაციას ფაილის და ფაილზე უფლებების შესახებ. ფაილი შეიძლება ჩაწერილი იყოს დისკზე სხვადასხვა ფრაგმენტად. C პროგრამისთვის ფაილი არის ბაიტბის მიმდევრობა. თითოეული ბაიტი შეიძლება იქნას წაკითხული და ჩაწერილი. ესაა ფაილის სტრუქტურა იუნიქს სისტემებში სადაც წარმოიშვა C. ვინაიდან სხვა სისტემებში შეიძლება იყოს ფაილის სხვაგვარი მოდელი C-ში უზრუნველყოფილია ფაილის წარმდგენის ორი გზა ტექსტური და ბინარული

8.1 ტექსტური და ბინარული მოდა

ზოგადად ყველა ფაილი ბინარულია ანუ ჩაწერილია როგორც ნულების და ერთიანების მიმდევრობა.

ტექტსური მოდა ნიშნავს, რომ ფაილი იყენებს ბინარულ კოდებს ასო ნიშნებისათვის. რაც ნიშნავს, რომ თუ ასეთ ფაილს გავხსნით ტექსტურ ედიტორში მისი შინაარსი გამოჩნდება ბეჭდური ასო ნიშნების საშუალებით.

ბინარული მოდა ნიშნავს, რომ ფაილში ინფორმაცია ჩაწერილია ზუსტად ისე როგორც მას ხედავს პროგრამა/ოპერატიული სისტემა.



სურ 8.1: ბინარული და ტექსტური ფორმატი

მოდის გარდა ფაილის შინაარსიც შეიძლება იყოს ტექსტური ან ბინარული. ასევე ფორმატიც შეიძლება იყოს ტექსტური ან ბინარული.

იუნიქსი იყენებს ერთი და იგივე ფაილის ფორმატს როგორც ტექსტური ასევე ბინარული ფაილებისთვის. "\n"აღნიშნავს ახალ ხაზზე გადასვლას იუნიქს სისტემაში და C-ში. იუნიქს დი-

რექტორიები შეიცავენ ინფორმაციას ფაილის ზომის შესახებ, რაც შეიძლება გამოყენებულ იქნას ფაილის დასასრულის დეტექტირებისათვის. სხვა სისტემებში ფაილებთან მუშაობის სხვა წესებია. მაგ. OSX მდე სისტემებში მაკინტოშ მანქანებზე ახალ ხაზზე გადასვლას აღნიშნავდა "\r"(Enter ღილაკი), ადრეულ MS-DOS სისტემაში კი "\r\n"და ფაილის დასასრულს "Ctrl-Z"კოდი, იმის მიუხედავად, რომ რეალური ფაილი გრძელდებოდა "Ctrl-Z"კოდის შემდეგ ნულებით, რათა ფაილის ზომა ყოფილიყო 256 ჯერადი. ვინდოუს სისტემაში ნოუთპად პროგრამით შექმნილ ფაილებს აქვთ MS-DOS ფაილის ფორმატი, თუმცა ახალმა პროგრამებმა შეიძლება გამოიყენონ იუნიქსის ფორმატი. სხვა სისტემებში შეიძლება ტექსტური ფაილის ყოველი ხაზი იყოს ერთი და იგივე ზომის, ანუ ზოგიერთი ხაზი შევსებულ იყოს ნულლ ნიშნით, თუ ეს აუცილებელია ერთი და იგივე სიგრძის მისაღწევად. ან შეიძლება ხაზის სიგძე კოდირებული იყოს ყოველი ხაზის დასაწყისში.

იმისათვის, რომ რაღაც წესით წაკითხულ იქნას ტექსტური ფაილი, C ენაში არის ფაილთან წვდომის ორი მოდა: ტექტსური და ბინარული. ბინარულ მოდაში ფაილის შემადგენელი ყოველი ბაიტი ხელმისაწვდომია პროგრამისთვის. ტექსტურ მოდაში კი რასაც პროგრამა ხედავს შეიძლება განსხვავებული იყოს ფაილის შემადგენლობისგან. როგორც ტექსტის წაკითხვისას ხაზის დამთავრების და ფაილის დამთავრების კოდები ინტერპრეტირებულია ისე როგრც ეს არის C ენაში. მაგალითად C პროგრამა, რომელიც დაკომპილირებული იყო მაკინტოშზე "\r"გადაიყვანს "\n"ფაილის წაკითხვისას და "\n"გადაიყვანს "\r"ფაილის ჩაწერისას. ან MS-DOS კომპილირებულ C პროგრამაში "\r \n"ინტერპრეტირდება როგორც "\n"წაკითხვისას ხოლო ჩაწერისას "\n"იწერება როგორც "\r \n".

მოცემული ფაილი შეიძლება გაიხსნას და წაკითხულ იქნას როგორც ტექსტურ ასევე ბინარულ მოდაშიც. ამ შემთხვევაში თუ ფაილი შექმნილი იყო MS-DOS პროგრამას შეუძლია "დაინახოს"როგორც "\r"ასევე "\n"კოდები ანუ ბინარულ მოდაში არ ხდება სპეციალური კოდების ავტომატურად ინტერპრეტირება. თუ საჭიროა, რომ მაგალითად დაიწეროს პროგრამა, რომელიც წაიკითხავს ფაილს როგორც იუნიქ ასევე დოს და მაკინტოშის ფორმატებში ამისათვის ფაილის წაკითხვა უნდა მოხდეს ბინარული მოდით, რომ შესაძლებელი იყოს ხაზის დასასრულის კოდების წაკითხვა/ინტერპრეტირება და შესაბამისი გადაწყვეტილებების მიღება.

სისტემის დონეზე დაბალი დონის ფაილში შეტანა გამოტანის ფუნქციები შეიძლება იყოს სხვადასხვა, მაგრამ C ბიბლიოთეკის ფუნქციები არის ერთი და იგივე, რათა გარანტირებული იყოს C კოდის პორტაბელობა სხვადასხვა სისტემებს შორის.

C პროგრამის გაშვებისას ავტომატურად ხელმისაწვდომია სამი ფაილი: სტანდარტული შეტანის ფაილი (stdin) სტანდარტლი გამოტანის ფაილი (stdout) და შეცდომური შეტყობინებების ფაილი (stderror). სტანდარტული შეტანის მოწყობილობა როგორც წესი არის კლავიატურა. სტანდარტული გამოტანის და შეცდომური შეტყობინებების მოწყობილობები როგორც წესი არის დისპლეი. სტანდარტული შეტანა უზრუნველყოფს მონაცემთა შეტანას პროგრამაში. ესაა ფაილი რომელსაც კითხულობს `getchar()` და `scanf()`. სტანდარტული გამოტანის ფაილს იყენებს `putchar()`,`puts()` და `printf()`.

8.1.1 ტექსტური ფაილის წაკითხვა/ჩაწერა

განვიხილოთ მაგალითი:

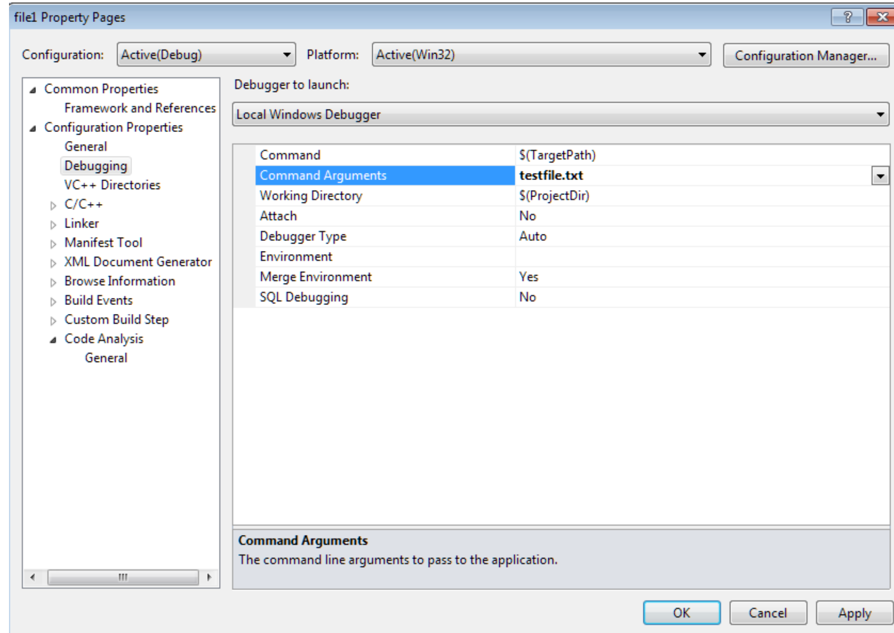
```
1 /* count.c -- using standard I/O */
2 #include <stdio.h>
```

```
3  #include <stdlib.h> // exit() prototype
4
5  int main(int argc, char *argv[])
6  {
7  int ch;          // place to store each character as read
8  FILE *fp;        // "file pointer"
9  unsigned long count = 0;
10 if (argc != 2)
11 {
12 printf("Usage: %s filename\n", argv[0]);
13 exit(EXIT_FAILURE);
14 }
15 if ((fp = fopen(argv[1], "r")) == NULL)
16 {
17 printf("Can't open %s\n", argv[1]);
18 exit(EXIT_FAILURE);
19 }
20 while ((ch = getc(fp)) != EOF)
21 {
22 putc(ch,stdout);  // same as putchar(ch);
23 count++;
24 }
25 fclose(fp);
26 printf("File %s has %lu characters\n", argv[1], count);
27
28 return 0;
29 }
```

ფაილში შეტანა გამოტანის ფუნქციები აღწერილია stdio.h ფაილში. მე-8 ხაზზე ხდება ფაილზე მიმთითებლის დეკლარირება. ფაილზე მიმთითებელი რეალურად უთითებს სტურქტურაზე სადაც მოცემულია ფაილის მონაცემები (შექმნის თარიღი, უკანაკნელი მიმართვის დრო, რომელ მომხმარებელს ეკუთვნის, ჩაწერა/წაკითხვა/გაშვების უფლებები და ა.შ.)

მე-9 ხაზზე ხდება პროგრამისთვის მიწოდებული არგუმენტების შემოწმება. პროგრამის გაშვების შემდეგ argc შენახული იქნება არგუმენტების რაოდენობა. არგუმენტების რაოდენობაში შედის თავად გაშვებული პროგრამის სახელი, სრული გზის ჩათვლით, ხოლო argv იქნება მითითება მასივზე (ანუ ზოგადად ორმაგი მასივი). ამ მასივში ინახება ტექსტური ხაზი რაც არის კონსოლზე. ზემოთ სურათზე ნაჩვენებია პროექტის პარამეტრებიდან როგორ უნდა მივაწოდოთ არგუმენტი. ამ შემთხვევაში მიეწოდება რარაც ტექსტური ფაილის სახელი. როცა ვუშვებთ ბრძანებატა ფანჯრიდან მაშინ ხელიტ უნდა მივუწეროთ საჭირო არგუმენტები. რა არგუმენტები მიეწოდება და როგორ ეს დამოკიდებულია პროგრამაზე. პროგრამის გაშვებისას უნდა შემოწმდეს თუ რა არგუმენტი მიეწოდა პროგრამას და შესაბამისად პროგრამამ უნდა მიიღოს გადაწყვეტილება შემდგომ ნაბიჯზე.

მოცემულ შემთხვევაში argc=2 ანუ არგუმენტების რაოდენობა უნდა იყოს 2 (ერთი თავად პროგრამის სახელი, მეორე მიწოდებული ტექსტური ფაილის). შესაბამისად მე-10 ხაზზე ხდება შემოწმება. თუ არგუმენტების რაოდენობა არაა ორი მაშინ პროგრამა მთავრდება შეცდომის



სურ 8.2: არგუმენტის მიწოდება პროექტიდან

შეტყობინებით `exit()` ფუნქცია დეკლარირებულია `stdlib.h`. `return 0;` იგივეა რაც `exit(0)` რაც სისტემას ატყობინებს, რომ პროგრამა წარმატებით დასრულდა.

თუ არგუმენტების რაოდენობა ორია მაშინ ბრძანებების შესრულება გადადის მე-15 ხაზზე. აქ მასივიდან `argv` იღება მეორე ელემენტი (შეგახსენებთ ინდექსირება იწყება ნულიდან), ანუ ფაილის სახელი რაც მივაწოდეთ პროგრამას გაშვებისას. ეს ფაილის სახელი მიეწოდება ფუნქციას `fopen`. ეს ფუნქცია ეცდება გახსნას ფაილი წასაკითხად("r") ტექსტურ მოდაში ("r"ნაცვლად შეიძლება იყოს "rt"სადაც "t"ნიშნავს, რომ ტექსტური ფაილია.) ამ შემთხვევაში ფაილი გაიხსნება მიმდინარე დირექტორიაში, სადაც არის პროგრამა. ცხადია შეგვიძლია მივაწოდოთ სრული გზაც დირექტორიების გამოყენებით. `fopen` შედეგი იქნება ფაილზე მიმთითებელი. თუ ფაილი წარმატებით გაიხსნა მაშინ 'fp' მნიშვნელობა არ იქნება NULL. სწორედ ეს მოწმდება მე-15 ხაზზე. ანუ თუ ფაილი ვერ გაიხსნა მაშინ გამოდის შეტყობინება შეცდომაზე და პროგრამა წყდება შეცდომის კოდის შეტყობინებით. თუ ფაილის გახსნა მოხდა პროგრამა გრძელდება.

`fopen` ბრძანებამ რადგან წარმატების გახსნა ფაილი ეხლა შეგვიძლია წაკითხვა ფაილიდან. ამ შემთხვევაში ხდება ფაილიდან თითოეული char წაკითხვა `getc`(არგუმენტად მიეწოდება მიმთითებელი ფაილზე).

მე-20 ხაზი: ფაილიდან წაკითხვა გრძელდება მანამ სანამ არ შეგვხდება ფაილის დასასრულის ასო ნიშანი (კოდი EOF(end of file)).

მე-21 ხაზი: გამოგვაქვს წაკითხული ასო ნიშანი ეკრანზე (შეგევძლო გამოგვეყენებინა `putchar` ფუნქციაც) და იზრდება მთვლელის მნიშვნელობა 1 ერთეულით. ანუ ეკრანზე გამოტანასთან ერთად ვითვლით ასო ნიშნების რაოდენობასაც.

მე-25 ხაზი: აქ უკვე ციკლს დამთავრებული აქვს წაკითხვა. საჭიროა ფაილის დახურვა `fclose(fp)`. ყურადღება მიაქციეთ, რომ ფაილი აუცილებლად დაიხუროს წაკითხვის შემდეგ. თუ ფაილი ღიაა წასაკითხად და მას დახურვის ბრძანების გარეშე /მანამდე კვლავ გავხსნით მაგ. ჩასაწერად და დაიწყებთ ჩაწერას რა მოხდება ვერავინ გეტყვით. იგივე ეხება როცა ფაილი

"r"	ტექსტური ფაილის გახსნა წასაკითხად
"w"	ფაილის გახსნა ჩასაწერად. არსებული ტექსტი ნულდება. ახალი ფაილი გადაეწერება ძველს
"a"	ფაილის გახსნა ჩასაწერად. ახალი ჩანაწერი დაემტება არსებულ ფაილს ბოლოში ან თუ ფაილი არ არსებობს იქმნება ახალი ფაილი
"r+"	ფაილის გახსნა გასაახლებლად. ანუ წასაკითხად და ჩასაწერად
"w+"	ფაილის გახსნა გასაახლებლად. ანუ წასაკითხად და ჩასაწერად. არსებული ტექსტი ნულდება. ახალი ფაილი გადაეწერება ძველს.
"a+"	ფაილის გახსნა გასაახლებლად(წაკითხვა და ჩაწერა), ახალი მონაცემები ემატება ფაილის ბოლოში თუ ფაილი არსებობს, თუ არა — იქმნება ახალი ფაილი.
"ab+", "a+b", "wb+","w+b", "ab+" "a+b","rb","wb", "ab"	ეს ოპციები მუშაობენ ზუსტად ისევე როგორც ზემოთ აღწერილი, მაგრამ ფაილი გახსნილია ბინარულ მოდაში.

ცხრილი 8.1: ფაილის გახსნის ოპციები.

გახსნილია ჩასაწერად. ქვემოთ მოყვანილია `fopen` შესაძლო მეორე პარამეტრები (პირველი ყოველთვის არის მიმთითებელი ფაილზე). თუ ფაილი არაა გახსნილი და ცდილობთ მის დახურვას `fclose()` მოგვცემს შეცდომას. თუ ფაილი წარმატებით დაიხურა `fclose()` აბრუნებს 0.

8.1.2 `getc()` და `putc()`

ეს ფუნქციები მუშაობენ ისევე როგორც `getchar()` და `putchar()` იმ განსხვავებით, რომ მიეწოდებათ ფაილზე მიმთითებელი და ასო ნიშნის ჩაწერა წაკითხვა ხდება ფაილიდან. `putc()` ფორმატია `putc(ch,pFile)`, სადაც c არის char ტიპის ცვლადი pFile მიმთითებელი ფაილზე (ან stdout, stderr);

8.1.3 ფაილის დასასრული

ფაილის დასასრულის საპოვნელად ხშირად დაშვებული შეცდომაა ფაილის ბოლოზე შემოწმება მანამ სანამ წაკითხული იქნება ასო ნიშანი ფაილიდან.

პროგრამის კოდი 8.1 მასივი

```
1 int ch;
2 FILE *pFile;
3 pFile=fopen("test.txt", "r");
4 while(ch!=EOF)
5 {
6 ch=getc(pFile);
7 putchar(ch);
8 }
```

სწორი მიდგომაა შემდეგი

```
int ch;
FILE *pFile;
ch=getc(pFIle);
pFile=fopen("test.txt", "r");
while(ch!=EOF)
{
ch=getc(pFile);
putchar(ch);
}
```

ან

```
int ch;
FILE *pFile;
ch=getc(pFIle);
pFile=fopen("test.txt", "r");
while((ch=getc(pFile)!=EOF)
{
putchar(ch);
}
```

8.1.4 fscanf() და fprintf()

ეს ფუნქციები მსგავსია scanf() და printf() ოღონდ მუშაობენ ფაილებთან. პირველ არგუმენტად მიეწოდებათ მიმთითებელი ფაილზე. ფაილი უკვე უნდა იყოს გახსნილი წასაკითხად ან ჩასაწერად ფუნქციის შესაბამისად. ქვემოთ მოყვანილი კოდი ფაილში ამატებს ახალ სიტყვებს ("a+"მოდა).

```
/* count.c -- using standard I/O */
#include <stdio.h>
#include <stdlib.h> // exit() prototype
#include <string.h>
#define MAX 41

int main(void)
{
FILE *fp;
char words[MAX];

if ((fp = fopen("wordy", "a+")) == NULL)
{
fprintf(stdout,"Can't open \"wordy\" file.\n");
exit(EXIT_FAILURE);
}

puts("Enter words to add to the file; press the #");
puts("key at the beginning of a line to terminate.");
```

```
20  while ((fscanf(stdin,"%40s", words) == 1)  && (words[0] != '#'))
21  fprintf(fp, "%s\n", words);
22
23  puts("File contents:");
24  rewind(fp);            /* go back to beginning of file */
25  while (fscanf(fp,"%s",words) == 1)
26  puts(words);
27  puts("Done!");
28  if (fclose(fp) != 0)
29  fprintf(stderr,"Error closing file\n");
30
31  return 0;
32  }
```

მე-5 ხაზზე შემოტანილი პარამეტრი აღნიშნავს სიტყვის მაქსიმალურ სიგრძეს. ფაილში ჩანაწერის ფორმატია ერთი სიტყვა ერთ ხაზზე. ფაილიდან მონაცემების წაკითხვისას უნდა ვიცოდეთ ფაილში ჩაწერის წესი/ფორმატი, თუ როგორაა მონაცემები ჩაწერილი. ზემოთ მოყვანილ მაგალითში ითვლება, რომ ყოველ ხაზზე წერია ერთი სიტყვა მაქსიმუმ $40-1$ ასო ნიშნით (ბოლო ასო ნიშანი არის ნულლ კოდი '\0' მასივში words).

მე-14 ხაზზე შეიძლებოდა ყოფილიყო printf ან puts . უბრალოდ ნაჩვენებია fprintf გამოყენების მაგალითი.

მე-14 ხაზზე ბრძანება ფაილს ხსნის გასაახლებლად (თუ არ არის ასეთი ფაილი იქმნება ახალი). მონაცემები დაემატება ფაილის ბოლოში.

მე-20 ხაზზე ხდება მონაცემების წაკითხვა ეკრანიდან. აქაც მოყვანილია fscanf გამოყენების მაგალითი. ნაცვლად შეიძლება გამოვიყენოთ scanf . იკითხება სიტყვა ეკრანიდან მაქსიმუმ 40 ასო ნიშანი..თუ მეტია ასო ნიშნების რაოდენობა დანარჩენი იგნორირდება. ასევე მოწმდება, რომ პირველი ასო ნიშანი არ იყოს # ანუ იგნორირებულია სიტყვები, რომლებიც იწყება ამ ნიშნით. შეგვიძლია დავუშვათ, რომ ეს იყოს კომენტარის დასაწყისი. სხვადასხვა ტიპის ფაილებში ცხადია შეიძლება იყოს სხვადასხვა, მაგ. ;- ე.წ. .ini ფაილებში, // C/C++ ან %

მე-20 ხაზზე წაკითხული სიტყვა იწერება ფაილში და გადასვლა ხდება ახალ ხაზზე.

24-ე ხაზზე არის ფუნქცია rewind("გადახვევა") ამ ბრძანებით მიმთითებელი გადადის ფაილის დასაწყისში.

25-26 ხაზზე ბრძანება ნიშნავს : წაიკითხე ფაილიდან მანამ სანამ შესაძლებელია ერთი სიტყვის წაკითხვა და გამოიტანე ეს სიტყვა ეკრანზე

28-29 ხაზი: ფაილის დახურვა.

განვიხილოთ კიდევ 1 მუშა მაგალითი, რომლის მსგავსი ხშირად გვხვდება პრაქტიკაში ფიზმათ. ამოცანებში:

პროგრამის კოდი 8.2 მასივის ჩაწერა/წაკითხვა

```
/* count.c -- using standard I/O */
#include <stdio.h>
#include <stdlib.h> // exit() prototype
#define XSIZE 10
#define YSIZE 10
int main(int argc, char *argv[])
{
int ch;           // place to store each character as read
FILE *fp;         // "file pointer"
char str[]="wertilebi";
char xsize[]="xsize=";
char ysize[]="ysize=";
double Onedim[XSIZE];
double Twodim[XSIZE][YSIZE];
char *filename1="onedim.txt";
char *filename2="twodim.txt";
char buffer[100]={'\0'};
int read;
int nx,ny;
int i,j;
for (i=0;i<XSIZE;i++)
{
Onedim[i]= (double)i;
}
fp=fopen(filename1, "w");
if (!fp)
{
printf("Can't open %s\n", filename1);
exit(EXIT_FAILURE);
}
fprintf(fp,"%s\n",str);
fprintf(fp,"%s%d\n",xsize,XSIZE);
for (i=0;i<XSIZE;i++)
{
fprintf(fp,"%lf\n",Onedim[i]);
}
fclose(fp);
//gavanuloT masivi
for (i=0;i<XSIZE;i++)
{
Onedim[i]= 0;
}
//cakitxva
fp=fopen(filename1, "r");
if (!fp)
```

```
46 {
47 printf("Can't open %s\n", filename1);
48 exit(EXIT_FAILURE);
49 }
50 
51 fscanf(fp,"%s",buffer);
52 *buffer='\0';
53 read=fscanf(fp, "%[^=]=%d",buffer, &nx);
54 if (read!=2)
55 {
56 puts("ver cavikixe xsize=nx striqoni");
57 exit(EXIT_FAILURE);
58 }
59 for (i=0;i<nx;i++)
60 {
61 fscanf(fp,"%lf",&Onedim[i]);
62 printf("%lf\n",Onedim[i]);
63 }
64 fclose(fp);
65 for (i=0;i<XSIZE;i++)
66 for (j=0;j<YSIZE;j++)
67 {
68 Twodim[i][j]= (double)(i*YSIZE+j);
69 }
70 fp=fopen(filename2, "w");
71 if (!fp)
72 {
73 printf("Can't open %s\n", filename1);
74 exit(EXIT_FAILURE);
75 }
76 fprintf(fp,"%s\n",str);
77 fprintf(fp,"%s%d\n",xsize,XSIZE);
78 fprintf(fp,"%s%d\n",ysize,YSIZE);
79 for (i=0;i<XSIZE;i++)
80 for (j=0;j<YSIZE;j++)
81 {
82 fprintf(fp,"%d,%d,%lf\n",i,j,Twodim[i][j]);
83 }
84 fclose(fp);
85 for (i=0;i<XSIZE;i++)
86 for (j=0;j<YSIZE;j++)
87 {
88 Twodim[i][j]= 0;
89 }
90 fp=fopen(filename2, "r");
91 if (!fp)
92 {
```

```
 93 printf("Can't open %s\n", filename2);
 94 exit(EXIT_FAILURE);
 95 }
 96 fscanf(fp,"%s",buffer);
 97 *buffer='\0';
 98 read=fscanf(fp, "%[^=]=%d",buffer, &nx);
 99 if (read!=2)
100 {
101 puts("ver cavikixe xsize=nx striqoni");
102 exit(EXIT_FAILURE);
103 }
104 read=fscanf(fp, "%[^=]=%d",buffer, &ny);
105 if (read!=2)
106 {
107 puts("ver cavikixe ysize=ny striqoni");
108 exit(EXIT_FAILURE);
109 }
110 while (1)
111 {
112 double val;
113 read=fscanf(fp,"%d,%d,%lf",&i,&j,&val);
114 if (read!=3) break;
115 printf("%d,%d,%lf\n",i,j,val);
116 Twodim[i][j]=val;
117 }
118 fclose(fp);
119 puts("esaa cakitxuli 2d masivi:");
120 for (i=0;i<XSIZE;i++)
121 for (j=0;j<YSIZE;j++)
122 {
123 printf("%d,%d,%lf\n",i,j,Twodim[i][j]);
124 }
125 return 0;
126 }
```

ზემოთმოყვანილ კოდში არის რამდენიმე სიახლე:

53-ე ხაზი ნიშნავს: წაიკითხე სიტყვა -"მდე (უფრო სწორედ განცალკევების სიმბოლოდ აღებულია -"მსგავსად შეიძლება წაკითხულ იქნას "და ან სხვა ასო ნიშნით გამოყოფილი სიდიდეების). როგორც წესი ასეთი წაკითხვა მუშაობს მარტივ შემთხვევებში. როცა ჩანაწერის ფორმატი რთულია და სჭირდება ანალიზი მაშინ ხაზი მთლიანად იკითხება როგორც სიტყვა და შემდეგ ხდება ანალიზი. fscanf აბრუნებს მთელ მნიშვნელობას თუ რამდენი არგუმენტი იქნა წარმატებით წაკითხული. რადგან ვკუთხულობთ ორ ცვლადს read ცვლადის მნიშვნელობა უნდა იყოს 2 თუ წაკითხვა წარმატებით დასრულდა. ეს მოწმდება 54-ე ხაზზე. იგივე კეთდება 113-ე ხაზზე. *ეს ცუდად დაწერილი კოდია.* კოდის დიდი ნაწილი მეორდება. კარგად დაწერილ კოდში კოდი არ უნდა მეორდებოდეს. რა ნაწილიც მეორდება ხშირად/გამოიყენება ჩაწერილ უნდა იქნას ფუნქციად. ასევე წინა კოდთან შედარებით სიახლე არის ის, რომ არ ვიყენებთ ინფორმაციას

ფაილში ჩაწერილი ელემენტების რაოდენობის შესახებ. როგორც წესი ხშირად ეს არ ვიცით. თუ ვიცით, რომ მაგალითად ფაილში იწერება ერთი და იგივე ტიპის მონაცემები, მაგ. კოორდინატები ყოველ ხაზზე ან პირველ ხაზზე სახ. გვარი, მეორე ხაზზე წელი, მესამეზე სამუშაო ადგილი, კომენტარი და ა.შ. ჯერ უნდა დავთვალოთ ხაზების რაოდენობა (აქ გათვალიწინებული უნდა იყოს ბოლო ხაზზე არის თუ არა ახალ ხაზზე გადასვლა. ეს იქნება ზედმეტი ხაზი), შემდეგ ვიცით რა რამდენი ხაზი სჭირდება ელემენტს, გვეცოდინება წასაკითხი ელემენტების რაოდენობა. შემდეგ გამოვიყენებთ ფუნქციას `rewind` რათა დავბრუნდეთ ფაილის დასაწყისში და წავიკითხავთ საჭირო ელემენტების რაოდენობას. თუ არ ვიცით ელემენტების რაოდენობა და ვიცით ფაილის ფორმატი მაშინ ფაილს ვკითხულობთ იმ თანმიმდევრობით რა ფორმატითაც არის ჩაწერილი და ფაილის წაკითხვა წყდება თუ `fscanf` ფუნქციამ ვერ შეძლო საჭირო ველის წაკითხვა. ეს კი მოხდება სამ შემთხვევაში:

1. დამთავრდა ფაილი.
2. წაკითხვის ფუნქცია ცუდადაა დაწერილი.
3. ფაილი არაა ჩაწერილი ისე როგორც ვკითხულობთ ანუ შეცდომაა ფაილში.

ზემოთ მოყვანილ შემთხვევაში ვიცით რა როგორაა ფაილი ჩაწერილი ვკითხულობთ ორ, მეორე შემთხვევაში სამ, ხაზს ცალ-ცალკე. შემდეგ ვიცით როგორაა კოორდინატები ჩაწერილი და ვკითხულობთ მათ ციკლში. თუ წაკითხვა ვერ მოხერხდა ე.ი. დამთავრებულა ფაილი. ეს მეთოდი უფრო გამოყენებადია ვიდრე ფაილის ბოლოზე შემოწმება. ფაილის ბოლოს დასატესტირებლად საჭიროა `char` (შეგახსენებთ, რომ ჯერ უნდა იქნას წაკითხული ასო ნიშანი ხოლო შემდეგ შემოწმებული). ბინარულ წაკითხვა/ჩაწერას და უნიკოდ (არა ინგლისურ ენოვან) ტექსტების ჩაწერა წაკითხვას მოგვიანებით პრაქტიკულ მეცადინეობაზე განვიხილავთ.