საქართველოს ტექნიკური უნივერსიტეტი

ლელა გაჩეჩილაძე, ნანა კურკუმული

დაპროგრამება C ენაზე

სტუ-ს „IT - კონსალტინგის ცენტრი“

საქართველოს ტექნიკური უნივერსიტეტი
1922 წლიდან
GEORGIAN TECHNICAL UNIVERSITY
SINCE 1922

ლელა გაჩეჩილაძე, ნანა კურკუმული

დაპროგრამება
C ენაზე

დამტკიცებულია:

სტუ-ის „IT-კონსალტინგის სამეცნიერო ცენტრის" სარედაქციო კოლეგიის მიერ, 2024 წ. 13 სექტემბერი, ოქმი N 1

თბილისი
2024

უაკ 004.5

წიგნში „დაპროგრამება C ენაზე“ განხილულია ისეთი მნიშვნელოვანი საკითხები, როგორიცაა: C ენის სინტაქსი, მისი ძირითადი კონსტრუქციები, მმართველი სტრუქტურები, მონაცემთა დინამიკური სტრუქტურები, მომხმარებლის მიერ შექმნილი ფუნქციები (მათ შორის, რეკურსიული ფუნქციები), ANSI C ფაილური სისტემა, ნაკადები, მიმთითებლები, მასივების დახარისხებისა და საჭირო მნიშვნელობის ელემენტების ძებნის თანამედროვე ალგორითმები და მათი შესაბამისი პროგრამული რეალიზაციები, ენის გამოყენების ფართო შესაძლებლობები.

სახელმძღვანელო მოიცავს თეორიულ წანამძღვრებს და მასში აღწერილია პროგრამული კოდის ჩაწერისა და მისი შესრულების პროცედურები. იგი განკუთვნილია დაპროგრამების C ენის ასათვისებლად.

რეცენზენტები:

პროფ. გია სურგულაძე – სტუ-ის „პროგრამული ინჟინერიის“ დეპარტამენტის ხელმძღვანელი, ტექნიკის მეცნიერებათა დოქტორი

პროფ. ნონა ოთხოზორია – სტუ-ის ინფორმატიკისა და მართვის სისტემების ფაკულტეტი

რედკოლეგია:

ა. ფრანგიშვილი (თავმჯდომარე), მ. ახობაძე, ზ. ბოსიკაშვილი, ზ. გასიტაშვილი, გ. გოგიჩაიშვილი, მ. თევდორაძე, ე. თურქია, თ. კაიშაური, რ. კაკუბავა, დ. კაპანაძე, თ. ლომინაძე, ნ. ლომინაძე, თ. ქვანია, ლ. პეტრიაშვილი, გ. სურგულაძე (რეადაქტორი), ი. ქართველიშვილი, ო. შონია, ა. ცინცაძე, ზ. წვერაიძე

© სტუ-ს „IT-კონსალტინგის სამეცნიერო ცენტრი“, 2024

ISBN: 978-9941-8-7281-5

ყველა უფლება დაცულია, ამ წიგნის არც ერთი ნაწილი (იქნება ეს ტექსტი, ფოტო, ილუსტრაცია თუ სხვა) არანაირი ფორმით და საშუალებით (იქნება ეს ელექტრონული თუ მექანიკური), არ შეიძლება გამოყენებულ იქნას გამომცემლის წერილობითი ნებართვის გარეშე.

სარჩევი

შესავალი....6
რატომ უნდა შევისწავლოთ დაპროგრამების ენა C ?.... 6
I თავი
C ენის ძირითადი ცნებები.... 8
1.1. იდენტიფიკატორები.... 8
1.2. კომენტარები.... 9
1.3. პროგრამული ინტერფეისი.... 9
1.4. ხშირად დაშვებული შეცდომები.... 11
1.5. მონაცემთა ტიპები.... 13
1.6. ცვლადები და კონსტანტები.... 14
1.7. მმართველი სიმბოლოები.... 16
1.8. მონაცემთა შეტანა. ფუნქცია scanf().... 17
1.9. scanf() ფუნქციაში მონაცემთა შეტანის ფორმატები.... 17
1.10. მონაცემთა გამოტანა. ფუნქცია printf().... 19
1.11. პირველი პროგრამა.... 22
II თავი
C ენის ოპერაციები და ოპერატორები.... 25
2.1. მინიჭების ოპერატორი.... 25
2.2. არითმეტიკის ოპერატორები.... 26
2.3. ინკრემენტისა და დეკრემენტის ოპერატორები.... 27
2.4. მინიჭების შედგენილი (ავტოასოციური) ოპერატორი.... 29
2.5. შედარების (თანადობის) ოპერატორები.... 30
2.6. ლოგიკური ოპერატორები.... 31
2.6.1. პირობითი ლოგიკური ოპერატორები.... 31
2.6.2 ბიტური ლოგიკური ოპერატორები.... 32
2.7. ბიტური ძვრის ოპერატორები.... 34
2.8. ტერნერული ოპერაცია.... 35
2.9. C ენის საკვანძო (დარეზერვებული) სიტყვები.... 36
2.10. ოპერაციების პრიორიტეტები.... 36
2.11. მათემატიკის ფუნქციები.... 40
III თავი
მმართველი სტრუქტურები.... 41
3.1. განშტოებადი სტრუქტურები.... 42
3.1.1. პირობითი ოპერატორი if.... 42
3.1.2 ჩალაგებული if ოპერატორები.... 45
3.1.3. მრავალრგოლიანი if-else-if სტრუქტურა.... 46
3.1.4. ოპერატორი switch.... 48

3.1.5. უპირობო გადასვლის ოპერატორი goto 51
3.2. ციკლური სტრუქტურები........ 52
3.2.1. ოპერატორი while 52
3.2.2. ოპერატორი do-while 56
3.2.3 ოპერატორი for........ 57
3.2.4. რთული (ჩადგმული) ციკლები 64
3.2.5. ოპერატორი break 66
3.2.6. ოპერატორი continue........ 67
IV თავი
მასივები........ 68
4.1.1. ერთგანზომილებიანი მასივები........ 69
4.1.2. მრავალგანზომილებიანი მასივები........ 75
4.1.3. სიმბოლოების მასივები 81
4.1.4. სტრიქონების შეტანა/გამოტანის ფუნქციები 83
4.1.5. string.h ბიბლიოთეკის ძირითადი ფუნქციები........ 85
V თავი
მომხმარებლის მიერ შექმნილი ფუნქციები 91
5.1. ფუნქციის პროტოტიპი 95
5.2. ფუნქციის ფაქტიური და ფორმალური პარამეტრები 96
5.3. ერთგანზომილებიანი მასივების გადაცემა ფუნქციებში 99
5.4. ორგანზომილებიანი მასივების გადაცემა ფუნქციებში........ 101
VI თავი
მასივების დახარისხებისა და ძებნის ალგორითმები და შესაბამისი პროგრამული რეალიზაციები 107
6.1.1. ბუშტისებრი დახარისხება 107
6.1.2. დახარისხება ჩასმით 109
6.1.3. დახარისხება ამორჩევით........ 111
6.1.4. რედიქსის დახარისხება 112
6.2.1. წრფივი ძებნა........ 116
6.2.2. ბინარული ძებნა........ 117
VII თავი
სტრუქტურები 120
7.1. სტრუქტურის ველების ინიციალება........ 121
7.2. სტრუქტურის ელემენტზე მიმართვა. სტრუქტურის ეგზემპლარის გამოცხადება 122
VIII თავი
მიმთითებლები (Pointers) 129
8.1.1. ოპერაციები მიმთითებლებზე........ 129

8.1.2. არითმეტიკული მოქმედებები და გამოსახულებები მიმთითებლებზე ... 131
8.1.3. უნარული ოპერაცია sizeof ... 134
8.1.4. ურთიერთკავშირი მასივებსა და მიმთითებლებს შორის ... 135
8.2. დინამიკური მეხსიერება ... 138
8.2.1. მეხსიერების დინამიკური გამოყოფა C ენაში ... 138
8.2.2. ორგანზომილებიანი მასივებისთვის. მეხსიერების დინამიკურად გამოყოფა ... 142
8.2.3. მეხსიერების ხელახალი განაწილება ... 143
8.3. მიმთითებლები სტრუქტურებზე ... 145
IX თავი
ნაკადები და ფაილები ... 151
9.1.1. ნაკადების ტიპები ... 151
9.1.2. ფაილები ... 152
9.1.3. შეტანა/გამოტანის სტანდარტული მოწყობილობები ... 153
9.1.4. ფაილური სისტემა ANSI C ... 154
9.1.5. მიმთითებელი ფაილზე. ფაილის გახსნის რეჟიმები ... 155
9.1.6. შეცდომების დამუშავება ... 157
9.1.7. ტექსტური ფაილების წაკითხვა და ჩაწერა ... 158
9.1.8. სიმბოლოების წაკითხვა და ჩაწერა ფაილში ... 159
9.2. ბინარული ფაილები ... 161
X თავი
რ ე კ უ რ ს ი ა ... 167
10.1.1. ჰანოის კოშკები ... 173
10.1.2. სწრაფი დახარისხების ალგორითმი და მისი პროგრამული რეალიზება რეკურსიული გზით ... 176
XI თავი
მონაცემთა დინამიკური სტრუქტურები ... 179
11.1 სტეკი ... 179
11.2 რიგი ... 189
11.3 ორმხრივი რიგი (დეკი) ... 192
11.4.1 გროვა (Heap) ... 201
11.4.2 გროვას დახარისხების ალგორითმი და მისი პროგრამული რეალიზება 208
ლიტერატურა ... 217

შესავალი

დაპროგრამების ენა **C** გასული საუკუნის 70-იან წლებში ამერიკელი პროგრა-მიტის, ჰარვარდის უნივერსიტეტის კურსდამთავრებულის **დენის მაკალისტერ რიტჩი**ს მიერ იქნა შემუშავებული **Bell Laboratory**-ში (ამერიკული კორპორაცია AT&T).

დენის რიტჩი

დენის რიტჩი - ადამიანი, რომელიც ბევრისთვის ცნო-ბილია და ბევრისთვისაც უცნობი. ადამიანი, რომელმაც გზა გაუხსნა პროგრამისტებს ახლებური ხედვისკენ. ადამიანი, რომელმაც დატოვა უდიდესი მემკვიდრეობა მომავალი თაობისთვის.

კენ ტომპსონთან ერთად დენის რიჩმა **C** ენა ოპერაციული სისტემა **UNIX**-ის შესაქმნელად შეიმუშავა. აღნიშნული ენის დაწერის ძირითადი მიზანი იყო ის, რომ C ყოფილიყო მოსახერხებელი დაპროგრამებისთვის.

ვებ-ში პრაქტიკულად ყველაფერი **C**-ს და **UNIX**-ის ბაზაზეა შექმნილი. **C** ენაზეა შემუშავებული ბრაუზერები. **UNIX**-ის ბირთვზე კი თითქმის მთელი ინტერნეტი მუშაობს და ეს უკანასკნელიც **C** ენაზეა დაწერილი, მათ შორის ვებ სერვერებიც.
დღესდღეობით **C** ენის სამი სტანდარტი არსებობს:

- არაოფიციალური, რომელიც ეყრდნობოდა დენის რიტჩის სახელმძღვანელოს „The C Programming Language“. C დაპროგრამების ენის სტანდარტი ANSI/ISO, რომელიც 1989 წელს გამოვიდა. ეს უკანასკნელი განსაზღვრავდა როგორც თავად C ენის სტრუქტურას, ასევე მის სტანდარტულ ბიბლიოთეკას. ამ სტანდარტს ხშირად მოიხსენიებენ როგორც ANSI C-ს.
- C99 სტანდარტი, რომელიც ANSI სტანდარტის გაფართოებას წარმოადგენდა. მასში ძირითადი სიახლეები გახლდათ 64-ბიტიანი პროცესორების და მრავალ-ენოვანი სიმბოლოების მხარდაჭერა.
- 2011 წლის 8 დეკემბერს გამოქვეყნებულ იქნა ახალი სტანდარტი C დაპროგ-რამების ენისათვის - C11 (ISO/IEC 9899:2011).

ამგვარად, **C** სისტემური დაპროგრამების ენად ითვლება, თუმცა ის გამოყენე-ბითი პროგრამების შესაქმნელადაც საკმაოდ მოსახერხებელია.

რატომ უნდა შევისწავლოთ დაპროგრამების ენა C ?

C არის დაპროგრამების მძლავრი ენა. იგი შეიძლება გამოყენებულ იქნას ისეთი პროგრამული უზრუნველყოფის შემუშავებისთვის, როგორიცაა ოპერაციული სისტემები, მონაცემთა ბაზები, კომპილატორები და ა.შ. C დაპროგრამირების ენა იდეალური არჩევანია დამწყებთათვის.

C გვეხმარება კომპიუტერის შიგა არქიტექტურის და იმის გაგებაში, თუ როგორ ინახება და ნაწილდება ინფორმაცია კომპიუტერში. მისი შესწავლის შემდეგ, ბევრად უფრო მარტივია დაპროგრამების სხვა ენების შესწავლა, ისეთების როგორიცაა: C++, Java, C#, Python და ა.შ.

C ენა ხასიათდება ღია წვდომაში მყოფ პროექტებზე მუშაობის შესაძლებ-ლობით. ზოგი გლობალური ღია პროექტი (open source), ისეთი როგორიცაა Linux-ის ბირთვი, Python-ის ინტერპრეტატორები, SQLite მონაცემთა ბაზები და ა.შ. C დაპროგრამების ენაზეა დაწერილი.

დაპროგრამების ენა C ერთ-ერთი ყველაზე სტაბილური და პოპულარული ენაა მსოფლიოში. იგი ეხმარება თქვენი სმარტფონების, ავტომობილის სანავიგაციო სისტემების, რობოტების, დრონების, მატარებლების და ელექტრონული მოწყობი-ლობების უმრავლესობის მუშაობის პროცესს. C შეიძლება გამოყენებულ იქნას ყველა იმ სიტუაციაში, სადაც სიჩქარე და მოქნილობა მნიშვნელოვანია, მაგა-ლითად, ჩაშენებულ სისტემებში ან მაღალი სიმძლავრისა და სირთულის გამოთვ-ლებში. იგი ერთ-ერთი უძველესი და პოპულარული ენაა, რომელიც დაპროგრა-მების ისეთი ენების საფუძველს წარმოადგენს, როგორიცაა C#, Java, JavaScript. დღეს ბევრი დეველოპერი ერიდება C ენის სწავლას, სხვები კი თვლიან, რომ C-ის სწავლა პირველ რიგში, C++-ის განვითარების ღირებულ საფუძველს წარმოადგენს. იგი ფართოდ გამოიყენება კომპიუტერულ მეცნიერებასა და დაპროგრამებაში.

ცნობილია ის ფაქტი, რომ იმპერატიულ დაპროგრამებაში პროგრამისტი ინსტრუქციების ერთობლიობას უზრუნველყოფს, რომელსაც კომპიუტერი პროგ-რამის მდგომარეობისა და მასში არსებული ინფორმაციის სტრუქტურის მანი-პულირების მიზნით მიჰყვება. ეს პარადიგმა იმ ნაბიჯებს აღწერს, რომლებიც კომპიუტერმა უნდა შეასრულოს პრობლემის გადასაჭრელად.

C არის იმპერატიული, პროცედურული ენა, რომელშიც რეალიზებულია სტრუქტურული დაპროგრამების მეთოდის შესაძლებლობები [3,4]. იგი ყველაზე პოპულარული დაპროგრამების ენაა პროგრამული უზრუნველყოფის შემუშავე-ბაში სისტემური დაპროგრამებისა და დაბალი დონის დაპროგრამების ამოცანებისთვის, მათ შორის hardware-ზე (ტექნიკური უზრუნველყოფა) პირდა-პირი კონტროლის ჩათვლით.

I თავი

C ენის ძირითადი ცნებები

C ენაზე დაწრილი პროგრამა ოპერატორებისგან შედგება. ყოველი ოპერატორი გარკვეული მოქმედების შესრულებას იწვევს. მათ ჩასაწერად ლათინური ანბანის ზედა და ქვედა რეგისტრის ასოები, ციფრები და სპეციალური სიმბოლოები გამოიყენება. ასეთ სიმბოლოებს მიეკუთვნება: წერტილი (.), მძიმე (,), ორწერტილი (:), წერტილ-მძიმე (;) და სხვ. ენაში გამოყენებულ სიმბოლოების ერთობლიობას **ენის ანბანი** ეწოდება.

C ენის ანბანს შეადგენს:

- ლათინური ანბანის ასოები: a–z, A–Z
- ათობითი ციფრები: 0-9
- გრაფიკული სიმბოლოები: ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
- ცარიელი სიმბოლოები: გამოტოვება (space), ჰორიზონტალური ტაბულაცია, ვერტიკალური ტაბულაცია, ახალი ხაზი.

პერსონალურ კომპიუტერში სიმბოლოები კოდების სახით ინახება და მათ **ASCII** კოდებს უწოდებენ.

1.1. იდენტიფიკატორები

C ენის მნიშვნელოვან ცნებას იდენტიფიკატორი წარმოადგენს. ის ობიექტის (ფუნქციის, ცვლადის, მუდმივას და ა.შ.) სახელის როლში გამოიყენება. მათი შერჩევა შემდეგი წესების გათვალისწინებით ხდება:

- იდენტიფიკატორი უნდა იწყებოდეს ლათინური ანბანის (a,…,z, A,…,Z) ასოთი ან ხაზგასმის (_) სიმბოლოთი;
- იდენტიფიკატორში შეიძლება გამოყენებულ იქნას: ლათინური ანბანის ასოები, ციფრები (0,...,9) და ხაზგასმის სიმბოლო. სხვა სიმბოლოების იდენტიფიკატორში გამოყენება დაუშვებელია!
- C ენა იდენტიფიკატორში ერთმანეთისაგან განასხვავებს მაღალი და დაბალი რეგისტრის სიმბოლოებს. ეს ნიშნავს, რომ იდენტიფიკატორები, მაგალითად: name, NAME, Name, NaMe და ა.შ. განსხვავებულია.

- იდენტიფიკატორი არ უნდა ემთხვეოდეს ენაში არსებულ დარეზერვებულ (საკვანძო) სიტყვას და ენის ამა თუ იმ სტანდარტული ფუნქციის სახელს.

  იდენტიფიკატორის სწორი დასახელების მაგალითებია:

  Counter, get_line, a, Parami _ab;

  იდენტიფიკატორის არასწორი დასახელების მაგალითებია: %ab, 12abc, -x.

1.2. კომენტარები

C ენაზე დაწერილ პროგრამებში მნიშვნელოვანი როლი აქვს კომენტარებს. ისინი ამარტივებენ პროგრამის წაკითხვას. მათი ჩაწერა პროგრამის ნებისმიერ ადგილას დასაშვებია. კომენტარი არაშესრულებადი ბრძანებაა, რაც იმას ნიშნავს, რომ კომპილატორის მიერ მისი იგნორირება ხდება.

C ენაში კომენტარის ორი სახე არსებობს:

- ერთსტრიქონიანი კომენტარი, რომელიც ორი სლეშის (//) სიმბოლოთი იწყება;
- მრავალსტრიქონიანი კომენტარი, რომელიც იწყება სიმბოლოებით /* (სლეში ვარსკვლავი) და სრულდება სიმბოლოებით */ (ვარსკვლავი სლეში).

რეკომენდაციები:

- ✓ დაიწყეთ პროგრამა მოკლე კომენტარებით, რომელიც აღწერს ალგორითმის ძირითად ეტაპებს, ცვლადებს მონაცემთა შენახვისათვის, შუალედური და გამომავალი შედეგებისათვის.
- ✓ არ ჩართოთ კომენტარი პროგრამის სტრიქონის შუაში.

1.3. პროგრამული ინტერფეისი

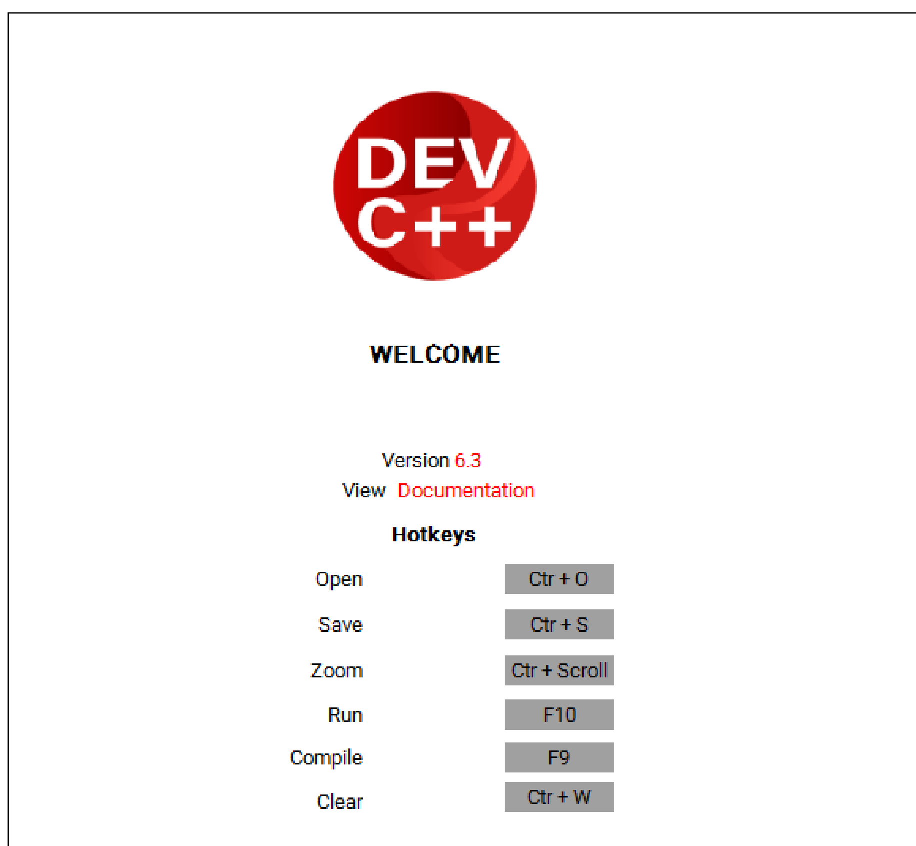
C ენაზე დაწერილი პროგრამის შესრულებაზე გაშვებამდე აუცილებელია პროგრამის ტრანსლირება (გადაყვანა მანქანურ კოდში), გამართვა და შემოწმება.

დღეს ყველა ეს მოქმედება გაერთიანებულია სპეციალური პროგრამა-გარსის შიგნით, რომელსაც პროგრამის შემუშავების ინტეგრირებული გარემო **(IDE – Integrated Development Environment)** ეწოდება. ის მოიცავს:

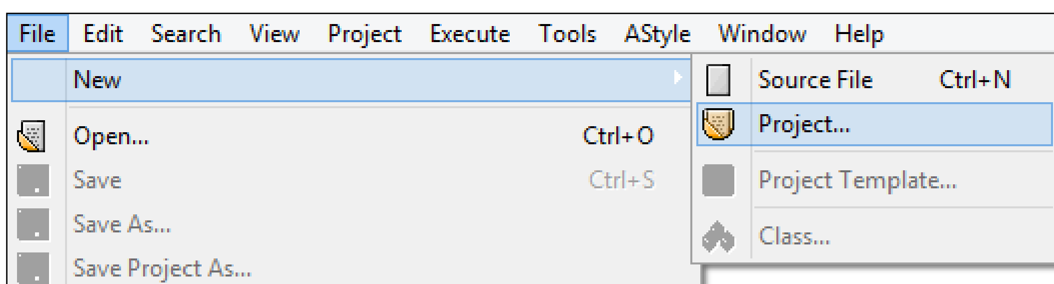
- **ტექსტურ რედაქტორს**, რომელიც გამოიყენება პროგრამების შექმნისა და რედაქტირებისთვის;

- **ტრანსლიატორს**, რომელიც პროგრამის ტექსტის თარგმნას ახდენს პროცესორის ბრძანებებად;
- **კომპილატორს** შესრულებადი (**exe**) ფაილის შესაქმნელად;
- **გამმართველს** პროგრამებში შეცდომების მოსაძებნად.

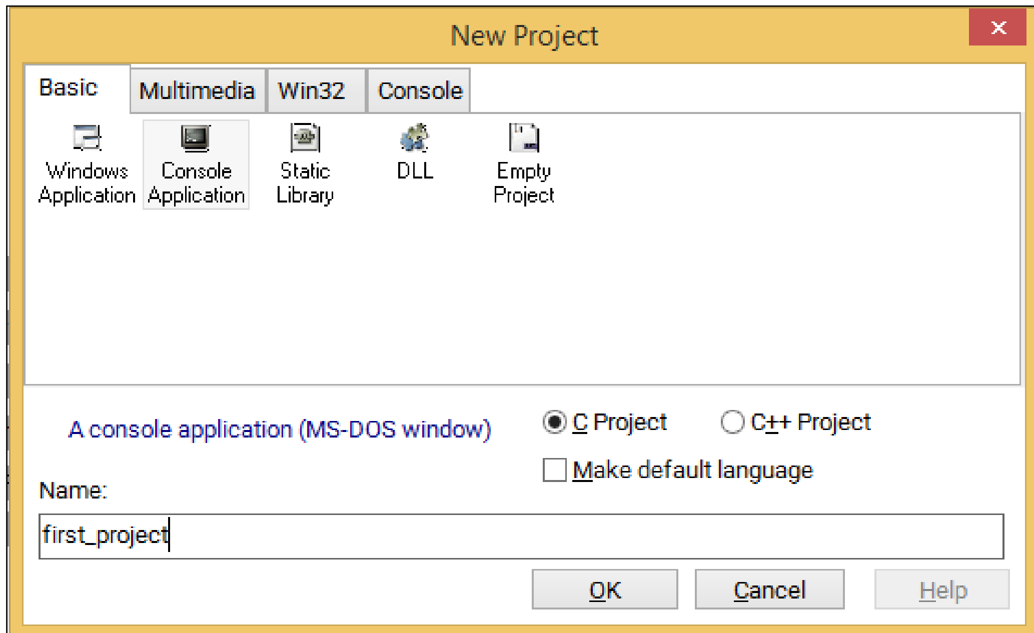
პირველ სურათზე წარმოდგენილია C დაპროგრამების ენის ინტეგრირებული გარემო, ხოლო მე-2, მე-3 და მე-4 სურათებზე - პროექტის შექმის, კომპილაციისა და შესრულებაზე გაშვების პროცესები.



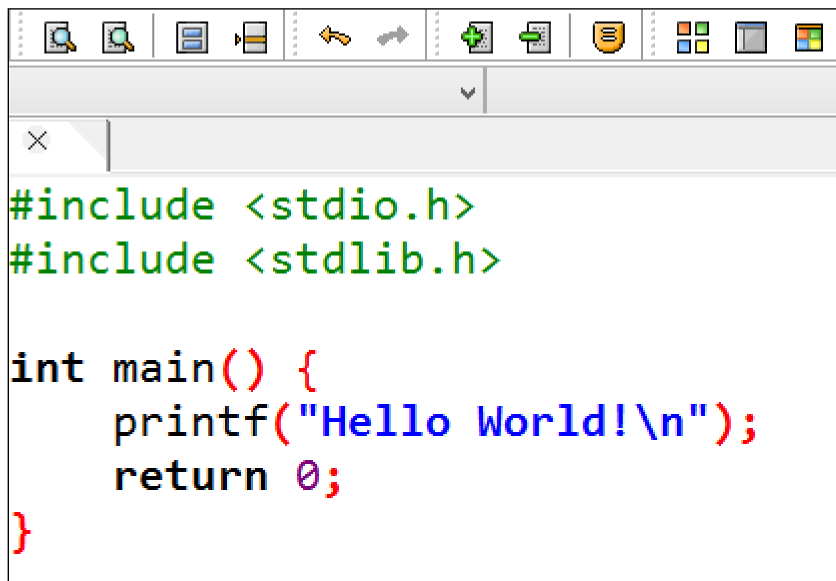
სურ. 1



სურ. 2



სურ. 3



სურ. 4

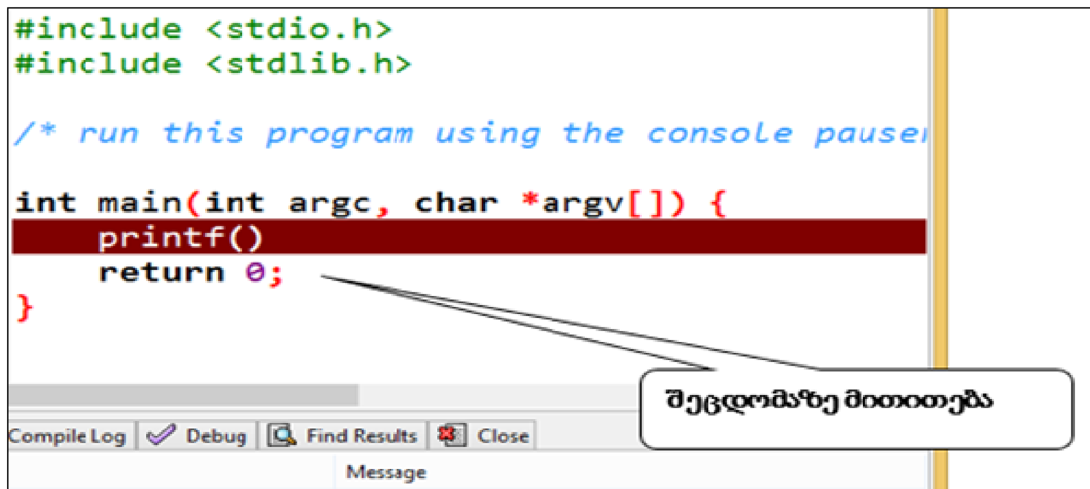
1.4. ხშირად დაშვებული შეცდომები

არავისთვის სიახლეს არ წარმოადგენს ის ფაქტი, რომ დამწყების პროგრამისტები პროგრამული კოდის წერის დროს ხშირად უშვებენ სხვადასხვა სახის (სინტაქსურ, ლოგიკურ და სემანტიკურ) შეცდომებს.

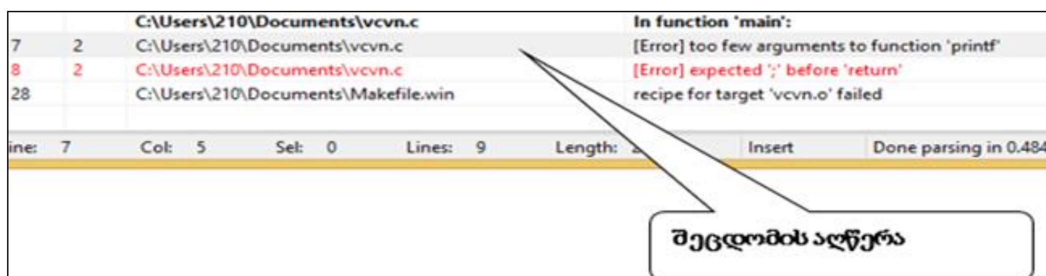
პირველ ცხრილში რამდენიმე ასეთი შეცდომაა წარმოდგენილი, ხოლო მე-5.1 და მე-5.2 სურათებზე - სინტაქსური შეცდომის ამსხველი ფანჯარა.

ცხრილი 1. ხშირად დაშვებული შეცდომები

xxx.h: No such file or directory	ვერ მოიძებნა სათაურის ფაილი 'xxx.h' (სახელი არასწორადაა მითითებული ან წაშლილია და ა.შ.)
'xxx‘ undeclared (first use this function)	ფუნქცია ან ცვლადი'xxx' უცნობია
missing terminating " character	დაუხურავია აპოსტროფები "
expected ;	ოპერატორის დასასრულს გამორჩენილია წერტილ-მძიმე წინა სტრიქონში
expected }	არ არის დახურული ფიგურული ფრჩხილი



სურ. 5.1



სურ. 5.2

1.5. მონაცემთა ტიპები

C ენაზე დაწრილი პროგრამა ახორციელებს სხვადასხვა ტიპის მონაცემების დამუშავებას. მონაცემები შეიძლება იყოს მარტივი, შედგენილი და სხვა ტიპის.

მარტივი მონაცემებია:

- მთელირიცხვა მონაცემები;
- ნამდვილრიცხვა მონაცემები;
- სიმბოლური მონაცემები;
- ლოგიკური მონაცემები.

შედგენილი (რთული) მონაცემებია:

- მასივი - ერთი და იმავე ტიპის ინდექსირებული ელემენტების ნაკრები;
- სტრიქონული ტიპი - მასივი, რომელიც სიმბოლოების სტრიქონს ინახავს;
- სტრუქტურა - სხვადასხვა ელემენტების ნაკრები, რომელიც ინახება, როგორც ერთი მთლიანი.

მონაცემთა სხვა ტიპებია:

- მიმთითებელი, რომელიც კომპიუტერის მეხსიერებაში ინახავს მისამართს. ეს უკანასკნელი რაიმე ინფორმაციაზე მიუთითებს, როგორც წესი - ცვლადზე.

ერთმანეთისაგან განასხვავებენ ცნებებს „მონაცემთა ტიპი“ და „ტიპის მოდიფიკატორი“. **მონაცემთა ტიპი** არის მაგალითად, მთელი ტიპის ცვლადი, ხოლო მოდიფიკატორი - მონაცემი ნიშნით ან მის გარეშე. ნიშნიანი მთელი ტიპის ცვლადი იღებს როგორც დადებით, ისე უარყოფით მნიშვნელობებს, ხოლო უნიშნო - მხოლოდ დადებით მნიშვნელობებს.

C ენაში არსებობს **მონაცემთა ხუთი ბაზისური ტიპი**, რომლებიც შემდეგი დარეზერ-ვებული სიტყვებით მოიცემა:

- **char** - სიმბოლური
- **int** - მთელი
- **float** - ნამდვილი
- **double** - ორმაგი სიზუსტის ნამდვილი რიცხვი
- **void** - მნიშვნელობა არ გააჩნია.

საბაზისო ტიპის ესა თუ ის ობიექტი შეიძლება მოდიფიცირებულ იქნას. ამისთვის გამოიყენება საკვანძო სიტყვები, რომელთაც **მოდიფიკატორები** ეწოდება. ესენია: **unsigned (უნიშნო), signed (ნიშნიანი), short (მოკლე) და long (გრძელი).**

მე-2 ცხრილში ნაჩვენებია მონაცემთა ბაზისური ტიპები.

ცხრილი 2. მონაცემთა ბაზისური ტიპები

Data type	Size(bytes)	Range	Format String
char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short	2	-32,768 to 32,767	%d
unsigned short	2	0 to 65535	%u
int	2	32,768 to 32,767	%d
unsigned int	2	0 to 65535	%u
long	4	-2147483648 to +2147483647	%ld
Unsinged long	4	0 to 4294967295	%lu
float	4	-3.4e-38 to +3.4e-38	%f
double	8	1.7 e-308 to 1.7 e+308	% lf
long double	10	3.4 e-4932 to 1.1 e+4932	%lf

1.6. ცვლადები და კონსტანტები

ტექნიკური თვალსაზრისით, **ცვლადი კომპიუტერის ოპერატიული მეხსიერების უჯრა ან უჯრების მიმდევრობაა, რომელსაც გააჩნია სახელი (იდენტიფიკატორი) და ინახავს რაიმე მნიშვნელობას.** ცვლადის მნიშვნელობა პროგრამის შესრულების პროცესში შეიძლება შეიცვალოს და უჯრაში ახალი მნიშვნელობის ჩაწერისას ძველი იშლება.

C ენაზე დაწერილ პროგრამაში ნებისმიერი ცვლადი გამოყენებამდე უნდა იქნეს გამოცხადებული, რაც იმას ნიშნავს, რომ მითითებული უნდა იყოს ცვლადის ტიპი და სახელი (იდენტიფიკატორი).

ცვლადის გამოცხადების ფორმა შემდეგია:

ცვლადის ტიპი ცვლადის სახელი;

მაგალითად:

int i; float k; double m;

თუ პროგრამაში ერთი და იმავე ტიპის რამდენიმე ცვლადის გამოცხადებაა საჭირო, ისინი შეიძლება ერთ სტრიქონში ერთმანეთისგან მძიმეებით გამოყოფის გზით გამოვაცხადოთ. მაგ: **int i, n, k, p, t;**

შესაძლებელია, ცვლადს გამოცხადების დროს კონკრეტული მნიშვნელობა მივანიჭოთ. ამ პროცესს **ცვლადის სტატიკური ინიციალება** ეწოდება. ამ დროს ჩაწერის ფორმა შემდეგია:

ცვლადის ტიპი ცვლადის სახელი=მნიშვნელობა;

მაგალითად:

int i=0, k, n, m=1;

float pi=3.1415, y;

char a='a';

კონსტანტა (მუდმივა) მონაცემია, რომლის მნიშვნელობა პროგრამის შესრულების პროცესში არ შეიძლება შეიცვალოს! ის ფიქსირებული მნიშვნელობის მქონე ობიექტია.

კონსტანტები არსებობს: რიცხვითი (მთელრიცხვა და მცოცავი წერტილით), სიმბოლური და სტრიქონული.

მთელრიცხვა კონსტანტა ათობითი, რვაობითი ან თექვსმეტობითი რიცხვია;

ათობითი კონსტანტა შედგება ერთი ან რამდენიმე ათობითი (0-9) ციფრისგან, რომელთაგან პირველი ციფრი ნული არ უნდა იყოს (ამ შემთხვევაში რიცხვი აღიქმება, როგორც რვაობითი). მაგალითად: **1, -29, 385.**

რვაობითი კონსტანტა (0-7) ციფრების მიმდევრობაა, რომელიც ყოველთვის ნულით იწყება. მაგალითად: **00, 071, -052, -03.**

თექვსმეტობითი კონსტანტა თექვსმეტობითი ციფრების (0-9, A-F) მიმდევრობაა, რომელსაც წინ უსწრებს ჩანაწერი: 0x ან 0X. მაგ: **0x0, 0x1, -0x2AF, 0x17.**

მცოცავწერტილიანი (ნამდვილრიცხვა) კონსტანტები წარმოდგენილია ორმაგი სიზუსტის მქონე მცოცავწერტილიანი რიცხვით და ის შემდეგი ნაწილებისგან შედგება: მთელი ნაწილი, მთელი და წილადი ნაწილების გამყოფი წერტილი,

წილადი ნაწილი, ექსპონენტას სიმბოლო e ან E. მაგალითად: **345, 3.14159, 2.1E5, .123E3, 4037e-5** და ა.შ.

სიმბოლური კონსტანტა ერთმაგ აპოსტროფებში მოთავსებული ერთი სიმბოლოა. მაგალითად: **'p'**. სიმბოლური კონსტანტების როლში მმართველი სიმბოლოებიც გამოიყენება, რომელთა ჩანაწერი ბექსლეშით (\) იწყება. მაგ: **'\t', '\n'** და ა.შ.

კონსტანტების გამოცხადების ფორმა შემდეგია:

const კონსტანტას ტიპი კონსტანტას სახელი=მნიშვნელობა;

მაგალითად: **const double PI=3.14;**

ცვლადებისგან განსხვავებით კონსტანტებს მნიშვნელობებს გამოცხადებისთანავე ვანიჭებთ! წინააღმდეგ შემთხვევაში, მივიღებთ შეტყობინებას შეცდომის შესახებ.

სტრიქონული კონსტანტა ორმაგ აპოსტროფებში (ბრჭყალებში) განთავსებული სიმბოლოების მიმდევრობაა. მაგ: **"Tbilisi", "Georgia"** და ა.შ.

1.7. მმართველი სიმბოლოები

მმართველია სიმბოლო, რომელიც გარკვეული მოქმედებების შესასრულებლად გამოიყენება. ეს მოქმედებებია: ახალ სტრიქონზე გადასვლა, ჰორიზონტალური ტაბულირება, სტრიქონის დასაწყისში გადასვლა და ა.შ. მმართველი სიმბოლოები მე-3 ცხრილშია წარმოდგენილი.

ცხრილი 3. მმართველი სიმბოლოები

მმართველი სიმბოლო	აღწერა
\'	ერთმაგი ბრჭყალი
\"	ორმაგი ბრჭყალი
\\	ირიბი დახრილი ხაზი
\0	Null (სიმბოლო, რომლის კოდია 0)
\a	ზარი
\b	უკან დაბრუნება (BackSpace)
\f	გვერდის გადაფურცვლა
\n	ახალ სტრიქონზე გადასვლა
\r	სტრიქონის დასაწყისში გადასვლა
\t	ჰორიზონტალური ტაბულირება
\v	ვერტიკალური ტაბულირება

1.8. მონაცემთა შეტანა
ფუნქცია scanf()

კლავიატურიდან მონაცემთა მნიშვნელობების ფორმატირებულ შეტანას უზრუნველყოფს **ფუნქცია scanf()** (მისი პროტოტიპი **stdio.h** სათაურის ფაილშია განთავსებული).

scanf() ფუნქციის ჩაწერის ზოგადი ფორმა შემდეგია:

scanf("ფორმატების სტრიქონი", მისამართი 1, მისამართი2,...);

ფორმატების სტრიქონი მონაცემთა შეტანის ფორმატია, რომლის **შემადგენელი ელემენტია ორმაგ აპოსტროფებში მოთავსებული ფორმატები სხვადასხვა ტიპის ცვლადების მნიშვნელობების შესატანად**.

შეტანის ფორმატის შემდეგ უნდა დავსვათ მძიმე და ჩამოვთვალოთ მეხსიერების უჯრების მისამართები, რომლებშიც უნდა ჩაიწეროს შეტანილი მნიშვნელობები.

ცვლადის მისამართის ფორმატირებისთვის ამპერსენდის (&) სიმბოლო გამოიყენება.

უნდა განვასხვავოთ ერთმანეთისაგან ჩანაწერები **a** და **&a.**

a არის **a** ცვლადის მნიშვნელობა;

&a არის **a** ცვლადის მისამართი.

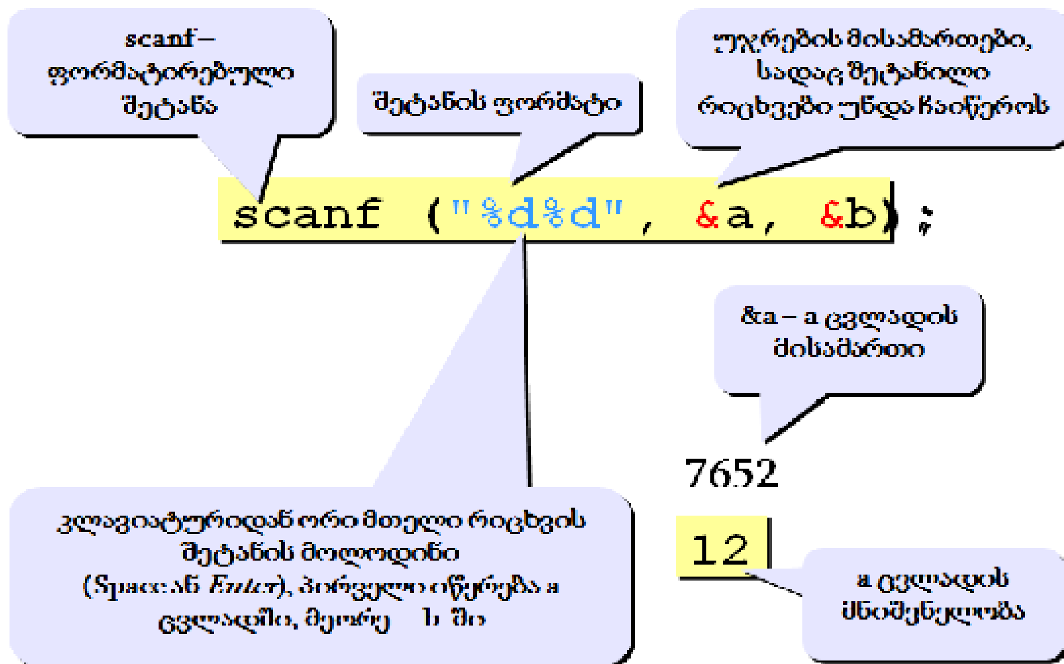
ფორმატების რაოდენობა სტრიქონში უნდა ემთხვეოდეს ცვლადების მისამართების რაოდენობას, ხოლო ცვლადების ტიპი - მითითებულ ფორმატს.

1.9. scanf() ფუნქციაში მონაცემთა შეტანის ფორმატები

- **%d — int** ტიპის ნიშნიანი მთელი რიცხვი თვლის ათობით სისტემაში;
- **%u —unsigned int** ტიპის მთელი რიცხვი;
- **%x — int** ტიპის ნიშნიანი მთელი რიცხვი თვლის თექვსმეტობით სისტემაში;
- **%o — int** ტიპის ნიშნიანი მთელი რიცხვი თვლის რვაობით სისტემაში;
- **%hd — short** ტიპის ნიშნიანი მთელი რიცხვი თვლის ათობით სისტემაში;
- **%hu — unsigned short** ტიპის მთელი რიცხვი;

- **%hx — short** ტიპის ნიშნიანი მთელი რიცხვი თვლის თექვსმეტობით სისტემაში;
- **%ld — long int** ტიპის ნიშნიანი მთელი რიცხვი თვლის ათობით სისტემაში;
- **%lu — unsigned long int** ტიპის მთელი რიცხვი;
- **%lx — long int** ტიპის ნიშნიანი მთელი რიცხვი თვლის თექვსმეტობით სისტემაში;
- **%f** — ნამდვილრიცხვა ფორმატი (**float** ტიპის მცოცავწერტილიანი რიცხვები);
- **%lf** — ორმაგი სიზუსტის ნამდვილრიცხვა ფორმატი (**double** ტიპის მცოცავწერტილიანი რიცხვები);
- **%e** — ნამდვილრიცხვა ფორმატი ექსპონენციალური ფორმით (**float** ტიპის მცოცავწერტილიანი რიცხვები ექსპონენციალურ ფორმაში);
- **%c** — სიმბოლური ფორმატი;
- **%s** — სტრიქონული ფორმატი.

მე-6 სურათზე წარმოდგენილია scanf() ფუნქციის სტრუქტურა.



სურ. 6

1.10. მონაცემთა გამოტანა

ფუნქცია printf()

ეკრანზე მონაცემთა ფორმატირებულ გამოტანას უზრუნველყოფს **ფუნქცია printf()** (მისი პროტოტიპი **stdio.h** სათაურის ფაილშია განთავსებული).

printf() ფუნქციის ჩაწერის ზოგადი ფორმა შემდეგია:

printf("ფორმატების სტრიქონი", ობიექტი1, ობიექტი2,...);

ფორმატების სტრიქონი მონაცემთა მნიშვნელობების გამოტანის ფორმატია, რომლის **შემადგენელი ელემენტებია:**

- **მმართველი სიმბოლოები;**
- **გამოსატანი ტექსტი (ინფორმაცია);**
- **ფორმატები სხვადასხვა ტიპის ცვლადების მნიშვნელობების გამოსატანად**.

საჭიროების შემთხვევაში ობიექტები შეიძლება გამოტოვებულ იქნეს.

გამოტანის ყოველი ფორმატი **%** სიმბოლოთი იწყება. ფორმატების სტრიქონის შემდეგ ეთითება მძიმით გამოყოფილი გამოსატანი ცვლადების სახელები. **%**-ის სიმბოლოების რაოდენობა ფორმატის სტრიქონში უნდა ემთხვეოდეს გამოსატანი ცვლადების რაოდენობას, ხოლო ყოველი ფორმატის ტიპი - გამოსატანი ცვლადის ტიპს.

მე-4, მე-5 და მე-6 ცხრილებში წარმოდგენილია მონაცემთა გამოტანის ფორმატების ნიმუშები.

ცხრილი 4. მონაცემთა გამოტანის ფორმატი მთელი რიცხვებისთვის

ნიმუში	შედეგი	კომენტარი
printf(" [%d]", 1234);	[1234]	მინიმალური დასაშვები ველი
printf(" [%6d]", 1234);	[1234]	6 პოზიცია, მონაცემი მარჯვენა კიდეში
printf(" [%-6d]", 1234);	[1234]	6 პოზიცია, მონაცემი მარცხენა კიდეში
printf(" [%2d]", 1234);	[1234]	რიცხვი არ ეტევა გამოყოფილ 2 პოზიციაში, ამიტომ გამოტანის არე ფართოვდება

ცხრილი 5. მონაცემთა გამოტანის ფორმატი ნამდვილი რიცხვებისთვის

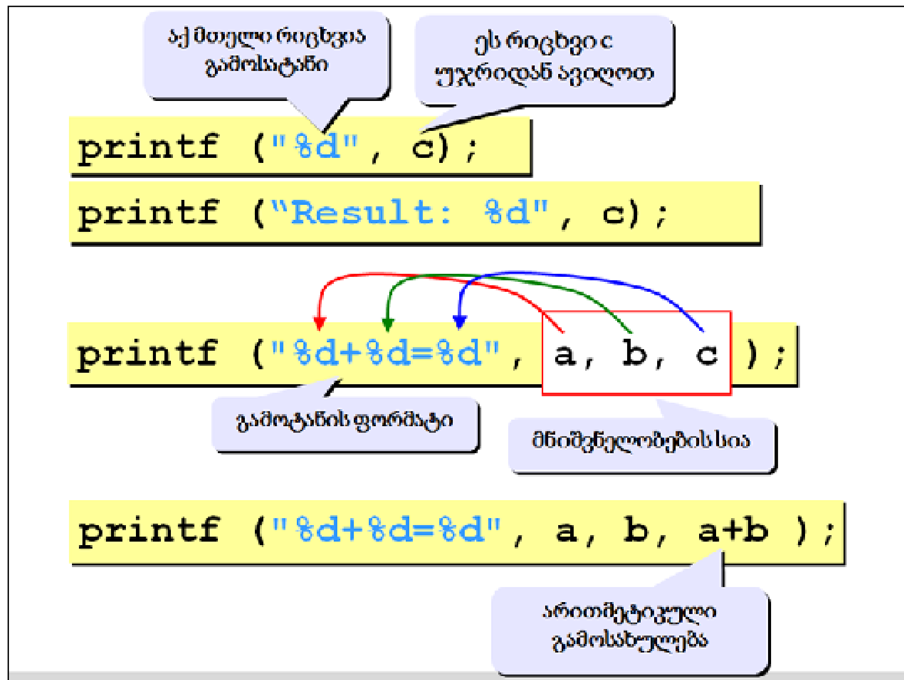
ნიმუში	შედეგი	კომენტარი
printf(“ [%f]”, 123.45);	[123.450000]	მინიმალური დასაშვები ველი, 6 ნიშანი წილად ნაწილში
printf(“ [%9.3d]”, 123.45);	[123.450]	სულ 9 პოზიცია, მათ შორის 3 წილად ნაწილში. მონაცემი მარჯვენა კიდეში
printf(“ [%-9.3d]”, 123.45);	[123.450]	სულ 9 პოზიცია, მათ შორის 3 წილად ნაწილში. მონაცემი მარცხენა კიდეში
printf(“ [%6.4d]”, 123.45);	[123.4500]	რიცხვი არ ეტევა გამოყოფილ 6 პოზიციაში (4 წილად ნაწილში), ამიტომ გამოტანის არე ფართოვდება

ფორმატი **%e** გამოიყენება სამეცნიერო გამოთვლებში ძალიან დიდი ან ძალიან მცირე რიცხვების გამოსატანად, მაგალითად.: ატომის ზომა ან მანძილი მზემდე.

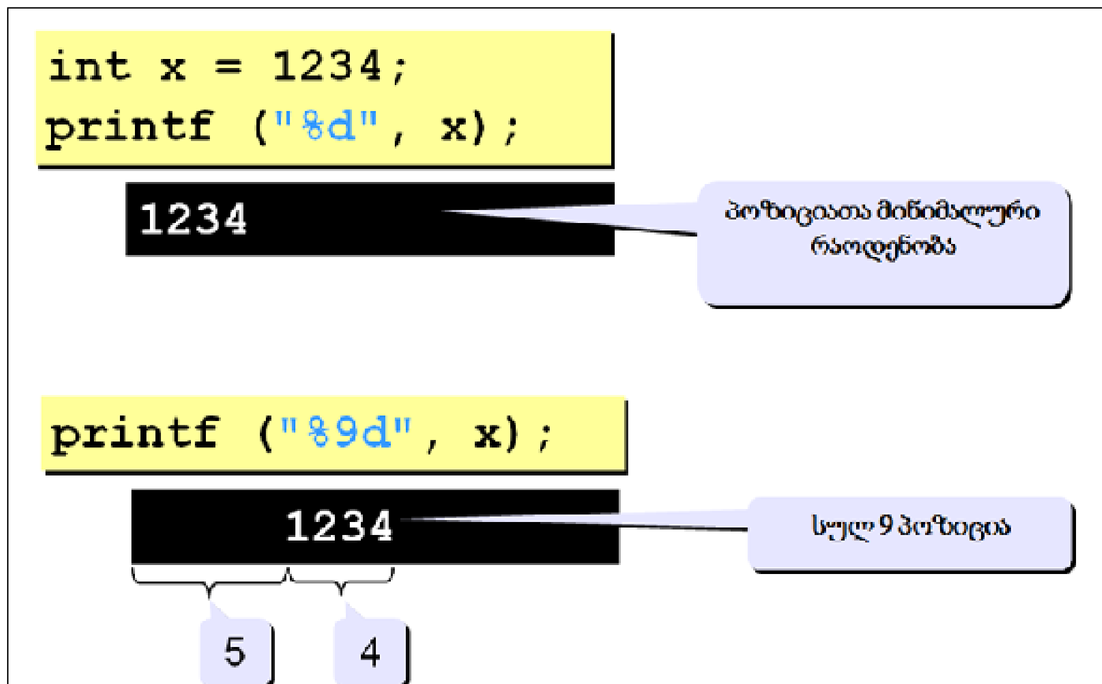
ცხრილი 6. მონაცემთა გამოტანის ფორმატი ნამდვილი რიცხვებისთვის ექსპონენციალური ფორმით

ნიმუში	შედეგი	კომენტარი
printf("[%e]", 123.45);	[1.234500e+02]	მინიმალური ველი, მანტისას წილად ნაწილში 6 ნიშანი
printf("[%12.3e]", 123.45);	[1.234e+02]	სულ 12 პოზიცია, აქიდან 3 მანტისას წილადი ნაწილისთვის. მონაცემი მარჯვენა კიდეში

printf() ფუნქციის სტრუქტურის ნიმუშები ნაჩვენებია მე-7 ÷ მე-9 სურათებზე.



სურ. 7



სურ. 8

```
float x = 123.4567;
printf ("%f", x);
```

```
123.456700
```

პოზიციათა მინიმალური რაოდენობა. 6 ციფრი წილად ნაწილში

```
printf ("%9.3f", x);
```

```
  123.456
```

სულ 9 პოზიცია. აქედან 3 ციფრი წილად ნაწილში

```
printf ("%e", x);
```

```
1.234560e+02
```

სტანდარტული სახე $1,23456 \cdot 10^2$

```
printf ("%10.2e", x);
```

```
  1.23e+02
```

სულ 10 პოზიცია. აქედან 2 ციფრი მანტისას წილად ნაწილში

სურ. 9

1.11. პირველი პროგრამა

წინამდებარე სახელმძღვანელოში პროგრამებს წარმოგიდგენთ Dev-C++ თავისუფალ ინტეგრირებულ გარემოში. პროგრამული კოდები განიხილება ვერსიისთვის: Dev-C++ 6.3 (ამ ეტაპზე ბოლო, ახალი ვერსიაა).

ახლა კი დადგა დრო, დავწეროთ ჩვენი პირველი პროგრამა (იხილეთ მე-10 სურათი) და განვმარტოთ მისი თვითოეული სტრიქონი.

```c
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

სურ. 10

პროგრამის პირველი სტრიქონი **#include <stdio.h>** პრეპროცესორის დირექტივას წარმოადგენს (პრეპროცესორი პროცესორის შემადგენელი ნაწილია), რომელიც კომპილატორს მიუთითებს, თუ რომელი ბიბლიოთეკის („ქუდის" ფაილის) ჩართვა ხდება პროგრამაში. აღნიშნული დირექტივა პროგრამაში მონაცემთა ნაკადურ შეტანა-გამოტანას ემსახურება.

ჩანაწერი:

```
int main()
{
    return 0;
}
```

მიგვანიშნებს მთავარ ფუნქციაზე, რომელიც შეიცავს არგუმენტებს (ჩვენ შემთხვევაში ის უარგუმენტო ფუნქციაა). თავად ფუნქციის ტანი მოთავსებულია ღია და დახურულ { } ფიგურულ ფრჩხილებს შორის. ფუნქციაში ბრძანება **return 0;** პროგრამის წარმატებით დასრულებაზე მიუთითებს.

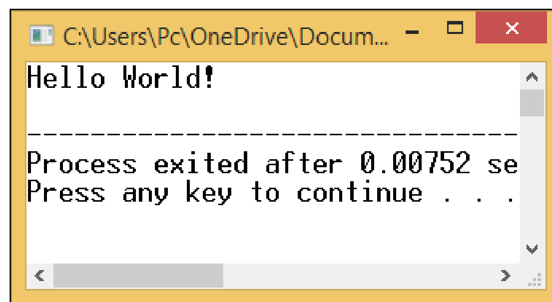
stdio.h ბიბლიოთეკის მიხედვით მონაცემთა გამოტანის ბრძანებას შემდეგი სახე აქვს:

```
printf("Hello World!\n");
```

"\n" – ნიშნავს ახალ ხაზს (new line), რომელიც გამოიტანს რა ახალ სტრიქონზე გადასვლის სიმბოლოს, ასუფთავებს გამოტანის ბუფერს.

იმისათვის, რომ პროგრამა დაკომპილირდეს და მოხდეს მისი შესრულებაზე გაშვება, საჭიროა დავაწვეთ F11 ფუნქციონალურ კლავიშს ან ინსტრუმენტების ზოლზე Compile & Run ინსტრუმენტს.

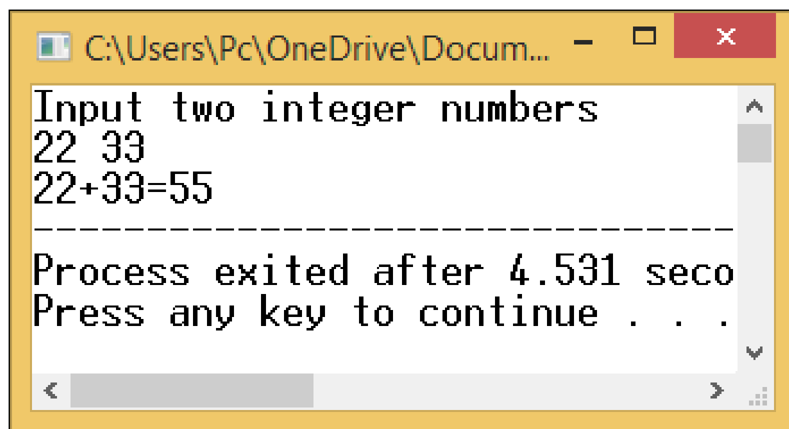
პროგრამის შესრულებაზე გაშვების შემდეგ, თუ იგი სინტაქსურ შეცდომებს არ შეიცავს, კონსოლზე გამოვა მე-11 სურათზე წარმოდგენილი ინფორმაცია.



სურ. 11

ქვემოთ წარმოდგენილია ორი მთელი რიცხვის ჯამის გამოთვლის პროგრამული კოდი, ხოლო მე-12 სურათზე - მისი შესრულების შედეგი.

```
#include <stdio.h>
int main() {
	int a, b, c;
	printf("Input two integer numbers\n");
	scanf("%d%d", &a, &b);
	c=a+b;
	printf("%d+%d=%d",a,b,c);
	return 0;
}
```



სურ. 12

II თავი
C ენის ოპერაციები და ოპერატორები

C ენაში ნებისმიერი გამოსახულება **ოპერანდებისგან** (ცვლადები, მუდმივები და ა.შ.) შედგება და ისინი ერთმანეთს ოპერაციის აღმნიშვნელი ნიშნებით (სიმბოლოებით) უკავშირდებიან. ოპერაციის ნიშანი, იგივე **ოპერატორი**, წარმოადგენს სიმბოლოს ან სიმბოლოთა ჯგუფს, რომელიც კომპილატორს ატყობინებს, თუ რა ოპერაცია უნდა შესრულდეს. ოპერაციები მკაცრი მიმდევრობით და პრიორიტეტების გათვალიწინებით სრულდება.

C ენის ძირითადი ოპერაციებია:

- მინიჭების ოპერაციები;
- არითმეტიკის ოპერაციები;
- თანადობის (შედარების) ოპერაციები;
- ლოგიკური ოპერაციები;
- ბიტური ძვრების ოპერაციები.

ოპერაციების შესრულების შედეგს რიცხვი წარმოადგენს.

ოპერანდების რაოდენობიდან გამომდინარე, ოპერაციები შეიძლება იყოს:

- უნარული (ერთოპერანდიანი);
- ბინარული (ორი ოპერანდისგან შემდგარი);
- ტერნერული (სამი ოპერანდისგან შემდგარი).

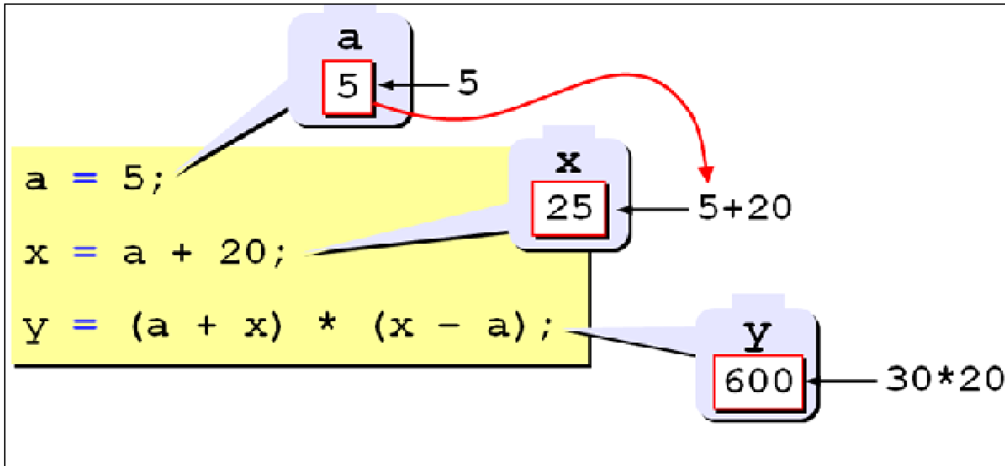
2.1. მინიჭების ოპერატორი

მინიჭების ოპერატორი ტოლობის (=) სიმბოლოთი აღინიშნება და ორ ეტაპად სრულდება:

- გამოითვლება ტოლობის ნიშნის მარჯვენა ნაწილში არსებული გამოსახულება;
- გამოთვლილი გამოსახულების შედეგი მინიჭების ოპერატორით (ტოლობის ნიშნით) ენიჭება ამავე ნიშნის მარცხენა ნაწილში მდგომ ოპერანდს.

მინიჭების ოპერატორის **ასოციატურობა** არის **მარჯვნიდან მარცხნივ**. ზოგადად, ოპერატორის **ასოციატურობის** ქვეშ იგულისხმება ოპერაციის შერულების მიმართულება.

მინიჭების ოპერატორის შესრულების ნიმუში მე-13 სურათზეა ნაჩვენები.



სურ. 13

2.2. არითმეტიკის ოპერატორები

არითმეტიკის ოპერატორები შეგვიძლია გამოვიყენოთ როგორც მთელი, ისე წილადი რიცხვებისათვის.

განვიხილოთ ზოგიერთი განმარტება.

- როდესაც გაყოფის ოპერატორს მთელი რიცხვებისათვის ვიყენებთ (ოპერანდები მთელრიცხვაა), მაშინ შედეგი მთელი რიცხვი იქნება, ნაშთი კი იგნორირებული რჩება.
- **მაშასადამე, მთელრიცხვა გაყოფის შედეგი მთელი რიცხვია!**
- ნაშთის მისაღებად **%** (მოდულით გაყოფის) ოპერატორი უნდა გამოვიყენოთ.
- **ყურადღება მიაქციეთ იმ ფაქტს, რომ მოდულით გაყოფის (%) ოპერატორის ოპერანდები აუცილებლად მთელრიცხვა უნდა იყოს!**
- არითმეტიკის ოპერატორები სრულდება პრიორიტეტების გათვალისწინებით მარცხნიდან მარჯვნივ: გამრავლება, გაყოფა, შეკრება, გამოკლება. ამ კანონზომიერებას მხოლოდ ფრჩხილები () არღვევს, რამე თუ, **პირველ რიგში, ფრჩხილებში მოთავსებული ოპერატორები სრულდება!**

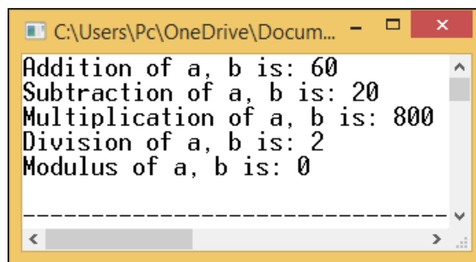
არითმეტიკის ოპერატორები მე-7 ცხრილშია წარმოდგენილი.

ცხრილი 7. არითმეტიკის ოპერატორები

ოპერატორი	მნიშვნელობა
+	შეკრება
-	გამოკლება (და უნარული მინუსი)
*	გამრავლება
/	გაყოფა
%	მოდულით გაყოფა (ნაშთის გამოყოფა)
++	ინკრემენტი
--	დეკრემენტი

ქვემოთ ნაჩვენებია არითმეტიკის ოპერატორების გამოყენების ამსახველი პროგრამული კოდი, ხოლო მისი შესრულების შედეგები მე-14 სურათზეა წარმოდგენილი.

```
#include <stdio.h>
int main() {
	int a=40, b=20, add, sub, mult, div, mod;
	add=a+b;
	sub=a-b;
	mult=a*b;
	div=a/b;
	mod=a%b;
	printf("Addition of a, b is: %d\n",add);
	printf("Subtraction of a, b is: %d\n",sub);
	printf("Multiplication of a, b is: %d\n",mult);
	printf("Division of a, b is: %d\n",div);
	printf("Modulus of a, b is: %d\n",mod);
	return 0;  }
```



სურ. 14

2.3. ინკრემენტისა და დეკრემენტის ოპერატორები

C ენაში ფართო გამოყენება აქვს **ინკრემენტისა** და **დეკრემენტის** ოპერატორებს.

ინკრემენტის ოპერატორი (++) ერთით ზრდის თავისი ოპერანდის (არგუმენ-ტის) მნიშვნელობას. ოპერანდი არის ცვლადი, რომელზეც ინკრემენტის ოპერაცია უნდა შესრულდეს.

დეკრემენტის ოპერატორი (--) ერთით ამცირებს თავისი ოპერანდის მნიშვნე-ლობას. მაგალითად,

x = x + 1;

ოპერატორი ასრულებს იმავე მოქმედებებს, რასაც ოპერატორი:

x++;   ან ++x;

ანალოგიურად,

x = x - 1;

ოპერატორი ასრულებს იმავე მოქმედებებს, რასაც ოპერატორი:

x--; ან --x;

როგორც ვხედავთ, ეს ოპერატორები შეიძლება მიეთითოს როგორც ოპერან-დამდე (**პრეფიქსული ფორმა**), ისე ოპერანდის შემდეგ (**პოსტფიქსური ფორმა**).

ინკრემენტისა და დეკრემენტის ოპერატორების ჩაწერის პრეფიქსულ და პოსტფიქსურ ფორმებს შორის შემდეგი განსხვავებაა:

- თუ ინკრემენტის ან დეკრემენტის ოპერატორი ოპერანდის წინაა, მაშინ ოპე-რანდის მნიშვნელობა ჯერ ერთით გაიზრდება (ინკრემენტის ოპერატორის შემთხვევაში) ან ერთით შემცირდება (დეკრემენტის ოპერატორის შემთხვე-ვაში) და შემდეგ მოხდება მისი გამოყენება გამოსახულებაში.
- თუ ინკრემენტის ან დეკრემენტის ოპერატორი მითითებულია ოპერანდის შემდეგ, მაშინ ჯერ ამ ოპერანდის გამოყენება მოხდება გამოსახულებაში და შემდეგ - მისი მნიშვნელობის ერთით გაზრდა (ინკრემენტის ოპერატორის შემთხვევაში) ან ერთით შემცირება (დეკრემენტის ოპერატორის შემთხვევა-ში).

აღნიშნული ოპერატორების შესრულების ნიმუშები მე-15 სურათზეა ნაჩვენები.

```
A
int a = 5;
int x = 10 + a++;
x = 15,   a = 6

B
int b = 5;
int y = 10 - ++b;
y = 4,   b = 6

C
int a = 12;
int x = 8 + a--;
x = 20,   a = 11

D
int b = 4;
int y = 9 / --b;
y = 3,   b = 3
```

სურ. 15

2.4. მინიჭების შედგენილი (ავტოასოციური) ოპერატორი

მინიჭების შედგენილ (შემოკლებულ) **ოპერატორში** არითმეტიკის ოპერატორე-ბი მინიჭების ოპერატორის მარცხნივაა მოთავსებული. მისი ჩაწერის სინტაქსი შემდეგია:

ცვლადი ოპერატორი= გამოსახულება;

შედგენილი (შემოკლებული) ოპერატორებია:

+= -= *= /= %= ;

მაგალითად: *x*=*x*+5; ჩანაწერის ექვივალენტურია ჩანაწერი *x*+=5;

x=*x*%10; ჩანაწერის ექვივალენტურია ჩანაწერი *x*%=10; და ა.შ.

მინიჭების შედგენილი ოპერატორები მინიჭების ჩვეულებრივ ოპერატორზე კომპაქტურია. მათი გამოყენება კოდის კომპილაციას აჩქარებს, რადგან ოპერან-დის შეფასება მხოლოდ ერთხელ ხდება.

მინიჭების ოპერატორით შესრულებული არითმეტიკული ოპერაციების მოკლე ჩანაწერები მე-8 ცხრილშია წარმოდგენილი.

ცხრილი 8. შემოკლებული ჩანაწერები C ენაში

სრული ჩანაწერი	შემოკლებული ჩანაწერი
a = a + 1; ინკრემენტი	a++;
a = a + b;	a += b;
a = a - 1; დეკრემენტი	a--;
a = a - b;	a -= b;
a = a * b;	a *= b;
a = a / b;	a /= b;
a = a % b;	a %= b;

2.5. შედარების (თანადობის) ოპერატორები

შედარების ოპერატორი ადარებს ორ მნიშვნელობას და გასცემს **bool** ტიპის **true** (ჭეშმარიტ, ერთის ტოლ) მნიშვნელობას, ან **false** (მცდარ, ნულის ტოლ) მნიშვნელობას. ეს ოპერატორები **განშტოებადი სტრუქტურის** პროგრამების ორგანიზებისთვის გამოიყენება.

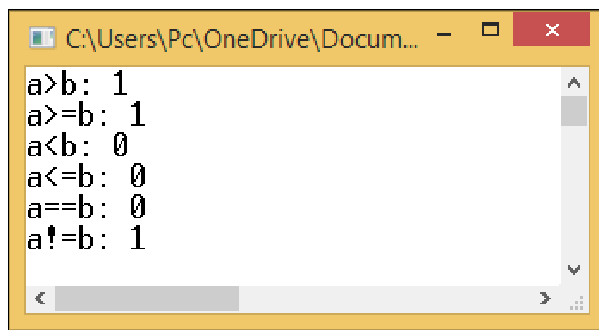
შედარების ოპერატორები მე-9 ცხრილშია წარმოდგენილი.

ცხრილი 9. შედარების ოპერატორები

ოპერატორი	მნიშვნელობა
==	ტოლია
!=	არ არის ტოლი
>	მეტია
<	ნაკლებია
>=	მეტია ან ტოლი
<=	ნაკლებია ან ტოლი

ქვემოთ ნაჩვენებია შედარების ოპერატორების გამოყენების ამსახველი პროგრამული კოდი, რომლის შედეგები წარმოდგენილია მე-16 სურათზე.

```
#include <stdio.h>
int main() {
	int a=40, b=20;
	printf("a>b: %d\n",a>b);
	printf("a>=b: %d\n",a>=b);
	printf("a<b: %d\n",a<b);
	printf("a<=b: %d\n",a<=b);
	printf("a==b: %d\n",a==b);
	printf("a!=b: %d\n",a!=b);
	return 0;
}
```



სურ. 16

2.6. ლოგიკური ოპერატორები

2.6.1. პირობითი ლოგიკური ოპერატორები

ლოგიკური ოპერატორი ოპერაციას **bool** ტიპის მნიშვნელობებზე ასრულებს. **ლოგიკური ოპერატორები ორ ჯგუფად იყოფა:**

- პირობითი;
- ბიტური

პირობითი ლოგიკური ოპერატორები პროგრამებში პირობების შემოწმების დროს გამოიყენება და სრულდება ნებისმიერ ობიექტზე. მისი შედეგი ერთის (გამოსახულების ჭეშმარიტობის შემთხვევაში) ან ნულის (გამოსახულების მცდარობის შემთხვევაში) ტოლია.

ზოგადად, ნულისგან განსხვავებული ნებისმიერი მნიშვნელობა პირობითი ლოგიკური ოპერატორების გამოყენების შემთხვევაში აღიქმება ჭეშმარიტ მნიშვნელობად.

პირობით ლოგიკურ ოპერატორებს მიეკუთვნება:

- ლოგიკური „და“ (&&) ოპერატორი;
- ლოგიკური „ან“ (||) ოპერატორი;
- ლოგიკური „არა“ (!) ოპერატორი.

ლოგიკური „და“ (იგივე კონიუნქცია) ოპერატორი ბინარულია (ორ ოპერანდიანი) და ჭეშმარიტ შედეგს იძლევა მხოლოდ თავისი ოპერანდების ჭეშმარიტების შემთხვევაში.

ლოგიკური „ან“ (იგივე დიზიუნქცია) ოპერატორი ბინარულია (ორ ოპერანდიანი) და ჭეშმარიტ შედეგს იძლევა თავისი ოპერანდებიდან ერთ-ერთის ჭეშმარიტების შემთხვევაშიც.

ლოგიკური „არა“ (უარყოფა) ოპერატორი უნარულია (ერთ ოპერანდიანი) და ის თავისი ოპერანდის მნიშვნელობას საწინააღმდეგო მნიშვნელობით ცვლის.

პირობითი ლოგიკური ოპერტორები მე-10 ცხრილშია წარმოდგენილი.

ცხრილი 10. პირობითი ლოგიკური ოპერატორები

X	Y	X OR Y	X AND Y	NOT X
true	true	true	true	false
true	false	true	false	false
false	true	true	false	true
false	false	false	false	true

2.6.2 ბიტური ლოგიკური ოპერატორები

ბიტური ლოგიკური ოპერატორები ოპერირებენ ბიტებზე, რომელთაგან თითოეულმა მათგანმა მხოლოდ ორი მნიშვნელობა შეიძლება მიიღოს: 0 ან 1.

ძირითადი ბიტური ოპერატორებია:

- **& კონიუნქცია (ლოგიკური „და“)** ბინარული ოპერატორია, რომლის შედეგი ერთის ტოლია მხოლოდ ერთნაირი ოპერანდების შემთხვევაში.
- **| დიზიუნქცია (ლოგიკური „ან“)** ბინარული ოპერატორია, რომლის შედეგი ერთის ტოლია, თუ ერთ-ერთი ოპერანდი უდრის ერთს.
- **˜ ინვერსია (ლოგიკური „არა“)** უნარული ოპერატორია, რომლის შედეგი ერთის ტოლია, თუ მისი ოპერანდი ნულია და ნულის ტოლია, თუ მისი ოპერანდი ერთს უდრის.
- **^ გამომრიცხავი „ან“** ბინარული ოპერატორია, რომლის შედეგი ერთის ტოლია, თუ მისი ოპერანდები განსხვავებულია (ერთი ოპერანდი ერთს უდრის, ხოლო მეორე - ნულს).

პირობითი ბიტური ოპერტორები მე-11 ცხრილშია წარმოდგენილი.

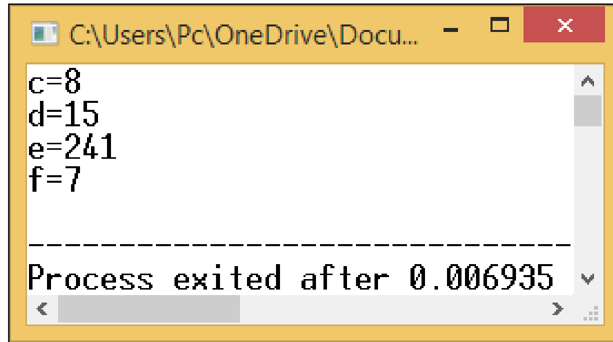
ცხრილი 11. პირობითი ბიტური ოპერატორები

a	b	a & b	a | b	~a	a ^ b
0	0	0	0	1	0
0	1	0	1	1	1
1	0	0	1	0	1
1	1	1	1	0	0

ქვემოთ წარმოდგენილია ბიტური ლოგიკური ოპერატორების გამოყენების ამსახველი პროგრამული კოდი, რომლის შესრულების შედეგები მე-17 სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
	unsigned char a=14;  //a=0000 1110
	unsigned char b=9;   //b=0000 1001
	unsigned char c, d, e, f;
	c=a&b;    //c=8=0000 1000
	d=a|b;    //d=15=0000 1111
	e=˜a;     //e=241=1111 0001
	f=a^b;  //f=7=0000 0111
	printf("c=%d\n",c);
	printf("d=%d\n",d);
	printf("e=%d\n",e);
	printf("f=%d\n",f);
```

```
        return 0;
}
```



სურ. 17

2.7. ბიტური ძვრის ოპერატორები

ბიტური ძვრების ოპერატორები მთელრიცხვა არითმეტიკაში გამოიყენება და შემდეგი სახით აღინიშნება:

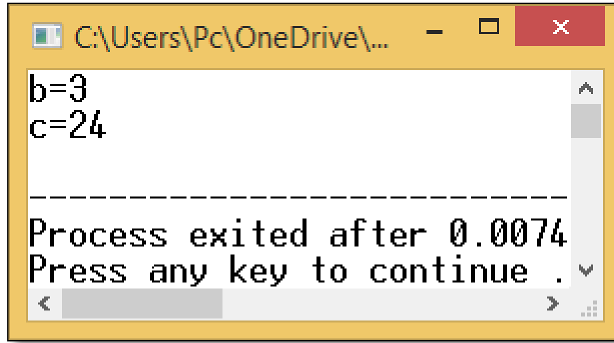
1. >> - მარჯვნივ ძვრა;
2. << - მარცხნივ ძვრა.

მთელი რიცხვის ერთი თანრიგით არითმეტიკული ძვრა მარჯვნივ (>>) შეესაბამება 2-ზე გაყოფას, ხოლო მთელი რიცხვის ერთი თანრიგით არითმეტიკული ძვრა მარცხნივ (<<) - 2-ზე გამრავლებას.

ზოგადად, ბიტური ოპერატორები გამოიყენება int და char ტიპის ცვლადებზე და მათ ვარიანტებზე (მაგალითად, long int). მათი გამოყენება დაუშვებელია float, double, void ტიპის ცვლადებზე და უფრო რთულ ტიპებზე.

ქვემოთ წარმოდგენილია ბიტური ძვრების ოპერატორების გამოყენების ამსახველი პროგრამული კოდი, რომლის შესრულების შედეგები მე-18 სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
        unsigned char a=6;  //a=0000 0110
        unsigned char b, c;
        b=a>>1;         //b=0000 0110 >> 1= 0000 0011=3
        c=a<<2;         //c=0000 0110 << 2=24
        printf("b=%d\n", b);
        printf("c=%d\n", c);
        return 0; }
```



სურ. 18

2.8. ტერნერული ოპერაცია

C ენაში არსებობს ერთადერთი სამოპერანდიანი (**ტერნერული**) ოპერაცია, რომლის ჩაწერის სინტაქსი შემდეგია:

პირობითი გამოსახულება? მოქმედება-1 : მოქმედება-2;

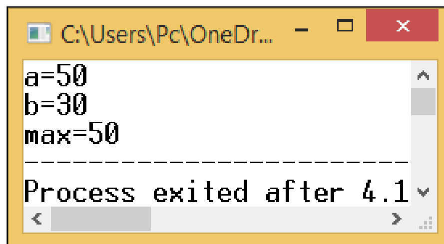
მაგალითად:

x=a>b? “Yes” : “No”;

თუ გამოსახულება a>b ჭეშმარიტია, x ცვლადი მიიღებს “Yes” მნიშვნელობას, წინააღმდეგ შემთხვევაში - “No” მნიშვნელობას.

ქვემოთ წარმოდგენილია ტერნერული ოპერციის გამოყენების ამსახველი პროგრამული კოდი, რომლის შესრულების შედეგები მე-19 სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
	int a,b, max;
	printf("a=");
	scanf("%d",&a);
	printf("b=");
	scanf("%d",&b);
	max=a>b? a : b;
	printf("max=%d", max);
	return 0; }
```



სურ. 19

2.9 C ენის საკვანძო (დარეზერვებული) სიტყვები

მე-12 ცხრილში წარმოდგენილია C ენაში არსებული საკვანძო (დარეზერ-ვებული) სიტყვები, რომელთა იდენტიფიკატორების როლში გამოყენების უფლება არ გვაქვს.

ცხრილი 12. C ენაში არსებული საკვანძო (დარეზერვებული) სიტყვები

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

2.10 ოპერაციების პრიორიტეტები

ჩვეულებრივ, გამოსახულებაში ჯერ პრიორიტეტული ოპერაციები სრულდება, შემდეგ კი - ნაკლებად პრიორიტეტული. მე-13 ცხრილში ოპერაციები დალაგებუ-ლია პრიორიტეტების კლების მიხედვით.

ცხრილი 13. ოპერაციების პრიორიტეტები

<table>
<tr><th>ოპერაცია</th><th>ასოციაციურობა</th></tr>
<tr><td>::</td><td>მარცხენა</td></tr>
<tr><td>( ) [ ] -></td><td>მარცხენა</td></tr>
<tr><td>! “ + (უნარული) -(უნარული) ++ -- &(უნარული) *(უნარული)
(ტიპის გარდაქმნა) static_cast const_cast dynamic_cast
reinterpret_cast sizeof new delete . . . typeid</td><td>მარჯვენა</td></tr>
<tr><td>. *(უნარული) ->*</td><td>მარცხენა</td></tr>
<tr><td>* / %</td><td>მარცხენა</td></tr>
<tr><td>+ -</td><td>მარცხენა</td></tr>
<tr><td><< >></td><td>მარცხენა</td></tr>
<tr><td>< <= > >=</td><td>მარცხენა</td></tr>
<tr><td>== !=</td><td>მარცხენა</td></tr>
<tr><td>&</td><td>მარცხენა</td></tr>
<tr><td>^</td><td>მარცხენა</td></tr>
<tr><td>|</td><td>მარცხენა</td></tr>
<tr><td>&&</td><td>მარცხენა</td></tr>
<tr><td>||</td><td>მარცხენა</td></tr>
<tr><td>?:(პირობის ოპერაცია)</td><td>მარჯვენა</td></tr>
<tr><td>= *= /= %= += -= &= ^= |= <<= >>=</td><td>მარჯვენა</td></tr>
<tr><td>‘</td><td>მარცხენა</td></tr>
</table>

შესაბამისად, ცხრილის ზედა ნაწილში მოთავსებულია მაღალი პრიორიტეტის ოპერაციები, ქვედა ნაწილში კი - დაბალი პრიორიტეტის. ერთ სტრიქონში მოთავსებულ ოპერაციებს ერთნაირი პრიორიტეტი აქვს. თუ გამოსახულებაში ფრჩხილები გამოყენებული არ არის, მაშინ თანაბარი პრიორიტეტის მქონე ოპერაციები იმ მიმდევრობით შესრულდება, რომელსაც მათი ასოციატურობა (ოპერაციის შესრულების მიმართულება) განსაზღვრავს. „მარცხენა" ასოციატურობის შემთხვევაში გამოსახულების ყველაზე მარცხენა ოპერაცია პირველ რიგში შესრულდება, შემდეგ კი მიმდევრობით სრულდება ყველა ოპერაცია მარცხნიდან მარჯვნივ.

პრიორიტეტების ცოდნა საჭიროა იმისათვის, რომ გამოსახულება სწორად ჩავწეროთ და შესაბამისად, მივიღოთ სწორი შედეგი.

ორიოდ სიტყვით განვმარტოთ ტერმინი **გამოსახულება**. ეს არის ოპერატორების, ცვლადებისა და ფუნქციების კომბინაცია.

ახლა კი განვიხილოთ წინამდებარე სახელმძღვანელოს მეორე თავთან დაკავშირებული რამდენიმე ამოცანა.

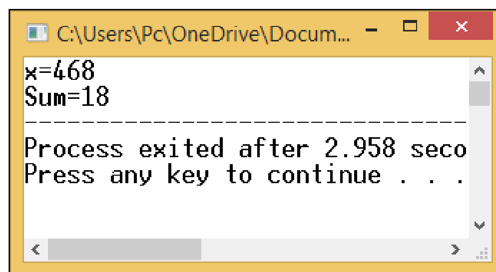
ამოცანა 1.

შევადგინოთ პროგრამა, რომელიც ნებისმიერ სამნიშნა ნატურალურ რიცხვში გამოთვლის მის ციფრთა ჯამს.

პროგრამული კოდი:

```
#include <stdio.h>
 int main() {
 int x;
 printf("x=");
 scanf("%d",&x);
 x=x/100 + x/10%10 + x%10;
 printf("Sum=%d", x);
 return 0; }
```

წარმოდგენილი პროგრამული კოდის შესრულების შედეგი მე-20 სურათზეა ნაჩვენები.



სურ. 20

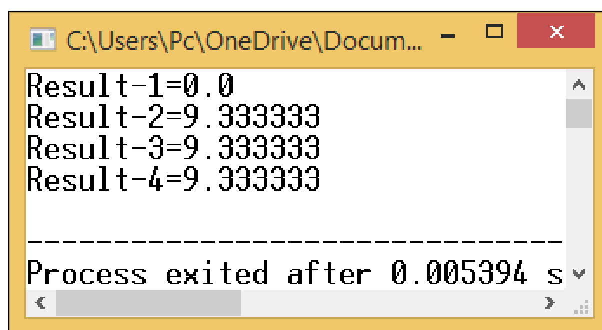
ამოცანა 2.

მოცემულია მთელი ტიპის a=28 ცვლადი. შევადგინოთ პროგრამა, რომელიც კონსოლზე გამოიტანს მოცემული ცვლადის მესამედის მთელ და ზუსტ მნიშვნე-ლობებს (ამ უკანასკნელის შემთხვევაში განიხილეთ ყველა შესაძლო ვარიანტი).

პროგრამული კოდი:

```
#include <stdio.h>
 int main() {
 int a=28;
 printf("Result-1=%.1f\n",a/3);
 printf("Result-2=%f\n",a/3.);
 printf("Result-3=%f\n",1.0*a/3);
 printf("Result-4=%f\n",(float)a/3);
 return 0;
}
```

წარმოდგენილი პროგრამული კოდის შესრულების შედეგი 21-ე სურათზეა ნაჩვენები.



სურ. 21

ამოცანა 3.

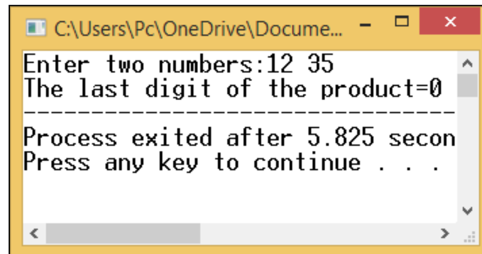
მოცემულია ორი მთელი რიცხვი. შევადგინოთ პროგრამა, რომელიც გაარკვევს, თუ რა ციფრით მთავრდება მათი ნამრავლი.

პროგრამული კოდი:

```
#include <stdio.h>
 int main() {
 int a, b;
 printf("Enter two numbers:");
 scanf("%d%d", &a, &b);
```

```
printf("The last digit of the product=%d", (a*b%10));
return 0; }
```

წარმოდგენილი პროგრამული კოდის შესრულების შედეგი 22-ე სურათზეა ნაჩვენები.



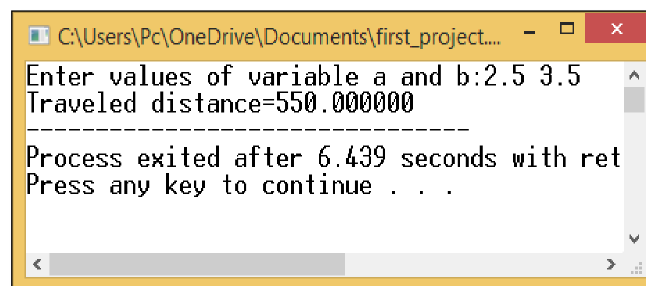
სურ. 22

ამოცანა 4.

გიორგიმ *a* საათის განმავლობაში იმგზავრა 80კმ/სთ სიჩქარით, ხოლო *b* საათის განმავლობაში 100კმ/სთ-ის სიჩქარით. დაწერეთ პროგრამა, რომელიც გამოთვლის სულ რამდენი კილომეტრი გაიარა გიორგიმ.

პროგრამული კოდი: (შესრულების შედეგები 23-ე სურათზეა ნაჩვენები)

```
#include <stdio.h>
int main() {
float a, b;
printf("Enter values of variable a and b:");  scanf("%f%f", &a, &b);
printf("Traveled distance=%f", (a*80+b*100));
return 0; }
```



სურ. 23

2.11. მათემატიკის ფუნქციები

მათემატიკის ფუნქციები **math.h** სტანდარტულ ბიბლიოთეკაშია აღწერილი (იხილეთ მე-14 ცხრილი). მათი უმრავლესობა არგუმენტის (ან არგუმენტების) სახით double ტიპის მონაცემს (ან მონაცემებს) შეიცავს და ამავე ტიპის შედეგს აბრუნებს.

ცხრილი 14. მათემატიკის ფუნქციები

ფუნქციის ჩანაწერი C-ში	განმარტება
sin(x)	*sinx* (*x* არგუმენტი მოიცემა რადიანებში)
cos(x)	*cosx* (*x* არგუმენტი მოიცემა რადიანებში)
tan(x)	*tgx* (*x* არგუმენტი მოიცემა რადიანებში)
asin(x)	*arcsinx* (*x* არგუმენტი მოიცემა რადიანებში)
acos(x)	*arccosx* (*x* არგუმენტი მოიცემა რადიანებში)
atan(x)	*arctgx* (*x* არგუმენტი მოიცემა რადიანებში)
sqrt(x)	$\sqrt{x}$ კვადრატული ფესვი *x*-დან
pow(x,y)	x^y *x* აყვანილია *y* ხარისხში
exp(x)	e^x ექსპონენციალური ფუნქცია
fabs(x)	$\lvert x \rvert$ *x*-ის მოდული (აბსოლუტური მნიშვნელობა)
log(x)	*lnx* ნატურალური ლოგარითმი (*e*-ს ფუძით)
log10(x)	*lgx* ათობითი ლოგარითმი (10-ის ფუძით)
ceil(x)	*x*-ის დამრგვალება მეტობით მომდევნო მთელ რიცხვამდე (არანაკლებ *x*-სა)
floor(x)	*x*-ის დამრგვალება ნაკლებობით მომდევნო მთელ რიცხვამდე (არაუმეტეს *x*-სა)
fmod(x,y)	*x*-ის *y*-ზე გაყოფის შედეგად მიღებული ნაშთი

III თავი

მმართველი სტრუქტურები

გამოთვლით პროცესში მოქმედებათა შესრულების თანმიმდევრობის მიხედვით ალგორითმები შეიძლება სამ ჯგუფად დაიყოს. პირველ ჯგუფს შეადგენს **წრფივი სტრუქტურის ალგორითმები**, მეორე ჯგუფს – **განშტოებული სტრუქტურის ალგორითმები**, ხოლო მესამე ჯგუფს კი – **ციკლური სტრუქტურის ალგორითმები**.

ალგორითმებს, რომელშიც ყველა მოქმედება, ელემენტარული ოპერაციები, წრფივი თანმიმდევრობით, ერთიმეორის მიყოლებით სრულდება და გამოთვლების მიმართულება საწყისი მონაცემების კონკრეტულ მნიშვნელობებზე არ არის დამოკიდებული, **წრფივი სტრუქტურის ალგორითმები** ეწოდება.

პრაქტიკაში ალგორითმების წრფივი სტრუქტურით წარმოდგენა იშვიათად არის შესაძლებელი. ასე მაგალითად: ყველაზე უმარტივესი გამოთვლითი პროცესის აღსაწერად, როგორიცაა:

$$y = \begin{cases} 2x^2 + 3, & x \le 0 \\ 2x^2 - 3, & x > 0 \end{cases}$$

წრფივი სტრუქტურა არ არის საკმარისი, რადგან x ცვლადის მოცემული მნიშვნელობის მიხედვით საჭიროა ერთ-ერთი ფორმულის ამორჩევა.

ამოცანების გადაწყვეტის შესაბამისი ალგორითმების შემუშავებისას ძალიან ხშირად აქვს ადგილი მსგავსი პროცედურების გამოყენებას, რომლის დროსაც საწყისი მონაცემების ან შუალედური შედეგების ანალიზის საფუძველზე გამოთვლითი პროცესის მიმართულება იცვლება.

ალგორითმებს, რომლებიც ასეთ პროცესებს იყენებენ, **განშტოებული სტრუქტურის ალგორითმებს** უწოდებენ, ხოლო **მიმართულებებს**, რომლის მიხედვითაც გამოთვლითი პროცესი მიმდინარეობს - **ალგორითმის შტოებს**.

როდესაც პროგრამული კოდის ესა თუ ის ფრაგმენტი მრავალჯერ მეორდება, აღნიშნულ პროცესს პროგრამისტები **ციკლს** უწოდებენ და ასეთ დროს ადგილი აქვს **ციკლური სტრუქტურის ალგორითმების** შემუშავებას.

ალგორითმების შესაბამისად, განასხვავებენ: წრფივი, განშტოებული და ციკლური სტრუქტურის პროგრამებს.

3.1. განშტოებადი სტრუქტურები
3.1.1 პირობითი ოპერატორი if

პირობითი ოპერატორები საშუალებას იძლევა გამოსახულების მნიშვნელობის ან ცვლადის მდგომარეობის მიხედვით პროგრამაში არჩეულ იქნას ბრძანებების შესრულების სხვადასხვა განშტოება.

if ოპერატორი - ესაა **C** პროგრამის განშტოების პირობითი შერჩევის ოპერატორი. ის გამოიყენება პროგრამის შესრულების სამართავად ორი სხვადასხვა მიმართულების შტოზე. ოპერატორის ჩაწერის ზოგადი ფორმა შემდეგია:

if(<პირობა>) <ოპერატორი1>; [else <ოპერატორი2>;]

აქ თითოეული **<ოპერატორი>** წარმოადგენს ან რომელიმე ერთ ოპერატორს ან ერთ ბლოკში (ფიგურულ ფრჩხილებში) გაერთიანებულ რამდენიმე ოპერატორს. **<პირობა>** ნებისმიერი გამოსახულებაა, რომელიც **bool** ტიპის შედეგს აბრუნებს. **else** აუცილებელი არაა.

if ოპერატორის მუშაობის პრინციპი შემდეგია:

თუ **<პირობა>** ჭეშმარიტია, მაშინ პროგრამა ასრულებს <**ოპერატორი1**>-ს. წინააღმდეგ შემთხვევაში იგი ასრულებს <**ოპერატორი2**>-ს (თუ ეს ოპერატორი არსებობს); **არცერთ შემთხვევაში პროგრამა არ შეასრულებს ორივე ოპერატორს ერთდროულად.**

კვადრატული ფრჩხილები ნიშნავს, რომ **else** სიტყვის ჩაწერა აუცილებელი არ არის. ასეთ შემთხვევაში, **if** ოპერატორის სინტაქსი იქნება:

if (პირობა) { ბლოკი1; }

თუ პირობა იღებს **true** (ჭეშმარიტ) მნიშვნელობას, მაშინ შესრულდება **ბლოკი1**, წინააღმდეგ შემთხვევაში კი - **ბლოკი1**-ის გარეთ პროგრამაში არსებული ოპერატორი.

თუ **if** ოპერატორის პირობის ჭეშმარიტობაზე ერთი ბრძანების შესრულებაა დამოკიდებული, ოპერატორის ტანის ამსახველი ღია და დახურული ფიგურული ფრჩხილების გამოყენება სავალდებულო არ არის, ხოლო, **თუ პირობის ჭეშმარიტობაზე დამოკიდებულ ბრძანებათა რიცხვი ერთს აღემატება, ფიგურული ფრჩხილების გამოყენება სავალდებულოა.**

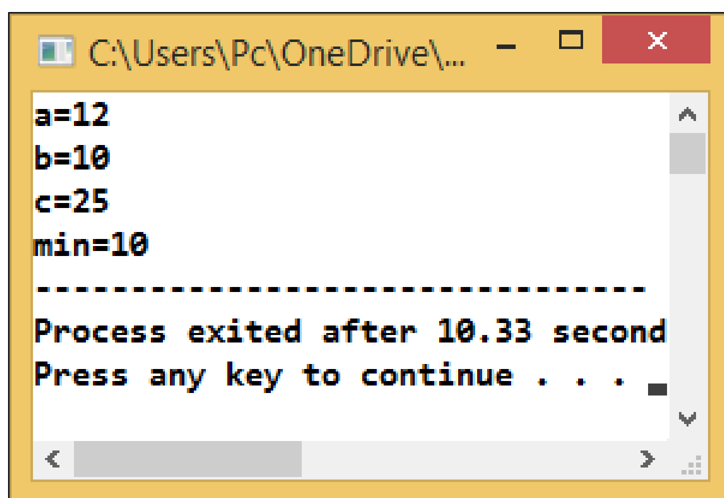
ახლა კი დადგა დრო, if და if/else ოპერატორების გამოყენებით გადავწყვიტოთ ამოცანები.

ამოცანა 5.

შევადგინოთ პროგრამა, რომელიც მთელი ტიპის სამ ცვლადს შორის განსაზღვრავს უმცირესს.

დასმული ამოცანის გადაწყვეტის პროგრამულ კოდს ქვემოთ წარმოდგენილი სახე აქვს, ხოლო მისი შესრულების შედეგი 24-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
	int a, b, c, min;
	printf("a=");
	scanf("%d", &a);
	printf("b=");
	scanf("%d", &b);
	printf("c=");
	scanf("%d", &c);
	min=a;
	if(min>=b)
	 min=b;
	 if(min>=c)
	 min=c;
	 printf("min=%d", min);
	return 0;
}
```



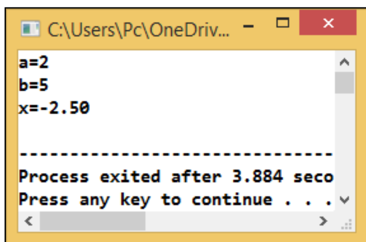
სურ. 24

ამოცანა 6.

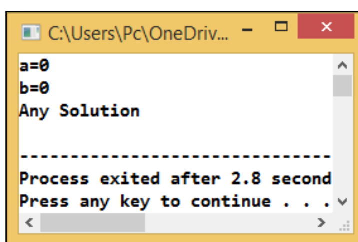
შევადგინოთ $ax + b = 0$ წრფივი განტოლების ამოხსნის პროგრამა a და b კოეფიციენტების ნებისმიერი მნიშვნელობების დროს.

დასმული ამოცანის გადაწყვეტის პროგრამულ კოდს ქვემოთ წარმოდგენილი სახე აქვს, ხოლო მისი შესრულების შედეგები 25.1÷25.3 სურათებზეა ნაჩვენები.

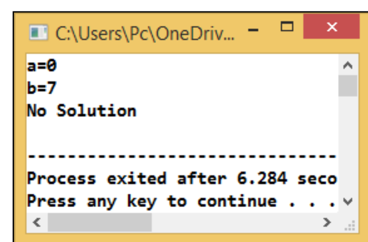
```
#include <stdio.h>
int main() {
	float a, b, x;
	printf("a=");
	scanf("%f", &a);
	printf("b=");
	scanf("%f", &b);
	if(a!=0){
		x=-b/a;
		printf("x=%3.2f\n", x);
	}
	else if(b==0)
	 printf("Any Solution\n");
	 else
	 printf("No Solution\n");
	return 0;
}
```



სურ. 25.1



სურ. 25.2



სურ. 25.3

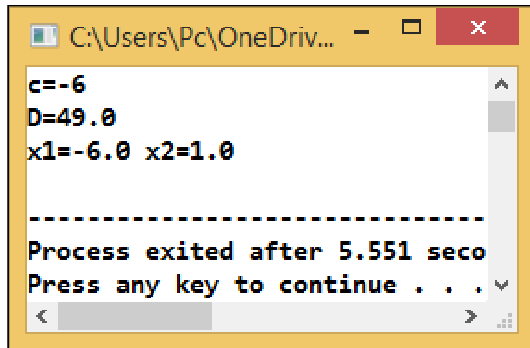
ამოცანა 7.

შევადგინოთ $ax^2 + bx + c = 0$ კვადრატული განტოლების ამოხსნის პროგრამა a, b და c კოეფიციენტების ნებისმიერი მნიშვნელობების დროს.

დასმული ამოცანის გადაწყვეტის პროგრამულ კოდს ქვემოთ წარმოდგენილი სახე აქვს, ხოლო მისი შესრულების ერთ-ერთი შედეგი 26-ე სურათზეა ნაჩვენები.

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

```
int main() {
        float a, b, c, x1, x2, x, D;
        printf("a=");
        scanf("%f", &a);
        printf("b=");
        scanf("%f", &b);
        printf("c=");
        scanf("%f", &c);
        D=b*b-4*a*c;
        printf("D=%2.1f\n", D);
        if(D>0){
        x1=(-b-sqrt(D))/(2*a);
        x2=(-b+sqrt(D))/(2*a);
        printf("x1=%2.1f\tx2=%2.1f\n", x1, x2);
        }
        else if(D==0){
                x=-b/(2*a);
                printf("x=%2.1f\n", x);
        }
        else
        printf("Two Complex Roots\n");
        return 0; }
```



სურ. 26

3.1.2 ჩალაგებული if ოპერატორები

პროგრამებში ხშირად გვხვდება ერთმანეთში ჩალაგებული **if** ოპერატორები. ამ შემთხვევაში უნდა გვახსოვდეს, რომ **else** ოპერატორი ყოველთვის უკავშირ-დება მის უახლოეს **if** ოპერატორს, რომელიც იმავე ბლოკში მდებარეობს და ჯერ არაა დაკავშირებული სხვა **else** ოპერატორთან. მაგალითად:

```
if(i == 10) {
if(j < 20) a =b;
```

```
if(k > 100) c = d; // ეს if ოპერატორი
else a = c; // დაკავშირებულია ამ else-თან
}
else a = d; // ეს else ოპერატორი დაკავშირებულია if(i == 10)-თან.
```

როგორც კომენტარებიდან ჩანს, ბოლო **else** ოპერატორი არაა დაკავშირებული **if(j < 20)** ოპერატორთან, ვინაიდან ის არ იმყოფება იმავე ბლოკში (მიუხედავად იმისა, რომ **if**-ის უახლოესი ოპერატორია, რომელთანაც **else** ოპერატორი ჯერ არაა დაკავშირებული). ბლოკში ბოლო **else** ოპერატორი დაკავშირებულია **if(k > 100)** ოპერატორთან, ვინაიდან იგი უახლოესია ბლოკის შიგნით.

3.1.3. მრავალრგოლიანი if-else-if სტრუქტურა

დაპროგრამებაში გავრცელებულია კონსტრუქცია, რომელიც აიგება ჩალაგებული **if** ოპერატორების მიმდევრობისაგან, მას უწოდებენ მრავალრგოლიან **if – else-if** სტრუქტურას და შემდეგი სახე აქვს:

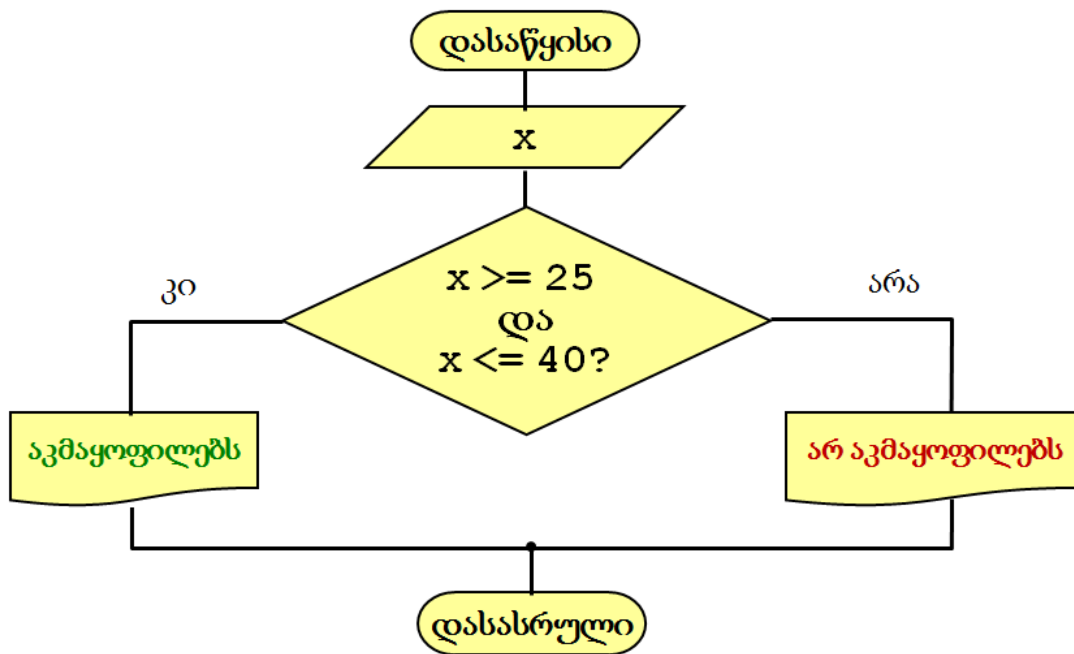
```
if (<პირობა>)
<ოპერატორი>;
else if (<პირობა>)
<ოპერატორი>;
 else if (<პირობა>)
<ოპერატორი>;
else
<ოპერატორი>;
```

if ოპერატორი სრულდება მიმდევრობით, ზემოდან ქვემოთ. როგორც კი **if** ოპერატორის მმართავი ერთ-ერთი პირობა გახდება **true** (ჭეშმარიტი), პროგრამა შეასრულებს ამ **if** ოპერატორთან დაკავშირებულ ოპერატორს და გამოტოვებს დანარჩენ მრავალრგოლიანი სტრუქტურის ნაწილს. თუ არცერთი პირობა არ აღმოჩნდება ჭეშმარიტი, პროგრამა შეასრულებს ბოლო **else** ოპერატორს.

ამოცანა 8.

კომპანია აგროვებს თანამშრომლებს 25-დან 40 წლის ჩათვლით. საჭიროა ადამიანის ასაკის შეყვანა და განსაზღვრა, აკმაყოფილებს თუ არა ის კომპანიის ასაკობრივ მოთხოვნას.

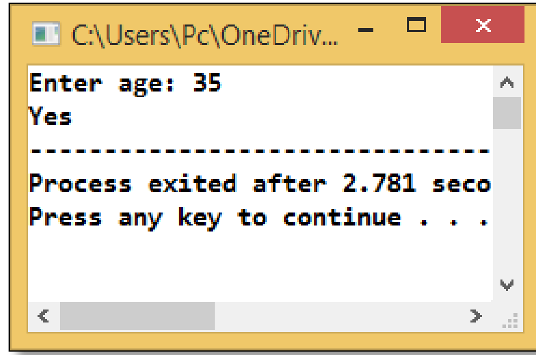
დასმული ამოცანის გადაწყვეტის შესაბამის ალგორითმის ბლოკ-სქემას 27-ე სურათზე ნაჩვენები სახე აქვს.



სურ. 27

ალგორითმის შესაბამისი პროგრამული კოდი ქვემოთ არის წარმოდგენილი, ხოლო მისი შესრულების შედეგი 28-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
	int age;
	printf("Enter age: ");
	scanf("%d", &age);
	if(age>=25 && age<=40)
	 printf("Yes");
	 else
	 printf("No");
	return 0;
}
```



სურ. 28

3.1.4. ოპერატორი switch

switch ოპერატორი საშუალებას გვაძლევს ვარიანტების სიიდან ამოვარჩიოთ საჭირო მოქმედება. მისი ჩაწერის სინტაქსი შემდეგია:

switch (<გამოსახულება>) {

case <მნიშვნელობა1>:

// ოპერატორების მიმდევრობა

break;

case <მნიშვნელობა 2>:

// ოპერატორების მიმდევრობა

break;

case <მნიშვნელობა N>:

// ოპერატორების მიმდევრობა

break;

default:

// ოპერატორები

}

გამოსახულება> უნდა იყოს byte, short, int ან char ტიპის. **<მნიშვნელობა>**-ში მითითებული ყოველი ტიპი **<გამოსახულება>**-ში მითითებული შედეგის ტიპის თანადი (თავსებადი) უნდა იყოს. case-ის თითოეული **<მნიშვნელობა>** უნდა იყოს უნიკალური კონსტანტა (მუდმივა და არა ცვლადი). ასევე, არაა დაშვებული case-ის მნიშვნელობების დუბლირება.

<გამოსახულება>-ის შედეგი დარდება case ოპერატორებში მითითებულ თითოეულ კონსტანტას. თუ მოხდება რომელიმესთან თანხვედრა, მაშინ შესრულება გრძელდება ამ case-ის შემდეგ ჩაწერილი ოპერატორიდან. ერთ case განშტოებასთან დაკავშირებულმა ოპერატორების მიმდევრობამ მართვა არ უნდა გადასცეს მომდევნო case განშტოების ოპერატორებს. ამის თავიდან ასაცილებლად თითოეული case განშტოების ოპერატორების მიმდევრობა break ოპერატორით უნდა დავასრულოთ.

თუ არცერთი კონსტანტის მნიშვნელობა არ დაემთხვევა **<გამოსახულების>** შედეგს, მაშინ პროგრამა ასრულებს default ოპერატორის შემდეგ ჩაწერილ ოპერატორებს. თუმცა, ამ ოპერატორის გამოყენება აუცილებელი არაა. default ოპერატორის არარსებობის შემთხვევაში და გამოსახულების შედეგის არცერთ კონსტანტასთან თანხვედრის შემთხვევაში, პროგრამა switch კონსტრუქციაში არაფერს ასრულებს. პროგრამის შესრულება გრძელდება switch კონსტრუქციის შემდეგი ოპერატორიდან.

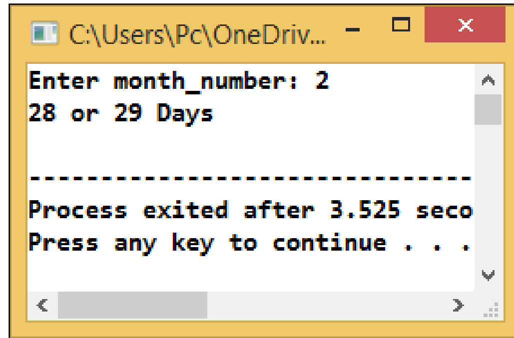
ამოცანა 9.

შევადგინოთ პროგრამა, რომელიც თვის აღმნიშვნელი რიცხვის მიხედვით განსაზღვრავს მასში დღეების რაოდენობას.

დასმული ამოცანის გადაწყვეტის პროგრამულ კოდს ქვემოთ წარმოდგენილი სახე აქვს, ხოლო მისი შესრულების შედეგი 29-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
	int month_number;
	printf("Enter month_number: ");
	scanf("%d", &month_number);
	switch(month_number){
		case 12: case 1: case 3: case 5: case 7: case 8: case 10:
		printf("31 Days\n"); break;
		case 4: case 6: case 9: case 11:
		printf("30 Days\n"); break;
		case 2:
		printf("28 or 29 Days\n"); break;
		default:
		printf("Wrong number\n");
	}
```

```
        return 0;
}
```



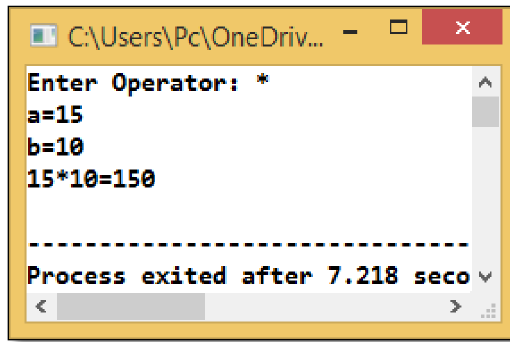
სურ. 29

ამოცანა 10.

switch ოპერატორის გამოყენებით შევადგინოთ მარტივი კალკულატორის ამსახველი პროგრამა.

დასმული ამოცანის გადაწყვეტის პროგრამულ კოდს ქვემოთ წარმოდგენილი სახე აქვს, ხოლო მისი შესრულების შედეგი 30-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
int a, b;
char operator;
printf("Enter Operator: ");
scanf("%c", &operator);
printf("a=");
scanf("%d", &a);
printf("b=");
scanf("%d", &b);
switch(operator){
   case '+':
         printf("%d+%d=%d\n",a,b,a+b); break;
   case '-':
         printf("%d-%d=%d\n",a,b,a-b); break;
   case '*':
         printf("%d*%d=%d\n",a,b,a*b); break;
   case '/':
         printf("%d/%d=%d\n",a,b,a/b); break;
  default:
         printf("Error! Operator is not correct!\n");}
return 0;
}
```



სურ. 30

3.1.5. უპირობო გადასვლის ოპერატორი goto

goto წარმოადგენს უპირობო გადასვლის ოპერატორს (ამასთან, მოძველებულ კონსტრუქციას), რომლის ჩაწერის სინტაქსს შემდეგი სახე აქვს:

goto ჭდე;

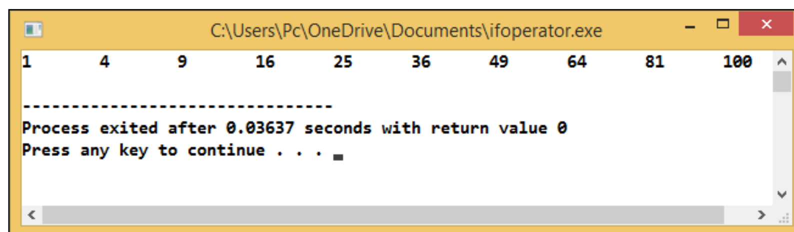
სადაც **ჭდის** ქვეშ ის იდენტიფიკატორი იგულისხმება, რომელიც ორწერტილით ბოლოვდება და პროგრამის იმ ადგილზე მიუთითებს, რომელსაც მართვა გადაეცემა.

ამოცანა 11.

შევადგინოთ პროგრამა, რომელიც goto ოპერატორის გამოყენებით 1÷10 ინტერვალიდან ყველა რიცხვის კვადრატს წარმოგვიდგენს კონსოლზე.

დასმული ამოცანის გადაწყვეტის პროგრამულ კოდს ქვემოთ წარმოდგენილი სახე აქვს, ხოლო მისი შესრულების შედეგი 31-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
int x=1;
start:
  printf("%d\t", x*x);
  x++;
  if(x<=10)
   goto start;
return 0; }
```



სურ. 31

3.2. ციკლური სტრუქტურები

ციკლი ალგორითმული სტრუქტურაა. როდესაც პროგრამული კოდის ესა თუ ის ფრაგმენტი მრავალჯერ მეორდება, აღნიშნულ პროცესს პროგრამისტები **ციკლს** უწოდებენ.

ყოველი ციკლი შედგება:

- **ციკლის გამეორების პირობის შემმოწმებელი ბლოკისაგან;**
- **ციკლის ტანისაგან.**

ციკლის ტანში არსებული ოპერაციები სრულდება მანამ, სანამ პირობის შემმოწმებელი ბლოკი ახდენს ჭეშმარიტი მნიშვნელობის დაბრუნებას.

ციკლური პროცესების განსახორციელებლად C დაპროგრამების ენაში ციკლის (იტერაციის) ოპერატორები: **while , do/while** და **for** გამოიყენება.

ციკლები არსებობს: **არითმეტიკული** და **იტერაციული**. ამასთან, ციკლები შეიძლება იყოს **მარტივი**, **რთული (ჩალაგებული)** და **უსასრულო.**

როდესაც შესასრულებელ ბრძანებათა გამეორების რიცხვი წინასწარ ცნობილია, საქმე გვაქვს არითმეტიკულ ციკლთან, ხოლო თუ გამეორებათა რიცხვი წინასწარ ცნობილი არ არის, მაშინ – იტერაციულ ციკლთან.

3.2.1. ოპერატორი while

while ციკლის ოპერატორის ჩაწერის სინტაქსი შემდეგია:

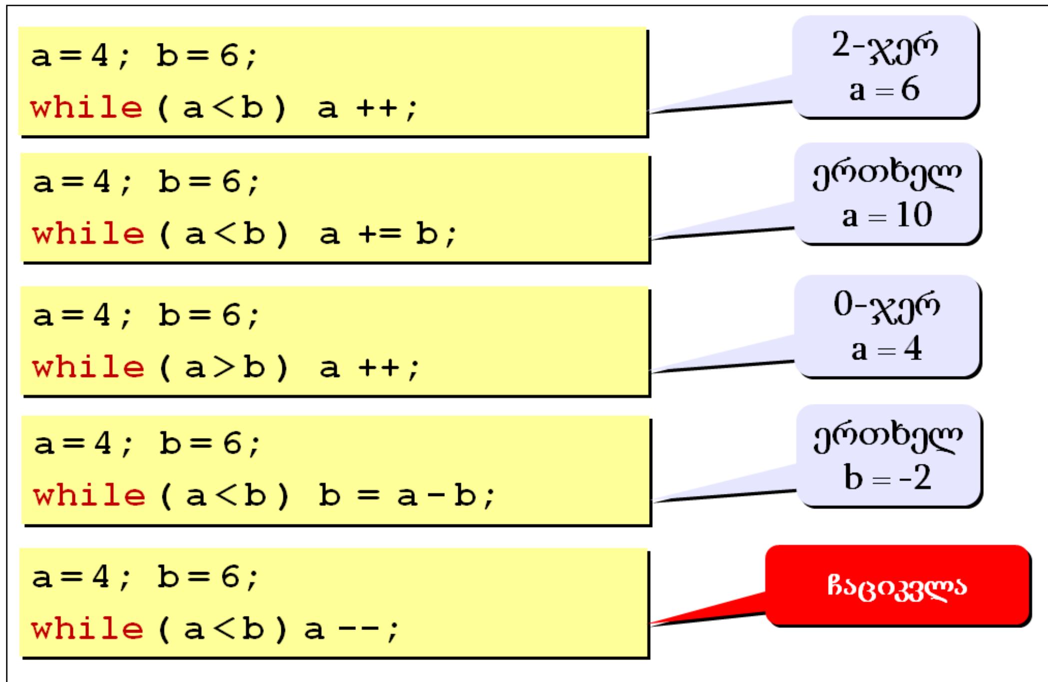
```
while(პირობა)
{
 ციკლის ტანი;
}
```

პირობის ქვეშ აქ ლოგიკის ან თანადობის ოპერაცია იგულისხმება, რომლის შედეგი ჭეშმარიტი ან მცდარი შეიძლება იყოს. შესაბამისად, **while** ციკლი მუშაობს მანამ, სანამ პირობა ჭეშმარიტია და როგორც კი პირობა გახდება მცდარი, მართვა ციკლის ტანის გარეთ არსებულ მომდევნო ოპერატორს გადაეცემა პროგრამაში.

ამგვარად, რადგან აღნიშნულ ციკლში პირობა მოწმდება დასაწყისში, თუ იგი თავიდანვე მცდარია, ციკლის ტანში არსებული ბრძანებები არ შესრულდება. თუ

ციკლში ერთი ბრძანებაა შესასრულებელი, ციკლის ტანის ამსახველი ფიგურული ფრჩხილების გამოყენება სავალდებულო არ არის, ხოლო თუ ბრძანებათა რიცხვი ერთს აღემატება, ფიგურული ფრჩხილების გამოყენება აუცილებელია!

სქემატურად წარმოვადგინოთ while ციკლის შემთხვევაში, ციკლის ტანში შესასრულებელ მოქმედებათა რაოდენობა (იხილეთ 32-ე სურათი).



სურ. 32

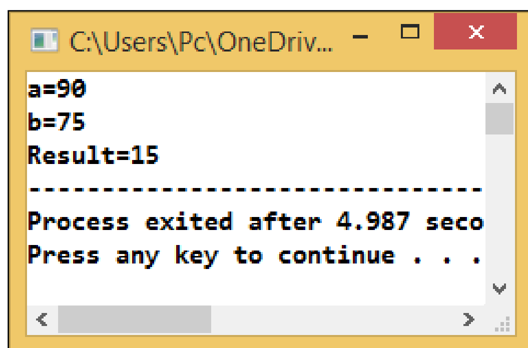
ამოცანა 12.

while ციკლის გამოყენებით, ევკლიდეს ალგორითმის საფუძველზე წარმოვად-გინოთ ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის განსაზღვრის პროგრამა.

```
#include <stdio.h>
int main() {
        int a, b;
        printf("a=");
        scanf("%d", &a);
        printf("b=");
        scanf("%d", &b);
        while(a!=b){
                if(a>b)
                 a-=b;
                 else
```

```
                b-=a;
        }
        printf("Result=%d", a);
        return 0;
}
```

პროგრამის შესრულების შედეგი 33-ე სურათზეა ნაჩვენები.



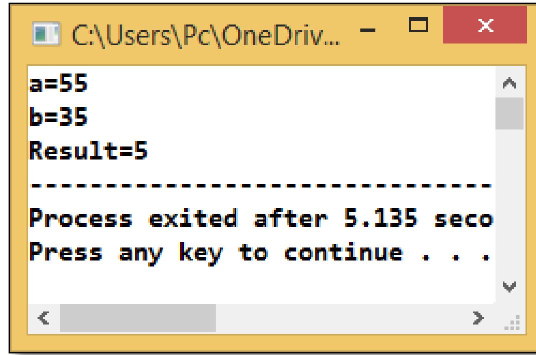
სურ. 33

ამოცანა 13.

while ციკლის გამოყენებით, ევკლიდეს **მოდიფიცირებული** ალგორითმის საფუძველზე წარმოვადგინოთ ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის განსაზღვრის პროგრამა.

```
#include <stdio.h>
int main() {
        int a, b;
        printf("a=");
        scanf("%d", &a);
        printf("b=");
        scanf("%d", &b);
        while(a!=0 && b!=0){
                if(a>b)
                 a%=b;
                 else
                  b%=a;
        }
        if(a!=0)
         printf("Result=%d", a);
         else
         printf("Result=%d", b);
        return 0; }
```

პროგრამის შესრულების შედეგი 34-ე სურათზეა ნაჩვენები.



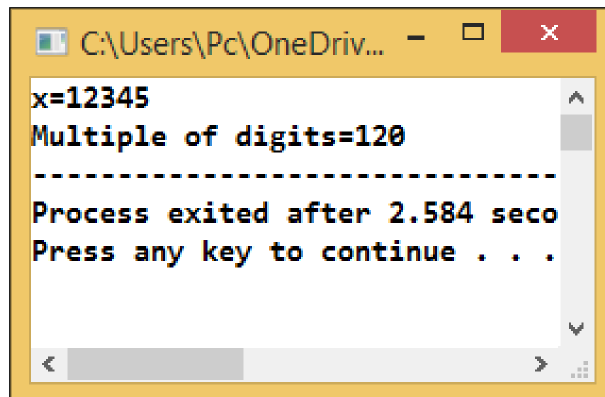
სურ. 34

ამოცანა 14.

შევადგინოთ პროგრამა, რომელიც ნებისმიერ ნატურალურ რიცხვში განსაზღვრავს მის ციფრთა ნამრავლს.

```
#include <stdio.h>
int main() {
	int x, p=1;
	printf("x=");
	scanf("%d", &x);
	while(x!=0){
		p*=x%10;
		x/=10;
	}
	 printf("Multiple of digits=%d", p);
	return 0;
}
```

პროგრამის შესრულების შედეგი 35-ე სურათზეა ნაჩვენები.



სურ. 35

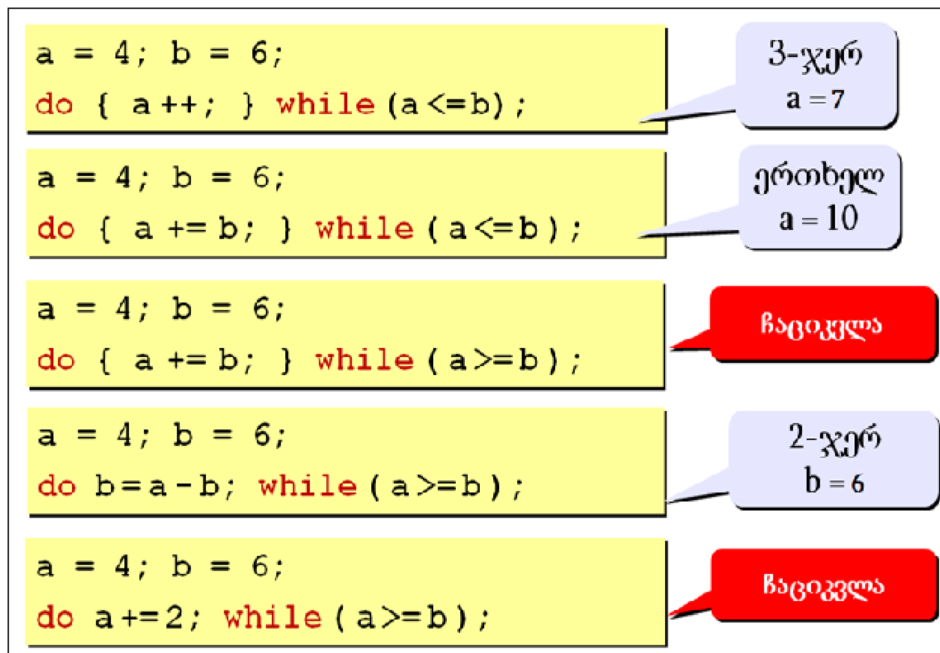
3.2.2. ოპერატორი do-while

პროგრამის წერისას, ზოგჯერ იქმნება სიტუაციები, როდესაც საჭიროა ციკლის ტანის ერთხელ მაინც შესრულება, მაშინაც კი, თუ ციკლის მმართველი გამოსა-ხულების შედეგი მცდარია. ამ შემთხვევაში **do-while** ციკლი გამოიყენება, რომლის **ჩაწერის ფორმა** შემდეგია:

do
{
ციკლის ტანი;
}
while(პირობა);

თუ ციკლში ერთი ოპერატორი სრულდება, მაშინ ფიგურული ფრჩხილების გამოყენება აუცილებელი არ არის, მაგრამ მათ ხშირად იყენებენ **do-while** ციკლის წაკითხვის გაუმჯობესების მიზნით, რადგან ის ადვილად შეიძლება აგვერიოს **while** ციკლში. **while** ციკლისგან განსხვავებით, რომელშიც ჯერ პირობა მოწმდება და შემდეგ სრულდება ციკლის კოდი, **do-while** ციკლში ჯერ ციკლის კოდი სრულ-დება, შემდეგ მოწმდება პირობა.

სქემატურად წარმოვადგინოთ do-while ციკლის შემთხვევაში, ციკლის ტანში შესასრულებელ მოქმედებათა რაოდენობა (იხილეთ 36-ე სურათი).



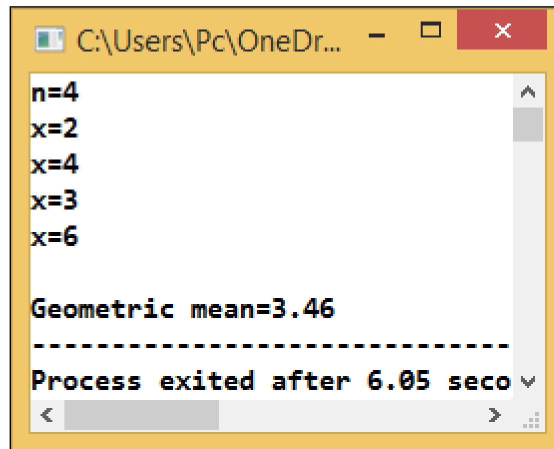
სურ. 36

ამოცანა 15.

do-while ოპერატორის გამოყენებით შევადგინოთ პროგრამა, რომელიც კლავია-ტურიდან ჩვენ მიერ შეტანილი *n* რაოდენობის რიცხვების საშუალო გეომეტრი-ულს გამოთვლის.

```
#include <stdio.h>
#include <math.h>
int main() {
        int n, i=1;
        float x, s=1;
        printf("n=");
        scanf("%d", &n);
        do{
         printf("x=");
         scanf("%f", &x);
         s*=x;
         i++;
        }while(i<=n);
        s=pow(s,1./n);
        printf("\nGeometric mean=%1.2f", s);
        return 0;
}
```

პროგრამის შესრულების შედეგი 37-ე სურათზეა ნაჩვენები.



სურ. 37

3.2.3 ოპერატორი for

for - პარამეტრიზებული ციკლია (გამეორებათა ფიქსირებული რიცხვით). აღ-ნიშნული ციკლის ორგანიზებისთვის შემდეგი ოპერაციების შესრულებაა საჭირო:

- **ინიციალება** - ციკლის მმართველი პარამეტრისთვის (მთვლელისთვის) საწყისი მნიშვნელობის მინიჭება;
- **პირობა** - ციკლის გამეორების პირობის შემოწმება;
- **მოდიფიცირება** - ციკლის მმართველი პარამეტრის (მთვლელის) მნიშვნე-ლობის ცვლილება.

ზემოთ აღნიშნული სამივე ოპერაცია მრგვალ ფრჩხილებში თავსდება და ერთმანეთისგან წერტილ-მძიმით (;) გამოიყოფა. უმეტეს შემთხვევაში, ციკლის მმართველ პარამეტრს მთელრიცხვა ცვლადი წარმოადგენს.

მმართველი პარამეტრის **ინიციალება** მხოლოდ ერთხელ ხდება და ისიც მაშინ, როდესაც for ციკლი შესრულებას იწყებს.

ციკლის გამეორების **პირობა** მოწმდება ციკლის ტანის ყოველი შესაძლო შესრულების წინ. როდესაც პირობა აღმოჩნდება მცდარი (ნულის ტოლი), ციკლი თავდება.

პარამეტრის **მოდიფიცირება** ხდება ციკლის ტანის ყოველი შესრულების შემ-დეგ. მმართველი პარამეტრის როგორც გაზრდა, ისე შემცირებაა შესაძლებელი.

თუ ციკლის მთვლელის საწყისი მნიშვნელობა საბოლოოზე ნაკლებია, მთვლე-ლის ცვლილების ბიჯი დადებითი უნდა იყოს (ასეთ მთვლელს **პირდაპირი მთვლელი** ეწოდება), წინააღმდეგ შემთხვევაში – უარყოფითი (აქ უკვე მთვლელს **უკუმთვლელი** ეწოდება). იმ შემთხვევაში, თუ ციკლის მთვლელის მნიშვნელობას არ შევცვლით ან არასწორად შევცვლით, უსასრულო ციკლის კლასიკურ მაგალითს მივიღებთ, რასაც პროგრამის ჩაციკვლას უწოდებენ.

for ციკლის ჩაწერის სინტაქსი შემდეგია:

```
for (მთვლელის საწყისი მნიშვნელობა; პირობა; მთვლელის მნიშვნელობის ცვლილება)
{
 ციკლის ტანი;
}
```

for ციკლის მუშაობის პრინციპი შემდეგში მდგომარეობს: თავდაპირველად მო-წმდება ციკლის პირობა და სანამ ის ჭეშმარიტია, სრულდება ციკლის ტანში არსე-ბული ბრძანებები, ხოლო როგორც კი პირობა გახდება მცდარი, ციკლის ტანში არსებული ბრძანებები არ შესრულდება და მართვა ციკლის ტანის გარეთ არსებულ მომდევნო ოპერატორს გადაეცემა პროგრამაში. თუ ციკლში მხოლოდ ერთი

ბრძანებაა რამდენჯერმე შესასრულებელი, ციკლის ტანის ამსახველი ფიგურული ფრჩხილების გამოყენება სავალდებულო არ არის, ხოლო თუ შესასრულებელ ბრძანებათა რიცხვი ერთს აღემატება, ფიგურული ფრჩხილების გამოყენება აუცილებელია. თუ ციკლის პირობა თავიდანვე მცდარია, ციკლის ტანში არსებული ბრძანებები არ შესრულდება და მართვა ციკლის ტანის გარეთ მომდევნო ოპერატორს გადაეცემა პროგრამაში.

for ციკლის ჩაწერისას შესაძლებელია მისი რომელიმე შემადგენელი ნაწილი არ მივუთითოთ, თუმცა გამოტოვებული ნაწილის ნაცვლად აუცილებელია წერტილმძიმის გამოყენება. ზემოთ განხილული მაგალითი სხვადასხვა პროგრამული ვარიანტის სახით შეგვიძლია წარმოვადგინოთ:

I ვარიანტი:

```
#include <stdio.h>
int main() {
     int x=1;
     for(; x<=10; x++)  //გამოტოვებულია მთვლელის საწყისი მნიშვნელობა.
     printf("%d\t", x);
     return 0; }
```

II ვარიანტი:

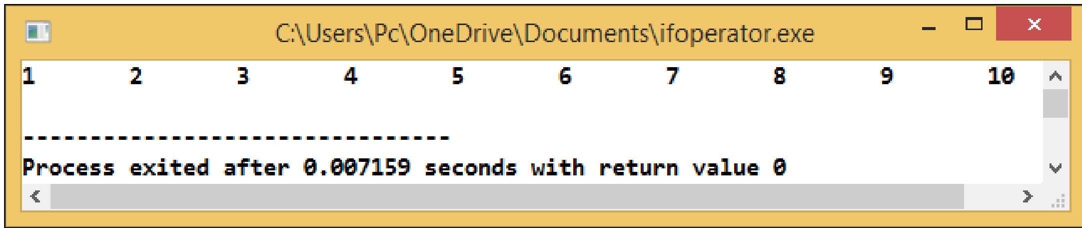
```
#include <stdio.h>
int main() {
     int x;
     for(x=1; x<=10;){ //გამოტოვებულია მთვლელის მნიშვნელობის ცვლილება.
      printf("%d\t", x);
      x++;
         }
     return 0;
}
```

III ვარიანტი:

```
#include <stdio.h>
int main() {
     int x=1;
     for(x; x<=10;){
      printf("%d\t", x);
      x++;
 }
```

```
    return 0;
}
```

ზემოთ წარმოდგენილი სამივე ვარიანტი ერთსა და იმავე შედეგებს გვაძლევს, რაც 38-ე სურათზეა ნაჩვენები.



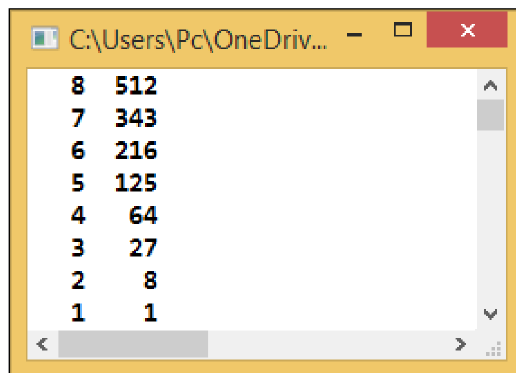
სურ. 38

ამოცანა 16.

უკუმთვლელის მქონე for ციკლის გამოყენებით შევადგინოთ პროგრამა, რომელიც 8-დან 1-ის ჩათვლით ციფრების კუბებს წარმოგვიდგენს.

```
#include <stdio.h>
#include <math.h>
int main() {
    int n, cubeN;
    for(n=8; n>=1; n--){
    cubeN=pow(n,3);
    printf("%4d %4d\n",n, cubeN);
}
    return 0; }
```

პროგრამის შესრულების შედეგები 39-ე სურათზეა ნაჩვენები.

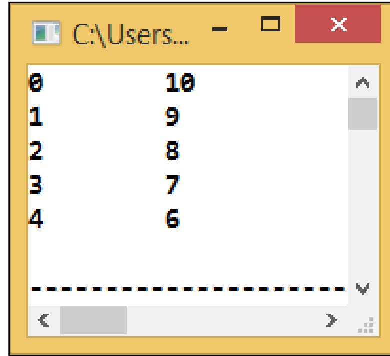


სურ. 39

for ციკლში ერთის ნაცვლად ორი მმართველი ცვლადის გამოყენებაც შეგვიძლია ისე, როგორც ეს ქვემოთ წარმოდგენილ პროგრამაშია ნაჩვენები:

```
#include <stdio.h>
int main() {
        int x1, x2;
for(x1=0, x2=10; x1<x2; x1++, x2--)
        printf("%d\t %d\n", x1, x2);
        return 0;
}
```

პროგრამის შესრულების შედეგები მე-40 სურათზეა ნაჩვენები.



სურ. 40

სქემატურად წარმოვადგინოთ for ციკლის შემთხვევაში, ციკლის ტანში არსე-ბული a ცვლადის მნიშვნელობები (იხილეთ 41-ე სურათი).

```
a=1;
for(i=1; i<4; i++) a++;
```
a = 4

```
a=1;
for(i=1; i<4; i++) a=a+i;
```
a = 7

```
a=1; b=2;
for(i=3; i>=1; i--)a+=b;
```
a = 7

```
a=1;
for(i=1; i>=3; i--)a=a+1;
```
a = 1

```
a=1;
for(i=1; i<=4; i--)a++;
```
ჩაციკვლა

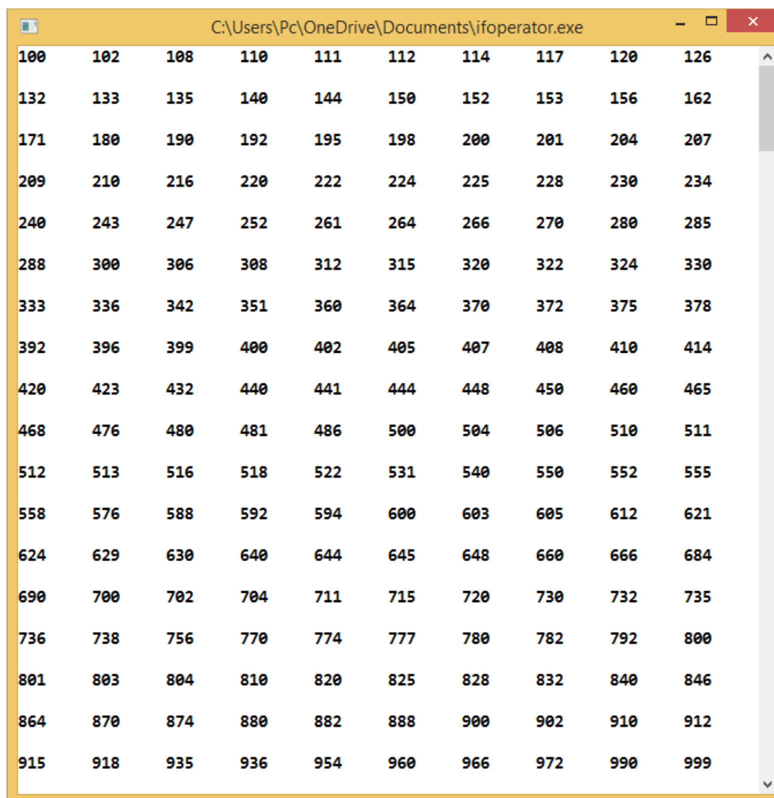
სურ. 41

ამოცანა 17.

შევადგინოთ პროგრამა, რომელიც კონსოლზე ყველა იმ სამნიშნა ნატურალურ რიცხვს წარმოგვიდგენს, რომელიც უნაშთოდ იყოფა თავის ციფრთა ჯამზე.

```
#include <stdio.h>
int main() {
int  a, b, c, s, n=0, x;
for(x=100; x<=999; x++)
 {
   a=x/100;
   b=x/10%10;
   c=x%10;
   s=a+b+c;
  if(x%s==0)
   {
     printf("%d\t", x);
     n++;
      if(n%10==0)
        printf("\n");
     }}
          return 0;  }
```

პროგრამის შესრულების შედეგები 42-ე სურათზეა ნაჩვენები.



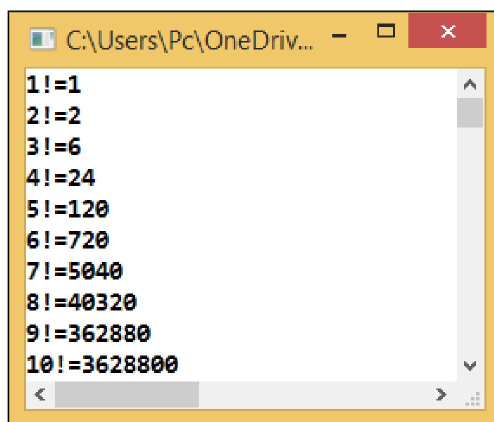
სურ. 42

ამოცანა 18.

შევადგინოთ პროგრამა, რომელიც 1-დან 10-ის ჩათვლით ნატურალური რიცხვების ფაქტორიალების მნიშვნელობებს გამოიტანს კონსოლზე.

```
#include <stdio.h>
int main() {
long f=1, x;
for(x=1; x<=10; x++)
 {
   f*=x;
   printf("%d!=%d\n", x, f);
 }
         return 0;
}
```

პროგრამის შესრულების შედეგები 43-ე სურათზეა ნაჩვენები.



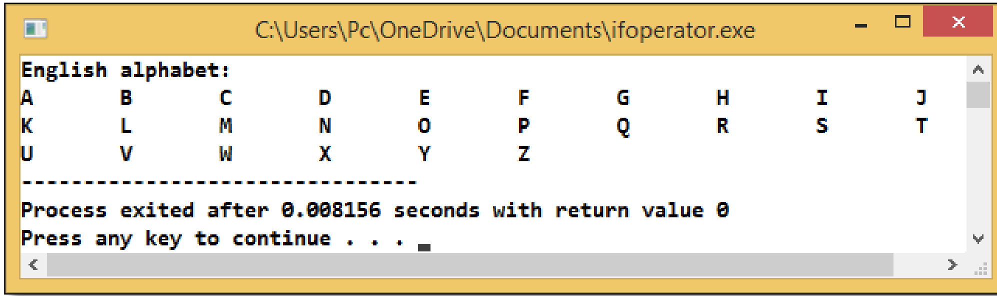
სურ. 43

ამოცანა 19.

შევადგინოთ ინგლისური ანბანის დიდი ასოების კონსოლზე წარმოდგენის პროგრამა.

```
#include <stdio.h>
int main() {
printf("English alphabet:\n");
char i;
for(i='A'; i<='Z'; i++)
 printf("%c\t", i);
         return 0;  }
```

პროგრამის შესრულების შედეგები 44-ე სურათზეა ნაჩვენები.



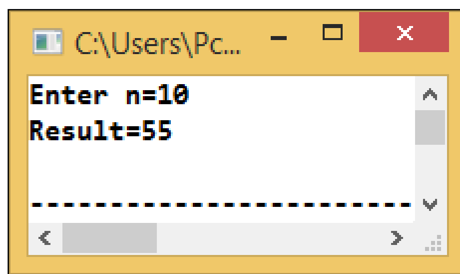
სურ. 44

ამოცანა 20.

შევადგინოთ ფიბონაჩის რიცხვითი მიმდევრობის მე-*n* წევრის მნიშვნელობის გამოთვლის პროგრამა.

```
#include <stdio.h>
int main() {
int n, fib1=1, fib2=1, fib, i;
printf("Enter n=");
scanf("%d", &n);
for(i=3; i<=n; i++){
  fib=fib1+fib2;
  fib1=fib2;
  fib2=fib;
 }
printf("Result=%d\n" ,fib);
        return 0;
}
```

პროგრამის შესრულების შედეგები 45-ე სურათზეა ნაჩვენები.



სურ. 45

3.2.4. რთული (ჩადგმული) ციკლები

დაპროგრამების ენებში ხშირად გამოიყენება ე.წ. **რთული** ანუ, ერთმანეთში ჩალაგებული ციკლები. ასეთი რთული ციკლების მუშაობის პრინციპი შემდეგია:

თავდაპირველად, გარე ციკლის მთვლელი იღებს საწყის მნიშვნელობას და მოწმდება გარე ციკლის პირობა. თუ ის ჭეშმარიტია, მუშაობას შიდა ციკლი იწყებს და სანამ ის თავის თავს არ ამოწურავს (ანუ, სანამ შიდა ციკლის პირობა არ გახდება მცდარი), გარე ციკლში გამოსვლა არ ხდება. ფაქტიურად, რთული ციკლები მუშა-ობს მანამდე, სანამ გარე ციკლის პირობა არ აღმოჩნდება მცდარი.

ამოცანა 21.

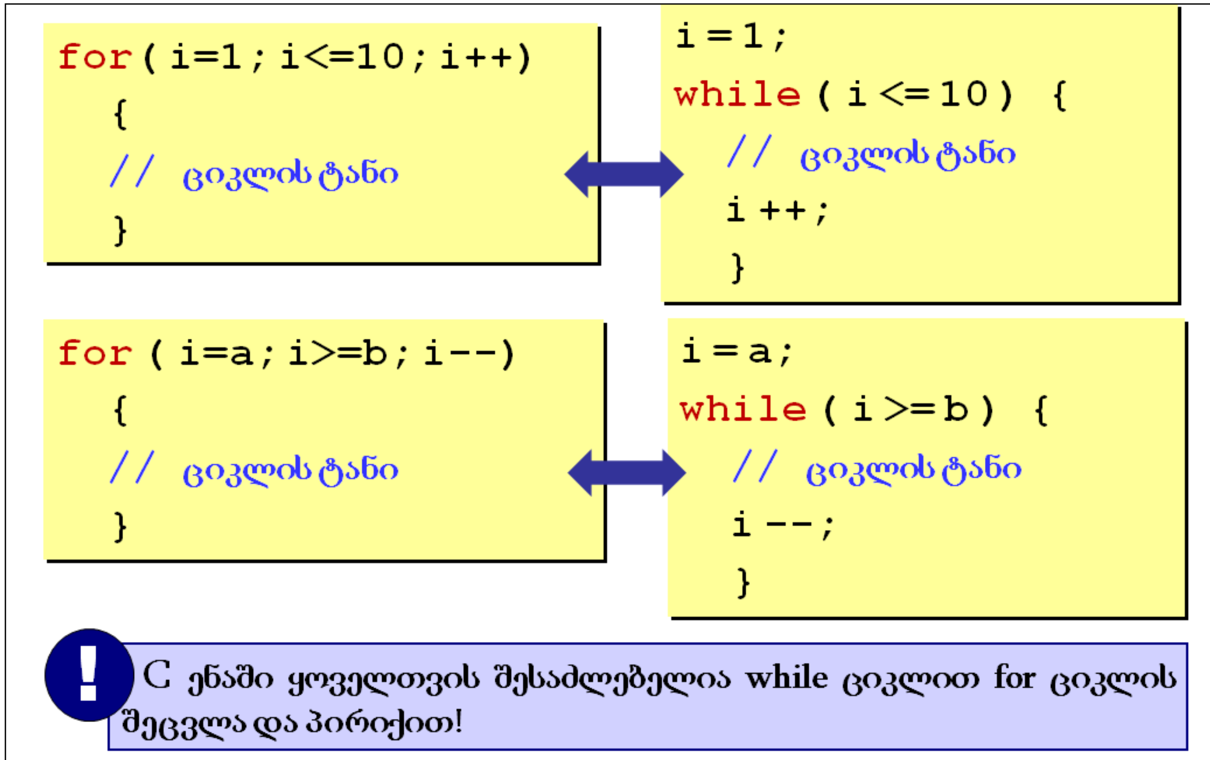
შევადგინოთ კონსოლზე ტოლფერდა სამკუთხედის წარმოდგენის პროგრამა.

```
#include <stdio.h>
int main() {
int height, i, j;
printf("height: ");
scanf("%d", &height);
for (i = 0; i < height; i++)
 {
    for (j = 1; j < height - i; j++)
      {
         printf(" ");
          }

for ( j = height - 2 * i; j <= height; j++)
         {          printf("*");
           }
         printf("\n"); }
return 0; }
```

პროგრამის შესრულების შედეგები 46-ე სურათზეა ნაჩვენები.

სურ. 46

47-ე სურათზე ნაჩვენებია შემთხვევა, თუ როგორ შეიძლება for ციკლის შეცვლა while ციკლით და პირიქით.



სურ. 47

3.2.5. ოპერატორი break

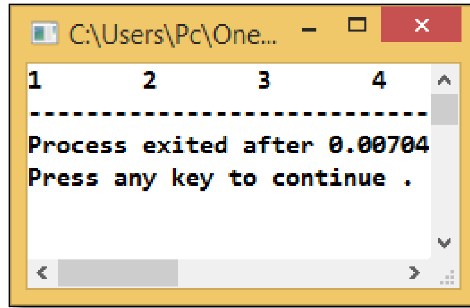
break ოპერატორი ციკლებიდან ალტერნატიული გამოსვლის საშუალებას წარმოადგენს, რაც იმას ნიშნავს, რომ გამომდინარე ციკლში არსებული ამა თუ იმ პირობის ჭეშმარიტებიდან, ხდება **ციკლური პროცესის შეწყვეტა** (ყველა იტერაციის შეუსრულებლობის მიუხედავად) და მართვა ციკლის ტანის მომდევნო ოპერატორს გადაეცემა პროგრამაში.

წარმოვადგინოთ **break** ოპერატორის გამოყენების ამსახველი პროგრამა:

```
#include <stdio.h>
int main() {
int i;
for(i=1; i<=9; i++)
{
 if(i==5) break;
 printf("%d\t", i);
}
```

```
return 0; }
```

პროგრამის შესრულების შედეგები 48-ე სურათზეა ნაჩვენები.



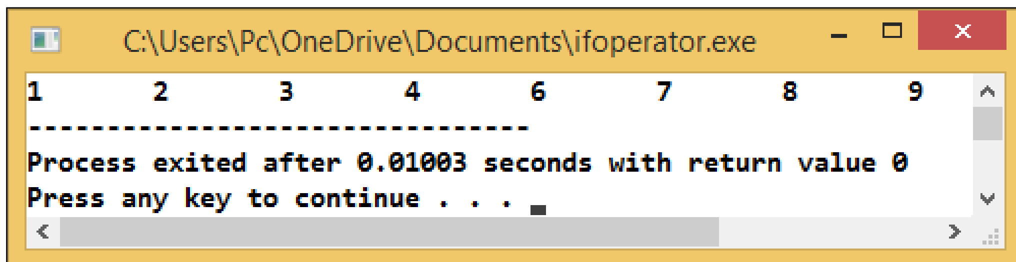
სურ. 48

3.2.6. ოპერატორი continue

continue ოპერატორი ახდენს ციკლის **მიმდინარე იტერაციის გამოტოვებას** და დანარჩენი იტერაციების შესრულებას. წინა 63-ე გვერდზე წარმოდგენილი პროგ-რამული კოდი შევცვალოთ იმ პირობით, რომ ეკრანზე ერთიდან ცხრის ჩათვლით ყველა ციფრი გამოვიტანოთ ხუთის გარდა.

```
#include <stdio.h>
int main() {
int i;
for(i=1; i<=9; i++)
{
 if(i==5) continue;
 printf("%d\t", i);
}
return 0; }
```

პროგრამის შესრულების შედეგები 49-ე სურათზეა ნაჩვენები.



სურ. 49

IV თავი
მასივები

მასივი კომპიუტერის მეხსიერებაში არსებულ მიმდევრობითი უჯრედების ჯგუფს წარმოადგენს, რომელთაც საერთო სახელი და ერთიდაიგივე ტიპი გააჩნიათ. ყოველი მასივი ხასიათდება **ზომით** და **განზომილებით**.

მასივში შემავალი ელემენტების რაოდენობა მის ზომას განსაზღვრავს. თითოეულ ელემენტს შეიძლება ერთი ან მეტი ინდექსი გააჩნდეს, რაც მასივის განზომილების განსაზღვრას განაპირობებს. თუ მასივში შემავალ ყოველ ელემენტს თითო ინდექსი გააჩნია, მას **ერთგანზომილებიანი მასივი** ეწოდება, ხოლო თუ მასივში ყოველი ელემენტი ორინდექსიანია, მას **ორგანზომილებიანი მასივი** ანუ **მატრიცა** ეწოდება.

განვიხილოთ მასივებთან დაკავშირებული ძირითადი ცნებები.

- **მასივის ელემენტი (მასივის ელემენტის მნიშვნელობა)** - მნიშვნელობა, რომელიც კომპიუტერის ოპერატიული მეხსიერების განსაზღვრულ უჯრაში ინახება (ეს უკანასკნელი მასივის საზღვრებშია განთავსებული), ამავდროულად მეხსიერების ამავე უჯრის მისამართი;
- **მასივის მისამართი** - მასივის საწყისი (ნულოვანი) ელემენტის მისამართი;
- **მასივის სახელი** - იდენტიფიკატორი, რომელიც მასივის ელემენტებზე მიმართვისთვის გამოიყენება;
- **მასივის ზომა** - მასივის ელემენტების რაოდენობა;
- **მასივის სიგრძე** - ბაიტების რაოდენობა, რომელიც კომპიუტერის მეხსიერებაში მასივის ყველა ელემენტის შესანახად არის გამოყოფილი:

მასივის სიგრძე = ელემენტის ზომა * ელემენტების რაოდენობა

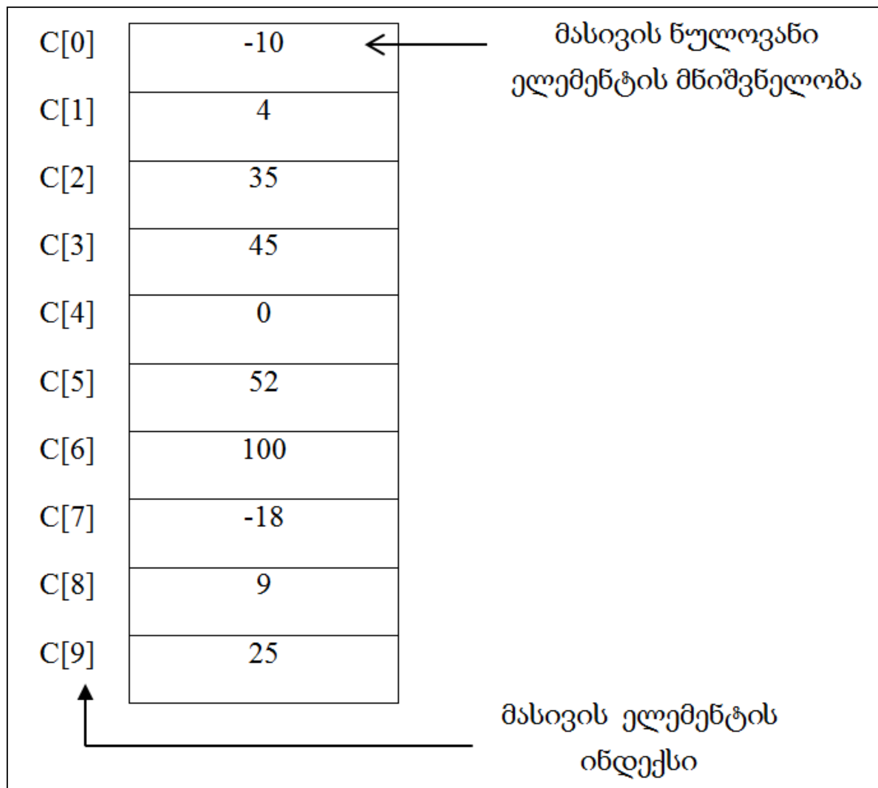
- **ელემენტის ზომა** - მასივის ერთი ელემენტის მიერ დაკავებული ბაიტების რაოდენობა.
- **ელემენტის მისამართი** - კომპიუტერის მეხსიერების საწყისი უჯრის მისამართი, სადაც ელემენტია განთავსებული;
- **ელემენტის ინდექსი** - მთელი ტიპის მნიშვნელობა ან გამოსახულება, რომლის შედეგი ასევე, მთელი ტიპისაა. გამოიყენება ელემენტზე მიმართვისთვის;

- **ელემენტის მნიშვნელობა** - კონკრეტული ელემენტის კონკრეტული მნიშვნე-ლობა.

4.1.1. ერთგანზომილებიანი მასივები

დაპროგრამების ენა C-ში ყოველ მასივს **ნულოვანი ელემენტი** გააჩნია, ანუ როგორც წესი, ნებისმიერი მასივის **პირველი ელემენტის ინდექსი ნულის ტოლია**.

50-ე სურათზე წარმოდგენილია 10-ელემენტიანი ერთგანზომილებიანი C მასივი.



სურ. 50

ამგვარად, მასივი არის სტრუქტურა, რომელიც მონაცემთა სახით მოიცავს ერთმანეთთან დაკავშირებულ ერთიდაიმავე ტიპის მქონე ელემენტებს.

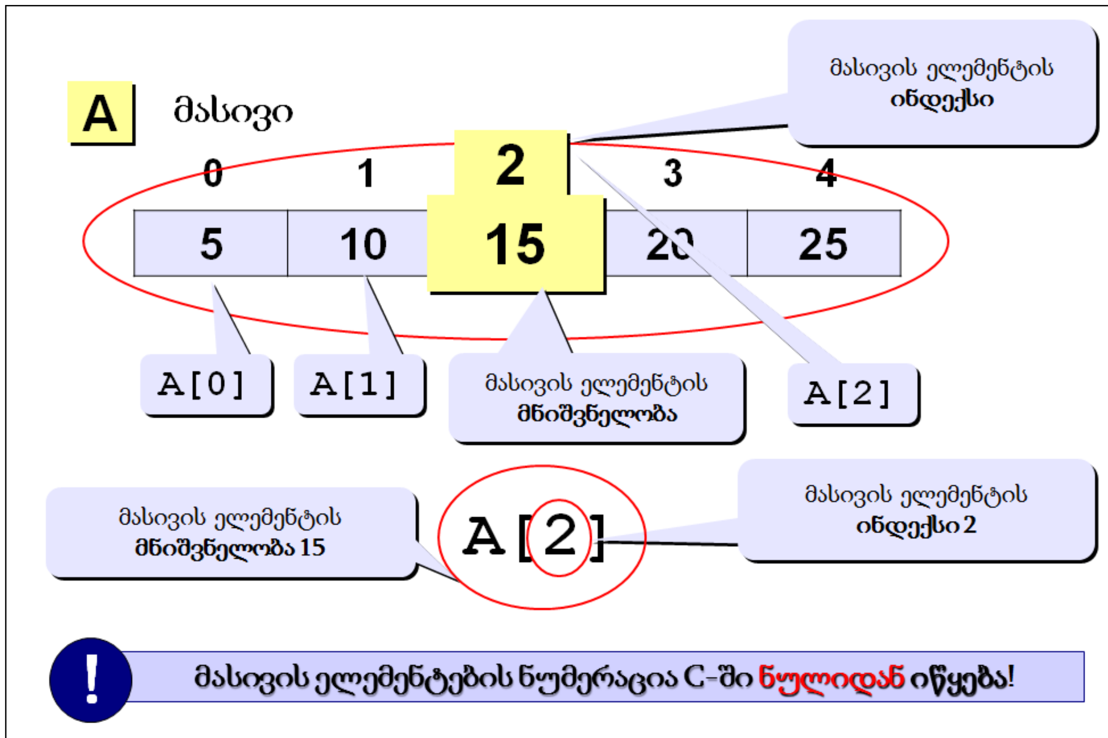
კომპიუტერის მეხსიერებაში ყოველი მასივი გარკვეულ მოცულობას იკავებს. მასივი, მსგავსად ნებისმიერი სხვა ობიექტისა, პროგრამაში გამოყენებამდე უნდა იქნას გაცხადებული, რათა კომპილატორმა მისთვის კომპიუტერის მეხსიერებაში საჭირო მოცულობის გამოყოფა წინასწარ მოახდინოს. ამ მიზნით, პროგრამისტმა აუცილებელია, მიუთითოს მასივში შემავალი ელემენტების ტიპი, მათი რაოდენობა და მასივის იდენტიფიკატორი.

შესაძლებელია ერთი და იმავე ტიპის მქონე ელემენტებისგან შემდგარი რამდენიმე მასივის ერთდროულად აღწერაც. მაგალითად:

int b[100], a[25];

ამ შემთხვევაში აღიწერება მთელრიცხვა ასი ელემენტისგან შემდგარი **b** მასივი და ამავე ტიპის ოცდახუთი ელემენტისგან შემდგარი **a** მასივი.

52-ე სურათზე ნაჩვენებია ერთგანზომილებიანი A მასივის სტრუქტურა.

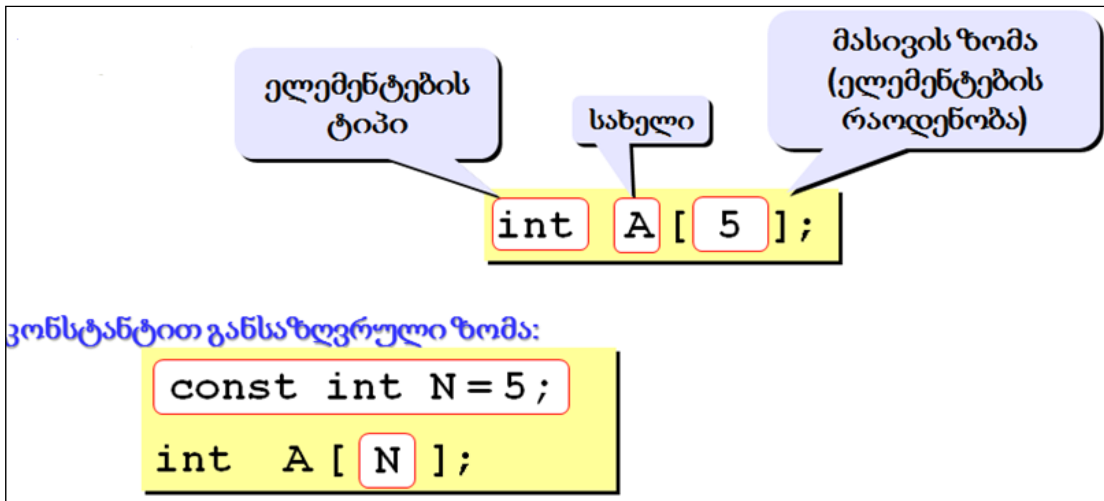


სურ. 51

შესაძლოა, გაგვიჩნდეს კითხვა: რისთვის არის საჭირო მასივის დეკლარირება? პასუხი შემდეგია:

- განისაზღვრება მასივის სახელი;
- განისაზღვრება მასივის ტიპი;
- განისაზღვრება ელემენტების რაოდენობა;
- ადგილი გამოიყოფა კომპიუტერის მეხსიერებაში.

52-ე სურათზე ნაჩვენებია მასივების დეკლარირების ნიმუშები.



სურ. 52

მასივის ელემენტებს საწყისი მნიშვნელობები შეიძლება მისი გაცხადების დროს მიენიჭოს. ამ პროცესს ელემენტების **ინიციალება** ეწოდება. აღნიშნულ შემთხვევაში მასივის ელემენტების მნიშვნელობები ფიგურულ ფრჩხილებში თავსდება და ერთიმეორისგან მძიმით გამოიყოფა. მაგალითად:

int A[7]={1, 2, 5, 10, 0, -5, 8};

თუ მასივში არსებული ელემენტების რაოდენობა მეტი აღმოჩნდება მნიშვნე-ლობათა რაოდენობაზე, დარჩენილი ელემენტები ავტომატურად იღებენ საწყის ნულოვან მნიშვნელობებს. მაგალითად:

int A[10]={ 0 };

აღნიშნული ოპერატორი საწყის ნულოვან მნიშვნელობას ცხადი სახით ანიჭებს A მასივის პირველ ელემენტს, ხოლო დარჩენილი ცხრა ელემენტი არაცხადი სახით იღებს ამ მნიშვნელობას. უნდა აღინიშნოს, რომ მასივის გაცხადების პროცესში მისი ელემენტები ავტომატურად არ იღებენ საწყის ნულოვან მნიშვნელობებს. პროგრამისტმა ამ დროს პირველ ელემენტს მაინც უნდა მიანიჭოს ნულოვანი მნიშვნელობა, რათა დარჩენილი ელემენტები ავტომატურად განულდეს.

53-ე სურათზე ნაჩვენებია მასივების დეკლარირებისა და ინიციალების ნიმუშები.

დეკლარირება:

```
int X[10], Y[10];
float zz, A[20];
char s[80];
```

ინიციალება:

```
int A[4] = { 8, -3, 4, 6 };
float B[2] = { 1. };
char C[3] = { 'A', '1', 'Ю' };
```

დანარჩენი ნულოვანი მნიშვნელობებით

! საწყისი მნიშვნელობები თუ მოცემული არ არის, უჯრებში „ნაგავი-“ა!

სურ. 53

მასივებთან მუშაობის დროს, დამწყები პროგრამისტები ხშირად უშვებენ გარკვეული სახის შეცდომებს. სწორედ, ეს შემთხვევებია წარმოდგენილი 54-ე სურათზე.

```
const int N = 10;
      float A[N];
```

```
int X[4.5];
```

```
int A[10];
A[10] = 0;
```

```
float X[5];
int n = 1;
X[n-2] = 4.5;
X[n+8] = 12.;
```

მასივის საზღვრებს გარეთ გასვლა

წილადი ნაწილი იკარგება (შეცდომა არ არის)

```
int X[4];
X[2] = 4.5;
```

```
float A[2] = { 1, 3.8 };
```

```
float B[2] = { 1., 3.8, 5.5 };
```

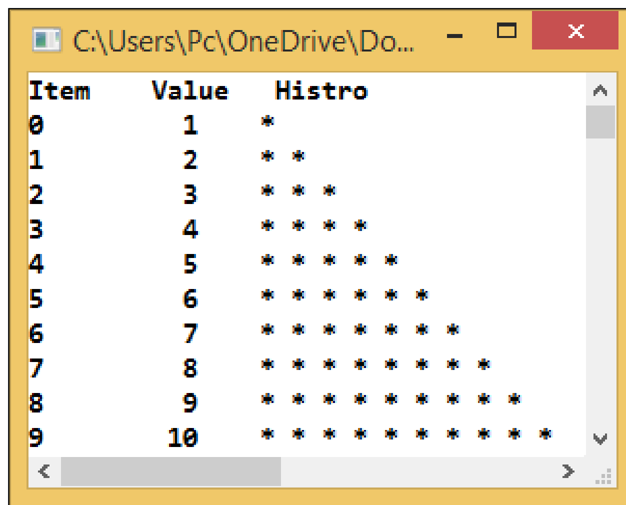
სურ. 54

ამოცანა 22.

შევადგინოთ პროგრამა, რომელიც მოცემული ათელემენტიანი მთელრიცხვა A მასივიდან წაიკითხავს ელემენტების მნიშვნელობებს და წარმოადგენს მასივს შესაბამის ჰისტროგრამასთან ერთად.

```
#include <stdio.h>
int main() {
const int size=10;
int A[size], i, j;
for(i=0; i<size; i++)
A[i]=i+1;
printf("Item\tValue\tHistro\n");
for(i=0;i<size;i++)
{
printf("%d %9d   ", i, A[i]);
for(j=1; j<=A[i]; j++)
printf(" *");
printf("\n");
}
return 0;
}
```

პროგრამის შესრულების შედეგები 55-ე სურათზეა ნაჩვენები.



სურ. 55

ამოცანა 23.

შევადგინოთ პროგრამა, რომელიც მოცემული 12-ელემენტიანი მთელრიცხვა A მასივიდან წაიკითხავს ელემენტების მნიშვნელობებს, გამოთვლის მასივის ელემენტების ჯამს, ნამრავლს, საშუალო არითმეტიკულ მნიშვნელობებს და განსაზღვრავს მასივში მაქსიმალური და მინიმალური ელემენტების მნიშვნელობებს.

```
#include <stdio.h>
int main() {
const int size=12;
int A[size], i, max, min;
double p=1;
float s=0;
printf("Current array:\n");
for(i=0; i<size; i++){
  A[i]=2*i+2;
  printf("%4d", A[i]);
  s+=A[i];
  p*=A[i];
}
max=A[0]; min=A[0];
for(i=0; i<size; i++){
  if(max<=A[i])
    max=A[i];
  if(min>=A[i])
    min=A[i];
}
printf("\nSum=%.f\n",s);
printf("Product=%.f\n",p);
s=s/size;
printf("Average=%.3f%\n",s);
printf("Max=%d\n",max);
printf("Min=%d\n",min);
return 0;
}
```

პროგრამის შესრულების შედეგები 56-ე სურათზეა ნაჩვენები.

```
C:\Users\Pc\OneDrive\Documents\ifoperat...
Current array:
   2   4   6   8  10  12  14  16  18  20  22  24
Sum=156
Product=1961990553600
Average=13.000
Max=24
Min=2

--------------------------------
```

სურ. 56

4.1.2. მრავალგანზომილებიანი მასივები

როგორც ზემოთ ავღნიშნე, დაპროგრამების ენა C-ში მასივებს შეიძლება რამდენიმე ინდექსი გააჩნდეს. მრავალგანზომილებიანი მასივების ტიპიური წარმომადგენელია ერთგვარ მნიშვნელობათა ცხრილები, რომლებიც ინფორმაციას სტრიქონებსა და სვეტებში მოიცავენ.

იმისათვის, რომ ამ ცხრილებში ესა თუ ის ელემენტი განისაზღვროს, საჭიროა მისი ორი ინდექსის მითითება. პირველი მათგანი – სტრიქონის ნომერია, ხოლო მეორე – იმ სვეტის ნომერი, რომელთა გადაკვეთაზეც ელემენტი იმყოფება.

იმ მასივებს ან ცხრილებს, რომელთა ცალკეულ ელემენტზე მისამართად ორი ინდექსის გამოყენებაა საჭირო, **ორგანზომილებიანი მასივები** ანუ **მატრიცები** ეწოდება. აღსანიშნავია, რომ მრავალგანზომილებიან მასივებში ელემენტებს შეიძლება ორზე მეტი ინდექსიც გააჩნდეს.

მრავალგანზომილებიანი მასივების ელემენტები ისევე იღებენ საწყის მნიშვნელობებს, როგორც ერთგანზომილებიანი მასივის ელემენტები მათი გაცხადების დროს.

57-ე სურათზე წარმოდგენილია ორგანზომილებიანი *a* მასივი, რომელიც სამ სტრიქონსა და ოთხ სვეტს მოიცავს. ამ შემთხვევაში ამბობენ, რომ საქმე გვაქვს მატრიცასთან განზომილებით სამი ოთხზე.

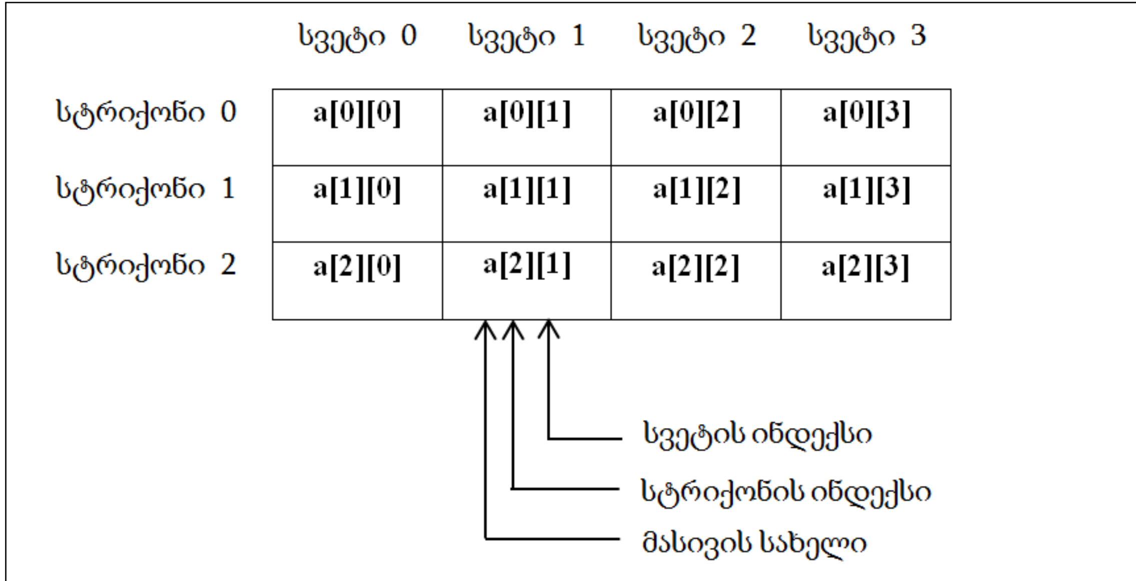
a მასივის ყოველი ელემენტი მისი სახელით განისაზღვრება და შემდეგი სახით ჩაიწერება: *a*[**i**][**j**], სადაც *a* - მასივის **იდენტიფიკატორია**, ხოლო **i** და **j** - **ინდექსები**, რომლებიც ერთმნიშვნელოვნად განსაზღვრავენ *a* მასივის ყოველ ელემენტს. უნდა აღინიშნოს, რომ მატრიცის რიგით პირველ ელემენტს, მსგავსად ერთგანზომილებიანი მასივებისა, ნულოვანი ინდექსები გააჩნია.

მატრიცაში ელემენტთა მნიშვნელობების დაჯგუფება სტრიქონებში ხდება და ისინი ფიგურულ ფრჩხილებში თავსდება.

a [**2**][**2**] მატრიცის ინიციალება შესაძლებელია შემდეგი სახით:

```
int a [2][2]={ {1, 2}, {3, 4} };
```

ამგვარად, $a[0][0]$ და $a[0][1]$ ელემენტებს ენიჭებათ შესაბამისად ერთისა და ორის ტოლი მნიშვნელობები, ხოლო $a[1][0]$ და $a[1][1]$ ელემენტებს – შესაბამისად სამისა და ოთხის ტოლი მნიშვნელობები.



სურ.57

თუ მატრიცის ინიციალება ხდება შემდეგი სახით:

int *a* [2][2]={ {1}, {3, 4} };

ეს ნიშნავს, რომ $a[0][0]$ ელემენტი იღებს ერთის ტოლ მნიშვნელობას, ხოლო $a[0][1]$ ავტომატურად ნულდება; $a[1][0]$ ელემენტს ენიჭება სამის, ხოლო $a[1][1]$ ელემენტს – ოთხის ტოლი მნიშვნელობა.

აღსანიშნავია, რომ კომპიუტერის მეხსიერებაში მასივის ყველა ელემენტი, მიუხედავად მათი ინდექსების რაოდენობისა, მიმდევრობით ინახება. ორგანზომი-ლებიან მასივებში კი პირველი სტრიქონი მეორე სტრიქონის წინ თავსდება. აქ ყოველი სტრიქონი შეიძლება განხილულ იქნას, როგორც ერთგანზომილებიანი მასივი.

58-ე სურათზე წარმოდგენილია მატრიცების დეკლარირებისა და ინიციალების ნიმუშები.

ცნობილია, რომ თუ $[a_{i\,j}]$mxn მატრიცაში n=m (ანუ სვეტების და სტრიქონების რაოდენობა ერთმანეთის ტოლია), მაშინ ასეთ მატრიცას **კვადრატულ მატრიცას** უწოდებენ. ხოლო *a* მატრიცის **ტრანსპონირებულია მატრიცა**, რომლის სტრიქო-ნებს, შესაბამისად, *a* მატრიცის სვეტები შეადგენენ, ხოლო სვეტებს კი – შესაბა-

მისად a მატრიცის სტრიქონები. a მატრიცის ტრანსპონირებული მატრიცა წარმოდგენილია, როგორც: a^t ანუ მათემატიკური თვალსაზრისით ეს მატრიცები 59-ე სურათზე ნაჩვენები სახით ჩაიწერება.

დეკლარირება და ინიცალება:

```
const int N = 3, M = 4;
int A[N][M];
float a[2][2] = {{3.2, 4.3}, {1.1, 2.2}};
char sym[2][2] = { 'a', 'b', 'c', 'd' };
```

კლავიატურიდან შეტანა

```
for ( j=0; j<M; j++ )
  for ( i=0; i<N; i++ ) {
    printf ( "A[%d][%d]=", i, j);
    scanf ( "%d", &A[i][j] );
    }
```

i j

```
A[0][0]=25
A[0][1]=14
A[0][2]=14
   ...
A[2][3]=54
```

სურ. 58

$$a = \begin{vmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{vmatrix}; \qquad a^t = \begin{vmatrix} a_{00} & a_{10} & a_{20} & a_{30} \\ a_{01} & a_{11} & a_{21} & a_{31} \\ a_{02} & a_{12} & a_{22} & a_{32} \\ a_{03} & a_{13} & a_{23} & a_{33} \end{vmatrix}$$

სურ. 59

როგორც ჩანაწერიდან ჩანს, მატრიცის ტრანსპონირების დროს საწყისი მატრიცის მთავარ დიაგონალზე არსებული ელემენტები თავთავის ადგილებზე უცვლელად რჩება.

ქვემოთ წარმოდგენილია მატრიცის ტრანსპონირების ამსახველი პროგრამული კოდი.

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

```
int main() {
	const int n=4;
	int a[n][n], i, j, t;
	printf("Current Matrix:\n");
	srand(time(0));
	for(i=0; i<n; i++){
	 for(j=0; j<n; j++){
		a[i][j]=10+rand()%71;
		printf("%d\t",a[i][j]);
	 }
	  printf("\n");
}
	printf("\nTransposed Matrix:\n");
	 for(i=0; i<n; i++)
	 for(j=i+1; j<n; j++){
		t=a[i][j];
		a[i][j]=a[j][i];
		a[j][i]=t;
	 }
	 for(i=0; i<n; i++){
	 for(j=0; j<n; j++)
		printf("%d\t",a[i][j]);
	  printf("\n");
}
	return 0;
}
```

პროგრამული კოდის შესრულების შედეგები მე-60 სურათზეა ნაჩვენები.

```
C:\Users\Pc\OneDriv...
Current Matrix:
43      78      59      80
51      71      76      49
46      49      31      29
56      41      25      29

Transposed Matrix:
43      51      46      56
78      71      49      41
59      76      31      25
80      49      29      29
```

სურ. 60

აღსანიშნავია, რომ წარმოდგენილ პროგრამულ კოდში მატრიცის ელემენტები [10;80] ინტერვალიდან შემთხვევითი რიცხვების გენერატორის საშუალებით იღებენ შემთხვევით მნიშვნელობებს ფორმულით: a[i][j]=10+rand()%71, ხოლო კვაზი-შემთხვევითი რიცხვების გამომუშავების მიზნით გამოყენებულია ფუნქცია srand(time(0)). სამივე (rand() და srand(time()) ფუნქციების მუშაობითვის პროგრამულ კოდში აუცილებელია <stdlib.h> ბიბლიოთეკის ჩართვა.

კვადრატული მატრიცის **მთავარ დიაგონალზე** განთავსებულია ერთნაირი ინდექსების მქონე ელემენტები. მთავარი დიაგონალის ზემოთ ყველა ელემენტის სტრიქონის ინდექსი სვეტის ინდექსზე ნაკლებია, ხოლო მთავარი დიაგონალის ქვემოთ - ყველა ელემენტის სტრიქონის ინდექსი სვეტის ინდექსზე მეტია.

რაც შეეხება **არამთავარ დიაგონალზე** განლაგებულ ელემენტებს, მათი სტრიქონის და სვეტის აღმნიშვნელი ინდექსების ჯამი ერთით ნაკლებია სტრიქონების (ან სვეტების, რადგან ეს სიდიდეები კვადრატულ მატრიცაში ერთმანეთის ტოლია) რაოდენობაზე. ე.ი. თუ მატრიცის განზომილებაა 5x5-ზე, არამთავარ დიაგონალზე არსებული ელემენტების შესაბამისი იდექსების ჯამი 4-ის ტოლია. არამთავარი დიაგონალის ზემოთ ყველა ელემენტის ინდექსების ჯამი 4-ზე ნაკლებია, ხოლო არამთავარი დიაგონალის ქვემოთ ყველა ელემენტის ინდექსების ჯამი 4-ზე მეტია.

61-ე სურათზე ეს ზოგადი პირობებია წარმოდგენილი კვადრატული მატრიცისთვის.

მთავარ დიაგონალზე:	**if(i==j)**
მთავარი დიაგონალის ზემოთ:	**if(i<j)**
მთავარი დიაგონალის ქვემოთ:	**if(i>j)**
არამთავარ (მცირე) დიაგონალზე:	**if(i+j==rows-1) ან if(i+j==columns-1)**
მცირე დიაგონალის ზემოთ:	**if(i+j<rows-1) ან if(i+j<columns-1)**
მცირე დიაგონალის ქვემოთ:	**if(i+j>rows-1) ან if(i+j>columns-1)**

სურ. 61

ამოცანა 24.

შევადგინოთ პროგრამა, რომლიც მთელირიცხვა *a*[4x5] მატრიცის ელემენტებს [10;50] ინტერვალიდან მიანიჭებს თანაბარი ალბათობით განაწილებულ კვაზი-შემთხვევით მნიშვნელობებს. გამოთვლის მატრიცის ელემენტების ჯამების მნიშვ-ნელობებს სტრიქონების მიხედვით, განსაზღვრავს მიღებულ შედეგებს შორის უდიდესს და კონსოლზე გამოიტანს ყველა ეტაპის შედეგს და იმ ელემენტებს, რომელთაც ჯამის მნიშვნელობაც სტრიქონების მიხედვით მაქსიმალურია.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
const int rows=4, columns=5;
int main() {
        int a[rows][columns], sum[rows], i, j, max, k;
        printf("Current matrix:\n");
        srand(time(0));
        for(i=0; i<rows; i++){
         for(j=0; j<columns; j++){
          a[i][j]=10+rand()%41;
           printf("%4d", a[i][j]);
   }
        printf("\n");
}
        for(i=0; i<rows; i++){
          sum[i]=0;
         for(j=0; j<columns; j++)
           sum[i]+=a[i][j];
           printf("sum[%d]=%d\n", i, sum[i]);
  }
  max=sum[0];
  for(i=0; i<rows; i++)
  if(max<=sum[i]){
        max=sum[i];
        k=i;
  }
  printf("\nmax=%d\n", max);
  for(j=0; j<columns; j++)
  printf("%4d", a[k][j]);
return 0;
}
```

პროგრამული კოდის შესრულების შედეგები 62-ე სურათზეა ნაჩვენები.

```
C:\Users\Pc\OneDriv...
Current matrix:
  45  25  34  32  16
  15  14  20  38  33
  13  37  21  20  24
  39  32  32  32  44
sum[0]=152
sum[1]=120
sum[2]=115
sum[3]=179

max=179
  39  32  32  32  44
```

სურ. 62

4.1.3. სიმბოლოების მასივები

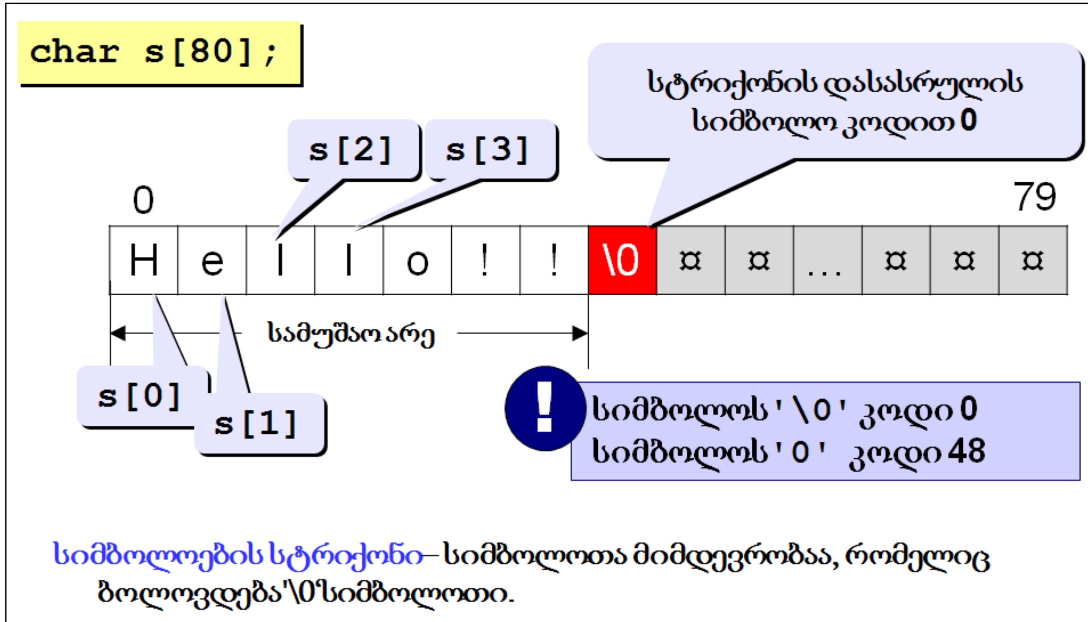
დაპროგრამების ენას C-ს არ გააჩნია მონაცემთა სტრიქონული ტიპის მხარდაჭერა, მაგრამ ის საშუალებას გვაძლევს სტრიქონები ორი სხვადასხვა გზით განვსაზღვროთ. პირველ შემთხვევაში გამოიყენება **სიმბოლოების მასივი**, ხოლო მეორეში - **მიმთითებელი** (ამ უკანასკნელს წინამდებარე სახელმძღვანელოს მომდევნო თავებში განვიხილავთ) მასივის პირველ სიმბოლოზე.

პროგრამაში სტრიქონები განისაზღვრება, როგორც:

- **სტრიქონული კონსტანტები;**
- **სიმბოლოების მასივები;**
- **მიმთითებელი სიმბოლურ ტიპზე;**
- **სტრიქონების მასივები**.

ამას გარდა, წინასწარ გათვალისწინებული უნდა იყოს სტრიქონების შესანახად მეხსიერების გამოყოფა. სიმბოლოების ნებისმიერი მიმდევრობა, რომელიც ორმაგ აპოსტროფებშია (“ “) მოთავსებული, განიხილება, როგორც **სტრიქონული კონსტანტა**.

63-ე სურათზე წარმოდგენილია სიმბოლოების სტრიქონის ნიმუში.



სურ. 63

სტრიქონის კონსოლზე კორექტულად გამოტანის მიზნით საჭიროა, ნებისმიერი სტრიქონი დასრულდეს ნულოვანი (**'\0'**) სიმბოლოთი, რომლის მთელრიცხვა მნიშვნელობა ნულის ტოლია. სტრიქონული კონსტანტას გამოცხადებისას, ნულოვანი სიმბოლო მას ავტომატურად ემატება. ამგვარად, სიმბოლოების მიმდევრობა, რომელიც სტრიქონულ კონსტანტას წარმოადგენს, კომპიუტერის ოპერატიულ მეხსიერებაში განთავსდება ნულოვანი ბაიტის ჩათვლით. მეხსიერების უჯრებში სტრიქონის ელემენტები მიმდევრობით ლაგდება. შესაბამისად, **სტრიქონი სიმბოლოების მასივს წამოადგენს**. ტრიქონის ყოველი სიმბოლოს კოდის შესანახად 1 ბაიტი გამოიყოფა. სტრიქონული კონსტანტები სტატიკურ მეხსიერებაში თავსდება. ორმაგ აპოსტროფებში მოთავსებული სიმბოლოების მიმდევრობის საწყისი მისამართი განიხილება, როგორც თავად სტრიქონის მისამართი.

სიმბოლოების მასივის განსაზღვრისას, აუცილებელია კომპიუტერს „ვაცნობოთ" მეხსერების საჭირო ზომა. ასე, მაგალითად: char m[82];

სიმბოლოების მასივის ნიმუში 64-ე სურათზეა წარმოდგენილი.

კომპილატორს თავადაც შეუძლია სიმბოლოების მასივის ზომის განსაზღვრა, თუ მასივის ინიციალება სტრიქონული კონსტანტის დეკლარირებისას ხდება. ასე, მაგალითად:

char m2[]="My Name is Lela.";

char m3[]={‘ M’, ‘y’, ‘ ‘, ‘c’, ‘i’, ‘t’, ‘y’,’\0’};

ამ შემთხვევაში, m2 და m3 წარმოადგენს მიმთითებლებს მასივების პირველ ელემენტებზე.

- m2 ეკვივალენტურია &m2[0]-ის;
- m2[0] ეკვივალენტურია **‘M’** -ის;
- m2[1] ეკვივალენტურია **‘y’** -ის;
- m3 ეკვივალენტურია &m3[0]-ის;
- m3[1] ეკვივალენტურია **‘y’** -ის.

სიმბოლოების მასივის გამოცხადებისას და მისი სტრიქონული კონსტანტის ინიციალების დროს, შესაძლებელია მასივის ზომის ცხადი სახით მითითება, მაგრამ მასივის მითითებული ზომა უნდა აღემატებოდეს ინიციალებული სტრიქონული კონსტანტის ზომას. **char m2[80]=“My name is Lela.”.**

სიმბოლოების მასივი:

```
char A[4] = { 'A', 'B', '[', 'M'};
char B[10];
```

მასივისთვის:

- ყოველი სიმბოლო ცალკეული ობიექტია;
- მასივის სიგრძე N, რომელიც მოცემულია მისი დეკლარირებისას

რა არის საჭირო:

- დამუშავდეს სიმბოლოების მიმდევრობა, როგორც ერთი მთლიანი;
- სტრიქონს უნდა გააჩნდეს ცვლადი სიგრძე.

სურ. 64

4.1.4. სტრიქონების შეტანა/გამოტანის ფუნქციები

სტრიქონის შეტანის მიზნით შეგვიძლია scanf() ფუნქციის გამოყენება, მაგრამ ამ უკანასკნელის დანიშნულებაა სიტყვის მიღება და არა სტრიქონის.

თუ “**%s**” შეტანის ფორმატს გამოვიყენებთ, სტრიქონი მომდევნო ცარიელ სიმბოლომდე შეიყვანება, რომელიც შეიძლება იყოს ინტერვალი (space), ტაბულაცია ან ახალ სტრიქონზე გადასვლა.

ინტერვალების (space) ჩათვლით სტრიქონის შესატანად გამოიყენება ფუნქცია:

char * gets(char *); ან მისი ეკვივალენტი: **char * gets_s(char *);**

სტრიქონის კონსოლზე გამოტანის მიზნით შეგვიძლია printf() ფუნქციის გამოყენება:

printf("%s", str); // str — მიმთითებელია სტრიქონზე.

ან შემოკლებული ფორმა:

printf(str);

სტრიქონის კონსოლზე გამოსატანად, ასევე შეგვიძლია მივმართოთ ფუნქციას:

int puts (char *s); რომელიც ბეჭდავს სტრიქონს და კურსორი მომდევნო სტრი-ქონზე გადააქვს (განსხვავებით printf() ფუნქციისგან). **puts()** ფუნქცია ასევე გამოი-ყენება აპოსტროფებში მოთავსებული სტრიქონული კონსტანტის კონსოლზე გა-მოსატანად.

სიმბოლოების შესატანად გამოიყენება ფუნქცია:

char getchar();

რომელიც აბრუნებს კლავიატურიდან შეტანილი სიმბოლოს მნიშვნელობას.

სიმბოლოების კონსოლზე გამოსატანად მივმართავთ ფუნქციას:

char putchar(char);

რომელიც გვაძლევს კონსოლზე გამოტანილი სიმბოლოს მნიშვნელობას და ეკრან-ზე გამოაქვს სიმბოლო, რომელიც მას არგუმენტის სახით გადაეცა.

სტრიქონებთან სამუშაოდ პროგრამაში საჭიროა **<string.h>** ბიბლიოთეკის ჩართ-ვა.

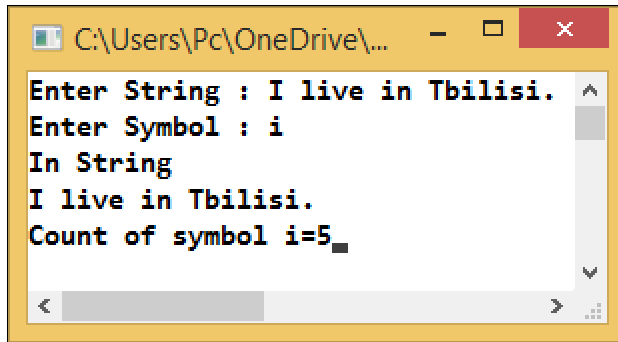
ამოცანა 25.

შევადგინოთ პროგრამა, რომელიც სტრიქონში საჭირო სიმბოლოს რაოდენობის დათვლის.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
int main() {
  char s[80], sym;
  int count, i;
  printf("Enter String : ");
  gets(s);
  printf("Enter Symbol : ");
```

```
 sym = getchar();
 count = 0;
 for (i = 0; s[i] != '\0'; i++)
 {
   if (s[i] == sym)
     count++;
 }
 printf("In String\n");
 puts(s);
 printf("Count of symbol %c=%d", sym, count);
 getchar(); getchar();
 return 0;
}
```

პროგრამული კოდის შესრულების შედეგები 65-ე სურათზეა ნაჩვენები.



სურ. 65

4.1.5. string.h ბიბლიოთეკის ძირითადი ფუნქციები

<string.h> ბიბლიოთეკის ძირითადი ფუნქციები მე-15 ცხრილშია წარმოდგენილი.

ცხრილი 15. string.h ბიბლიოთეკის ძირითადი ფუნქციები

ფუნქცია	აღწერა
char *strcat(char *s1, char *s2)	s1-თან აერთიანებს s2-ს და აბრუნებს s1-ს.
char *strncat(char *s1, char *s2, int n)	s1-თან აერთიანებს s2-ის არაუმეტეს n სიმბოლოს (სტრიქონს ასრულებს სიმბოლოთი '\0') და აბრუ-

	ნებს s1-ს.
char *strcpy(char *s1, char *s2)	s2 სტრიქონს აკოპირებს s1-ში '\0' სიმბოლოს ჩათვლით და აბრუნებს s1-ს.
char *strncpy(char *s1, char *s2, int n)	s1-ში აკოპირებს s2-ის არაუმეტეს n სიმბოლოს და აბრუნებს s1-ს.
int strcmp(char *s1, char *s2)	s1და s2 სტრიქონებს ერთმანეთს ადარებს და აბრუნებს 0-ს, თუ სტრიქონები ევივალენტურია.
int strncmp(char *s1, char *s2, int n)	ერთმანეთს ადარებს s1და s2 სტრი-ქონების არაუმეტეს n სიმბოლოს და აბრუნებს 0-ს, თუ სტრიქონების საწყისი n სიმბოლოები ევივა-ლენტურია.
int strlen(char *s)	აბრუნებს s სტრიქონში სიმბო-ლოების რაოდენობას.
char *strset(char *s, char c)	ავსებს სტრიქონს s სიმბოლოებით, რომელთა კოდი c-ს ტოლია და აბრუნებს მიმთითებელს s სტრი-ქონზე.
char *strnset(char *s, char c, int n)	ცვლის s სტრიქონის პირველ n სიმბოლოებს სიმბოლოებით, რო-მელთა კოდი c-ს ტოლია და აბრუ-ნებს მიმთითებელს s სტრიქონზე.

სტრიქონის სიგრძის განსაზღვრა სქემატურად 66-ე სურათზეა წარმოდგენილი, ხოლო სტრიქონების კოპირების სქემა - 67-ე სურათზე.

ბიბლიოთეკის ჩართვა:

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

სტრიქონის სიგრძე: ***strlen*** (*string length*)

```
char q[80] = "qwerty";
int n;
n = strlen ( q );
```

n = 6

! სიგრძის განსაზღვრისას სიმბოლო '\0' არ განიხილება!

სურ. 66

strcpy (***string copy***)

```
char q1[10]="qwerty", q2[10]="01234";

strcpy ( q1, q2 );
```

სად

საიდან

! q1-ის ძველი მნიშვნელობა იშლება!

სტრიქონის «კუდის» კოპირება

```
char q1[10]="qwerty", q2[10]="01234";
strcpy ( q1, q2+2 );
```

q2 = &q2[0]

q2+2 = &q2[2]

q1	2	3	4	\0	t	y	\0	¤	¤	¤

q2	0	1	2	3	4	\0	¤	¤	¤	¤

სურ. 67

68-ე სურათზე წარმოდგენილია strncpy ფუნქციის მუშაობის სქემა, ხოლო 69-ე სურათზე - სტრიქონების გაერთიანების სქემა.

strncpy – n რაოდენობის სიმბოლოს კოპირება

```
char q1[10]="qwerty", q2[10]="01234";
strncpy ( q1+2, q2, 2 );
```

q1+2 = &q1[2]

q1	q	w	0	1	t	y	\0	¤	¤	¤

q2	0	1	2	3	4	\0	¤	¤	¤	¤

! ფუნქცია strncpy არ ამატებს '\0' სიმბოლოს სტრიქონის ბოლოს!

სურ. 68

strcat (string concatenation) = მეორე სტრიქონის პირველის ბოლოში კოპირება

```
char q1[10]="qwe", q2[10]="0123";
strcat ( q1, q2 );
```

q1	q	w	e	0	1	2	3	\0	¤	¤

q2	0	1	2	3	\0	¤	¤	¤	¤	¤

```
char q1[10]="qwe", q2[10]="0123";
strcat ( q1, q2+2 );
```

q1	q	w	e	2	3	\0	¤	¤	¤	¤

q2	0	1	2	3	\0	¤	¤	¤	¤	¤

სურ. 69

ქვემოთ ნაჩვენებია string.h ბიბლიოთეკის ძირითადი ფუნქციების გამოყენების ამსახველი პროგრამა.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  char m1[80] = "The first string";
  char m2[80] = "The second String";
  char m3[80];
  strncpy(m3, m1, 6);
  puts("Result of strncpy(m3, m1, 6)");
  puts(m3);
  strcpy(m3, m1);
  puts("\nResult of strcpy(m3, m1)");
  puts(m3);
  puts("\nResult of strcmp(m3, m1) ");
  printf("%d", strcmp(m3, m1));
  strncat(m3, m2, 5);
  puts("\nResult of strncat(m3, m2, 5)");
  puts(m3);
  strcat(m3, m2);
  puts("\nResult of strcat(m3, m2)");
  puts(m3);
  puts("\nCount of symbols in m1 String  strlen(m1) : ");
  printf("%d\n", strlen(m1));
  _strnset(m3, 'f', 7);
  puts("\nResult of strnset(m3, 'f', 7)");
  puts(m3);
  _strset(m3, 'k');
```

```
  puts("\nResult of strnset(m3, 'k')");
  puts(m3);
  getchar();
  return 0;
}
```

პროგრამული კოდის შესრულების შედეგები 70-ე სურათზეა ნაჩვენები.

```
C:\Users\Pc\OneDrive\Documents\...
Result of strncpy(m3, m1, 6)
The fi

Result of strcpy(m3, m1)
The first string

Result of strcmp(m3, m1)
0
Result of strncat(m3, m2, 5)
The first stringThe s

Result of strcat(m3, m2)
The first stringThe sThe second String

Count of symbols in m1 String  strlen(m1) :
16

Result of strnset(m3, 'f', 7)
fffffffst stringThe sThe second String

Result of strnset(m3, 'k')
kkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkkk
```

სურ. 70

V თავი
მომხმარებლის მიერ შექმნილი ფუნქციები

ფუნქცია წარმოადგენს ბრძანებათა ჯგუფს, რომელსაც აქვს კონკრეტული სახელი, ანუ იდენტიფიკატორი, ხოლო ამ ჯგუფში მითითებული ბრძანებების შესრულება მხოლოდ ფუნქციის გამოძახების შედეგად არის შესაძლებელი.

ფუნქცია პროგრამის დამოუკიდებელი ერთეულია, რომელიც კონკრეტული ამოცანის რეალიზებას ემსახურება. ის ერთგვარი ქვეპროგრამაა და პროგრამაში განსაზღვრულ მოქმედებებს ასრულებს.

ფუნქციის სიგნატურა მისი გამოყენების წესებს განსაზღვრავს. ტრადიციულად, სიგნატურა ფუნქციის აღწერას წარმოადგენს, რომელიც მოიცავს: ფუნქციის სახელს, ფორმალური პარამეტრებისა და მათი ტიპების ჩამონათვალს და დასაბრუნებელი შედეგის ტიპს (ასეთის არსებობის შემთხვევაში) ან დარეზერვებულ სიტყვას: **void**.

ფუნქციის სემანტიკა მისი რეალიზების საშუალებას განსაზღვრავს და როგორც წესი, ის ფუნქციის ტანს წარმოადგენს.

C ენაში განასხვავებენ მომხმარებლის მიერ შექმნილი ფუნქციების ორ ძირითად სახეს:

- ფუნქციები, რომლებიც ასრულებენ რა გამოთვლით ოპერაციებს, მიღებულ შედეგებს პროგრამის ძირითად ნაწილში (როგორც წესი, main() ფუნქციაში) აბრუნებენ. ასეთი ფუნქციები აუცილებლად უნდა შეიცავდნენ ერთს მაინც **return** ოპერატორს, რომელიც გამოთვლილ შედეგს ფუნქციიდან დააბრუნებს.

- ფუნქციები, რომლებიც ასრულებენ რა სხვადასხვა სახის მანიპულაციებს, მიღებულ შედეგებს არ აბრუნებენ და საჭიროების შემთხვევაში თავად შეუძლიათ მათი დაბეჭდვა. ეს **void** სპეციფიკაციის მქონე ფუნქციებია. სიტყვა **void** ცარიელს, დაუკავებელს ნიშნავს.

ზოგად შემთხვევაში ფუნქციის განსაზღვრას (**function definition**) შემდეგი სახე აქვს:

ფუნქციის სათაური
{
ფუნქციის ტანი;
}

ფუნქციის **სათაური** (**function header**) მოიცავს:

- დასაბრუნებელი შედეგის ტიპს. ხოლო ფუნქციებისთვის, რომლებიც მი-ღებულ შედეგებს არ აბრუნებენ, ამ შემთხვევაში მიეთითება ტიპი **void**;
- ფუნქციის სახელს (იდენტიფიკატორს);
- ერთმანეთისგან მძიმეებით გამოყოფილ პარამეტრებისა და მათი ტიპების სიას, რომელიც მრგვალ ფრჩხილებში თავსდება. ამასთან, ყოველი პარამეტ-რის წინ მისი ტიპი იწერება (უპარამეტრებო ფუნქციის შემთხვევაში მრგვალ ფრჩხილებში არსებული სივრცე ცარიელი რჩება).

ფუნქციის ტანი (**function body**) მოიცავს კოდს, რომელიც ფუნქციის გამოძა-ხების დროს სრულდება. ეს არის ერთი ან რამდენიმე ბრძანების შემცველი პროგ-რამული კოდის ფრაგმენტი, რომლის საზღვრები ღია და დახურულ ფიგურულ ფრჩხილებს შორის თავსდება. იმისათვის, რომ ფუნქციაში აღწერილი ბრძანებები (მოქმედებები) შესრულდეს, საჭიროა ფუნქციის გამოძახება (**function call**), რომლის ჩაწერის სინტაქსი შემდეგია:

ფუნქციის სახელი (არგუმენტების სია);

ყურადსაღებია ის ფაქტი, რომ ფუნქციის განსაზღვრა ნებისმიერ სხვა ფუნქციაში დაუშვებელია.

მომხმარებლის მიერ შექმნილი ნებისმიერი ფუნქცია ჯერ უნდა აღვწეროთ და შემდეგ მოვახდინოთ მისი გამოძახება. ამ წესის დარღვევა მხოლოდ ფუნქციის პროტოტიპის გამოყენების შემთხვევაშია შესაძლებელი.

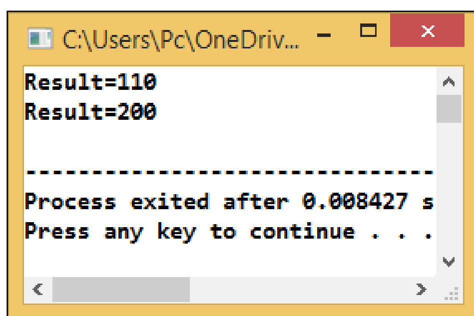
დადგა დრო, მოვახდინოთ მომხმარებლის მიერ შექმნილი ფუნქციების გამო-ყენების დემონსტრირება.

შევადგინოთ ორი მთელრიცხვა ცვლადის შეკრების პროგრამა.

```
//return ოპერტორის შემცველი ფუნქცია
#include <stdio.h>
int sum(int x, int y){
	return x+y;
}
```

```
int main() {
        int res1, res2;
        res1=sum(50, 60);
        res2=sum(120, 80);
        printf("Result-1=%d\n", res1);
        printf("Result-2=%d", res2);
        return 0;
}
//void სპეციფიკაციის ფუნქცია
#include <stdio.h>
void sum(int x, int y){
        printf("Result=%d\n",x+y);
}
int main() {
        sum(50, 60);
        sum(120, 80);
        return 0;
}
```

ცხადია, ორივე პროგრამული კოდის შესრულების შემთხვევაში 71-ე სურათზე ნაჩვენები ერთი და იგივე შედეგები მიიღება.



სურ. 71

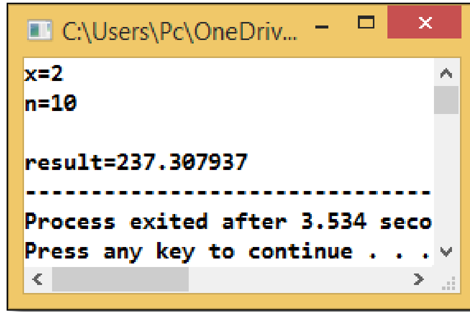
ამოცანა 26.

მომხმარებლი მიერ შექმნილი ფუნქციის გამოყენებით შევადგინოთ შემდეგი მწკრივის $s = 1 + x + \frac{x^2}{2} + \frac{x^3}{3} + \cdots \frac{x^n}{n}$ წევრების ჯამის გამოთვლის პროგრამა.

```
//void სპეციფიკაციის ფუნქცია
#include <stdio.h>
void function1(double x, int n) // function header
{
double p=1, s=0;
int i;
for(i=1; i<=n; i++)
{
```

```
p*=x;   // function body
s+=p/i;
}
printf("\nresult=%lf", s);
}
int main(){
double x;
int n;
printf("x=");
scanf("%lf", &x);
printf("n=");
scanf("%d", &n);
function1(x,n);  // function call
return 0;
}

//return ოპერტორის შემცველი ფუნქცია
#include <stdio.h>
double function1(double x, int n) // function header
{
double p=1, s=0;
int i;
for(i=1; i<=n; i++)
{
p*=x;   // function body
s+=p/i;
}
return s;
}
int main(){
double x, res;
int n;
printf("x=");
scanf("%lf", &x);
printf("n=");
scanf("%d", &n);
res=function1(x,n);  // function call
printf("\nresult=%lf", res);
return 0;
}
```

აქაც ცხადია, ერთი და იმავე საწყისი მონაცემების დროს, ორივე პროგრამული კოდის შესრულების შემთხვევაში 72-ე სურათზე ნაჩვენები ერთი და იგივე შედეგი მიიღება.



სურ. 72

5.1. ფუნქციის პროტოტიპი

ფუნქციის პროტოტიპის განსაზღვრა, გარკვეულწილად, ახლოსაა ფუნქციის სათაურის განსაზღვრასთან, მაგრამ მისგან იმით განსხვავდება, რომ არ აქვს ტანი და არ საჭიროებს ფუნქციის პარამეტრების იდენტიფიკატორების ჩაწერას, რამეთუ, კომპილატორი მათ იგნორირებას ახდენს. პროტოტიპის განსაზღვრა (განსხვავებით ფუნქციის სათაურისგან) აუცილებლად წერტილ-მძიმით სრულდება. ფუნქციის პარამეტრების ტიპები ერთმანეთისგან მძიმეებით გამოიყოფა და ფუნქციის სახელის შემდეგ მრგვალ ფრჩხილებში თავსდება.

ამგვარად, ფუნქციის პროტოტიპის განსაზღვრის ზოგადი სახე შემდეგია:

შედეგის ტიპი (ან void) ფუნქციის სახელი (პარამეტრების ტიპები);

დოკუმენტირების თვალსაზრისით ფუნქციის პროტოტიპში შესაძლებელია გამოყენებულ იქნას პარამეტრების იდენტიფიკატორები, თუმცა, კომპილატორისთვის ამას არანაირი მნიშვნელობა არ აქვს. ფუნქციის პროტოტიპის დასასრულს წერტილ-მძიმის არარსებობა სინტაქსური ხასიათის შეცდომად ითვლება.

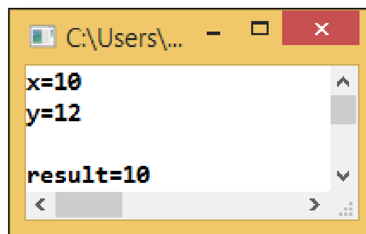
ფუნქციის პროტოტიპი მნიშვნელოვანია იმ თვალსაზრისით, რომ ის კომპილატორს საშუალებას აძლევს შეამოწმოს ფუნქციაში გადაცემული ფორმალური და ფაქტიური პარამეტრების ტიპების შესაბამისობა.

მოვახდინოთ ფუნქციის პროტოტოტიპის გამოყენების დემონსტრირება ორ მთელ რიცხვს შორის მინიმალურის განსაზღვრისთვის. პროგრამული კოდი წარმოვადგინოთ ორ ვარიანტად: ჯერ return ოპერატორის შემცველი ფუნქციით და შემდეგ void სპეციფიკაციის ფუნქციით.

```
//return ოპერატორის შემცველი ფუნქცია
#include <stdio.h>
```

```
int mini(int, int);  //prototipe
int main(){
int x, y, res;
printf("x=");
scanf("%d", &x);
printf("y=");
scanf("%d", &y);
res=mini(x,y);   // function call
printf("\nresult=%d", res);
return 0; }
int mini(int a, int b){
        return a<b? a : b;}

// void სპეციფიკაციის ფუნქცია
#include <stdio.h>
void mini(int, int); //prototipe
int main(){
int x, y;
printf("x=");
scanf("%d", &x);
printf("y=");
scanf("%d", &y);
mini(x,y);   // function call
return 0; }
void mini(int a, int b){
        int res = a<b? a : b;
        printf("\nresult=%d", res);}
```

აქაც ცხადია, ერთი და იმავე საწყისი მონაცემების დროს, ორივე პროგრამული კოდის შესრულების შემთხვევაში 73-ე სურათზე ნაჩვენები ერთი და იგივე შედეგი მიიღება.



სურ. 73

5.2. ფუნქციის ფაქტიური და ფორმალური პარამეტრები

ფუნქციის **ფაქტიური არგუმენტი** არის სიდიდე, რომელიც ფუნქციის გამოძახების დროს ენიჭება ფორმალურ არგუმენტს.

ამგვარად, **ფორმალური არგუმენტი** ცვლადია გამომძახებელ ფუნქციაში, ხოლო **ფაქტიური არგუმენტი** კონკრეტული მნიშვნელობაა, რომელიც ამ ცვლადს ფუნქციის გამოძახების დროს ენიჭება.

ფაქტიური არგუმენტი შეიძლება იყოს მუდმივი სიდიდე, ცვლადი ან გამოსა-ხულება. თუ ფაქტიური არგუმენტი გამოსახულების სახითაა წარმოდგენილი, მაშინ მისი მნიშვნელობა ჯერ გამოითვლება და შემდეგ გადაეცემა ფუნქციას.

პროგრამის ფუნქციებად დაყოფა შემდეგ უპირატესობებს იძლევა:

- ფუნქცია შეიძლება გამოვიძახოთ პროგრამის ნებისმიერი ადგილიდან, რაც პროგრამული კოდის გამეორებას თავიდან აგვაცილებს;
- ერთი და იგივე ფუნქცია შეიძლება სხვადასხვა პროგრამაში გამოვიყენოთ;
- ფუნქციები მოდულური პროგრამების დაპროექტებას უწყობს ხელს;
- ფუნქციის გამოყენებით პროგრამის წაკითხვა და გაგება უფრო მარტივია და ის ასევე, აჩქარებს პროგრამის გამართვის (შეცდომების გასწორების) პრო-ცესს.

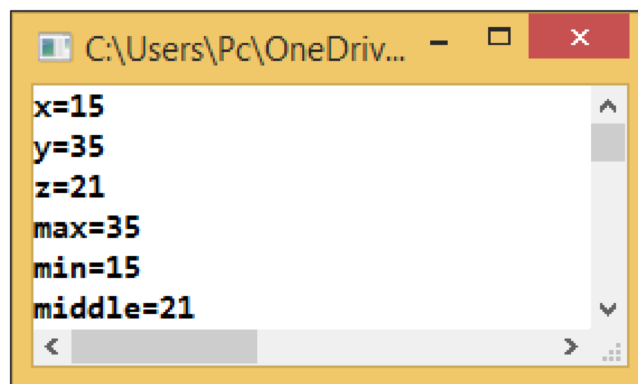
ამოცანა 27.

მოცემულია სამი *x*, *y* და *z* მთელი ტიპის ცვლადი. განვსაზღვროთ მათ შორის უდიდესი, უმცირესი და შუალედური მნიშვნელობის მქონე ცვლადები (გამოვიყე-ნოთ return ოპერატორის შემცველი ფუნქციები).

```
#include <stdio.h>
int maxi(int, int, int);  //prototipe
int mini(int, int, int);  //prototipe
int mid(int, int, int);  //prototipe
int main(){
int x, y, z;
printf("x=");
scanf("%d", &x);
printf("y=");
scanf("%d", &y);
printf("z=");
scanf("%d", &z);
printf("max=%d\n",maxi(x,y,z));   // function call
printf("min=%d\n",mini(x,y,z));   // function call
printf("middle=%d\n",mid(x,y,z));   // function call
return 0; }
```

```
int maxi(int a, int b, int c){
        int max=a;
        if(max<b)
        max=b;
        if(max<c)
        max=c;
        return max;
        }

int mini(int a, int b, int c){
        int min=a;
        if(min>b)
        min=b;
        if(min>c)
        min=c;
        return min;
        }
int mid(int a, int b, int c){
int middle;
if(a>b && a<c || a<b && a>c)
middle=a;
else if(b>a && b<c || b<a && b>c)
middle=b;
else
middle=c;
return middle;
}
```

პროგრამის შესრულების შედეგები 74-ე სურათზეა ნაჩვენები.



სურ. 74

5.3. ერთგანზომილებიანი მასივების გადაცემა ფუნქციებში

ხშირ შემთხვევაში შესაძლებელია და საჭიროც არის, ამა თუ იმ ფუნქციაში არგუმენტის სახით გადავცეთ მასივი. ასეთ დროს სასურველია (მაგრამ არა სავალდებულო), მასივის ზომაც ერთ-ერთი არგუმენტის სახით წარმოვადგინოთ ფუნქციაში, რაც ამ უკანასკნელს საშუალებას მისცემს, მასივში არსებული ელემენტების რაოდენობა წინასწარ განსაზღვროს. ცხადია, მასივის ზომა ფუნქციის პროტოტიპის ან მის აღწერამდეც შეიძლება განისაზღვროს როგორც გლობალური სახის მთელი ტიპის მუდმივი სიდიდე (რადგან სტატიკური მასივის ელემენტების რაოდენობა, ანუ მისი ზომა უცვლელია).

მასივის სახელი სხვა არაფერია, თუ არა მისი რიგით პირველი (ნულოვანი) ელემენტის მისამართი კომპიუტერის მეხსიერებაში. რადგან ფუნქციას მასივის საწყისი მისამართი გადაეცემა, ამიტომ მისთვის ცნობილია, თუ სად ინახება მასივი. აქედან გამომდინარე ცხადია, რომ როდესაც ფუნქცია თავის ტანში ახდენს მასივის ელემენტების მოდიფიცირებას, ამით იგი მასივის რეალურ ელემენტებს ცვლის.

განვიხილოთ იმ **ფუნქციის პროტოტიპი**, რომელშიც პარამეტრების სახით ერთგანზომილებიანი მასივი და მისი ზომაა გადაცემული:

void modifyArray(int [ ], const int); **//*prototipe***

აღნიშნული პროტოტიპის მიხედვით თუ ვიმსჯელებთ, საქმე გვაქვს ფუნქციასთან, რომლის სახელია **modifyArray**. ამასთან, ფუნქციას პირველი პარამეტრის სახით მთელრიცხვა მასივი გააჩნია, ხოლო მეორე პარამეტრის სახით – მისი ზომა (მთელი ტიპის მუდმივი სისდიდე).

არცერთი პარამეტრის სახელი პროტოტიპში მითითებული არ არის, რადგან კომპილატორი მაინც მოახდენდა მათ იგნორირებას, თუმცა დოკუმენტირების თვალსაზრისით შესაძლებელი იყო მათი სახელების ჩაწერა.

ზემოთ წარმოდგენილი პროტოტიპის შესაბამის **ფუნქციის განსაზღვრას** შემდეგი სახე აქვს:

void modifyArray(int A [ ], const int size)

{ function body; }

ამ შემთხვევაში, ფუნქციაში მითითებულია როგორც ერთგანზომილებიანი მთელრიცხვა მასივის იდენტიფიკატორი (**A**), ისე მისი ზომის დასახელებაც (**size**) და ბუნებრივია, აქ განიხილება ფუნქციის ტანიც (**function body**).

იმისათვის, რომ მოვახდინოთ წარმოდგენილი ფუნქციის გამოძახება, პირველ რიგში, საჭიროა მივუთითოთ ფუნქციის სახელი, ხოლო შემდეგ - მრგვალ ფრჩხილებში მოთავსებული და ერთმანეთისგან მძიმით გამოყოფილი არგუმენტების იდენტიფიკატორები. მაშასადამე, მოცემული **ფუნქციის გამოძახების** შემთხვევაში გვექნება შემდეგი ჩანაწერი:

modifyArray(array, size);

ფუნქციაში პირველი არგუმენტის სახით განიხილება ის რეალური **array** მასივი, რომელიც ფუნქციის აღწერისას გამოყენებულ **A** მასივს (პარამეტრს) შეესაბამება, ხოლო მეორე არგუმენტი (**size**) წარმოადგენს მოცემული მასივის ზომას. აღსანიშნავია, რომ მასივი იდენტიფიკატორით **array** იმავე ტიპის ელემენტებისგან უნდა შედგებოდეს, რომლებისგანაც **A** მასივი შედგება და იგი ფუნქციის გამოძახებამდე აუცილებლად უნდა იქნას განსაზღვრული.

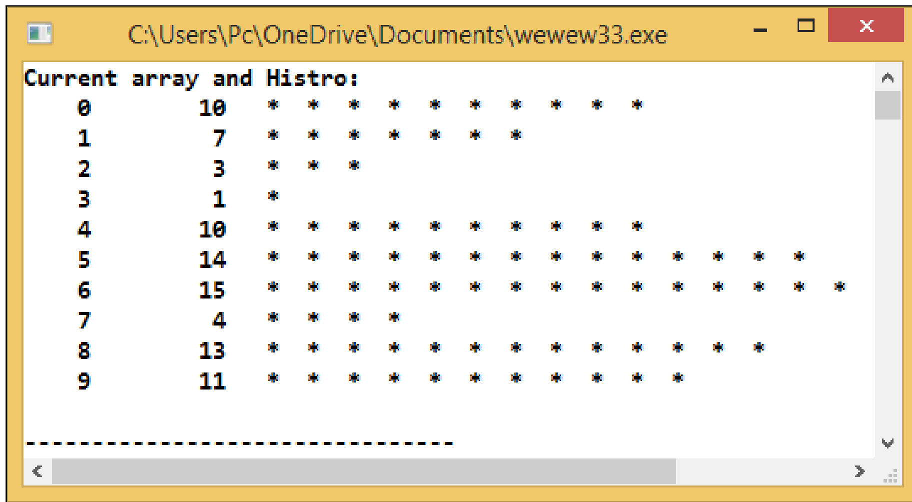
ამოცანა 28.

სამომხმარებლო ფუნქციაში პარამეტრის სახით გადაცემული ერთგანზომილებიანი მასივის გამოყენებით შევადგინოთ პროგრამა, რომელიც მასივის ელემენტებს [1;15] ინეტრვალიდან მიანიჭებს თანაბარი ალბათობით განაწილებულ კვაზიშემთხვევით მთელრიცხვა მნიშვნელობებს და კონსოლზე წარმოადგენს მასივის ელემენტებს ჰისტროგრამასთან ერთად.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void function(int [], const int);   //prototipe
int main() {
        const int size=10;
        int A[size];
        printf("Current array and Histro:\n");
        function(A, size);
        return 0;
}
void function(int B[], const int n){
```

```
        srand(time(0));
        for(int i=0; i<n; i++){
                B[i]=1+rand()%15;
                printf("%5d%10d  ",i, B[i]);
                for(int j=1; j<=B[i]; j++)
                        printf(" * ");
                        printf("\n");
        }
}
```

პროგრამული კოდის შესრულების შედეგები 75-ე სურათზეა ნაჩვენები.



სურ. 75

5.4. ორგანზომილებიანი მასივების გადაცემა ფუნქციებში

ცნობილია, რომ, თუ ფუნქციაში პარამეტრის სახით ერთგანზომილებიანი მასივია გადაცემული, მაშინ ფუნქციის აღწერის, ისევე როგორც ფუნქციის პროტოტიპის განსაზღვრისას პარამეტრების სიაში არსებული კვადრატული ფრჩხილები ცარიელი რჩება და მასივის ზომა ცალკე პარამეტრის სახით განიხილება. ანალოგიურად, როდესაც ფუნქციის ერთ-ერთ პარამეტრს მრავალგანზომილებიანი მასივი წარმოადგენს, მისი პირველი ინდექსის (სტრიქონის) მითითება საჭირო არ არის, ხოლო სხვა დანარჩენი ინდექსების ჩაწერა აუცილებელია. ინდექსების განზომილებას კომპილატორი იყენებს კომპიუტერის მეხსიერების იმ უჯრედების განსაზღვრისათვის, რომლებშიც თავსდება მრავალგანზომილებიანი მასივის ელემენტები.

როგორც წესი, თუ ორგანზომილებიანი მასივი (მატრიცა) ფუნქციის ერთ-ერთ პარამეტრად განიხილება, მაშინ მისი სვეტებისა და სტრიქონების რაოდენობის აღმნიშვნელი მონაცემები განისაზღვრება, როგორც მთელი ტიპის გლობალური სა-ხის მუდმივი სიდიდეები.

წარმოვადგინოთ იმ **ფუნქციის პროტოტიპი**, რომელშიც ერთ-ერთი პარამეტრის სახით ორგანზომილებიანი მასივია (მატრიცაა) გადაცემული:

void function(int [ ][columns]); //*prototipe*

ზემოთ განსაზღვრული პროტოტიპის შესაბამის **ფუნქციის აღწერას** შემდეგი სახე აქვს:

void function(int A [ ][columns])

{

function body;

}

მოცემული **ფუნქციის გამოძახების** შემთხვევაში შემდეგი ჩანაწერი გვექნება:

function(array);

ამოცანა 29.

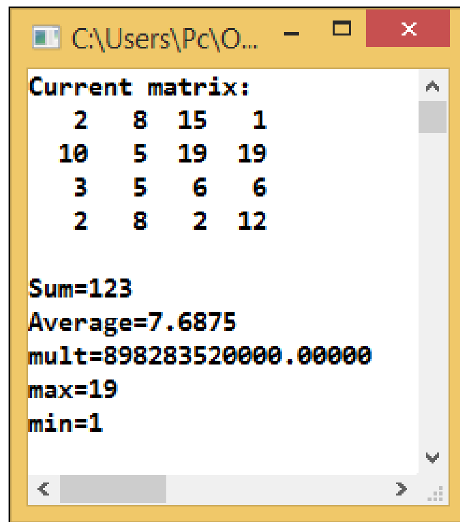
სამომხმარებლო ფუნქციაში პარამეტრის სახით გადაცემული ორგანზომილე-ბიანი მასივის გამოყენებით შევადგინოთ პროგრამა, რომელიც მასივის ელემენ-ტებს [1;20] ინეტრვალიდან მიანიჭებს თანაბარი ალბათობით განაწილებულ კვაზიშემთხვევით მთელრიცხვა მნიშვნელობებს, გამოთვლის მასივის ელემენ-ტების ჯამს, საშუალო არითმეტიკულ მნიშვნელობას, ნამრავლს და განსაზღვრავს მასივში მაქსიმალურ და მინიმალურ მნიშვნელობებს.

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

const int rows=4;
const int columns=4;
void input(int [][columns]);
void print(int [][columns]);
void sum_average(int [ ][columns]);
void mult(int [][columns]);
void max_min(int [][columns]);
```

```
int main() {
int A[rows][columns];
input(A);
printf("Current matrix:\n");
print(A);
sum_average(A);
mult(A);
max_min(A);
return 0;
}
void input(int B[][columns])
{
int i, j;
for(i=0; i<rows; i++)
for(j=0; j<columns; j++)
 B[i][j]=1+rand()%20;
}
void print(int B[][columns])
{
 int i, j;
for(i=0; i<rows; i++){
for(j=0; j<columns; j++)
printf("%4d",B[i][j]);
printf("\n");
}  }
void sum_average(int B[][columns])
{
double s=0;
int i, j;
for(i=0; i<rows; i++)
for(j=0; j<columns; j++)
s+=B[i][j];
printf("\nSum=%.0lf\n", s);
printf("Average=%.4lf\n",s/(rows*columns)); }
void mult(int B[][columns])
{
double m=1;
int i, j;
for(i=0; i<rows; i++)
for(j=0; j<columns; j++)
m*=B[i][j];
printf("mult=%.5lf\n",m);
}
```

```
void max_min(int B[][columns])
{
        int i, j;
int max=B[0][0], min=B[0][0];
for(i=0; i<rows; i++)
for(j=0; j<columns; j++)
{
if(max<=B[i][j])
max=B[i][j];
if(min>=B[i][j])
min=B[i][j];
}
printf("max=%d\n", max);
printf("min=%d\n", min);
}
```

პროგრამული კოდის შესრულების შედეგები 76-ე სურათზეა ნაჩვენები.



სურ. 76

ამოცანა 30.

სამომხმარებლო ფუნქციაში პარამეტრის სახით გადაცემული ორგანზომილებიანი მასივის გამოყენებით შევადგინოთ პროგრამა, რომელიც მასივის ელემენტებს [10;35] ინეტრვალიდან მიანიჭებს თანაბარი ალბათობით განაწილებულ კვაზიშემთხვევით მთელრიცხვა მნიშვნელობებს, დაახარისხებს მასივის ელემენტებს ზრდადობით ჯერ სტრიქონების და შემდეგ სვეტების მიხედვით და ამოცანის ყველა ეტაპის შედეგს წარმოგვიდგენს კონსოლზე.

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

```
#include <time.h>
const int rows=4;
const int columns=4;
void input(int [][columns]);
void print(int [][columns]);
void sort1(int [ ][columns]);
void sort2(int [ ][columns]);

int main() {
int A[rows][columns];
input(A);
printf("Current matrix:\n");
print(A);
printf("\nSorted matrix by rows:\n");
sort1(A);
print(A);
printf("\nSorted matrix by columns:\n");
sort2(A);
print(A);
return 0;  }
void input(int B[][columns])
{
int i, j;
for(i=0; i<rows; i++)
for(j=0; j<columns; j++)
 B[i][j]=10+rand()%26;  }
 void print(int B[][columns]) {
 int i, j;
for(i=0; i<rows; i++){
for(j=0; j<columns; j++)
printf("%4d",B[i][j]);
printf("\n"); } }
void sort1(int B[][columns])
{
        int i, j, m, t;
        for(m=1; m<(rows*columns); m++)
        for(i=0; i<rows; i++)
         for(j=0; j<columns-1; j++)
          if(B[i][j]>B[i][j+1]){
                t=B[i][j];
                B[i][j]=B[i][j+1];
                B[i][j+1]=t; }  }
void sort2(int B[][columns]) {
        int i, j, m, t;
```

```
for(m=1; m<(rows*columns); m++)
for(j=0; j<columns; j++)
 for(i=0; i<rows-1; i++)
  if(B[i][j]>B[i+1][j]){
          t=B[i][j];
          B[i][j]=B[i+1][j];
          B[i+1][j]=t; } }
```

პროგრამული კოდის შესრულების შედეგები 77-ე სურათზეა ნაჩვენები.

```
C:\Users\Pc\OneDriv...
Current matrix:
  25  17  26  16
  17  30  22  14
  10  34  21  23
  21  15  13  33

Sorted matrix by rows:
  16  17  25  26
  14  17  22  30
  10  21  23  34
  13  15  21  33

Sorted matrix by columns:
  10  15  21  26
  13  17  22  30
  14  17  23  33
  16  21  25  34
```

სურ. 77

VI თავი
მასივების დახარისხებისა და ძებნის ალგორითმები და შესაბამისი პროგრამული რეალიზაციები

მონაცემთა დახარისხება განსაზღვრული წესით ზრდადობის ან კლებადობის მიხედვით თავისი არსით კომბინატორული ამოცანების კლასს მიეკუთვნება. სპე-ციალისტთა აზრით, კომპიუტერული დროის დაახლოებით 25% სისტემატიურად დახარისხების ამოცანებზე იხარჯება. ამიტომ, აღნიშნული ალგორითმები გან-საკუთრებულ ყურადღებას იმსახურებენ.

როგორც წესი, ყოველი ორგანიზაცია ამათუიმ მონაცემთა დახარისხებას ახ-დენს, ხოლო ხშირ შემთხვევებში, საჭირო ხდება მონაცემთა მნიშვნელოვანი მოცუ-ლობის დახარისხება. გამომდინარე ზემოთ თქმულიდან, წარმოგიდგენთ ერთგანზომილებიან მასივებში მონაცემთა დახარისხების რამდენიმე ალგორითმს.

6.1.1. ბუშტისებრი დახარისხება

განვიხილოთ ათი ელემენტისგან შემდგარი a მასივი და მისი ელემენტები დავალაგოთ ზრდადობის მიხედვით ბუშტისებრი დახარისხების ალგორითმის გამოყენებით. დახარისხების აღნიშნულ ალგორითმს "ჩაძირვის" ალგორითმსაც უწოდებენ, რადგან მასივის უმცირესი მნიშვნელობა, მსგავსად წყალში ჰაერის ბუშ-ტისა, სულ ზევით (მასივის დასაწყისისკენ) მიიწევს, ხოლო უდიდესი მნიშვნე-ლობა სულ უფრო უახლოვდება წყლის ფსკერს (მასივის ბოლოს) ანუ აღინიშნება მისი ერთგვარი "ჩაძირვა".

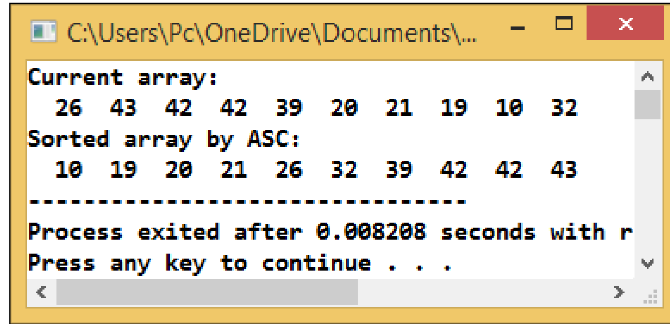
აღნიშნული ალგორითმი მასივის რამდენიმე დათვალიერებას საჭიროებს. ყო-ველ ეტაპზე დარდება მასივის წყვილი მეზობელი a_i და a_{i+1} ელემენტები და ისინი გადაადგილდება მხოლოდ იმ შემთხვევაში, როდესაც სრულდება პირობა: $a_i > a_{i+1}$; ხოლო თუ $a_i \leq a_{i+1}$, მოწმდება მასივის მომდევნო მეზობელი წყვილი და ა.შ. ბოლო ელემენტამდე. ბუშტისებრ დახარისხების ალგორითმში პირველივე ეტაპის დასასრულს მასივის უდიდესი მნიშვნელობის მქონე ელემენტი გარანტირებულად იკავებს მასივის ბოლოს თავის ადგილს. მომდევნო ეტაპზე შემდეგი უდიდესი მნიშვნელობის მქონე ელემენტი თავსდება საჭირო ადგილას მასივში და პროცესი

გრძელდება მანამ, სანამ არ მოხდება მასივის ელემენტების სრული დახარისხება ზრდადობის მიხედვით. თუ მასივი n რაოდენობის ელემენტისგან შედგება, აღნიშნული ალგორითმი ელემენტების დასახარისხებლად საჭიროებს $n-1$ რაოდენობის ეტაპის შესრულებას და ყოველ ეტაპზე $n-1$ რაოდენობის შედარების ოპერაციის განხორციელებას.

ბუშტისებრი დახარისხების ალგორითმის შესაბამისი პროგრამული კოდი ქვემოთ არის წარმოდგენილი, ხოლო მისი რეალიზების შედეგები 78-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define size 10
void input_function(int []);   //prototipe
void print_function(int []);   //prototipe
void bubble_sort(int []);  //prototipe
int main() {

        int A[size];
        printf("Current array:\n");
        input_function(A);
        print_function(A);
        printf("\nSorted array by ASC:\n");
        bubble_sort(A);
        print_function(A);
        return 0;
}
void input_function(int B[]){
        srand(time(0));
        for(int i=0; i<size; i++)
                B[i]=10+rand()%41;
}
void print_function(int B[]){
        for(int i=0; i<size; i++)
         printf("%4d",B[i]);
}
void bubble_sort(int B[]){
        for(int m=1; m<size; m++)
        for(int i=0; i<size-1; i++)
          if(B[i]>B[i+1]){
                int temp=B[i];
```

```
                B[i]=B[i+1];
                B[i+1]=temp;
        }
}
```



სურ. 78

6.1.2. დახარისხება ჩასმით

დახარისხების აღნიშნული ალგორითმის არსი შემდეგია: ალგორითმის ყოველ ბიჯზე მასივის ელემენტების შემავალი მონაცემებიდან ერთ ელემენტს ვირჩევთ და მასივში საჭირო პოზიციაზე ვათავსებთ. აღსანიშნავია, რომ ერთელემენტიანი მასივი დახარისხებულად ითვლება. ეს პროცესი გრძელდება მანამ, სანამ ელემენ-ტების შემავალი მონაცემების ნაკრები არ ამოიწურება. დასახარისხებელი მასივი ორ ნაწილად შეიძლება დავყოთ: დახარისხებულ და დასახარისხებელ ნაწილებად. დახარისხების პროცესის დასაწყისში პირველი ელემენტი (ნულოვანი ინდექსის მქონე) დახარისხებულად ითვლება, ხოლო ყველა დანარჩენი - დასახარისხებლად.

დაწყებული მასივის რიგით მეორე ელემენტიდან ბოლომდე ალგორითმი დასახარისხებელი ნაწილიდან ელემენტს დახარისხებული მასივის საჭირო პოზი-ციაზე ათავსებს. შესაბამისად, დახარისხების ყოველ ბიჯზე მასივის დახარისხე-ბული ნაწილი ერთი ელემენტით იზრდება, ხოლო დასახარისხებელი ნაწილი ერთი ელემენტით მცირდება. ამრიგად, ალგორითმის ძირითადი ციკლი იწყება არა ნულოვანი, არამედ ინდექსით პირველი ელემენტიდან.

„დახარისხება ჩასმით“ ალგორითმის შესაბამისი პროგრამული კოდი ქვემოთ არის წარმოდგენილი, ხოლო მისი რეალიზების შედეგები 79-ე სურათზეა ნაჩვე-ნები.

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

```
#include <stdlib.h>
#include <time.h>
#define size 10
void input_function(int []);  //prototipe
void print_function(int []);  //prototipe
void insertion_sort(int []);  //prototipe
int main() {

        int A[size];
        printf("Current array:\n");
        input_function(A);
        print_function(A);
        printf("\nSorted array by ASC:\n");
        insertion_sort(A);
        print_function(A);
        return 0;
}
void input_function(int B[]){
        srand(time(0));
        for(int i=0; i<size; i++)
                B[i]=10+rand()%41;
}
void print_function(int B[]){
        for(int i=0; i<size; i++)
         printf("%4d",B[i]);
}
void insertion_sort(int B[]){
        for(int i=1; i<size; i++){
        int temp=B[i];
        int j;
        for(j=i; j>0 && temp<B[j-1]; j--)
                B[j]=B[j-1];
                B[j]=temp;
         } }
```

```
C:\Users\Pc\OneDrive\Documents\wew...
Current array:
  40  25  18  28  25  36  25  46  42  15
Sorted array by ASC:
  15  18  25  25  25  28  36  40  42  46
--------------------------------
Process exited after 0.007922 seconds with retu
Press any key to continue . . .
```

სურ. 79

6.1.3. დახარისხება ამორჩევით

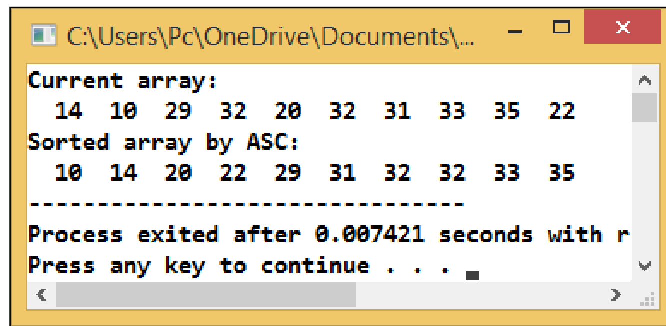
აღნიშნული ალგორითმი მასივის დახარისხების ყველაზე ბუნებრივ ალგორითმს წარმოადგენს. თავდაპირველად, მასივში მინიმალური მნიშვნელობის ელემენტი განისაზღვრება და ის მასივის პირველ ელემენტს ჩაანაცვლებს (ცხადია, თუ მასივის პირველ ელემენტს არ გააჩნია მინიმალური მნიშვნელობა). შემდეგ, დარჩენილი $n-1$ რაოდენობის ელემენტიდან კვლავ განისაზღვრება მინიმალური ელემენტი და ის მეორე ელემენტს ჩაანაცვლებს. პროცესი გრძელდება მასივის ელემენტების ამოწურვამდე. ალგორითმის ბიჯები შემდეგია:

- განისაზღვრება მასივის მინიმალური მნიშვნელობის ელემენტი;
- ზემოაღნიშნული ელემენტი პირველ დაუხარისხებელ პოზიციაში თავსდება;
- დახარისხდება მასივის ბოლო (კუდი) ისე, რომ უკვე დახარისხებული ელემენტები დახარისხების შემდგომ ეტაპებზე აღარ განიხილება და გამოირიცხება.

„დახარისხება ამორჩევით" ალგორითმის შესაბამისი პროგრამული კოდი ქვემოთ არის წარმოდგენილი, ხოლო მისი რეალიზების შედეგები მე-80 სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define size 10
void input_function(int []);   //prototipe
void print_function(int []);   //prototipe
void selection_sort(int []);  //prototipe
int main() {
  int A[size];
  printf("Current array:\n");
  input_function(A);
  print_function(A);
  printf("\nSorted array by ASC:\n");
  selection_sort(A);
  print_function(A);
  return 0;
}
void input_function(int B[]){
```

```
        srand(time(0));
         for(int i=0; i<size; i++)
                 B[i]=10+rand()%41;
}
void print_function(int B[]){
         for(int i=0; i<size; i++)
          printf("%4d",B[i]);
}
void selection_sort(int B[]){
         for(int i=0; i<size-1; i++){
         int min=i;
         for(int j=i+1; j<size; j++)
         if(B[j]<B[min])
           min=j;
           int temp=B[i];
           B[i]=B[min];
           B[min]=temp;
   }
}
```



სურ. 80

6.1.4. რედიქსის დახარისხება

რედიქსის დახარისხება არის დახარისხების უნიკალური ალგორითმი, რომელიც მუშაობს იმ ძირითადი პრინციპით, რომ რიცხვები არის ციფრების ანსამბლი.

რედიქსის დახარისხების ალგორითმი გამოიყენება მხოლოდ მთელი რიცხვე-ბის დასახარისხებლად ზრდადობით ან კლებადობით, რადგან მთელ რიცხვებს აქვთ მხოლოდ ერთი მათემატიკური კომპონენტი - **ციფრები**.

აღნიშნული ალგორითმი განვიხილოთ 10-ელემენტიანი მასივის მაგალითზე, რომლის საწყისი სახე 81-ე სურათზეა ნაჩვენები.

0	1	2	3	4	5	6	7	8	9
15	120	53	32	167	81	75	36	9	60

სურ. 81

- **ბიჯი 1.** მაქსიმალური ელემენტის მოძებნა;
- **ბიჯი 2.** მაქსიმალური მნიშვნელობის რიცხვში შემავალი ციფრების რაოდენობის დათვლა;

  ჩვენს შემთხვევაში მაქიმალური რიცხვია 167 და ის 3 ციფრისგან შედგება. ე.ი. ციკლი ასეულების თანრიგამდე უნდა მივიდეს (შესრულდეს 3-ჯერ).
- **ბიჯი 3.** რიცხვების დალაგება უმცირესი თანრიგის (ერთეულების) ციფრების მიხედვით ზრდადობით (იხილეთ 82-ე სურათი);
- **ბიჯი 4.** რიცხვების დალაგება შემდეგი თანრიგის (ათეულების) ციფრების მიხედვით (იხილეთ 83-ე სურათი);
- **ბიჯი 5.** პროცესის გაგრძელება შემდეგი (ასეულების) თანრიგისთვის (იხილეთ 84-ე სურათი).

0	1	2	3	4	5	6	7	8	9
15	120	53	36	167	81	75	32	9	60

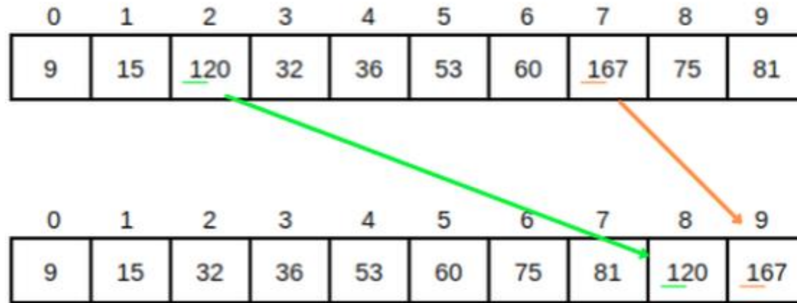
0	1	2	3	4	5	6	7	8	9
120	60	81	32	53	15	75	36	167	9

სურ. 82

0	1	2	3	4	5	6	7	8	9
120	60	81	32	53	15	75	36	167	9

0	1	2	3	4	5	6	7	8	9
9	15	120	32	36	53	60	167	75	81

სურ. 83



სურ. 84

როგორც ვხედავთ, მასივი დახარისხებულია მისი შემადგენელი ელემენტების ზრდადი მნიშვნელობების მიხედვით.

რედიქსის დახარისხების ალგორითმის შესაბამის პროგრამულ კოდს ქვემოთ ნაჩვენები სახე აქვს, ხოლო მისი რეალიზების შედეგები 85-ე სურათზეა წარმოდგენილი.

```
#include <stdio.h>
// find  max
int get_max(int a[], int n) {
  int max = a[0], i;
  for (i = 1; i < n; i++)
    if (a[i] > max)
      max = a[i];
  return max;
}
void radix_sort (int a[], int n){
   int bucket[10][10], bucket_cnt[10];
   int i, j, k, r, NOP = 0, divisor = 1, lar, pass;
   lar = get_max (a, n);
   while (lar > 0){     //Counting the number of corders
                        //in the number of maximum value
      NOP++;
      lar/= 10;
   }
for (pass = 0; pass < NOP; pass++){
      for (i = 0; i < 10; i++){
         bucket_cnt[i] = 0;
      } for (i = 0; i < n; i++){
         r = (a[i] / divisor) % 10;
         bucket[r][bucket_cnt[r]] = a[i];
         bucket_cnt[r] += 1;  }
i = 0;
```

```
    for (k = 0; k < 10; k++){
      for (j = 0; j < bucket_cnt[k]; j++){
        a[i] = bucket[k][j];
        i++; } }
divisor *= 10;
    printf ("After pass %d : ", pass + 1);
    for (i = 0; i < n; i++)
      printf ("%d ", a[i]);
    printf ("\n\n");
} }
int main() {
        int i, n, a[10];
  printf ("Enter the number of items : ");
  scanf ("%d", &n);
  printf ("\nEnter items: ");
  for (i = 0; i < n; i++)
    scanf ("%d", &a[i]);
  printf("\n\n");
  radix_sort (a, n);
  printf ("Sorted items : ");
  for (i = 0; i < n; i++)
    printf ("%d ", a[i]);
        return 0;
}
```

```
C:\Users\Pc\OneDrive\Documents\wewew33.exe
Enter the number of items : 10

Enter items: 132 234 345 10 32 345 754 230 45 76


After pass 1 : 10 230 132 32 234 754 345 345 45 76

After pass 2 : 10 230 132 32 234 345 345 45 754 76

After pass 3 : 10 32 45 76 132 230 234 345 345 754

Sorted items : 10 32 45 76 132 230 234 345 345 754
--------------------------------
Process exited after 20.59 seconds with return value 0
```

სურ. 85

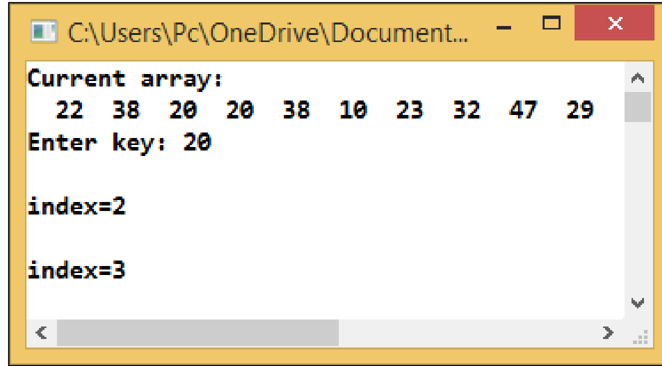
6.2.1. წრფივი ძებნა

წრფივი ძებნის ალგორითმის არსი შემდეგში მდგომარეობს: წინასწარ მოიცემა განსაზღვრული მნიშვნელობის მქონე ცვლადი და ის თანმიმდევრობით შედარდება მასივის ყველა ელემენტის მნიშვნელობას. თუ მასივის რომელიმე ელემენტის მნიშვნელობა დაემთხვევა მოცემული ცვლადის მნიშვნელობას, ალგორითმი ამ ელემენტის ინდექსს გამოიტანს; წინააღმდეგ შემთხვევაში ალგორითმი დასრულდება შესაბამისი შეტყობინებით.

წრფივი ძებნის ალგორითმის შესაბამის პროგრამულ კოდს ქვემოთ ნაჩვენები სახე აქვს, ხოლო მისი რეალიზების შედეგები 86-ე სურათზეა წარმოდგენილი.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define size 10
void input_function(int []);   //prototipe
void print_function(int []);   //prototipe
void linear_search(int [], int);  //prototipe
int main() {

        int A[size], key;
        printf("Current array:\n");
        input_function(A);
        print_function(A);
        printf("\nEnter key: ");
        scanf("%d", &key);
        linear_search(A, key);
        return 0;
}
void input_function(int B[]){
        srand(time(0));
        for(int i=0; i<size; i++)
                B[i]=10+rand()%41;
}
void print_function(int B[]){
        for(int i=0; i<size; i++)
         printf("%4d",B[i]);
}
void linear_search(int B[], int key){
        int s=0;
        for(int i=0; i<size; i++)
```

```
        if(key==B[i]){              printf("\nindex=%d\n", i);
            s++;
        }
        if(s==0)
         printf("\nNo Solution");
}
```



სურ. 86

6.2.2. ბინარული ძებნა

განვიხილოთ თხუთმეტი ელემენტისგან შემდგარი *A* მასივი და დიხოტომიის ანუ ბინარული ძებნის ალგორითმით მოვძებნოთ საჭირო მნიშვნელობის ელემენტი მასში.

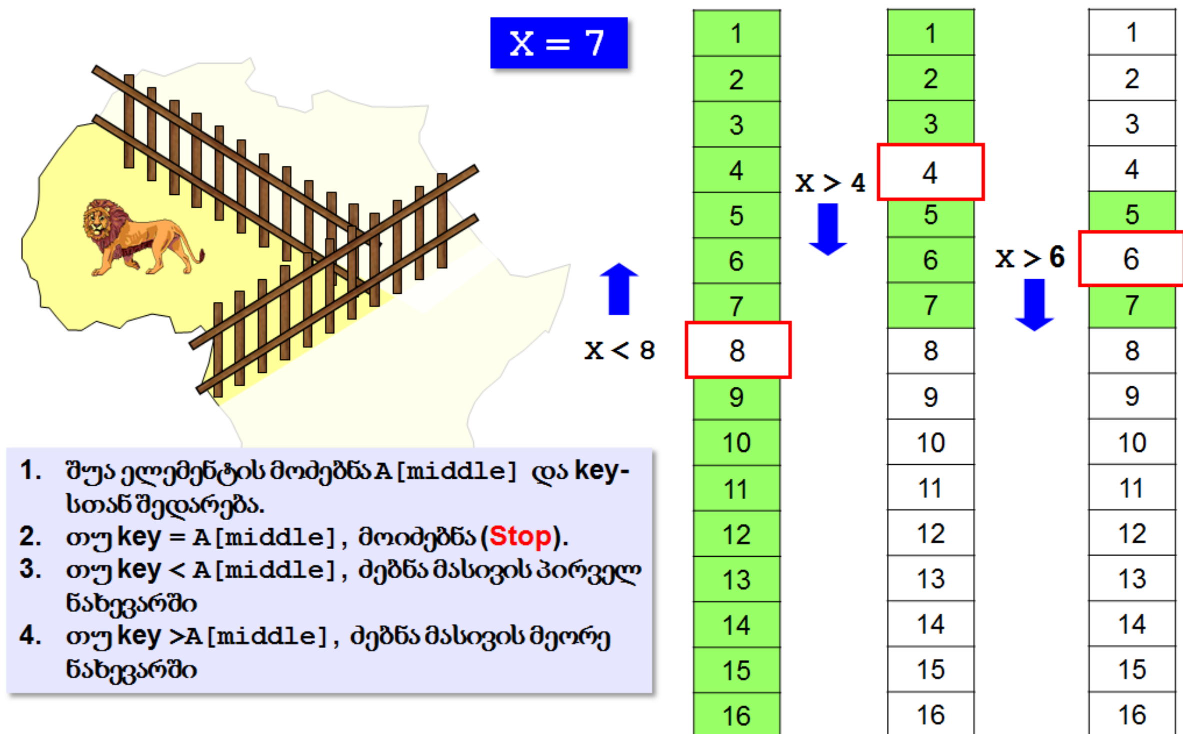
აღნიშნული ალგორითმის არსი შემდეგში მდგომარეობს:

ალგორითმის მუშაობის აუცილებელი პირობა მასივთა წინასწარ დახარისხებაში მდგომარეობს, ე.ი. ორმაგი ძებნის პროცესი ხორციელდება მხოლოდ დახარისხებულ მასივებში.

დავუშვათ, *A* მასივის ელემენტები დალაგებულია ზრდადობის მიხედვით. ალგორითმი ითვალისწინებს მასივის შუა ელემენტის მოძებნას, რომლის მნიშვნელობაც წინასწარ მოცემული ცვლადის მნიშვნელობას უნდა შედარდეს; თუ მათი მნიშვნელობები დაემთხვა, ეს ნიშნავს, რომ საჭირო ელემენტი მასივში მოიძებნა და პროცესი წყდება. წინააღმდეგ შემთხვევაში, თუ ცვლადის მნიშვნელობა მასივის შუა ელემენტის მნიშვნელობაზე ნაკლები აღმოჩნდება, ძიება წარმოებს მასივის პირველ ნახევარში, ხოლო თუ ცვლადის მნიშვნელობა გადააჭარბებს მასივის შუა ელემენტის მნიშვნელობას, ძიება საწყისი მასივის მეორე ნახევარში ხორციელდება.

ამდენად, ძებნის პირველივე ეტაპზე ადგილი აქვს მასივის განახევრებას, შემდეგ მიღებული ნახევრებიდან ერთ-ერთის იგნორირების გზით და მასივის მეორე ნახევრის შუაზე გაყოფით, ძიება წარმოებს საწყისი მასივის მეოთხედში და ა.შ. პროცესი გრძელდება მანამ, სანამ მასივის ერთ-ერთი ნახევრის შუა ელემენტის მნიშვნელობა არ დაემთხვევა მოცემული ცვლადის მნიშვნელობას, ან სანამ საჭირო ელემენტი არ მოიძებნება.

ბინარული ძებნის ალგორითმის შესაბამისი სქემა 87-ე სურათზეა ნაჩვენები.



სურ. 87

ბინარული ძებნის ალგორითმის შესაბამის პროგრამულ კოდს ქვემოთ ნაჩვენები სახე აქვს, ხოლო მისი რეალიზების შედეგი 88-ე სურათზეა წარმოდგენილი.

```
#include <stdio.h>
#define size 15
void input_function(int []);   //prototipe
void print_function(int []);   //prototipe
void binary_search(int [], int);  //prototipe
int main() {

        int A[size], key;
        printf("Current array:\n");
        input_function(A);
```

```
        print_function(A);
        printf("\nEnter key: ");
        scanf("%d", &key);
        binary_search(A, key);
        return 0;
}
void input_function(int B[]){
        for(int i=0; i<size; i++)
                B[i]=1+2*i;
}
void print_function(int B[]){
        for(int i=0; i<size; i++)
         printf("%4d",B[i]);
}
void binary_search(int B[], int key){
        int s=0, low=0, high=size-1, middle;
        while(low<=high){
            middle=(low+high)/2;
                if(key==B[middle]){
                 printf("\nindex=%d", middle);
                  s++;
                    break;
     }
        else if(key>B[middle])
                low=middle+1;
                else
                high=middle-1;
  }
          if(s==0)
            printf("\nNo Solution");
}
```

```
C:\Users\Pc\OneDrive\Documents\wewew33.exe
   1   3   5   7   9  11  13  15  17  19  21  23  25  27  29
Enter key: 23

index=11
--------------------------------
```

სურ. 88

VII თავი
სტრუქტურები

სტრუქტურა საერთო სახელის მქონე ურთიერთდაკავშირებული ერთიდაიმავე ან სხვადასხვა ტიპის ობიექტების ერთობლიობაა. სტრუქტურის ელემენტების (წევრების, ველების) როლში შეიძლება გამოყენებულ იქნას: ცვლადები, მასივები, მიმთითებლები ან სხვა სტრუქტურები.

სტრუქტურა საშუალებას გვაძლევს შევქმნათ მონაცემთა ახალი ტიპი. სტრუქ-ტურის სახელი მის ტიპზე მიუთითებს.

სტრუქტურა ერთმანეთთან დაკავშირებულ ობიექტების ჯგუფს განიხილავს არა როგორც ცალკეული ელემენტების ერთობლიობას, არამედ, როგორც ერთ მთლიანს.

სტრუქტურა მონაცემთა რთული ტიპია, რომელიც მონაცემთა მარტივი ტიპე-ბისგან შედგება.

სტრუქტურის გამოცხადების (დეკლარირების) ზოგადი ფორმა შემდეგია:

struct სტრუქტურის სახელი {

ტიპი ელემენტი1-ის სახელი;

ტიპი ელემენტი2-ის სახელი;

………………………………………………

ტიპი ელემენტი n-ის სახელი;

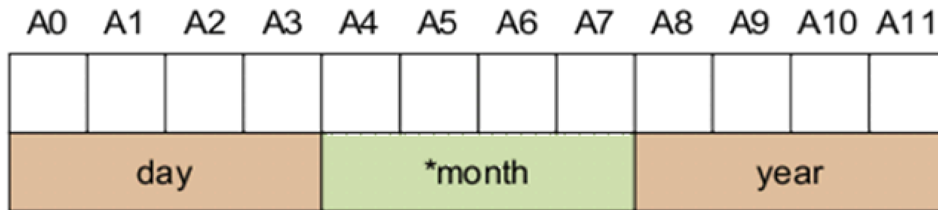
};

სტრუქტურის დეკლარირებაში დახურული ფიგურული ფრჩხილის შემდეგ (}) **წერტილ-მძიმის დასმა აუცილებელია**, რადგან ის ამ პროცესის დასრულებაზე მიუთითებს.

სტრუქტურის გამოცხადების ნიმუში:

```
struct date{
int day;            //4 ბაიტი
char  *month;   //4 ბაიტი
int year;          //4 ბაიტი
};
```

სტრუქტურის ველები კომპიუტერის ოპერატიულ მეხსიერებაში იმ თანმიმდევრობით განთავსდება, როგორც დეკლარირების პროცესში არის წარმოდგენილი. აღნიშნულ მაგალითში (იხილეთ 89-ე სურათი) date სტრუქტურა 12 ბაიტს იკავებს მეხსიერებაში.



სურ. 89

7.1. სტრუქტურის ველების ინიციალება

სტრუქტურის დეკლარირების დროს შესაძლებელია ერთი სტრუქტურის შიგნით (ველის სახით) მეორე სტრუქტურის მოთავსება. ასე, მაგალითად:

```
struct persone {
char lastname[20];
char firstname[20];
struct date bd;
};
```

სტრუქტურის ველების ინიციალება ორი გზით არის შესაძლებელი:

- სტრუქტურის ელემენტებზე მნიშვნელობების მინიჭებით იმ ცვლადის გამოცხადების პროცესში, რომელიც მოცემული სტრუქტურის ტიპს მიეკუთვნება;
- სტრუქტურის ელემენტებზე საწყისი მნიშვნელობების მისანიჭებლად შეტანა/გამოტანის ფუნქციების (printf(), scanf()) გამოყენებით.

ინიციალების პირველი გზა შემდეგი ფორმისაა:

struct სტრუქტურის სახელი ცვლადის სახელი={ელემენტი1-ის მნიშვნელობა, ელემენტი2-ის მნიშვნელობა, ... , ელემენტი n-ის მნიშვნელობა};

ასე მაგალითად:

struct date bd={16, “December”, 1967};

7.2. სტრუქტურის ელემენტზე მიმართვა. სტრუქტურის ეგზემპლარის გამოცხადება

სტრუქტურის ელემენტზე მიმართვისთვის საჭიროა: სტრუქტურის ტიპის ცვლადის სახელის და თავად ელემენტის სახელის მითითება, ხოლო მათ შორის - წერტილის დასმა.

ცვლადის სახელი. სტრუქტურის ელემენტის სახელი;

ასე, მაგალითად:

printf("%d %s %d",bd.day, bd.month, bd.year);

სტრუქტურის ტიპის ცვლადის სახელი შესაძლებელია სტრუქტურის დეკლარირების დროს მიეთითოს. ამ შემთხვევაში, სტრუქტურის ცვლადის სახელი სტრუქტურის დეკლარირების დამამთავრებელი დახურული ფიგურული ფრჩხილის (}) შემდეგ იწერება. ასე, მაგალითად:

struct complex_type {

doubl ereal;

double imag;

} number; // სტრუქტურის ცვლადის სახელი

სტრუქტურის ეგზემპლარის გამოცხადება არაფრით განსხვავდება ნებისმიერი სხვა ტიპის მონაცემის გამოცხადებისგან. ასე, მაგალითად:

struct dinner week_days [7]; //სტრუქტურის ტიპის მასივი

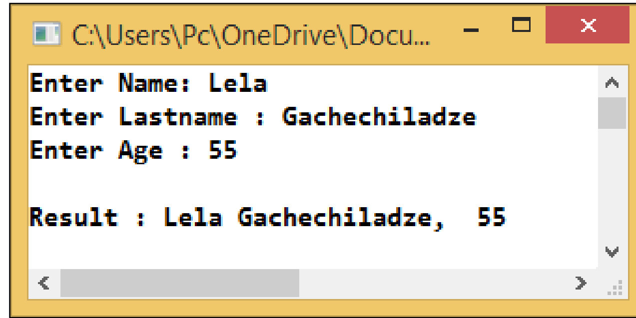
struct dinner best_one; // სტრუქტურის ტიპის ცვლადი

struct dinner *p; //მიმთითებელი სტრუქტურის ტიპის ცვლადზე

ქვემოთ წარმოდგენილია სტრუქტურის გამოყენების ამსახველი მარტივი პროგრამული კოდი, რომლის შესრულების შედეგები 90-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
struct persone {
  char firstname[20];
  char lastname[20];
  int age;
};
int main() {
  struct persone p;
  printf("Enter Name: ");
```

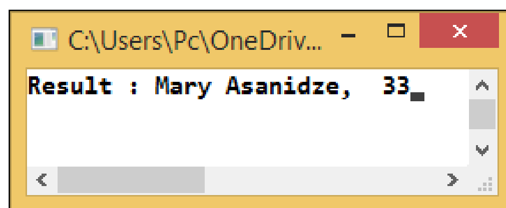
```
 scanf("%s", p.firstname);
 printf("Enter Lastname : ");
 scanf("%s", p.lastname);
 printf("Enter Age : ");
 scanf("%d", &p.age);
 printf("\nResult : %s %s,  %d", p.firstname, p.lastname, p.age);
 getchar(); getchar();
 return 0; }
```



სურ.90

წარმოდგენილი კოდის მოდიფიცირებულ ვარიანტს ქვემოთ ნაჩვენები სახე აქვს, ხოლო მისი შესრულების შედეგი იხილეთ 91-ე სურათზე.

```
#include <stdio.h>
#include <string.h>
struct persone {
 char firstname[20];
 char lastname[20];
 int age;
};
int main() {
 struct persone p;
 strcpy(p.firstname, "Mary");
 strcpy(p.lastname, "Asanidze");
 p.age=33;
 printf("Result : %s %s,  %d", p.firstname, p.lastname, p.age);
 getchar(); getchar();
 return 0;
}
```



სურ. 91

იმისათვის, რომ სტრუქტურის ტიპის ცვლადის გამოცხადებისას არ გამოვიყენოთ დარეზერვებული სიტყვა **stuct**, საჭიროა სტრუქტურის გამოცხადებისას მივმართოთ **typedef ინსტრუქციას!**

ასე, მაგალითად:

```
typedef struct {
  char firstname[20];
  char lastname[20];
  int age;
} persone ;
```

შესაბამისად, ზემოთ წარმოდგენილი პროგრამული კოდი მიიღებს სახეს:

```
#include <stdio.h>
#include <string.h>
typedef struct  {
  char firstname[20];
  char lastname[20];
  int age;
}persone;
int main() {
  persone p;
  strcpy(p.firstname, "Mary");
  strcpy(p.lastname, "Asanidze");
  p.age=33;
  printf("Result : %s %s,  %d", p.firstname, p.lastname, p.age);
  getchar(); getchar();
  return 0;}
```

პროგრამის შესრულების შედეგი, ცხადია, დამეთხვევა 91-ე სურთზე ნაჩვენებ შედეგს.

წარმოვადგინოთ სტრუქტურის გამოყენების ამსახველი კიდევ რამდენიმე პროგრამული კოდი.

```
#include <stdio.h>
struct date {
  int day;
  char month[20];
  int year;
};
struct persone {
  char firstname[20];
  char lastname[20];
  struct date bd;
};
int main() {
  struct persone p;
  printf("Enter Name: ");
  scanf("%s", p.firstname);
  printf("Enter Lastname : ");
  scanf("%s", p.lastname);
  printf("Enter birthday\nNumber: ");
  scanf("%d", &p.bd.day);
  printf("Month: ");
  scanf("%s", p.bd.month);
  printf("Year: ");
  scanf("%d", &p.bd.year);
  printf("\nResult : %s %s,  %d %s %d", p.firstname, p.lastname, p.bd.day, p.bd.month,
p.bd.year);
  getchar(); getchar();
  return 0;
}
```

პროგრამის შესრულების შედეგი 92-ე სურათზეა ნაჩვენები.

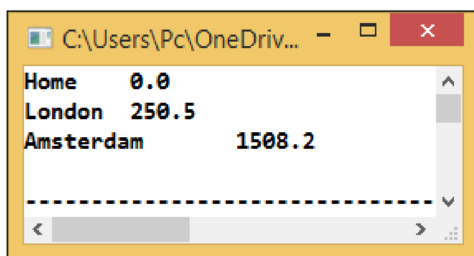
```
C:\Users\Pc\OneDrive\Documents\...
Enter Name: Nick
Enter Lastname : Sivsivadze
Enter birthday
Number: 29
Month: March
Year: 2021

Result : Nick Sivsivadze,  29 March 2021
```

სურ. 92

```
struct dinner{
char place[100];
float cost;
```

```
};
struct dinner three_days_dinner[]={{"Home", 0.},
                                   {"London", 250.5},
                                   {"Amsterdam", 1508.2}};
int main(){
        int i;
        for(i=0; i<3; i++)
        printf("%s\t%.1f\n", three_days_dinner[i].place, three_days_dinner[i].cost);
        return 0; }
```

პროგრამის შესრულების შედეგი 93-ე სურათზეა ნაჩვენები.



სურ. 93

ამოცანა 31.

შევქმნათ სტრუქტურა სახელწოდებით student. აღნიშნული სტრუქტურა უნდა მოიცავდეს შემდეგ ველებს:

- სტუდენტის სახელს (firstname);
- სტუდენტის გვარს (lastname);
- ფაკულტეტის დასახელებს (name_of_faculty);
- ჯგუფის ნომერს (number_of_group);
- კურსის ნომერს (course).

პირველი სამი ველი წარმოვადგინოთ სიმბოლური მასივების სახით, ხოლო ბოლო ორი ველი - მთელი ტიპის მონაცემების სახით.

მოვახდინოთ სტრუქტურის ველების ინიციალება და შედეგები კონსოლზე გამოვიტანოთ printf() ფუნქციის საშუალებით.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
 typedef struct {
char firstname[20];
char lastname[30];
```

```
char name_of_faculty[20];
int number_of_group;
int curse;
} student;
int main() {
 student me;
 printf("Enter FirstName: ");
 scanf("%s", me.firstname);
 printf("Enter LastName: ");
 scanf("%s", me.lastname);
 printf("Enter Name of faculty: ");
 scanf("%s",me.name_of_faculty);
 printf("Enter Nnumber of group: ");
 scanf("%d",&me.number_of_group);
 printf("Enter Nnumber of curse: ");
 scanf("%d",&me.curse);
printf("%s\t%s\t%s\t%d\t%d",me.firstname,me.lastname,me.name_of_faculty,
me.number_of_group,  me.curse);
return 0;
 }
```

პროგრამის შესრულების შედეგები 94-ე სურათზეა ნაჩვენები.

```
C:\Users\Pc\OneDrive\Documents\wew...
Enter FirstName: Levan
Enter LastName: Dadiani
Enter Name of faculty: Informatics
Enter Nnumber of group: 108150
Enter Nnumber of curse: 3
Levan   Dadiani Informatics     108150  3
--------------------------------
```

სურ. 94

ამოცანა 32.

შევადგინოთ პროგრამა, რომელიც სტრუქტურის ტიპის მასივის გამოყენებით მსოფლიოს ოთხ ქალაქსა და სახლში ვახშამზე დახარჯული თანხის ჯამურ მაჩვენებლს და დახარჯული თანხის მაქსიმალურ ოდენობას კონსოლზე გამო-იტანს.

```
#include <stdio.h>
typedef struct {
```

```
char place[100];
int cost;
} dinner;
dinner array[]={ {"Home", 0},
                  {"London", 250},
                 {"Amsterdam", 1500},
                 {"Barcelona", 1600},
                 {"Paris", 2500}
                               };
int main(){
        int i;
        int s=0,  max=array[0].cost;
        for(i=0; i<5; i++){
        printf("%s\t%d\n", array[i].place, array[i].cost);
        s+=array[i].cost;
        if(max<=array[i].cost)
         max=array[i].cost;
}
printf("\nSum of costs=%d", s);
printf("\nMax of costs=%d", max);
return 0;
}
```

პროგრამის შესრულების შედეგები 95-ე სურათზეა ნაჩვენები.

```
C:\Users\Pc\On...
Home    0
London  250
Amsterdam       1500
Barcelona       1600
Paris   2500

Sum of costs=5850
Max of costs=2500
---------------------------
```

სურ. 95

VIII თავი
მიმთითებლები (Pointers)

მიმთითებლები წარმოადგენს ცვლადებს, რომელთაც საკუთარი მნიშვნელობების სახით კომპიუტერის ოპერატიული მეხსიერების მისამართები აქვს. მეორეს მხრივ, მიმთითებელი იმ ცვლადის მისამართს შეიცავს, რომელსაც კონკრეტული მნიშვნელობა აქვს. ამ თვალსაზრისით, ცვლადის სახელი პირდაპირ მიმართავს თავის მნიშვნელობას, ხოლო მიმთითებელი – არაპირდაპირი სახით.

ცვლადის ამა თუ იმ მნიშვნელობაზე მიმთითებლის საშუალებით მიმართვას, **არაპირდაპირი დამისამართება** ეწოდება.

მიმთითებელი, მსგავსად ნებისმიერი ცვლადისა, პროგრამაში გამოყენებამდე უნდა განისაზღვროს. ასე, მაგალითად:

int *xPtr, x;

ჩანაწერი წარმოგვიდგენს **int** ტიპის ცვლადს იდენტიფიკატორით **xPtr** (ე.ი. მიმთითებელი მთელი ტიპის მონაცემზე) და იკითხება შემდეგნაირად: „ცვლადი **xPtr** წარმოადგენს მიმთითებელს მთელი ტიპის რიცხვზე“. ცვლადი **x** კი განსაზღვრულია, როგორც მთელი რიცხვი და არა მიმთითებელი მთელი ტიპის მონაცემზე. სიმბოლო “ * “ (ვარსკვლავი) ამ შემთხვევაში მხოლოდ **xPtr** ცვლადს ეკუთვნის.

ყოველ ცვლადს, რომელიც მიმთითებლის სახით განიხილება, განსაზღვრის დროს წინ უნდა უსწრებდეს სიმბოლო **“ * “**. მაგალითად, ჩანაწერი

double *xPtr, *yPtr ;

გვიჩვენებს, რომ **xPtr** და **yPtr** ცვლადები წარმოადგენს მიმთითებლებს **double** ტიპის მონაცემებზე. ზემოაღნიშნული “ * " სიმბოლოს გამოყენება ნიშნავს, რომ ცვლადი განისაზღვრება მიმთითებლის სახით.

8.1.1. ოპერაციები მიმთითებლებზე

მიმთითებლებზე შესაძლებელია განვახორციელოთ **უნარული დამისამართების ოპერაცია**, რომელიც თავისი ოპერანდის მისამართს აბრუნებს და **"&"** (ამპერსენდი) სიმბოლოთი აღინიშნება.

განვიხილოთ პროგრამული კოდის შემდეგი ფრაგმენტი:

int y=15;

int *yPtr;

yPtr=&y;

თავდაპირველად აქ ადგილი აქვს **int** (მთელი) ტიპის **y** ცვლადის ინიციალებას, რომელსაც 15-ის ტოლი მნიშვნელობა ენიჭება. შემდეგ განისაზღვრება მიმთითებელი **int** ტიპის მონაცემზე **yPtr**, ხოლო უნარული დამისამართების ოპერაციის გზით (**yPtr=&y**) **yPtr** მიმთითებელს მნიშვნელობის სახით ენიჭება **y** ცვლადის მისამართი.

მიმთითებლებზე ხორციელდება **არაპირდაპირი დამისამართების ოპერაციაც,** რომელიც ვარსკვლავის სიმბოლოთი (*) აღინიშნება. იგი იმ ობიექტის მნიშვნელობის დაბრუნებას ახდენს, რომელზეც მისი ოპერანდი (მიმთითებელი) მიუთითებს.

მიმთითებლების დეკლარირებისა და მისამართის ჩაწერის ნიმუშები 96-ე სურათზეა ნაჩვენები.

მიმთითებლის დეკლარირება:

```
char   *pC; // სიმბოლოს მისამართი
            // (ან მასივის ელემენტის)
int    *pI; // მთელი ტიპის ცვლადის მისამართი
float *pF;  // ნამდვილი ტიპის ცვლადის მისამართი
```

როგორ ჩავწეროთ მისამართი?

```
int m = 5, *pI;
int A[2] = { 3, 4 };
pI = &m;       // m ცვლადის მისამართი
pI = &A[1];  // მასივის A[1] ელემენტის მისამართი
pI = NULL;    // ნულოვანი მისამართი
```

```
scanf("%d", &m);
```

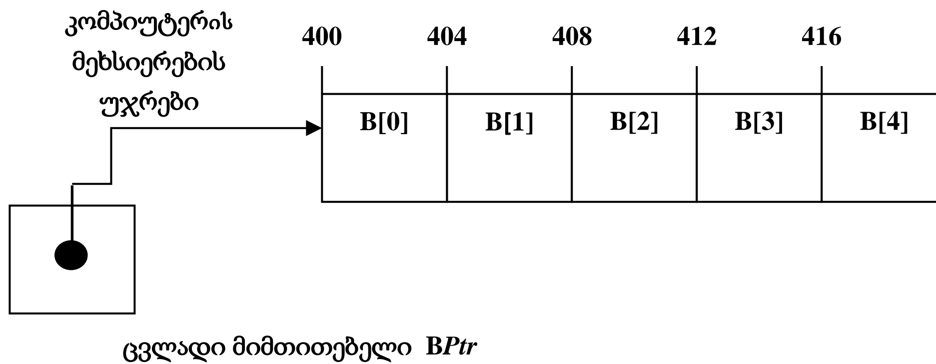
სურ. 96

8.1.2. არითმეტიკული მოქმედებები და გამოსახულებები მიმთითებლებზე

მიმთითებლები შესაძლებელია ოპერანდების სახით იქნეს გამოყენებული არითმეტიკულ, მინიჭებისა და შედარების ოპერაციებში. კერძოდ, მიმთითებლებზე შემდეგი არითმეტიკული ოპერაციები სრულდება:

- ინკრემენტისა (++) და დეკრემენტის (--) ოპერაციები;
- მიმთითებელს შეიძლება მივუმატოთ (+ ან +=) ან გამოვაკლოთ (-ან -=) მთელი ტიპის მნიშვნელობა;
- ერთ მიმთითებელს შეიძლება გამოვაკლოთ მეორე მიმთითებელი.

განვიხილოთ ხუთი ელემენტისგან შემდგარი მთელრიცხვა მასივი: int B[5]. დავუშვათ, რომ მისი პირველი (ე.ი. ნულოვანი ინდექსის მქონე) ელემენტი კომპიუტერის მეხსიერების მე-400 უჯრედშია მოთავსებული, ხოლო BPtr მიმთითებელს მნიშვნელობის სახით მასივის B[0] ელემენტის მისამართი აქვს მინიჭებული, ანუ: **BPtr = &B[0]** (იხილეთ 97-ე სურათი).



სურ. 97

BPtr მიმთითებელმა საწყისი მნიშვნელობა შემდეგი ოპერაციის შესრულების შედეგადაც შეიძლება მიიღოს: **BPtr = B**, რადგან ცნობილია, რომ მასივის სახელი წარმოადგენს მისი პირველი ელემენტის მისამართს კომპიუტერის მეხსიერებაში.

დავუშვათ, მთელი ტიპის თითოეული მონაცემი მეხსიერებაში ოთხ ბაიტს იკავებს (თუმცა, იგი დამოკიდებულია კონკრეტული ტიპის კომპიუტერზე). ახლა ვცადოთ და მიმთითებელს მივუმატოთ სამი. შედეგად მივიღებთ არა 403-ს, არამედ 412-ს, რადგან, როდესაც მიმთითებელს ამა თუ იმ მთელ რიცხვს ვუმა-

ტებთ, იგი ჯერ იმ ობიექტის ზომაზე მრავლდება, რომელზეც მიმთითებელი მიუთითებს და მხოლოდ ამის შემდეგ სრულდება მიმატების ოპერაცია. მაშასადამე, სამი თავდაპირველად მრავლდება ოთხზე (რადგან წინასწარ დავუშვით, რომ ყოველი int ტიპის მონაცემი ოთხ ბაიტს იკავებს) და შემდეგ ემატება 400, რის გამოც შედეგი 412 -ის ტოლია. BPtr +=3 ოპერაციის შესრულებით მიმთითებელი იმ ელემენტზე მიუთითებს, რომელიც კომპიუტერის მეხსიერების 412-ე უჯრედშია მოთავსებული, ანუ B[3]-ზე. თუ მასივს მონაცემთა სხვა ტიპი ექნებოდა, მაშინ ბუნებრივია, სრულიად განსხვავებულ შედეგს მივიღებდით, რადგან მონაცემთა ყოველი ტიპი გარკვეულ მოცულობას იკავებს კომპიუტერის მეხსიერებაში.

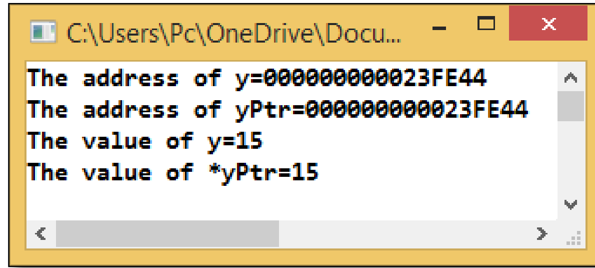
თუ მიმთითებელს აკლდება რაიმე მთელი რიცხვი, იგი ჯერ იმ ობიექტის ზომაზე მრავლდება, რომელზეც მიმთითებელი მიუთითებს, და შემდეგ სრულდება გამოკლების ოპერაცია. როდესაც არითმეტიკული ოპერაციები სრულდება მიმთითებლებზე, რომლებიც სიმბოლური ტიპის მონაცემებისგან შემდგარ მასივზე მიუთითებენ, ამ შემთხვევაში, მიღებული შედეგები მათემატიკურ შედეგებს ემთხვევა, რადგან ყოველი სიმბოლო მხოლოდ ერთ ბაიტს იკავებს. თუ მიმთითებელი ერთით იზრდება ან მცირდება, შესაძლებელია ინკრემენტისა (++) და დეკრემენტის (--) ოპერაციების გამოყენება, რომლებიც მიმთითებელს ზრდის (ან ამცირებს) ისე, რომ იგი მასივის მომდევნო (ან წინა) ელემენტზე მიუთითებდეს. როდესაც ერთ მიმთითებელს მეორეს ვაკლებთ, სხვაობის ოპერაცია იმ ელემენტების ინდექსებს შორის სრულდება, რომლებზეც ეს მიმთითებლები მიუთითებენ. მაგალითად, ოპერაცია: B3Ptr-BPtr გვაძლევს სამის ტოლ შედეგს (რადგან 3-0=3).

მიმთითებლები შეიძლება ერთმანეთს შევადაროთ იმ შემთხვევაში, თუ კი ისინი ერთსა და იმავე მასივის ელემენტებზე მიუთითებენ. ამასთან, აღნიშნული ოპერაციისას უნდა შედარდეს ის მისამართები, რომლებიც მიმთითებლებში ინახება.

განვიხილოთ მიმთითებლებზე უნარული და არაპირდაპირი დამისამართების ოპერაციების გამოყენების სადემონსტრაციო პროგრამა, რომლის შესრულების შედეგი 98-ე სურათზეა წარმოდგენილი.

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

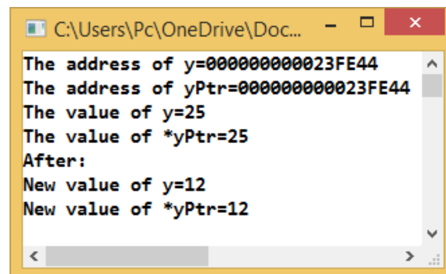
```
int y=15;
int *yPtr;
yPtr=&y;
printf("The address of y=%p\n",&y);
printf("The address of yPtr=%p\n",yPtr);
printf("The value of y=%d\n",y);
printf("The value of *yPtr=%d\n",*yPtr);
return 0;
}
```



სურ. 98

მიმთითებლის მნიშვნელობის ცვლილება იმ ცვლადის მნიშვნელობის ცვლილებასაც იწვევს, რომელზეც ის მიუთითებს! ვნახოთ შესაბამისი პროგრამული კოდი. მისი შესრულების შედეგი 99-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main() {
int y=25;
int *yPtr;
yPtr=&y;
printf("The address of y=%p\n",&y);
printf("The address of yPtr=%p\n",yPtr);
printf("The value of y=%d\n",y);
printf("The value of *yPtr=%d\n",*yPtr);
*yPtr=12;
printf("After:\n");
printf("New value of y=%d\n",y);
printf("New value of *yPtr=%d\n",*yPtr);
return 0; }
```



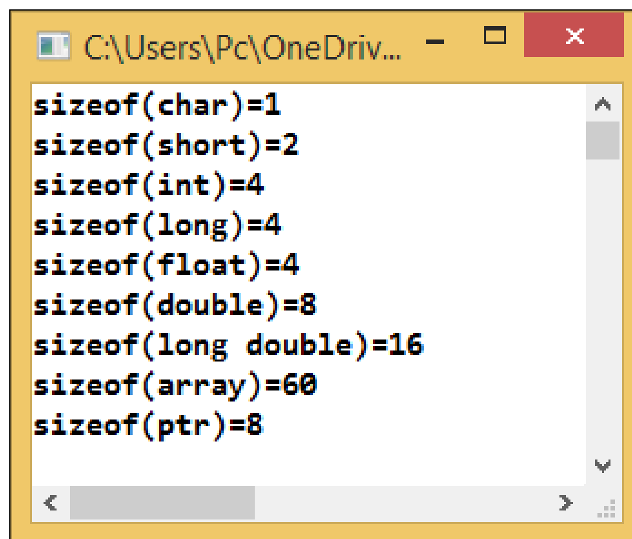
სურ. 99

8.1.3. უნარული ოპერაცია sizeof

დაპროგრამების ენა C-ში **sizeof** უნარული ოპერაცია გამოიყენება, რომელიც თავისი ოპერანდის ზომას ბაიტებში განსაზღვრავს. თუ აღნიშნული ოპერაციის ოპერანდს ამა თუ იმ მონაცემის (ცვლადის, მასივის, მუდმივას) სახელი წარმოადგენს, ამ შემთხვევაში ოპერანდის ფრჩხილებში მოთავსება საჭირო არ არის, ყველა დანარჩენ შემთხვევაში კი - აუცილებელია. ასე, მაგალითად:

```
#include <stdio.h>
int main() {
int array[15];
int *ptr=array;
printf("sizeof(char)=%d\n",sizeof(char));
printf("sizeof(short)=%d\n",sizeof(short));
printf("sizeof(int)=%d\n",sizeof(int));
printf("sizeof(long)=%d\n",sizeof(long));
printf("sizeof(float)=%d\n",sizeof(float));
printf("sizeof(double)=%d\n",sizeof(double));
printf("sizeof(long double)=%d\n",sizeof(long double));
printf("sizeof(array)=%d\n",sizeof array);
printf("sizeof(ptr)=%d\n",sizeof ptr);
return 0;
}
```

წარმოდგენილი პროგრამის შესრულების შედეგები მე-100 სურათზეა ნაჩვენები.



სურ. 100

8.1.4. ურთიერთკავშირი მასივებსა და მიმთითებლებს შორის

C ენაში მასივები და მიმთითებლები ერთმანეთთან მჭიდრო კავშირშია. მათი გამოყენება თითქმის ერთმანეთის ეკვივალენტურად შეიძლება. ხშირ შემთხვევაში, მასივის სახელი განიხილება როგორც მუდმივი მიმთითებელი.

დავუშვათ, განსაზღვრულია ხუთი ელემენტისგან შემდგარი მთელრიცხვა B[5] მასივი და bPtrt მთელი ტიპის ცვლადი-მიმთითებელი. რადგან მასივის სახელი (ინდექსის გარეშე) წარმოადგენს მიმთითებელს მის პირველ ელემენტზე, ამიტომ **BPtrt** მიმთითებელს შემდეგი ოპერატორით შეგვიძლია მივანიჭოთ მასივის პირველი ელემენტის მისამართი:

BPtrt =B;

თუმცა, იგივე ოპერაცია სხვაგვარადაც სრულდება, კერძოდ:

BPtrt =&B[0];

იმისათვის, რომ მასივის B[4] ელემენტს მივმართოთ, შეიძლება შემდეგი ჩანაწერი გამოვიყენოთ: ***(BPtr+4).**

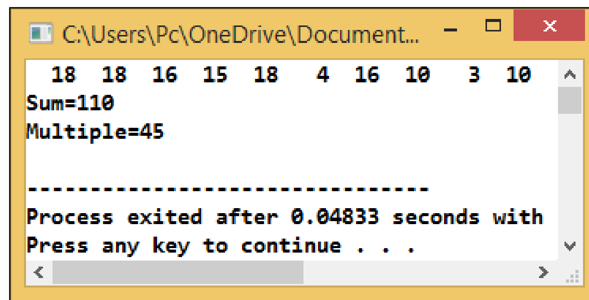
&B[4] მისამართი ასევე შემდეგი ოპერაციის გამოყენებით ჩაიწერება: BPtr+4;

ხოლო *(B+4) გამოსახულება მასივის B[4] ელემენტს მიმართავს. რადგან მასივის სახელი მუდმივ მიმთითებელს წარმოადგენს (იგი ყოველთვის მასივის დასაწყისზე მიუთითებს). B+=4 გამოსახულება მცდარია, ვინაიდან იგი ცდილობს მასივის სახელის (ამ შემთხვევაში, მნიშვნელობის) მოდიფიცირებას, რაც დაუშვებელია!

ქვემოთ წარმოდგენილია მასივებსა და მიმთითებლებს შორის კავშირის ამსახველი პროგრამა, რომლის შესრულების შედეგები 101-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
const int size=10;
int a[size], i;
int s=0;
long p=1;
int *aPtr;
aPtr=a;
srand(time(0));
for(i=0; i<size; i++){
        *(aPtr+i)=1+rand()%25;
```

```
        printf("%4d",*(aPtr+i));
        if(*(aPtr+i)%2==0)
        s+=*(aPtr+i);
        else
        p*=*(aPtr+i);
}
printf("\nSum=%d\n", s);
printf("Multiple=%ld\n", p);
return 0;
}
```

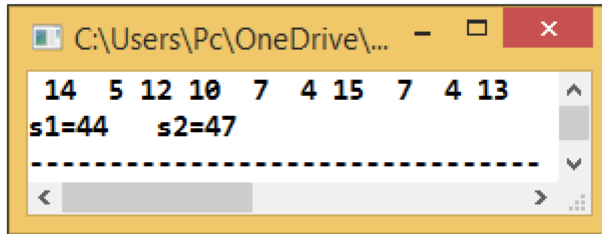


სურ. 101

ახლა კი წარმოვადგინოთ მასივებსა და მიმთითებლებს შორის კავშირის ამსახველი პროგრამა სამომხმარებლო ფუნქციების გამოყენებით, რომლის შესრულების შედეგები 102-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define size 10
void input(int *);
void print(int *);
void function(int *);
int i, s1=0, s2=0;
int main() {
int a[size];
input(a);
print(a);
function(a);
return 0;
}
void input(int *aPtr){
        srand(time(0));
for(i=0; i<size; i++)
        *(aPtr+i)=1+rand()%15;
}
void print(int *aPtr){
        for(i=0; i<size; i++)
```

```
        printf("%3d", *(aPtr+i));
}
void function(int *aPtr){
        for(i=0; i<size; i++){
                if(*(aPtr+i)%2==0)
        s1+=*(aPtr+i);
        else
        s2+=*(aPtr+i);
        }
        printf("\ns1=%d\ts2=%d", s1, s2);
}
```



სურ. 102

ამგვარად, შეგვიძლია შემდეგი დასკვნის გაკეთება:

- მიმთითებელი ცვლადია, რომელშიც შეიძლება სხვა ცვლადის მისამართის შენახვა;
- მიმთითებლის დეკლარირებისას საჭიროა იმ ცვლადების ტიპის მითითება, რომლებზეც ის მიუთითებს, ხოლო მისი სახელის (იდენტიფიკატორის) წინ საჭიროა ვარსკვლავის (*) სიმბოლოს დასმა;
- ცვლადის წინ დაწერილი **&** (ამპერსენდი) ნიშანი მის მისამართს აღნიშნავს;
- მიმთითებლის წინ დაწერილი ვარსკვლავის (*) ნიშანი პროგრამის მუშა ნაწილში (და არა დეკლარირების დროს) აღნიშნავს იმ უჯრის მნიშვნელობას, რომელზეც მიმთითებელი მიუთითებს;
- ნულოვანი მიმთითებლის აღნიშვნის მიზნით **NULL** კონსტანტა გამოიყენება;
- მიმთითებლის კონსოლზე გამოსატანად გამოიყენება ფორმატი **%p**.

8.2. დინამიკური მეხსიერება

ფიქსირებული ზომის (სტატიკური) მასივის შექმნისას, მისთვის კომპიუტერის ოპერატიული მეხსიერების განსაზღვრული უბანი გამოიყოფა.

დავუშვათ, გვაქვს ხუთი ნამდვილრიცხვა ელემენტისგან შემდგარი მასივი:

double numbers[5] = {1.0, 2.0, 3.0, 4.0, 5.0};

აღნიშნული მასივის შესანახად გამოყოფილი მეხსიერების მოცულობა 40 ბაიტია (5*8 =40, სადაც 8 ბაიტი double ტიპის მონაცემის ზომაა). თუმცა, ეს ყოველთვის მოსახერხებელი არ არის. ზოგჯერ აუცილებელია, რომ მასივის ელემენტების რაოდენობა და შესაბამისად, მასივის შესანახად გამოყოფილი მეხსიერების ზომა დინამიურად განისაზღვროს. მაგალითად, როდესაც მომხმარებელს სურს მასივის ზომა პროგრამის შესრულებაზე გაშვების პროცესში მიუთითოს. აღნიშნულ შემთხვევაში, მასივის შესაქმნელად ჩვენ შეგვიძლია მეხსიერების დინამიური გამოყოფა გამოვიყენოთ.

მეხსიერების დინამიკური გამოყოფის მართვის მიზნით მთელი რიგი ფუნქციები გამოიყენება, რომლებიც **stdlib.h** ქუდის ფაილშია განსაზღვრული.

8.2.1. მეხსიერების დინამიკური გამოყოფა C ენაში

მეხსიერების დინამიკური გამოყოფის ფუნქციების გამოყენების მიზნით საჭიროა მიმთითებლის აღწერა, რომელიც მასივის ელემენტების შესანახ საწყის მისამართს წარმოადგენს.

int *p // მიმთითებელი int ტიპზე

სტატიკური მასივის საწყისი მისამართი კომპილატორის მიერ მასივის გამოცხადების (დეკლარირების) დროს განისაზღვრება და მისი შეცვლა დაუშვებელია!

დინამიური მასივის საწყისი მისამართი, პროგრამის შესრულების პროცესში, ენიჭება მასივზე გამოცხადებულ მიმთითებელს.

მეხსიერების დინამიკური გამოყოფის ფუნქციები ოპერატიულ მეხსიერებაში “პოულობენ” საჭირო სიგრძის უწყვეტ უბანს და ახდენენ ამ უბნის საწყისი მისამართის დაბრუნებას. ეს ფუნქციებია:

- **malloc();**

- **calloc();**
- **realloc();**
- **free().**

malloc() ფუნქციის პროტოტიპს შემდეგი სახე აქვს:

void* malloc(მასივის ზომა ბაიტებში);

calloc() ფუნქციის პროტოტიპს შემდეგი სახე აქვს:

void* calloc(ელემენტების რაოდენობა, ელემენტის ზომა ბაიტებში);

აღნიშნული ფუნქციების გამოყენება პროგრამებში ითხოვს **<malloc.h>** ბიბლიოთეკის ჩართვას.

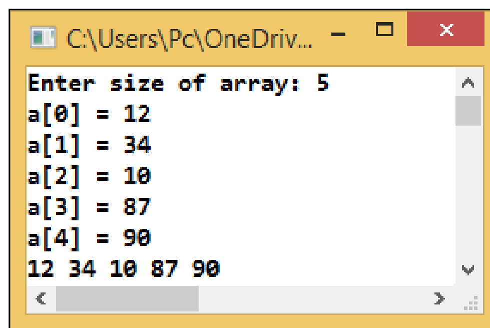
რადგან ორივე ფუნქცია მნიშვნელობის სახით აბრუნებს მიმთითებელს void (ცარიელ) ტიპზე, საჭირო ხდება დასაბრუნებელი შედეგის ტიპის ცხადი სახით გარდასახვა.

malloc() ფუნქციაში არგუმენტის სახით გადაცემული მასივის ზომის ბაიტებში განსაზღვრის მიზნით, საჭიროა მასივში არსებული ელემენტების რაოდენობის ერთი ელემენტის ზომაზე გამრავლება. რადგან მასივის ელემენტების როლში შეიძლება გამოყენებულ იქნას როგორც მონაცემთა მარტივი, ისე შედგენილი ტიპები (მაგალითად, სტრუქტურები), ელემენტის ზომის ზუსტი განსაზღვრის მიზნით, ზოგად შემთხვევაში, რეკომენდებულია **sizeof(ტიპი)** ფუნქციის გამოყენება. ეს უკანასკნელი განსაზღვრავს მითითებული ტიპის ელემენტის მიერ დაკავებულ ბაიტების რაოდენობას.

calloc() და **malloc()** ფუნქციების გამოყენებით დინამიკურად გამოყოფილი მეხსიერება შეგვიძლია გამოვათავისუფლოთ **free(მიმთითებელი)** ფუნქციის საშუალებით. დაპროგრამების კარგ სტილად (ტონად) ითვლება დინამიკურად გამოყოფილი მეხსიერების გამოთავისუფლება იმ შემთხვევისთვის, თუ მისი გამოყენება პროგრამაში აღარ ხორციელდება. თუმცა, აღსანიშნავია, რომ თუ დინამიკურად გამოყოფილი მეხსიერება ცხადი სახით არ თავისუფლდება, პროგრამის შესრულების პროცესის დამთავრებისას ის მაინც გამოთავისუფლდება.

ქვემოთ წარმოდგენილია ერთგანზომილებიანი დინამიკური მასივის ორგანიზებისა და მისი ელემენტების კონსოლური შეტანა/გამოტანის პროგრამა, რომლის შესრულების შედეგები 103-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main()
{
 int *a;
 int i, n;
 printf("Enter size of array: ");
 scanf("%d", &n);
 a = (int*)malloc(n * sizeof(int));
 for (i = 0; i<n; i++)
 {
  printf("a[%d] = ", i);
  scanf("%d", &a[i]);
 }
 for (i = 0; i<n; i++)
  printf("%d ", a[i]);
 free(a);
 getchar();
 return 0;
}
```



სურ. 103

ახლა კი შევადგინოთ ერთგანზომილებიანი დინამიური მასივის ორგანიზებისა და წრფივი ძებნის ალგორითმის შესაბამისი პროგრამა სამომხმარებლო ფუნქციების გამოყენებით. პროგრამის შესრულების შედეგები 104-ე სურათზეა ნაჩვენები.

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

```
void input(int []);
void print(int []);
void search(int []);
int n, i;
int main() {
int *a;
printf("Enter array size: ");
scanf("%d", &n);
a=(int *)malloc(n*sizeof(int));
input(a);
print(a);
search(a);
return 0;
}
void input(int a[]){
        srand(time(0));
for(i=0; i<n; i++)
        a[i]=1+rand()%20;
}
void print(int a[]){
        for(i=0; i<n; i++)
        printf("%d  ", a[i]);
}
void search(int a[]){
        int key, s=0;
        printf("\nEnter key: ");
        scanf("%d", &key);
        for(i=0; i<n; i++)
                if(key==a[i]){
                        printf("\nIndex=%d", i);
                        s++;
                }
                if(s==0)
                 printf("\nNo Solution");
}
```

```
C:\Users\Pc\OneDrive\Doc...
Enter array size: 10
6  17  6  17  14  3  13  16  10  13
Enter key: 13

Index=6
Index=9
--------------------------------
```

სურ. 104

8.2.2. ორგანზომილებიანი მასივებისთვის მეხსიერების დინამიკურად გამოყოფა

ორგანზომილებიანი მასივი (მატრიცა) კომპიუტერის ოპერატიულ მეხსიე-რებაში ერთგვარი ლენტის ფორმით თავსდება, რომელიც სტრიქონებში არსებული ელემენტებისგან შედგება. ამასთან, მატრიცის ნებისმიერი ელემენტის ინდექსი შეგვიძლია გამოვთვალოთ ფორმულით:

index = i*m+j;

სადაც **i** მიმდინარე სტრიქონის ნომერია, **j** - მიმდინარე სვეტის ნომერი, ხოლო **m** - სვეტების რაოდენობა.

განვიხილოთ მატრიცა განზომილებით 3x4. მწვანედ გამოყოფილი ელემენტის ინდექსი (იხილეთ 105-ე სურათი) განისაზღვრება, როგორც: **index = 1*4+2=6.**

მატრიცის შესანახად გამოყოფილი მეხსიერების მოცულობა გამოითვლება ფორმულით: **n*m*(ელემენტის ზომა)**, სადაც n მატრიცის სტრიქონების რაოდენო-ბაა, ხოლო m - სვეტების რაოდენობა.

სურ. 105

მატრიცის ელემენტებზე მიმთითებლის საშუალებით სწორ მიმართვას შემდეგი სახე აქვს: ***(p+i*m+j),** სადაც **p** მიმთითებელია მასივზე, **m** - მატრიცაში სვეტების რაოდენობა, **i** - სტრიქონის ინდექსი და **j** - სვეტის ინდექსი.

ქვემოთ წარმოდგენილია ორგანზომილებიანი დინამიკური მასივის ორგანიზე-ბისა და მისი ელემენტების კონსოლური შეტანა/გამოტანის პროგრამული კოდი, რომლის შესრულების შედეგები 106-ე სურათზეა ნაჩვენები.

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

```
#include <stdlib.h>
int main()
{
  int *a;
  int i, j, n, m;
  printf("Enter number of rows: ");
  scanf("%d", &n);
  printf("Enter number of columns: ");
  scanf("%d", &m);
  a = malloc(n*m * sizeof(int));
  for (i = 0; i<n; i++)  {
    for (j = 0; j<m; j++)  {
      printf("a[%d][%d] = ", i, j);
      scanf("%d", (a + i*m + j));  }}

  for (i = 0; i<n; i++)  {
    for (j = 0; j<m; j++)  {
      printf("%5d ", *(a + i*m + j));  }
    printf("\n"); }
  free(a);
  getchar();
  return 0;
}
```

```
Enter number of rows: 2
Enter number of columns: 3
a[0][0] = 13
a[0][1] = 24
a[0][2] = 35
a[1][0] = 46
a[1][1] = 31
a[1][2] = 21
   13    24    35
   46    31    21
```

სურ. 106

8.2.3. მეხსიერების ხელახალი განაწილება

იმ შემთხვევაში, როდესაც მასივის შესანახად გამოსაყოფი მეხსიერების წინასწარ მითითება არ შეიძლება (მაგალითად, როცა ელემენტების მნიშვნელობების შეტანა არ ხდება საჭირო ბრძანებამდე), მაშინ მასივის ზომის გაზრდის მიზნით შემდეგი მოქმედებები უნდა განვახორციელოთ:

- გამოვყოთ მეხსიერების ბლოკი ზომით $n + 1$ (ერთით მეტი, ვიდრე მასივის მიმდინარე ზომაა);
- მოვახდინოთ მასივში შენახული მნიშვნელობების კოპირება მეხსიერების ხელახლა გამოყოფილ არეში;
- გამოვათავისუფლოთ მასივის შესანახად ადრე გამოყოფილი მეხსიერება;
- მიმთითებელი მასივის საწყის მისამართზე გადავაადგილოთ ხელახლა გამოყოფილი მეხსიერების არეს დასაწყისში;
- დავამატოთ მასივში ბოლოს შეტანილი მნიშვნელობა.

ბოლოს მითითებული მოქმედების გარდა, ყველა დანარჩენ მოქმედებას ასრულებს **realloc()** ფუნქცია, რომლის ჩაწერის სინტაქსი შემდეგია:

void* realloc (void * ptr, size_t size);

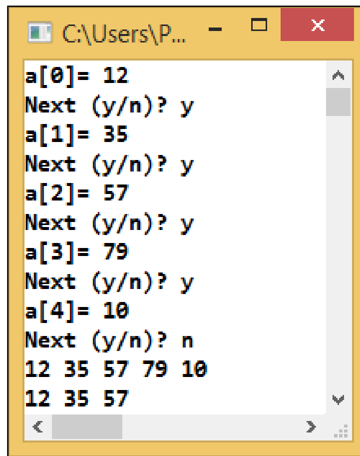
სადაც **ptr** მიმთითებელია **malloc(), calloc()** ან **realloc()** ფუნქციების მიერ ადრე გამოყოფილ მეხსიერების ბლოკზე, რაც ახალ ადგილას უნდა იქნას გადატანილი. თუ ეს პარამეტრი უდრის **NULL**-ს, ახალი ბლოკი გამოიყოფა და ფუნქცია აბრუნებს მასზე მიმთითებელს. **size** - ბლოკისთვის გამოყოფილი მეხსიერების ახალი ზომაა ბაიტებში წარმოდგენილი. თუ **size=0,** ადრე გამოყოფილი მეხსიერება თავისუფლდება და ფუნქცია ნულოვან მიმთითებელს აბრუნებს.

მეხსიერების ბლოკი ზომით შეიძლება გაიზარდოს ან შემცირდეს. მეხსიერების ბლოკის შიგთავსი ინახება მაშინაც კი, თუ ახალ ბლოკს ნაკლები ზომა აქვს, ვიდრე ძველს, თუმცა გადაიყრება ის მონაცემები, რომლებიც ახალი ბლოკის ფარგლებს სცდება.

ქვემოთ წარმოდგენილია **realloc()** ფუნქციის გამოყენების ამსახველი პროგრამული კოდი, რომლის შესრულების შედეგები 107-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <malloc.h>
int main()
{
  int *a = NULL, i = 0, elem, j;
  char c;
  do {
    printf("a[%d]= ", i);
    scanf("%d", &elem);
```

```
    a = (int*)realloc(a, (i + 1) * sizeof(int));
    a[i] = elem;
    i++;
    getchar();
    printf("Next (y/n)? ");
    c = getchar();
  } while (c == 'y');
  for (j = 0; j < i; j++)
    printf("%d ", a[j]);
  if (i>2) i -= 2;
  printf("\n");
  a = (int*)realloc(a, i * sizeof(int));
  for (j = 0; j < i; j++)
    printf("%d ", a[j]);
  getchar();
  return 0;
}
```



სურ. 107

8.3. მიმთითებლები სტრუქტურებზე

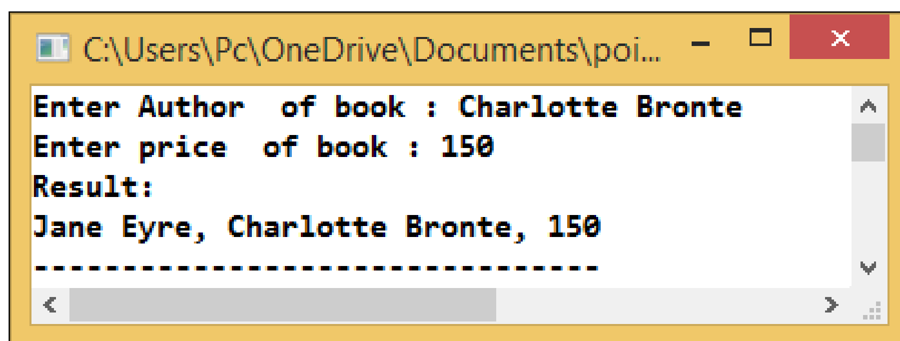
სტრუქტურის ელემენტებზე წვდომა მიმთითებლებით შესაძლებელია. ამისათვის აუცილებელია მიმთითებლის სტრუქტურის მისამართით ინიციალება.

მიმთითებლის საშუალებით სტრუქტურის ველებზე მიმართვისთვის ისრის (->) ოპერატორი გამოიყენება. ასე, მაგალითად: **მიმთითებელი -> ველი;** ან **(*მიმთითებელი). ველი;**

მიმთითებლის ქვეშ აქ იგულისხმება მიმთითებელი სტრუქტურაზე, ხოლო ველის ქვეშ - სტრუქტურის ველი.

ქვემოთ წარმოდგენილია სტრუქტურებზე მიმთითებლების გამოყენების ამსახველი პროგრამული კოდი, რომლის შესრულების შედეგები 108-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
struct book
{
  char title[15];
  char author[15];
  int value;
};
int main()
{

  struct book *lib;
  lib = (struct book*)malloc(3 * sizeof(struct book));
   printf("Enter name  of book : ");
    gets(lib->title);
    printf("Enter Author  of book : ");
    gets(lib->author);
    printf("Enter price  of book : ");
   scanf("%d", &(lib)->value);
  printf("Result:\n%s, %s, %d", lib->title, lib->author, lib->value);
  return 0;
}
```



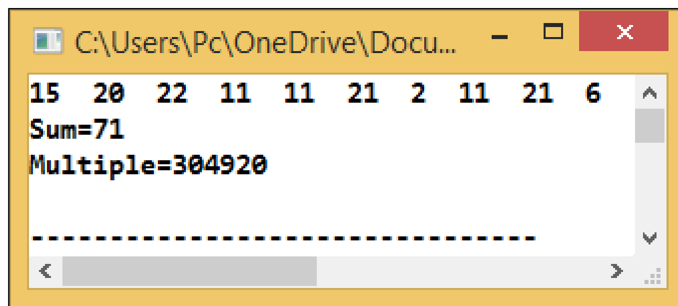
სურ. 108

ამოცანა 33.

მიმთითებლების გამოყენებით შევადგონეთ პროგრამა, რომელიც int a[10] მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ მნიშვნელობებს; გამოთვლის მოცემული მასივის ლუწინდექსიანი ელემენტების ჯამს და კენტინდექსიანი ელემენტების ნამრავლს.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
const int size=10;
int a[size], i;
int s=0;
long p=1;
int *aPtr;
aPtr=a;
srand(time(0));
for(i=0; i<size; i++){
        *(aPtr+i)=1+rand()%25;
        printf("%d  ",*(aPtr+i));
        if(i%2==0)
        s+=*(aPtr+i);
        else
        p*=*(aPtr+i);   }
printf("\nSum=%d\n", s);
printf("Multiple=%ld\n", p);
return 0;
}
```

ზემოთ წარმოდგენილი პროგრამული კოდის შესრულების შედეგები 109-ე სურათზეა ნაჩვენები.



სურ. 109

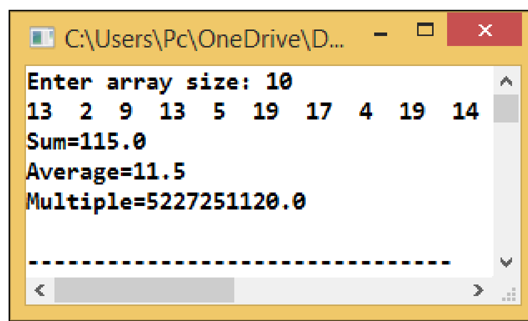
ამოცანა 34.

შევადგინოთ პროგრამა, რომელიც მეხსიერების დინამიური დანაწილების გათვალისწინებით მოახდენს ერთგანზომილებიანი მასივის ორგანიზებას: მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ შემთხვევით მნიშვნელობებს, გამოთვლის მასივის ელემენტების ჯამს, ნამრავლს და საშუალო არითმეტიკულ მნიშვნელობებს, მასივთან ერთად კონსოლზე გამოიტანს მიღებულ შედეგებს და პროგრამის დასასრულს მოახდენს მეხსიერების დინამიურ გამოთავისუფლებას.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
void input(int []);
void print(int []);
void function(int []);
int n, i;
double s=0, p=1;
int main() {
int *a;
printf("Enter array size: ");
scanf("%d", &n);
a=(int *)malloc(n*sizeof(int));
input(a);
print(a);
function(a);
free(a);
return 0;
}
void input(int a[]){
        srand(time(0));
for(i=0; i<n; i++)
        a[i]=1+rand()%20;
}
void print(int a[]){
        for(i=0; i<n; i++)
        printf("%d  ", a[i]);
}
void function(int a[]){
```

```
	for(i=0; i<n; i++){
		s+=a[i];
		p*=a[i];
	}
	printf("\nSum=%.1lf\n", s);
	printf("Average=%.1lf\n", s/n);
	printf("Multiple=%.1lf\n", p);
}
```

დასმული ამოცანის გადაწყვეტის პროგრამული კოდის შესრულების შედეგები 110-ე სურათზეა ნაჩვენები.



სურ. 110

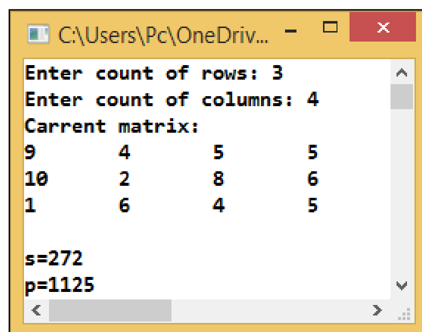
ამოცანა 35.

შევადგინოთ პროგრამა, რომელიც მეხსიერების დინამიური დანაწილების გათვალისწინებით მოახდენს ორგანზომილებიანი მასივის ორგანიზებას: მასივის ელემენტებს მიანიჭებს მთელი ტიპის ნებისმიერ შემთხვევით მნიშვნელობებს, ცალ-ცალკე გამოთვლის მასივის ლუწი მნიშვნელობის ელემენტების კვადრატების ჯამს და კენტი მნიშვნელობის ელემენტების ნამრავლს, მიღებულ შედეგებს შეადარებს ერთმანეთს, მასივთან ერთად კონსოლზე გამოიტანს ყველა შედეგს და პროგრამის დასასრულს მოახდენს მეხსიერების დინამიურ გამოთავისუფლებას.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int main() {
	int *a;
	int i, j, n, m, t;
	double s=0, p=1;
	printf("Enter count of rows: ");
	scanf("%d", &n);
```

```
printf("Enter count of columns: ");
scanf("%d", &m);
a=(int*)malloc(n*m*sizeof(int));
srand(time(0));
printf("Carrent matrix:\n");
for(i=0; i<n; i++){
        for(j=0; j<m; j++){
         t=a + i*m + j;
         t=1+rand()%10;
         if(t%2==0)
         s+=t*t;
         else
         p*=t;
         printf("%d\t", t); }
        printf("\n"); }
printf("\ns=%.0lf\n", s);
printf("p=%.0lf\n", p);
if(s>p)
printf("s>p");
else if(s<p)
printf("s<p");
else
printf("s=p");
free(a);
getchar();
return 0; }
```

დასმული ამოცანის გადაწყვეტის პროგრამული კოდის შესრულების შედეგები 111-ე სურათზეა ნაჩვენები.



სურ. 111

IX თავი
ნაკადები და ფაილები

C ენაში მონაცემთა შეტანა/გამოტანის სისტემის ფუნდამენტს ნაკადებისა და ფაილების კონცეფციები წარმოადგენს. აღნიშნული სისტემა პროგრამისტსა და აპარატურას შორის აბსტრაქციის დონეს ასახავს. ამ აბსტრაქციას **ნაკადი** ეწოდება, ხოლო თავად მოწყობილობას - **ფაილი**. მნიშვნელოვანია, ვიცოდეთ თუ როგორ ხდება მათი ურთიერთქმედება.

C ენაში მონაცემთა შეტანა/გამოტანის სისტემა განკუთვნილია სხვადასხვა მოწყობილობებთან (დისკური მოწყობილობები, ტერმინალი და სხვ.) სამუშაოდ. მიუხედავად იმისა, რომ ეს მოწყობილობები ერთმანეთისგან განსხვავებულია, სისტემა მათ გარდაქმნის ერთიან ლოგიკურ მოწყობილობად - ნაკადად.

ნაკადების ქცევა ერთიმეორის მსგავსია. რადგან ნაკადები არ არის დამოკიდებული მოწყობილობებზე, ამიტომ ინფორმაციის ჩასაწერად როგორც ფაილში (დისკზე), ისე სხვა მოწყობილობაზეც, ერთი და იგივე ფუნქციები შეგვიძლია გამოვიყენოთ.

9.1.1. ნაკადების ტიპები

ნაკადების ორი ტიპი არსებობს: ტექსტური და ორობითი (ბინარული).

ტექსტური ნაკადები - სიმბოლოების მიმდევრობაა. ტექსტურ ნაკადებში ზოგიერთი სიმბოლო შეიძლება გარდაიქმნას გარემოს მოთხოვნების შესაბამისად. ამიტომ, შესაძლებელია არ არსებობდეს ერთმნიშვნელოვანი შესაბამისობა ჩაწერილ ან წაკითხულ სიმბოლოებსა და გარე მოწყობილობის სიმბოლოებს შორის. ასევე, შესაძლოა წაკითხული ან ჩაწერილი სიმბოლოების რაოდენობა არ ემთხვეოდეს გარე მოწყობილობის სიმბოლოების რაოდენობას.

ორობითი ნაკადი - ბაიტების თანმიმდევრობაა, რომელიც სრულ შესაბამისობაშია გარე მოწყობილობის ბაიტებთან. ე.ი. ამ შემთხვევაში, ადგილი არ აქვს სიმბოლოების გარდაქმნას. თუმცა, ნულოვანი ბაიტების გარკვეული რაოდენობა შეიძლება დაემატოს ორობით ნაკადს. ეს ნულოვანი ბაიტები კი შეიძლება გამოყე-

ნებულ იქნას ინფორმაციის წარმოდგენის უნიფიცირებისთვის, მაგალითად, ლოგიკური დისკის სექტორის შესავსებად.

9.1.2. ფაილები

C ენაში ფაილები ლოგიკური კონცეფციაა, რომელიც დისკური ფაილებიდან დაწყებული და ტერმინალით დამთავრებული, ყველგან გამოიყენება.

ნაკადი კონკრეტულ ფაილს უკავშირდება ფაილის გახსნის ოპერაციით. თუ ფაილი გახსნილია, ე.ი. შესაძლებელია ფაილსა და პროგრამას შორის ინფორმაციის გაცვლის განხორციელება. ყველა ფაილი ერთნაირი შესაძლებლობების მქონე არ არის. მაგალითად, დისკურ ფაილზე წვდომა თავისუფალია, მაგრამ მოდემზე - არა. ეს ასახავს C ენაში მონაცემთა შეტანა/გამოტანის სისტემის მნიშვნელოვან ასპექტს: ყველა ნაკადი ერთნაირია, მაგრამ ფაილები - არა. დისკზე წვდომის უმცირესი ნაწილი არის სექტორი. ინფორმაცია იწერება ან იკითხება დისკიდან სექტორების მიხედვით. ამიტომ, მაშინაც კი, თუ პროგრამას მონაცემების მხოლოდ 1 ბაიტი სჭირდება, მაინც მთელი სექტორი იქნება წაკითხული. ეს მონაცემები ბუფერში (მეხსიერების უბანში) თავსდება მანამდე, სანამ მათ პროგრამა არ გამოიყენებს. მონაცემების ფაილში შეტანის დროს ადგილი აქვს „ბუფერიზაციის" პროცესს, რომელიც გრძელდება, სანამ ინფორმაციის მთელი სექტორი არ დაგროვდება. მხოლოდ ამის შემდეგ არის შესაძლებელი მონაცემების ფიზიკურად ფაილში ჩაწერა.

ნაკადის კავშირი ფაილთან ნადგურდება ნაკადის დახურვის ოპერაციით. ნაკადის დახურვა იწვევს ბუფერის მთელი შიგთავსის გარე მოწყობილობაზე გადაგზავნას. ამ პროცესს ბუფერის გასუფთავებას უწოდებენ და ის უზრუნველ-ყოფს, რომ ბუფერში ინფორმაცია არ დარჩეს. როდესაც პროგრამის მუშაობა ნორმალურად მთავრდება, ყველა ფაილი ავტომატურად იხურება. თუმცა, უმჯო-ბესია ფაილები თავად დავხუროთ **fclose()** ფუნქციის გამოყენებით, განსაკუთ-რებით მაშინ, როდესაც ფაილი აღარ არის საჭირო, რადგან ზოგიერთმა მოვლენამ შესაძლოა, ბუფერის დისკზე ჩაწერას ხელი შეუშალოს. მაგალითად, ფაილი არ

იწერება, თუ პროგრამა მთავრდება **abort()** ფუნქციის გამოძახებით, ან თუ მომხმარებელი პროგრამის დასრულებამდე გამორთავს კომპიუტერს.

9.1.3. შეტანა/გამოტანის სტანდარტული მოწყობილობები

პროგრამის დასაწყისში წინასწარ განსაზღვრული ხუთი ტექსტური ნაკადია გახსნილი, კერძოდ: stdin, stdout, stderr, stdaux და stdprn. ისინი შეესაბამება მე-16 ცხრილში წარმოდგენილ შეტანა/გამოტანის სტანდარტულ მოწყობილობებს.

ცხრილი 16. შეტანა/გამოტანის სტანდარტულ მოწყობილობები

ნაკადი	მოწყობილობა
stdin	კლავიატურა
stdout	ეკრანი
stderr	ეკრანი
stdaux	პირველი მიმდევრობითი პორტი
stdprn	პრინტერი

პირველი სამი ნაკადი განსაზღვრულია ANSI C სტანდარტით და ნებისმიერი კოდი, რომელიც მათ იყენებს, სრულად გადატანითია. ბოლო ორი განსაზღვრულია ბორლანდის მიერ და ისინი არ ხასიათდება სხვა კომპილატორებზე გადატანითობის თვისებით. ოპერაციული სისტემების უმეტესობა იძლევა შეტანა/გამოტანის გადამისამართების შესაძლებლობას, ამიტომ ქვეპროგრამები, რომლებიც კითხულობენ ან წერენ ამ ნაკადებში, შეიძლება სხვა მოწყობილობებზე გადამისამართდეს. გადამისამართება არის პროცესი, როდესაც ინფორმაცია, რომელიც ჩვეულებრივ გადადის ერთ მოწყობილობაზე, გადამისამართდება სხვა მოწყობილობაზე ოპერაციული სისტემის მიერ. არასოდეს არ უნდა გახსნათ ან დახუროთ ეს ფაილები ცხადი სახით!

9.1.4. ფაილური სისტემა ANSI C

ANSI C ფაილური სისტემა არის შეტანა/გამოტანის (I/O) სისტემის ნაწილი, რომელიც საშუალებას გვაძლევს ინფორმაცია წავიკითხოთ ან ჩავწეროთ ფაილებ-ში. იგი შეიცავს რამდენიმე ურთიერთდაკავშირებულ ფუნქციას. **stdio.h** სათაურის ფაილი ჩართული უნდა იყოს ნებისმიერ პროგრამაში, რომელიც ამ ფუნქციებს იყე-ნებს.

stdio.h სათაურის ფაილი განსაზღვრავს რამდენიმე მაკროსს, რომელთა შორის მნიშვნელოვანია შემდეგი მაკროსები: **NULL**, **EOF**, **FOPEN_MAX**, **SEEK_SET**, **SEEK_CUR** და **SEEK_END**. მაკროგანსაზღვრა **NULL** არის ნულოვანი მიმთითებელი. **EOF** მაკროსი, სტანდარტულად, განისაზღვრება როგორც -1 და მისი მნიშვნელობა ბრუნდება, როდესაც შეტანის ფუნქცია ცდილობს წაიკითხოს ფაილის ბოლო. **FOPEN_MAX** მაკროსი მთელრიცხვა მნიშვნელობაა, რომელიც განსაზღვრავს ერთდროულად გახსნილი ფაილების მაქსიმალურ რაოდენობას. სხვა მაკროსები გამოიყენება **fseek()** ფუნქციასთან ერთად, რომელიც ფაილზე თავისუფალ წვდომას უზრუნველყოფს.

მე-17 ცხრილში წარმოდგენილია **ANSI C** ფაილური სისტემის ტიპიური ფუნქციები.

ცხრილი 17. **ANSI C** ფაილური სისტემის ტიპიური ფუნქციები

ფუნქციის სახელი	დანიშნულება
open()	ფაილის გახსნა
fclose()	ფაილის დახურვა
putc()	სიმბოლოს ჩაწერა ფაილში
fputc()	**putc()**-ს ანალოგია
getc()	ფაილიდან სიმბოლოს კითხვა
fgetc()	**getc()**-ს ანალოგია

fseek()	მითითებულ ბაიტზე გადასვლა ფაილში
fprintf()	იგივე მოქმედება ფაილში, რაც **printf()**-ის მოქმედება კონსოლზე
fscanf()	იგივე მოქმედება ფაილთან, რაც **scanf()**-ის მოქმედება კონ-სოლთან
feof()	**True** მნიშვნელობის დაბრუნება ფაილის ბოლოში გასვლი-სას
ferror()	**True** მნიშვნელობის დაბრუნება შეცდომის აღმოჩენის დროს
rewind()	ფაილის პოზიციის ინდიკატორის ჩამოგდება ფაილის და-საწყისში
remove()	ფაილის წაშლა
flush()	ბუფერის გასუფთავება

9.1.5. მიმთითებელი ფაილზე. ფაილის გახსნის რეჟიმები

მიმთითებელი ფაილზე არის მიმთითებელი ინფორმაციაზე, რომელიც გან-საზღვრავს ფაილის სხვადასხვა პარამეტრებს, მათ შორის მის სახელს, მდგომა-რეობას და მიმდინარე პოზიციას. მიმთითებელი ფაილზე კონკრეტული დისკის ფაილის იდენტიფიცირებას ახდენს და გამოიყენება ნაკადის მიერ შეტა-ნა/გამოტანის ოპერაციების შესასრულებლად. მიმთითებელი ფაილზე არის **FILE** ტიპის ცვლადი-მიმთითებელი. ფაილების წასაკითხად ან ჩასაწერად პროგრამამ უნდა გამოიყენოს მიმთითებლები ფაილებზე. ფაილური ცვლადი-მიმთითებლის შესაქმნელად გამოიყენება ოპერატორი: **FILE *fp;**

ფაილის გახსნის დროს, მას შეტანა-გამოტანის ნაკადი უკავშირდება, ხოლო ამ უკანასკნელს - **FILE** ტიპის სტანდარტული სტრუქტურა, რომელიც **stdio.h** ფაილშია განსაზღვრული. **FILE** სტრუქტურა ფაილის შესახებ საჭირო ინფორმაციას მოიცავს.

ფაილის გახსნა **fopen()** ფუნქციით ხდება. ის **FILE** ტიპის სტრუქტურაზე აბრუნებს მიმთითებელს. ფუნქციის ჩაწერის სინტაქსი შემდეგია:

FILE *fopen(name, type);

name გასახსნელი ფაილის სახელია (გზის ჩათვლით), ხოლო **type** მიმთითებელია სიმბოლოების სტრიქონზე, რომელიც ფაილზე წვდომის საშუალებას - ფაილის გახსნის რეჟიმს განსაზღვრავს.

C ენაში ფაილის გახსნის შემდეგი რეჟიმები გამოიყენება:

- "**r**" - ხსნის ტექსტურ ფაილს კითხვისთვის (ამ დროს ფაილი უნდა არსებობდეს);
- "**w**" - ხსნის ცარიელ ტექსტურ ფაილს ჩაწერის მიზნით. თუ ფაილი არსებობს, მისი შიგთავსი იშლება;
- "**a**" - ხსნის ტექსტურ ფაილს მონაცემების ჩამატების მიზნით. თუ ფაილი არ არსებობს, ის იქმნება;
- "**r+**" - ხსნის ტექსტურ ფაილს ჩაწერისა და წაკითხვისთვის (ამ დროს ფაილი უნდა არსებობდეს). ჩაწერა შესაძლებელია ფაილის ნებისმიერ ადგილას ფაილის ბოლოს გარდა. ამ დროს ფაილის ზომის გაზრდა დაუშვებელია.
- "**w+**" - ხსნის ცარიელ ფაილს ჩაწერისა და წაკითხვის მიზნით. თუ ფაილი არსებობს, მისი შიგთავსი იშლება;
- "**a+**" - ხსნის ფაილს წაკითხვისა და მონაცემების ჩამატების მიზნით. თუ ფაილი არ არსებობს, ის იქმნება. მონაცემების ჩაწერა აქ ფაილის ბოლოშიც არის შესაძლებელი.
- **wb"** - ბინარულ ფაილს ხსნის მონაცემების ჩაწერის მიზნით;
- **"rb"** - ბინარულ ფაილს ხსნის მონაცემების წაკითხვის მიზნით;
- **"ab"** - ბინარულ ფაილს ხსნის მონაცემების ჩამატების მიზნით;
- **"w+b"** - ბინარულ ფაილს ხსნის მონაცემების ჩაწერის/წაკითვის მიზნით;
- **"r+b"** - ბინარულ ფაილს ხსნის მონაცემების წაკითხვის/ჩაწერის მიზნით;
- **"a+b"** - ბინარულ ფაილს ხსნის მონაცემების წაკითხვის/ჩამატების მიზნით.

fclose() ფუნქცია ემსახურება იმ ნაკადის ან ნაკადების დახურვას, რომელიც დაკავშირებულია **fopen()** ფუნქციით გახსნილ ფაილებთან. დასახური ნაკადი **fclose()** ფუნქციის არგუმენტით განისაზღვრება. ფუნქციის მიერ დაბრუნებული

ნულის ტოლი მნიშვნელობა ნიშნავს, რომ ნაკადი წარმატებით დაიხურა, ხოლო **EOF** მუდმივა შეცდომაზე მიუთითებს.

9.1.6. შეცდომების დამუშავება

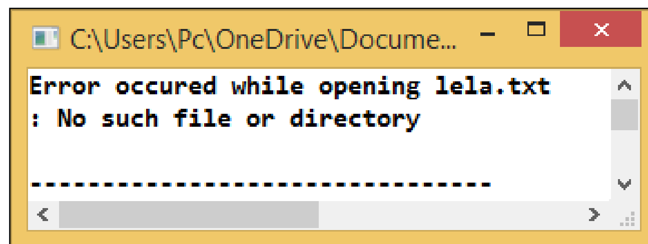
ფაილის გახსნის ან შექმნის პროცესში შესაძლოა შეცდომებმა იჩინოს თავი. მაგალითად მაშინ, როდესაც ფაილის წაკითხვის რეჟიმში გახსნისას აღმოჩნდება, რომ ფაილი არ არსებობს ან ფაილის შესანახად მეხსიერების არასაკმარისი მოცულობაა გამოყოფილი და ა.შ. შეცდომის წარმოქმნის შემთხვევაში **fopen()** ფუნქცია **NULL** მნიშვნელობას აბრუნებს. ფუნქციის შედეგის შემოწმებით ჩვენ შეგვიძლია შეცდომის დამუშავება მოვახდინოთ.

შეცდომის კონსოლზე გამოსატანად **perror()** ფუნქცია გამოიყენება, ხოლო პროგრამის მუშაობის დასრულებას ამ დროს **exit(0)** ფუნქცია ემსახურება.

აღნიშნულის საილუსტრაციოდ შემდეგი პროგრამული კოდი განვიხილოთ:

```
#include <stdio.h>
 int main()
{
   FILE *f;
   if((f=fopen("D:\lela.txt", "r"))==NULL)
   {
      perror("Error occured while opening lela.txt\n");
      exit(0);
   }
   fclose(f);
   return 0;  }
```

ზემოთ წარმოდგენილი პროგრამული კოდის შესრულების შედეგი 112-ე სურათზეა ნაჩვენები.



სურ.112

9.1.7. ტექსტური ფაილების წაკითხვა და ჩაწერა

fputs() ფუნქცია ფაილში ახდენს სტრიქონის ჩაწერას. ეს უკანასკნელი განიხილება, როგორც სიმბოლოთა ნაკრები, რომელიც **'\0'** სიმბოლოთი ბოლოვდება. ფუნქციის პროტოტიპს შემდეგი სახე აქვს:

int fputs(const char *s, FILE *stream);

სადაც პირველი პარამეტრი ჩასაწერი სტრიქონია, ხოლო მეორე - მიმთითებელი ფაილურ ნაკადზე. შედეგის სახით ფუნქცია არაუარყოფით მთელ რიცხვს აბრუნებს. ჩაწერის პროცესში შეცდომის წარმოქმნის შემთხვევაში **EOF** მნიშვნელობა ბრუნდება. სტრიქონის ჩაწერის დროს მისი დამაბოლოვებელი **'\0'** სიმბოლო ფაილში არ იწერება.

fgets() ფუნქცია ფაილიდან სტრიქონის წასაკითხად გამოიყენება. მის პროტოტიპს შემდეგი სახე აქვს:

char * fgets(char *s, int n, FILE *sream);

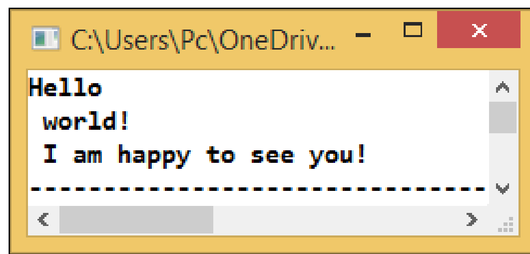
სადაც პირველი პარამეტრი **char *s** წასაკითხი სტრიქონია, მეორე პარამეტრი **int n** მიუთითებს წასაკითხი სიმბოლოების რაოდენობაზე, ხოლო მესამე პარამეტრი **FILE *sream** ფაილური ნაკადია, საიდანაც მონაცემები იკითხება.

გამოძახების დროს ფუნქცია ფაილიდან კითხულობს არაუმეტეს $n-1$ სიმბოლოს. ის კითხვას ასრულებს, როდესაც "წაიკითხავს" $n-1$ სიმბოლოს ან ხვდება ახალ სტრიქონზე გადასვლის მმართველ სიმბოლოს (\n). ყველა წაკითხული სიმბოლო **s** სტრიქონში იწერება, მათ შორის **\n** სიმბოლოც და ყოველი სტრიქონის ბოლოს იწერება \0 სიმბოლო. წარმატებით დასრულების შემთხვევაში, ფუნქცია აბრუნებს **s** მიმთითებელს, ხოლო შეცდომის ან ფაილის დასასრულზე გასვლის შემთხვევაში **NULL** მნიშვნელობას.

ქვემოთ წარმოდგენილია ტექსტურ ფაილში მონაცემების ჩაწერისა და წაკითხვის საილუსტრაციო პროგრამა, რომლის შესრულების შედეგი 113-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main()
{
   char * message = "Hello \n world!\n I am happy to see you!";
   char * filename = "E://lela1.txt";
```

```
char c[256];
FILE *f;
// write to file
if((f= fopen(filename, "w"))==NULL)
{
   perror("Error occured while opening file");
   return 1;  }
// write string
fputs(message, f);
fclose(f);
// read from file
if((f= fopen(filename, "r"))==NULL)
{
   perror("Error occured while opening file");
   return 1;  }
while((fgets(c, 256, f))!=NULL)
{
   printf("%s", c); }

fclose(f);
return 0;  }
```



სურ. 113

9.1.8. სიმბოლოების წაკითხვა და ჩაწერა ფაილში

ბინარულ ფაილებთან მუშაობა სიმბოლური წაკითხვა-ჩაწერის გზით ხდება. სიმბოლოს წასაკითხად **getc()** ფუნქცია გამოიყენება. მას შემდეგი პროტოტიპი აქვს: **int getc(FILE *stream);**

პარამეტრის როლში ფუნქციას ფაილურ ნაკადზე მიმთითებელი გადაეცემა, ხოლო ფუნქციის დასაბრუნებელ მნიშვნელობას ფაილიდან წაკითხული სიმბოლო, უფრო სწორად, მისი შესაბამისი რიცხვითი კოდი წარმოადგენს.

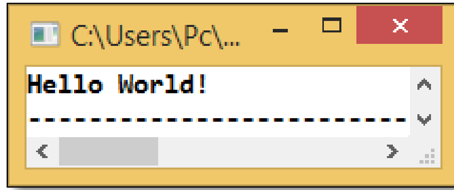
ფაილში სიმბოლოს ჩასაწერად **putc()** ფუნქცია გამოიყენება, რომელსაც შემდეგი პროტოტიპი აქვს: **int putc(int c, FILE *stream);**

პარამეტრების როლში ფუნქციას გადაეცემა მიმთითებელი ფაილურ ნაკადზე და ფაილში ჩასაწერი სიმბოლო. ფუნქციის შედეგს ჩაწერილი სიმბოლო წარმოადგენს.

ქვემოთ წარმოდგენილია სიმბოლური მასივის ჩაწერისა და წაკითხვის საილუსტრაციო პროგრამა, რომლის შესრულების შედეგი 114-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int write(char * filename, char message[], int n);
int read(char * filename);
int i;
int main() {
   char hi[] = "Hello World!";
   char * filename = "E://lela2.txt";
   int n = sizeof(hi)/sizeof(hi[0]);
   write(filename, hi, n);
   read(filename);
   return 0;
}
int write(char * filename, char message[], int n) {
   FILE * f;
   if((f= fopen(filename, "w"))==NULL) {
      perror("Error occured while opening file");
      return 1;
}
for(i=0; i<n; i++) {  // write to file
      putc(message[i], f);
}
   fclose(f);
   return 0;
}
int read(char * filename) {
   FILE * f;
   char sym;
   if((f= fopen(filename, "r"))==NULL) {
      perror("Error occured while opening file");
      return 1; }
   while((sym=getc(f))!= EOF)   // read from file
   {
     printf("%c", sym);
```

```
  }
  fclose(f);
return 0;
}
```



სურ. 114

9.2. ბინარული ფაილები

ტექსტური ფაილები მონაცემებს ტექსტის სახით ინახავს. ეს ნიშნავს, რომ თუ ფაილში ჩვენ ჩავწერთ მთელ რიცხვს, მაგალითად 12345, ჩაიწერება ხუთი სიმბოლო, რაც მონაცემთა ხუთ ბაიტს წარმოადგენს, მიუხედავად იმისა, რომ რიცხვი მთელი ტიპის მონაცემის საზღვრებს არ სცდება.

ტექსტური ფაილები საშუალებას გვაძლევს, ინფორმაცია ადამიანისთვის გასაგები სახით შევინახოთ, თუმცა შესაძლებელია მონაცემების ბინარული სახით შენახვაც. სწორედ ამ მიზნით გამოიყენება ბინარული ფაილები.

ფაილში ჩაწერა ხორციელდება **fwrite()** ფუნქციით, რომლის პროტოტიპს შემდეგი სახე აქვს:

size_t fwrite (const void * ptr, size_t size, size_t count, FILE * stream);

ფუნქცია აბრუნებს ფაილში წარმატებით ჩაწერილი ელემენტების რაოდენობას. არგუმენტების სახით იღებს: მიმთითებელს მასივზე, ერთი ელემენტის ზომას, ელემენტების რაოდენობას და მიმთითებელს ფაილურ ნაკადზე.

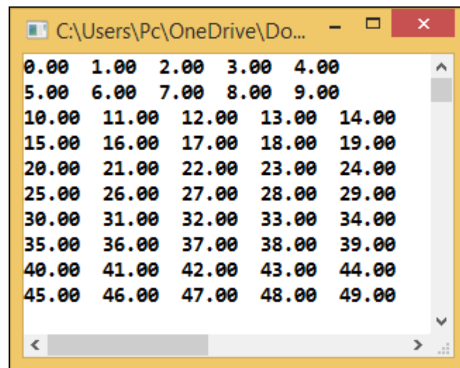
ფაილიდან წაკითხვა ხორციელდება **fread()** ფუნქციით, რომლის პროტოტიპს შემდეგი სახე აქვს:

size_t fread (void * ptr, size_t size, size_t count, FILE * stream);

ფუნქცია აბრუნებს წარმატებით წაკითხული ელემენტების რაოდენობას და ათავსებს მას **ptr** მისამართზე. წაკითხვა ხდება **count** რაოდენობის ელემენტის.

ქვემოთ წარმოდგენილია **fread()** და **fwrite()** ფუნქციების გამოყენების საილუს-ტრაციო პროგრამა, რომლის შესრულების შედეგები 115-ე სურათზეა ნაჩვენები.

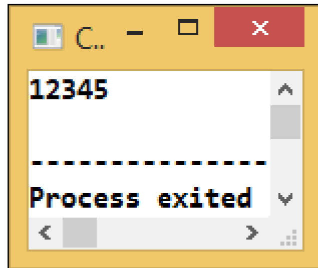
```
#include <stdio.h>
int main(void)
{ int i;
FILE *fp;
float balance[50]; int n=0;
/* Open to wb*/
if((fp=fopen("E://balance", "wb"))==NULL) {
printf("Cannot open file.");
return 1;  }
for(i=0; i<50; i++) balance[i] = (float) i;
/* Save array balance */
fwrite(balance, sizeof balance, 1, fp) ;
fclose(fp);
/* Zero values*/
for(i=0; i<50; i++) balance[i] = 0.0;
/* Open to rb */
if((fp=fopen("E://balance","rb"))==NULL) {
printf("cannot open file");
return 1; }
/*Read array balance */
fread(balance, sizeof balance, 1, fp);
/* Output */
for(i=0; i<50; i++){
printf("%.2f  ", balance [i]);
n++;
if(n%5==0)
printf("\n");
}
fclose(fp);
return 0; }
```



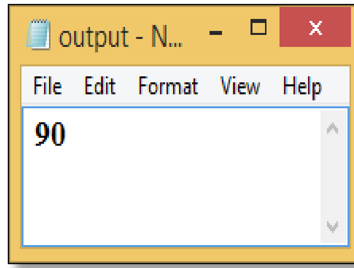
სურ. 115

ახლა ვნახოთ ბინარულ ფაილში მონაცემების ჩაწერის საილუსტრაციო პროგრამა, რომლის შედეგები 116(ა,ბ)-ე სურათებზეა წარმოდგენილი.

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define ERROR_FILE_OPEN -3
void main() {
   FILE *output = NULL;
   int number;
   output = fopen("E:/output.bin", "wb");
   if (output == NULL) {
      printf("Error opening file");
      getch();
      exit(ERROR_FILE_OPEN);
   }
   scanf("%d", &number);
   fwrite(&number, sizeof(int), 1, output);
   fclose(output);
}
```



სურ. 116ა

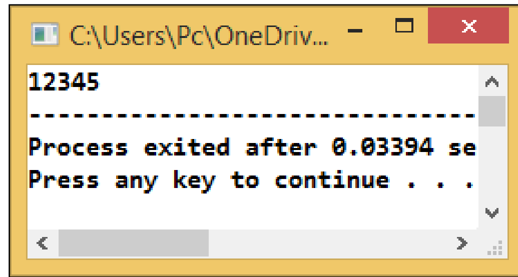


სურ. 116ბ

ახლა მოვახდინოთ ზემოთ შექმნილი ბინარული ფაილის წაკითხვა და ინფორმაციის კონსოლზე გამოტანა. შედეგი 117-ე სურათზეა ნაჩვენები.

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define ERROR_FILE_OPEN -3
void main() {
   FILE *input = NULL;
   int number;
   input = fopen("E:/output.bin", "rb");
   if (input == NULL) {
      printf("Error opening file");
```

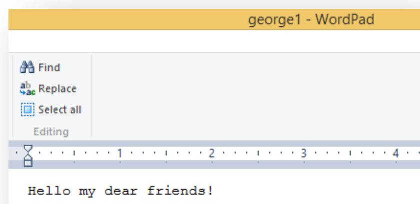
```
        getch();
        exit(ERROR_FILE_OPEN);
    }
    fread(&number, sizeof(int), 1, input);
    printf("%d", number);
    fclose(input);
}
```



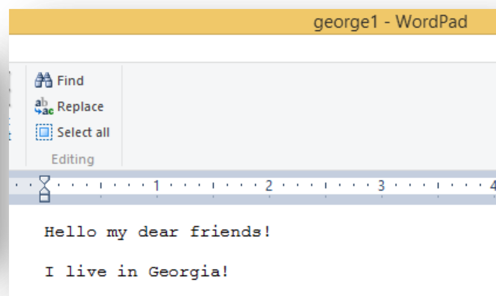
სურ. 117

ქვემოთ წარმოდგენილია ტექსტურ ფაილში მონაცემების ჩაწერის მარტივი საილუსტრაციო პროგრამა, რომლის შესრულების შედეგები 118(ა,ბ)-ესურათებზეა ნაჩვენები.

```
#include <stdio.h>
int main(){
FILE *f=fopen("E://george.txt","w");
fprintf(f, "Hello my dear friends!\n");
fclose(f);
return 0;
}

#include <stdio.h>
int main(){
FILE *f=fopen("E://george.txt","a");
fprintf(f, "\nI live in Georgia!\n");
fclose(f);
return 0;
}
```



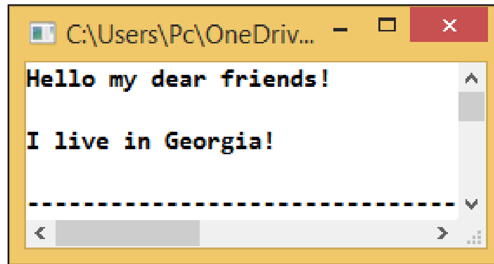
სურ. 118ა



სურ. 118ბ

ამჯერად, უფრო მარტივად მოვახდინოთ ტექსტურ ფაილში ჩაწერილი ინფორმაციის წაკითხვა და კონსოლზე გამოტანა. ამ მიზნით, ქვემოთ წარმოდგენილ პროგრამულ კოდს მივმართოთ, რომლის შესრულების შედეგი 119-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int main(){
char array[256];
FILE *f=fopen("E://george1.txt","r");
while((fgets(array, 256,f))!=NULL)
printf("%s", array);
fclose(f);
return 0;
}
```



სურ. 119

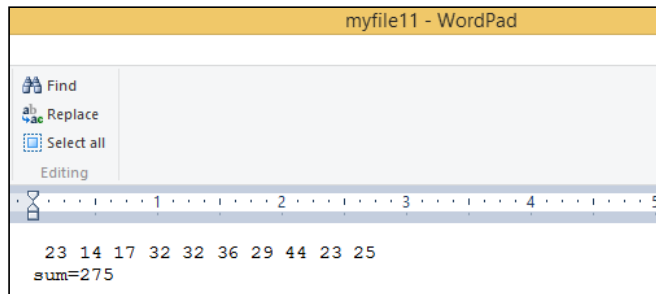
ამოცანა 36.

შევადგინოთ პროგრამა, რომელიც 10-ელემენტიანი მთელრიცხვა მასივის ელემენტებს [10;50] ინტერვალიდან მიანიჭებს თანაბარი ალბათობით განაწილებულ კვაზიშემთხვევით მნიშვნელობებს და გამოთვლის მასივის ელემენტების ჯამს. საწყის მასივს და გამოთვლილ მნიშვნელობას კი ტექსტურ ფაილში გამოიტანს.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define size 10
int main() {
int a[size], i, s=0;
srand(time(0));
for(i=0; i<size; i++){
a[i]=10+rand()%41;
  s+=a[i];
}
```

```
FILE *f;
f=fopen("E://myfile11.txt", "w");
 if(f!=NULL){
  for(i=0; i<size; i++)
   fprintf(f,"%3d", a[i]);
fprintf(f, "\nsum=%d", s);
}
fclose(f);
return 0;
}
```

დასმული ამოცანის გადაწყვეტის პროგრამული კოდის შესრულების შედეგი 120-ე სურათზეა ნაჩვენები.



სურ. 120

ამრიგად, შეგვიძლია დავასკვნათ, რომ:

ფაილი გარე მეხსიერების უბანია, რომელიც მონაცემთა მასივის შესანახად არის გამოყოფილი. მონაცემები, რომლებსაც ფაილები მოიცავს, მრავალფეროვანია. ეს არის ალგორითმულ ან მანქანურ ენაზე დაწერილი პროგრამები, საწყისი მონაცემები, განკუთვნილი პროგრამების სამუშაოდ, პროგრამების შესრულების შედეგები, ტექსტები, გრაფიკული გამოსახულებები და ა.შ.

კატალოგი (საქაღალდე, დირექტორია) ინფორმაციის მატარებელზე არსებული ბაიტების ერთობლიობაა, რომელიც მოიცავს ქვეკატალოგებსა და ფაილებს და გამოიყენება ფაილურ სისტემაში ფაილების ორგანიზების გამარტივების მიზნით.

ფაილური სისტემა ოპერაციული სისტემის ფუნქციონალური ნაწილია, რომელიც ფაილებზე ოპერაციების შესრულებას უზრუნველყოფს.

ფაილური სისტემის ნიმუშებია: **FAT (FAT – File Allocation Table,** ფაილების განთავსების ცხრილი), **NTFS, UDF** (გამოიყენება კომპაქტ-დისკებზე).

არსებობს **FAT** ფაილური სისტემის სამი ძირითადი ვერსია: FAT12, FAT16 და FAT32. ისინი ერთმანეთისგან იმ ბიტების რაოდენობით განსხვავდება, რომელიც კლასტერის ნომრის შესანახად გამოიყოფა.

X თავი
რ ე კ უ რ ს ი ა

ფუნქციებს, რომელთაც საკუთარი თავის გამოძახება შეუძლია, **რეკურსიული ფუნქციები** ეწოდება.

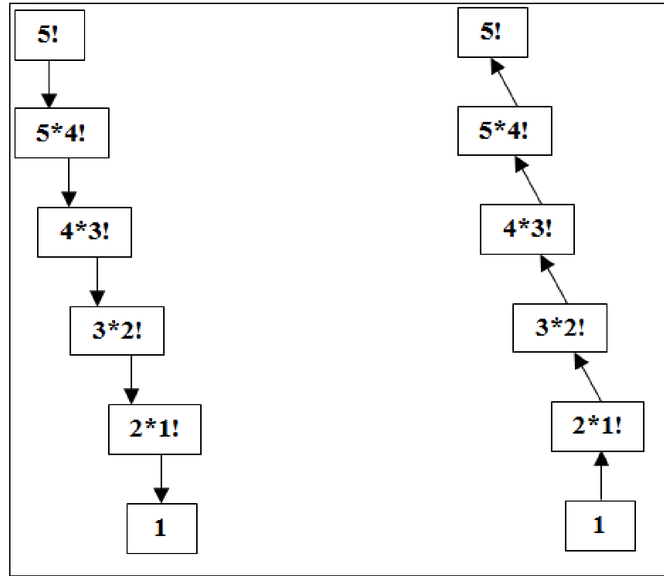
ზოგად შემთხვევაში, რეკურსიული ამოცანა ცალკეულ ეტაპებად იყოფა. ამოცანის გადასაწყვეტად ხდება რეკურსიული ფუნქციის გამოძახება, რომელმაც "იცის", თუ როგორ ამოხსნას ამოცანის უმარტივესი ნაწილი, ე.წ. საბაზო ამოცანა. თუ ფუნქციის გამოძახება მხოლოდ საბაზო ამოცანის გადაწყვეტის მიზნით ხდება, მაშინ იგი უბრალოდ აბრუნებს მიღებულ შედეგს, ხოლო უფრო რთული ამოცანის გადასაწყვეტად, იგი ამოცანას ორ ნაწილად ყოფს. ამასთან, ერთი ნაწილის ამოხსნა ფუნქციამ იცის, ხოლო მეორისა – არა. რეკურსია რომ შესრულდეს, ამოცანის მეორე ნაწილი საწყისი ამოცანის მსგავსი უნდა იყოს, მაგრამ უფრო გამარტივებული. რადგან ეს ახალი ამოცანა საწყისი ამოცანის მსგავსია, ფუნქცია საკუთარი თავის ახალი ასლის გამოძახებას ახდენს იმ მიზნით, რომ მუშაობა დაიწყოს შედარებით მარტივ პრობლემაზე. ამ პროცესს **რეკურსიული გამოძახება**, ანუ **რეკურსიის ბიჯი** ეწოდება და იგი, უმეტეს შემთხვევაში, შეიცავს **return** ოპერატორს. რეკურსიის პროცესის დასრულებისათვის აუცილებელია საწყისი ამოცანა დაყვანილ იქნას საბაზომდე.

მათემატიკიდან ცნობილია, რომ მთელი, არაუარყოფითი n რიცხვის ფაქტორიალი შემდეგი ფორმულით გამოითვლება:

ამასთან, 1!=1 და 0!=1. მაშასადამე, თუ n =5, მაშინ 5!=5*4*3*2*1=120.

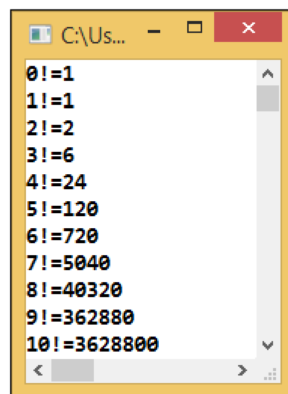
5-ის ფაქტორიალის მნიშვნელობის გამოთვლის პროცესი და რეკურსიული გამოძახებების თანამიმდევრობა 121-ე სურათზეა ნაჩვენები.

არაუარყოფითი მთელი რიცხვების ფაქტორიალების მნიშვნელობების რეკურსიული გზით გამოთვლის პროგრამა ქვემოთ არის წარმოდგენილი, ხოლო მისი შესრულების შედეგები 122-ე სურათზეა ნაჩვენები.



სურ. 121

```
#include <stdio.h>
#include <stdlib.h>
unsigned long factorial(unsigned long);
int main() {
int i;
for(i=0; i<=10; i++)
printf("%d!=%ld\n", i, factorial(i));
return 0;
}
unsigned long factorial(unsigned long n)
{
 if(n<=1)
 return 1;
 else
 return n*factorial(n-1);
 }
```



სურ. 122

მათემატიკიდან ასევე ცნობილია, რომ ფიბონაჩის რიცხვითი მიმდევრობა 0-ით და 1-ით იწყება და ხასიათდება იმ თვისებით, რომ მიმდევრობის ყოველი მომდევნო წევრის მნიშვნელობა წინა ორი წევრის მნიშვნელობათა ჯამის ტოლია. ბუნებაში ფიბონაჩის რიცხვითი მიმდევრობა სპირალის ფორმას აღწერს და მათემატიკურად შემდეგი სახე აქვს:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

ლეონარდო პიზელი ფიბონაჩი

რეკურსიული გზით ფიბონაჩის რიცხვითი მიმდევრობა შეიძლება შემდეგნაირად განისაზღვროს:

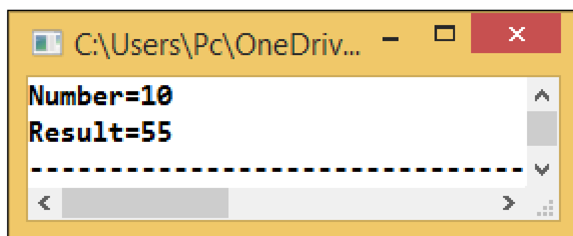
Fibonacci(0)=0;

Fibonacci(1)=1;

Fibonacci(n)= Fibonacci(n-1)+ Fibonacci(n-2);

ქვემოთ წარმოდგენილია ფიბონაჩის რიცხვითი მიმდევრობის მე-*n* წევრის მნიშვნელობის რეკურსიული გზით გამოთვლის პროგრამა, რომლის შესრულების შედეგი 123-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
//Fibonacci Numbers...
unsigned long  fibonacci(unsigned long );

int main() {
unsigned long  result, number;
printf("Number=");
scanf("%lu", &number);
result=fibonacci(number);
printf("Result=%lu", result);
return 0;
}
unsigned long fibonacci(unsigned long  n)
{
if(n==0 || n==1)
return n;
else
return fibonacci(n-1)+fibonacci(n-2); }
```



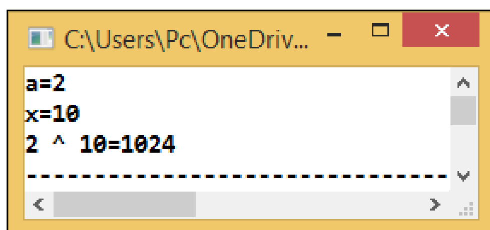
სურ. 123

ამოცანა 37.

შევადგინოთ $y = a^x$ გამოსახულების მნიშვნელობის გამოთვლის პროგრამა რეკურსიული გზით, სადაც a და x ცვლადები ნატურალური რიცხვებია.

```
#include <stdio.h>
#include <stdlib.h>
unsigned long function(int, int);
int main() {
int a, x;
unsigned long z;
printf("a=");
scanf("%d", &a);
printf("x=");
scanf("%d", &x);
z=function(a, x);
printf("%d ^ %d=%d",a, x, z);
return 0;
}
unsigned long function(int a, int x)
{
if(x==0)
  return 1;
if(x==1)
 return a;
else
 return a*function(a, x-1); }
```

პროგრამის შესრულების შედეგი 124-ე სურათზეა ნაჩვენები.



სურ. 124

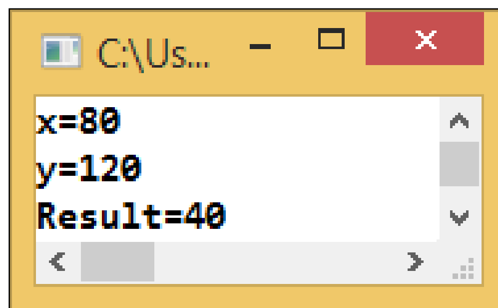
ამოცანა 38.

ევკლიდეს მოდიფიცირებული ალგორითმის გამოყენებით, რეკურსიული გზით გამოვთვალოთ ორი ნატურალური რიცხვის უდიდესი საერთო გამყოფის მნიშვნელობა.

ევკლიდე ალექსანდრიელი

```
#include <stdio.h>
#include <stdlib.h>
int function(int, int);
int main() {
int x, y, result;
printf("x=");
scanf("%d", &x);
printf("y=");
scanf("%d", &y);
result=function(x,y);
printf("Result=%d",result);
return 0;
}
int function(int x, int y)
{
if(y==0)
  return x;
else
  return function(y, x%y);
  }
```

პროგრამის შესრულების შედეგი 125-ე სურათზეა ნაჩვენები.



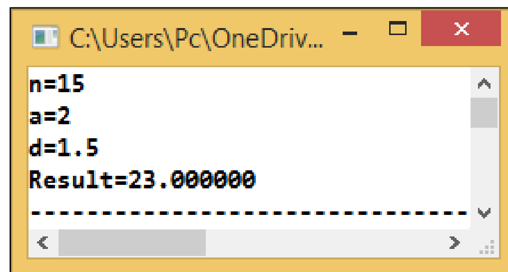
სურ. 125

ამოცანა 39.

შევადგინოთ რეკურსიული გზით არითმეტიკული პროგრესიის მე-*n* წევრის მნიშვნელობის გამოთვლის პროგრამა.

```
#include <stdio.h>
#include <stdlib.h>
double arifm(int, double, double);
int main() {
int n;
double a, d;
printf("n=");
scanf("%d", &n);
printf("a=");
scanf("%lf", &a);
printf("d=");
scanf("%lf", &d);
printf("Result=%lf",arifm(n,a,d));
return 0;
}
double arifm(int n, double a, double d){
if(n<1)
 return 0;
if(n==1)
 return a;
        return arifm(n-1, a, d)+d;
}
```

პროგრამის შესრულების შედეგი 126-ე სურათზეა ნაჩვენები.



სურ. 126

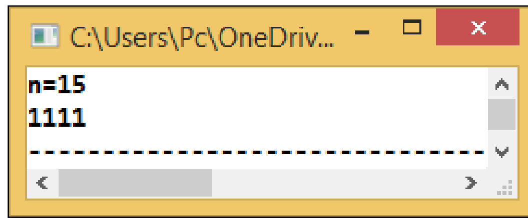
ამოცანა 40.

შევადგინოთ ნატურალური ათობითი რიცხვის ორობითში წარმოდგენის პროგ-რამა რეკურსიული გზით.

```
#include <stdio.h>
#include <stdlib.h>
int printBin(int);
int main() {
int n;
printf("n=");
```

```
scanf("%d", &n);
printBin(n);
return 0;
}
int printBin(int n){
 if(n==0)
    return 0;
  printBin(n/2);
  printf("%d",n%2);
}
```

პროგრამის შესრულების შედეგი 127-ე სურათზეა ნაჩვენები.



სურ. 127

10.1.1. ჰანოის კოშკები

ჰანოის კოშკები XIX საუკუნის ერთ-ერთი პოპულარული თავსატეხია.

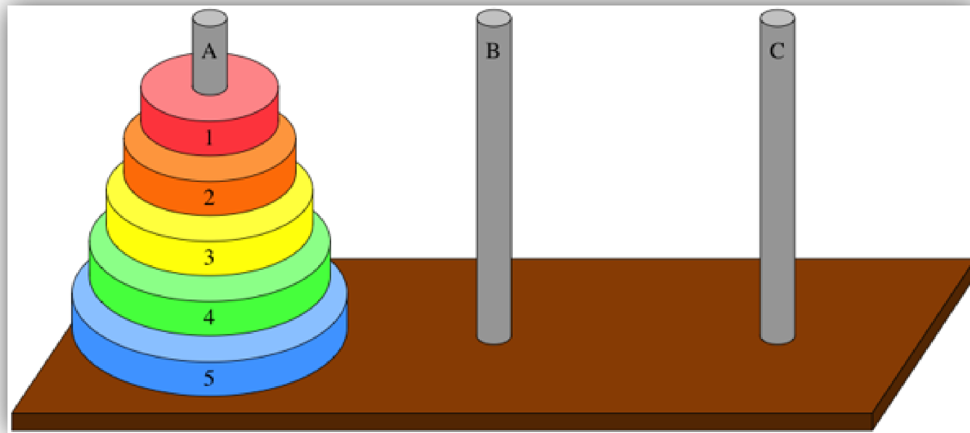
ძველი ინდური ლეგენდის თანახმად, ქალაქ ბენარესში მთავარი ტაძრის გუმბათის ქვეშ, იქ, სადაც დედამიწის ცენტრია, ბრინჯაოს მოედანზე ალმასის სამი ღერძი (მათ კოშკებად მოიხსენიებენ) დგას. სამყაროს შექმნის დღეს ერთ-ერთ ღერძზე 64 ფირფიტა იქნა ჩამოცმული ისე, რომ მათი ზომა ქვემოდან ზემოთ მცირდებოდა. ღმერთმა ბერებს დაავალა, ერთი ღერძიდან მეორეზე დამხმარე - მესამე ღერძის გამოყენებით გადაეტანათ ეს ფირფიტები, მაგრამ გაეთვალისწინე-ბინათ შემდეგი წესები:

- ერთ გადატანაზე შესაძლებელი იყო მხოლოდ ერთი ფირფიტის გადატანა.
- დიდი ზომის ფირფიტა პატარა ფირფიტის ზემოდან არასდროს არ უნდა აღმოჩენილიყო.
- შესაძლებელი იყო მხოლოდ ერთი დამხმარე ღერძის გამოყენება.

ლეგენდის თანახმად, როდესაც ბერები 64 ფირფიტას გადაიტანენ, ქვეყნის დასასრული (აპოკალიფსი) დადგება.

64 ფირფიტისთვის 18 446 744 073 709 551 615 გადალაგებაა საჭირო, რასაც 584 542 046 091 წელი დასჭირდება. ასე, რომ აპოკალიფსი მალე არ დადგება!

A ძელიდან ფირფიტის გადატანა შესაძლებელია B ან C ძელებზე (იხ. 128-ე სურათი). ორი ფირფიტის გადასაადგილებლად სამი მოქმედებაა საჭირო: **A->C, A->B, C->B.** სამი ფირფიტის შემთხვევაში შვიდი მოქმედება სრულდება, ხოლო ოთხის შემთხვევაში - თხუთმეტი.



სურ. 128

შევადგინოთ რეკურსიული ფუნქცია ფირფიტების ნებისმიერი რაოდენობისთვის:

ფუნქციას ოთხი პარამეტრი აქვს:

- **disc** – ფირფიტების რაოდენობა;
- **first** – ღერძი, რომელზეც განთავსებულია ფირფიტები;
- **last** – ღერძი, რომელზეც გადაგვაქვს ფირფიტები;
- **temp** – დამხმარე ღერძი, ფირფიტების დროებით განსათავსებლად.

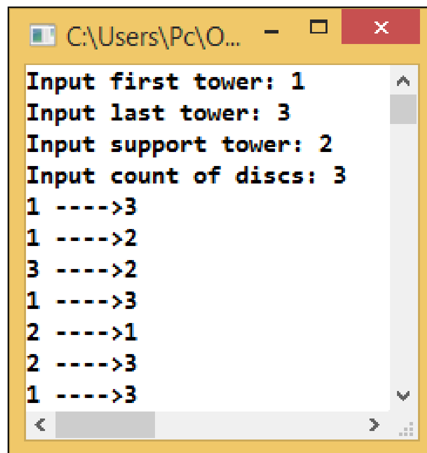
n რაოდენობის ფირფიტის A-დან B ღერძე C დამხმარე ღერძის გამოყენებით ფირფიტების გადასატანად ვიქცევით შემდეგნაირად:

- A-დან C-ზე გადაგვაქვს $n-1$ რაოდენობის ფირფიტა. დამხმარე ღერძის როლში ვიყენებთ B-ს;
- ბოლო ფირფიტა A-დან B ღერძზე გადაგვაქვს;

- $n-1$ რაოდენობის ფირფიტა C-დან B ღერძზე გადაგვაქვს. დამხმარე ღერძის როლში ვიყენებთ A-ს; ერთი ფირფიტის გადატანა ალგორითმში იმით გამოიხატება, რომ კონსოლზე გამოიტანება შესაბამისი სვლა.

ქვემოთ წარმოდგენილია ჰანოის კოშკების ამოცანის პროგრამული რეალიზება, რომლის შედეგები 129-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
int HanoiTowers(int, int, int, int);
int main() {
int disc, first, temp, last;
printf("Input first tower: ");
scanf("%d", &first);
printf("Input last tower: ");
scanf("%d", &last);
printf("Input support tower: ");
scanf("%d", &temp);
printf("Input count of discs: ");
scanf("%d", &disc);
HanoiTowers(disc,first,last, temp);
return 0;
}
int HanoiTowers(int disc, int first, int last, int temp){
 if(disc!=0){
   HanoiTowers(disc-1, first, temp, last);
  printf("%d ---->%d\n", first, last);
  HanoiTowers(disc-1, temp,last,first);
    }
}
```



სურ. 129

10.1.2. სწრაფი დახარისხების ალგორითმი და მისი პროგრამული რეალიზება რეკურსიული გზით

სწრაფი დახარისხება მიეკუთვნება „გათიშე და იბატონე" ალგორითმების კლასს. ისევე როგორც, ყველა „გათიშე და იბატონე" ალგორითმი, ისიც საწყის მასივს ორ ქვემასივად ყოფს, ხოლო შემდეგ რეკურსიული გზით ახარისხებს ქვემასივებს. მთელი პროცესი სამ ეტაპს მოიცავს:

- **საყრდენის (Pivot) შერჩევა** - მასივში ვირჩევთ ე.წ. საყრდენ ელემენტს (ხშირ შემთხვევაში, ეს მასივის მარცხენა ან მარჯვენა განაპირა ელემენტია);
- **დაყოფა** - მასივი ისე უნდა გადავაწყოთ, რომ ყველა ელემენტი რომლის მნიშვნელობა საყრდენ ელემენტზე ნაკლებია, განთავსდეს ამ ელემენტის მარცხნივ (წინ), ხოლო ყველა ელემენტი, რომლის მნიშვნელობა მეტია საყრდენი ელემენტის მნიშვნელობაზე, განთავსდეს მის შემდეგ (მარჯვნივ); ელემენტები, რომელთაც საყრდენი ელემენტის მნიშვნელობა აქვთ, შეიძლება ნებისმიერი მიმართულებით განთავსდეს;
- **გამეორება** - რეკურსიულად გამოვიყენოთ ზემოაღნიშნული ბიჯები ელემენტების ქვემასივზე, რომელთა მნიშვნელობები ნაკლებია საყრდენი ელემენტის მნიშვნელობაზე, და ცალკე ელემენტების ქვემასივზე, რომელთა მნიშვნელობებიც მეტია საყრდენი ელემენტის მნიშვნელობაზე.

რეკურსიის საბაზისო შემთხვევა არის ერთეულოვანი ზომის მასივები, რომლებიც არასდროს საჭიროებს დახარისხებას.

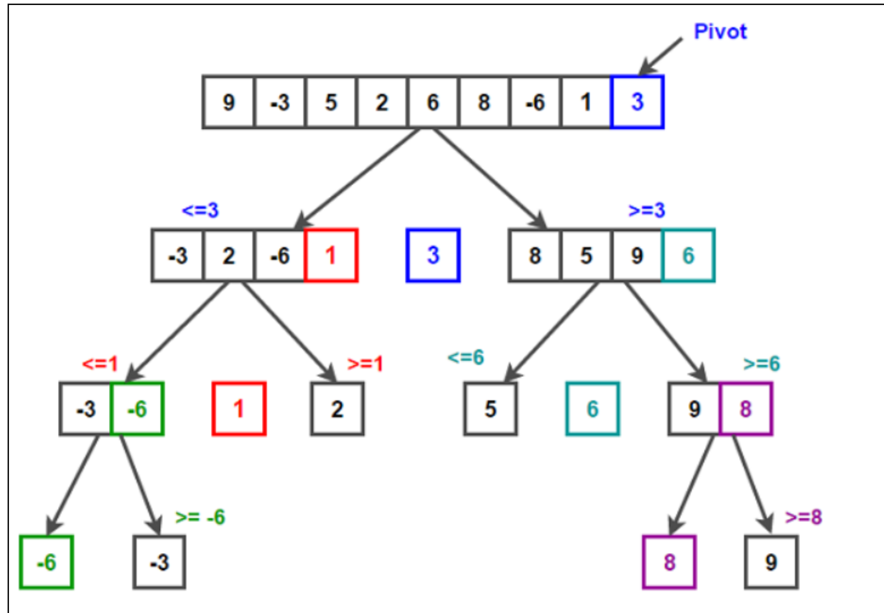
სწრაფი დახარისხების ალგორითმის ერთ-ერთი ნიმუში სქემის სახით 130-ე სურათზეა ნაჩვენები.

ქვემოთ წარმოდგენილია სწრაფი დახარისხების ალგორითმის შესაბამისი პროგრამული კოდი, რომლის შედეგი 131-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
int partition(int a[], int start, int end)
{
	int i;
	int pivot = a[end];
	int pIndex = start;
```

```
            for (i = start; i < end; i++)
    {
        if (a[i] <= pivot)
        {
            int k=a[i];
            a[i]=a[pIndex];
            a[pIndex]=k;
            pIndex++;
        }
    }
    int t=a[pIndex];
            a[pIndex]=a[end];
            a[end]=t;
     return pIndex;
 }
void quicksort(int a[], int start, int end)
{
    if (start >= end) {
        return;
    }
    int pivot = partition(a, start, end);
    quicksort(a, start, pivot - 1);
    quicksort(a, pivot + 1, end);
}

int main() {
            int i, n;
            int *a ;
            printf("n=");
            scanf("%d", &n);
            a=(int *)malloc(n * sizeof(int));
            for (i = 0; i < n; i++) {
                        a[i]=1+rand()%35;
                        printf("%3d", a[i]);
            }

    //int n = sizeof(a)/sizeof(a[0]);
    quicksort(a, 0, n - 1);
    printf("\n");
    for (i = 0; i < n; i++)
        printf("%3d", a[i]);
return 0;
}
```

ზემოთ წარმოდგენილი პროგრამული კოდის შესრულების შედეგები სვადასხვა ზომის მასივებისთვის 132(ა,ბ)-ე სურათებზეა ნაჩვენები.



სურ. 131

```
C:\Users\Pc\OneDrive\Documents\pointers6789.e...
n=15
  7 23 35  6 25 10 34 29 13 35  1  6  7 28 22
  1  6  6  7  7 10 13 22 23 25 28 29 34 35 35
--------------------------------
```

სურ. 132ა

```
C:\Users\Pc\OneDrive\Documents\pointers6789.exe
n=21
  7 23 35  6 25 10 34 29 13 35  1  6  7 28 22  2 21  8 33 12 17
  1  2  6  6  7  7  8 10 12 13 17 21 22 23 25 28 29 33 34 35 35
--------------------------------
```

სურ. 132ბ

XI თავი
მონაცემთა დინამიკური სტრუქტურები

ხშირად პროგრამებში საჭირო ხდება ისეთი მონაცემების გამოყენება, რომელთა ზომა და სტრუქტურა პროგრამის მუშაობის დროს უნდა შეიცვალოს. დინამიკური მასივები აქ ვერ დაგვეხმარება, რადგან წინასწარ შეუძლებელია იმის თქმა, თუ მეხსიერების რა მოცულობის გამოყოფა დაგვჭირდება - ეს მხოლოდ პროგრამის მუშაობის პროცესში შეიძლება გაირკვეს.

ასეთ შემთხვევებში სპეციალური სტრუქტურის მონაცემები გამოიყენება, რომ-ლებიც ერთმანეთთან მიმართვებით დაკავშირებულ ცალკეულ ელემენტებს წარ-მოადგენენ.

თითოეული ელემენტი (კვანძი) შედგება მეხსიერების ორი ნაწილისგან: მონაცემთა ველისა და მიმართვებისგან. მიმართვები იმავე ტიპის სხვა კვანძების მისამართებია, რომლებთანაც ეს ელემენტი ლოგიკურად არის დაკავშირებული.

მიმართვების ორგანიზებისთვის C ენა ცვლად-მიმთითებლებს იყენებს. რო-დესაც ასეთ სტრუქტურას ემატება ახალი კვანძი, მეხსიერების ახალი ბლოკი გამოიყოფა და (მიმართვის საშუალებით) მყარდება კავშირები ამ ელემენტსა და არსებულ ელემენტებს შორის. ნულოვანი მიმთითებელი (**NULL**) ჯაჭვის საბოლოო ელემენტის აღნიშვნის მიზნით გამოიყენება.

11.1 სტეკი

სტეკი (Stack) ქართულად (დასტას, შეკვრას) ნიშნავს. ის თითქმის ყველა დაპროგრამების ენაში გამოიყენება, მისი სახელიც და ფუნქციონალიც ძალიან გავს რეალურ ცხოვრებაში დასტებს და შეკვრებს.

სტეკი მონაცემთა აბსტრაქტული ტიპია, რომელიც ორგანიზებულია **LIFO** (**Last In First Out** – ბოლო შესული პირველი გადის) პრინციპით.

ალან ტიურინგი

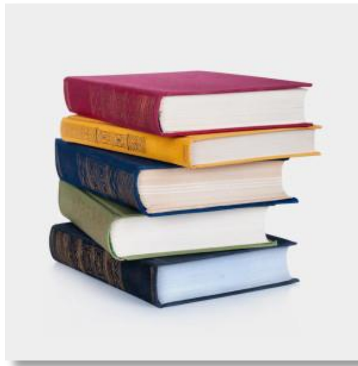
სტეკის ცნება 1946 წელს ინგლისელმა მათემატიკოსმა, კრიპტოანალიტიკოსმა **ალან ტიურინგმა** შემოიტანა.

როგორც ვიცით, რეალურ ცხოვრებაში რაიმე დასტაში, ნივთის ჩამატება ან დასტიდან ნივთის ამოღება ყოველთვის დასტის წვეროდან ხდება. ზუსტად იგივე პრინციპით მუშაობს **Stack** მონაცემთა სტრუქტურა.

ამგვარ მონაცემთა სტრუქტურებს ჩვენ **LIFO**-ს ვეძახით, რომელიც, როგორც ზემოთ აღვნიშნე, შემდეგნაირად იშიფრება „**Last in first out**"- სადაც ბოლოს ჩამატებული ელემენტი ყოველთვის პირველი წაიშლება.

Stack-ის ტერმინოლოგიით ჩამატებას **PUSH** ეწოდება, ხოლო წაშლას - **POP**. სტეკის მუშაობის პრინციპს რეალურ ცხოვრებაშიც ყოველდღიურად ვხვდებით. მაგალითად, ნებისმიერ კაფეტერიაში ან რესტორანში არის ერთმანეთზე დალაგებული თეფშები, სადაც თეფშის აღება ყოველთვის წვეროდან შეგვიძლია.

წარმოვიდგინოთ წიგნები დალაგებული სიმაღლეზე (იხილეთ 133-ე სურათი).



სურ. 133

სურათიდან ვხედავთ, რომ წიგნის ყდა რომელიც ჩვენთვის ხილვადია, არის მხოლოდ ვარდისფერ ყდიანი წიგნი, რომელიც წვეროშია მოქცეული. იმისთვის რომ სხვა წიგნებსაც მივწვდეთ, აუცილებელია ჯერ წვეროში მოთავსებული ვარდისფერ ყდიანი წიგნი ავიღოთ. ამ მაგალითით იმის თქმას ვცდილობთ, რომ ყოველთვის, როდესაც წიგნების აღებას მოვინდომებთ, დალაგებისგან განსხვავებით, წიგნის აღების პროცედურა ყოველთვის რევერსულია.

კიდევ ერთი მაგალითი ჩვენი ყოველდღიურობიდან:

ყველა ჩვენგანს გვაქვს ინტერნეტ ბრაუზერი რომელსაც ინტერნეტში მუშაობისთვის ვიყენებთ, ხოლო ყველა ბრაუზერს აქვს Back ღილაკი. როგორც კი ვებ-გვერდიდან, ვებ-გვერდზე გადავდივართ წარმოვიდგინოთ, რომ ეს ვებ-გვერდები ინახება სტეკში (ვგულისხმობთ web-page ის URL-ს).

ამჟამინდელი გვერდი, სადაც ახლა ვიმყოფებით, არის სტეკის წვეროში მოქცეული და თუ Back-ღილაკს დავაწვებით, მაშინ დავიწყებთ რევერსულად URL-ების სტეკიდან ამოღებას და წინა გვერდზე გადასვლას.

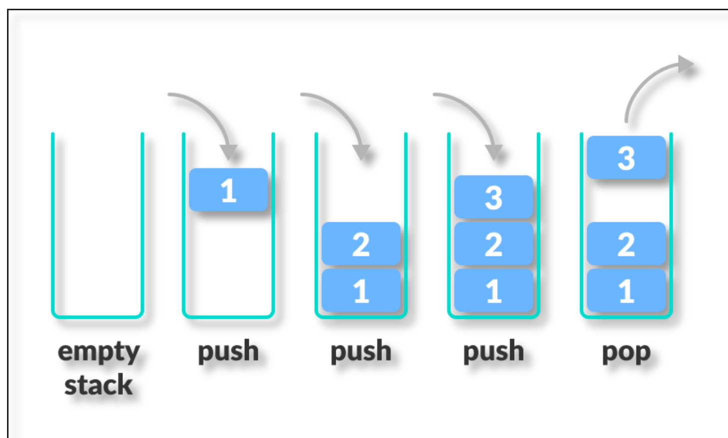
ამგვარად, **სტეკი** ელემენტების მოწესრიგებული ნაკრებია, სადაც ახალი ელემენტის დამატება და არსებული ელემენტების წაშლა მხოლოდ ერთი ბოლოდან არის შესაძლებელი და მას **სტეკის მწვერვალს** უწოდებენ.

სტეკთან მუშაობის დროს საჭიროა განისაზღვროს, თუ როგორ სრულდება ორი ოპერაცია - ელემენტის დამატება სტეკის მწვერვალზე (**push**) და ელემენტის მოხსნა სტეკის მწვერვალიდან (**pop**).

სტეკთან მიმართებაში შემდეგი ოპერაციები განიხილება:

- **push** – სტეკში ელემენტის მოთავსება;
- **pop** - სტეკის მწვერვალიდან ელემენტის წაშლა;
- **isEmpty** - შემოწმება, ცარიელია თუ არა სტეკი;
- **isFull** - შემოწმება, სავსეა თუ არა სტეკი;
- **peek** - საშუალებას იძლევა მივიღოთ ელემენტი სტეკის მწვერვალიდან მის (ელემენტის) წაუშლელად.

ზემოაღნიშნული ოპერაციები სქემის სახით 134-ე სურათზეა წარმოდგენილი.



სურ. 134

სტეკის მუშაობა შეგვიძლია შემდეგი სახით წარმოვადგინოთ:

- TOP მიმთითებელი გამოიყენება სტეკის "ზედა„ (მწვერვალის) ელემენტის გასაანალიზებლად.

- სტეკის ინიციალების შემდეგ, მისი მნიშვნელობა -1-ის ტოლია. ამ გზით ჩვენ შეგვიძლია მარტივად შევამოწმოთ სტეკი სავსეა თუ არა: თუ TOP == -1, სტეკი ცარიელია.
- როდესაც ელემენტს სტეკში „ვათავსებთ“, TOP მნიშვნელობა ერთით იზრდება. შემდეგი ელემენტი TOP+1 პოზიციაზე იქნება.
- როდესაც pop მეთოდს ვიყენებთ, ამით ჩვენ ვშლით იმ ელემენტს, რომელზეც TOP მიმთითებელი მიუთითებს. რის შემდეგაც, მისი მნიშვნელობა ერთით მცირდება.
- ახალი ელემენტის დამატებამდე ვამოწმებთ, სავსეა თუ არა სტეკი.
- ელემენტის წაშლამდე ვამოწმებთ, ცარიელია თუ არა სტეკი.

სტეკის რეალიზების მიზნით, ხშირ შემთხვევაში, მასივი გამოიყენება; თუმცა სიის გამოყენებაც არის შესაძლებელი.

ამჯერად, მოვახდინოთ სტეკის რეალიზება სტრუქტურის გამოყენებით, სადაც სტრუქტურის ერთ-ერთ ელემენტად მასივი განიხილება. შევიმუშაოთ სტეკის ელემენტებით შევსების, ელემენტების წაშლის და სტეკის შემმოწმებელი სამომხმარებლო ფუნქციები.

stack სტრუქტურის შექმნის კოდს შემდეგი სახე აქვს:

```
struct stack {
  int items[MAX];
  int top;
};
```

stack სტრუქტურის ტიპის ელემენტის გამოცხადება:

```
typedef struct stack st;
```

ცარიელი სტეკის შექმნა:

```
void createEmptyStack(st *s) {
  s->top = -1;
}
```

ფუნქცია, რომელიც ამოწმებს, სავსეა თუ არა სტეკი:

```
int isfull(st *s) {
  if (s->top == MAX - 1)
    return 1;
  else
```

```
    return 0;
}
```

ფუნქცია, რომელიც ამოწმებს, ცარიელია თუ არა სტეკი:

```
int isempty(st *s) {
  if (s->top == -1)
    return 1;
  else
    return 0;
}
```

სტეკის ელემენტებით შევსების ფუნქცია (push):

```
void push(st *s, int newitem) {
  if (isfull(s)) {
    printf("Stack is full");
  } else {
    s->top++;
    s->items[s->top] = newitem;
  } }
```

სტეკიდან ელემენტის წაშლის ფუნქცია (pop)

```
void pop(st *s) {
  if (isempty(s)) {
    printf("\n stack is empty \n");
  } else {
    printf("Deleted item= %d", s->items[s->top]);
    s->top--;
  }
printf("\n");
}
```

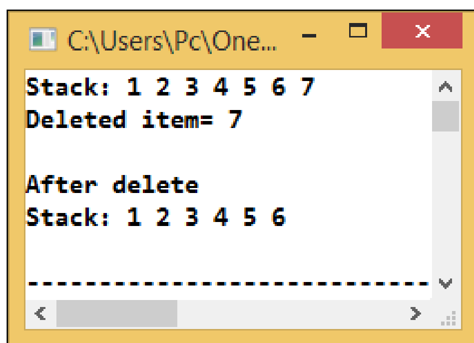
სტეკის კონსოლზე გამოტანის ფუნქცია:

```
void printStack(st *s) {
      int i;
  printf("Stack: ");
  for (i = 0; i < count; i++) {
    printf("%d ", s->items[i]);
  }
  printf("\n");
}
```

სტეკის რეალიზების სრულყოფილი პროგრამული კოდი ქვემოთ არის წარმოდგენილი, რომლის შესრულების შედეგები 135-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int count = 0;
// Create stack
struct stack {
  int items[MAX];
  int top;
};
typedef struct stack st;
void createEmptyStack(st *s) {
  s->top = -1;
}
// Is stack full?
int isfull(st *s) {
  if (s->top == MAX - 1)
    return 1;
  else
    return 0;
}
// Is stack empty?
int isempty(st *s) {
  if (s->top == -1)
    return 1;
  else
    return 0;
}
// Add items to stack
void push(st *s, int newitem) {
  if (isfull(s)) {
    printf("Stack is full");
  } else {
    s->top++;
    s->items[s->top] = newitem;
  }
  count++;
}
// Delete items from stack
void pop(st *s) {
  if (isempty(s)) {
    printf("\n stack is empty \n");
```

```
  } else {
    printf("Deleted item= %d", s->items[s->top]);
    s->top--;
  }
  count--;
  printf("\n");
}
// Stack output
void printStack(st *s) {
        int i;
  printf("Stack: ");
  for (i = 0; i < count; i++) {
    printf("%d ", s->items[i]); }
  printf("\n"); }
// function main
int main() {
  int ch;
  st *s = (st *)malloc(sizeof(st));
  createEmptyStack(s);
  push(s, 1);
  push(s, 2);
  push(s, 3);
  push(s, 4);
  push(s, 5);
  push(s, 6);
  push(s, 7);
  printStack(s);
  pop(s);
  printf("\nAfter delete\n");
  printStack(s);
}
```

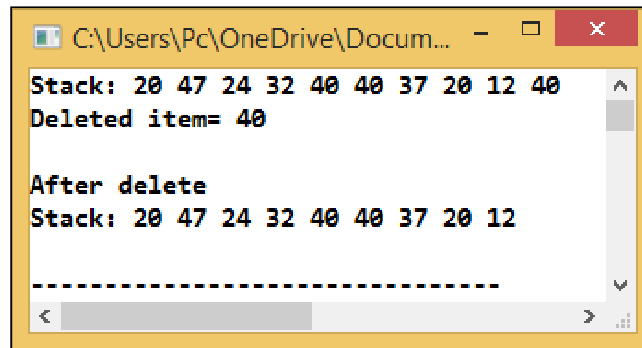


სურ. 135

თუ ზემოთ წარმოდგენილი სტეკის ელემენტებით შევსების (pop) ფუნქციას ჩავსვამთ ციკლში, ხოლო ელემენტებს კი თანაბარი ალბათობით განაწილებული

შემთხვევითი რიცხვების გენერირების გზით მივანიჭებთ მთელრიცხვა მნიშვნე-ლობებს, მივიღებთ 136-ე სურათზე ნაჩვენებ შედეგს (პროგრამული კოდი იგივე რჩება, იცვლება მხოლოდ მთავარი main() ფუნქცია):

```
int main() {
  int ch;
  int i;
  st *s = (st *)malloc(sizeof(st));
  createEmptyStack(s);
  srand(time(0));
for(i=0; i<MAX; i++)
  push(s, (10+rand()%41));
  printStack(s);
  pop(s);
  printf("\nAfter delete\n");
  printStack(s);
}
```



სურ. 136

სტეკის რეალიზების კიდევ ერთი საინტერესო პროგრამული კოდი ქვემოთ არის წარმოდგენილი, რომლის შესრულების შედეგები 137-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#define MAXSIZE 5
struct stack
{
   int stk[MAXSIZE];
   int top;
};
typedef struct stack STACK;
STACK  s;
void push(void);
int  pop(void);
void display(void);
```

```
void main ()
{
   int choice;
   int option = 1;
   s.top = -1;
printf ("STACK OPERATION\n");
   while (option)
   {
      printf ("------------------------------------------\n");
      printf ("      1    -->    PUSH               \n");
      printf ("      2    -->    POP               \n");
      printf ("      3    -->    DISPLAY               \n");
      printf ("      4    -->    EXIT           \n");
      printf ("------------------------------------------\n");
printf ("Enter your choice\n");
      scanf    ("%d", &choice);
      switch (choice)
      {
      case 1:
         push();
         break;
      case 2:
         pop();
         break;
      case 3:
         display();
         break;
      case 4:
         return;
      }
      fflush (stdin);
      printf ("Do you want to continue(Type 0 or 1)?\n");
      scanf    ("%d", &option); }  }
void push ()
{
   int num;
   if (s.top == (MAXSIZE - 1))
   {
      printf ("Stack is Full\n");
      return; }
   else
   {
      printf ("Enter the element to be pushed\n");
      scanf ("%d", &num);
```

```
        s.top = s.top + 1;
        s.stk[s.top] = num; }
    return; }
int pop ()
{
    int num;
    if (s.top == - 1)
    {
        printf ("Stack is Empty\n");
        return (s.top);
    }
    else
    {
        num = s.stk[s.top];
        printf ("poped element is = %dn", s.stk[s.top]);
        s.top = s.top - 1; }
    return(num);  }
void display ()
{
    int i;
    if (s.top == -1)
    {
        printf ("Stack is empty\n");
        return;
    }
    else
    {
        printf ("\n The status of the stack is \n");
        for (i = s.top; i >= 0; i--)
        {
            printf ("%d\n", s.stk[i]);
        }
    }
    printf ("\n");
}
```

```
C:\Users\Pc\OneDrive\Documents\pointers6789.exe
STACK OPERATION
------------------------------------------
      1     -->     PUSH
      2     -->     POP
      3     -->     DISPLAY
      4     -->     EXIT
------------------------------------------
Enter your choice
1
Enter the element to be pushed
2
Do you want to continue(Type 0 or 1)?
1
------------------------------------------
      1     -->     PUSH
      2     -->     POP
      3     -->     DISPLAY
      4     -->     EXIT
------------------------------------------
Enter your choice
2
poped element is = 2nDo you want to continue(Type 0 or 1)?
1
------------------------------------------
      1     -->     PUSH
      2     -->     POP
      3     -->     DISPLAY
      4     -->     EXIT
------------------------------------------
Enter your choice
4

--------------------------------
```

სურ. 137

11.2 რიგი

რიგი ემორჩილება **FIFO** პრინციპს - (**"first in - first out"). **რიგში პირველი ელემენტი რიგიდან პირველი გადის.

რიგი ელემენტების მოწესრიგებული ნაკრებია, რომელშიც ახალი ელემენტების დამატება ერთი ბოლოდან არის შესაძლებელი (მას **რიგის ბოლო** ეწოდება), ხოლო არსებული ელემენტების რიგიდან წაშლა შესაძლებელია მეორე ბოლოდან, რომელსაც **რიგის დასაწყისი** ეწოდება.

რიგის კარგი ნიმუშია მაღაზიაში სალაროსთან არსებული ადამიანების რიგი.

ზოგადად რიგი, როგორც მონაცემთა სტრუქტურა, მასობრივი მომსახურების მოდელირების ამოცანებში გამოიყენება, მაგალითად: ბანკში კლიენტების მომსახურების დროს.

როგორ მუშაობს რიგი?

enqueu არის მეთოდი, რომელიც რიგში ელემენტს ამატებს, ხოლო **dequeu** – მეთოდი, რომელიც რიგიდან ელემენტის წაშლას ემსახურება.

რიგის რეალიზება დაპროგრამების ნებისმიერ ენაზეა შესაძლებელი და ყველა ენაში რიგები ერთმანეთის მსგავსია.

რიგი შემდეგნაირად მუშაობს:

- რეალიზდება ორი მიმთითებელი: FRONT და REAR;
- FRONT არის მიმთითებელი რიგის პირველ ელემენტზე;
- REAR არის მიმთითებელი რიგის ბოლო ელემენტზე;
- FRONT და REAR მნიშვნელობები თავდაპირველად -1-ის ტოლი უნდა იყოს.

საბაზო ოპერაციები რიგებზე შემდეგია:

- **enqueu** ამატებს ელემენტს რიგის ბოლოში;
- **pop** - ემსახურება რიგის დასაწყისიდან ელემენტის წაშლას;
- **peek -** საშუალებას გვაძლევს რიგიდან ელემენტი მივიღოთ მის (ელემენტის) წაუშლელად;
- **IsEmpty** - ამოწმებს, ცარიელია თუ არა რიგი;
- **IsFull** - ამოწმებს, სავსეა თუ არა რიგი.

ოპერაცია enqueu-ს მუშაობის პრინციპი:

- ვამოწმებთ, სავსეა თუ არა რიგი;
- რიგში პირველი ელემენტის ჩამატებისას FRONT მიმთითებელს ვანიჭებთ 0-ის ტოლ მნიშვნელობას;
- ვზრდით REAR მიმთითებლის მნიშვნელობას 1-ით;
- ახალ ელემენტს ვამატებთ იმ პოზიციაზე, რომელზეც REAR მიმთითებელი მიუთითებს.

ოპერაცია dequeu-ს მუშაობის პრინციპი:

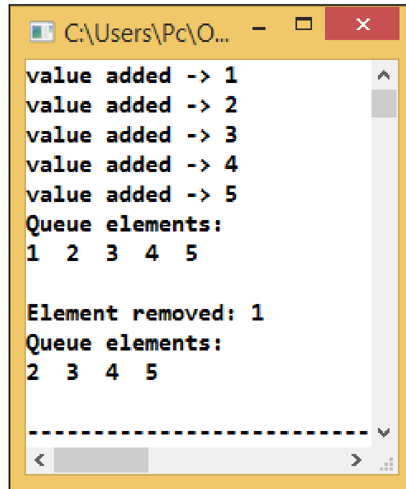
- ვამოწმებთ, ცარიელია თუ არა რიგი;
- ვიღებთ მნიშვნელობას, რომელზეც FRONT მიმთითებელი მიუთითებს;
- ვზრდით FRONT მიმთითებლის მნიშვნელობას 1-ით;

- რიგის ბოლო ელემენტის წაშლის დროს FRONT და REAR მიმთითებლებს ვანიჭებთ -1-ის ტოლ მნიშვნელობას.

ქვემოთ წარმოდგენილია რიგის ორგანიზების და მასზე ოპერაციების შესრულების ამსახველი პროგრამული კოდი, რომლის რეალიზების შედეგები 138-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#define SIZE 5
void enQueue(int);
void deQueue();
void display();
int items[SIZE], front = -1, rear = -1;
int main() {
 // Adding 5 items to the queue
 enQueue(1);
 enQueue(2);
 enQueue(3);
 enQueue(4);
 enQueue(5);
 display();
 // The deQueue function removes the first element
 deQueue();
 // Now there are 4 elements inside the queue
 display();
 return 0; }
void enQueue(int value) {
 if (rear == SIZE - 1)
  printf("\nThe queue is full");
 else {
  if (front == -1)
   front = 0;
  rear++;
  items[rear] = value;
  printf("\nvalue added -> %d", value);
      }
}
void deQueue() {
 if (front == -1)
  printf("\nQueue is empty");
 else {
  printf("\nElement removed: %d", items[front]);
  front++;
```

```
  if (front > rear)
    front = rear = -1;} }
// The function prints the queue to the console
void display() {
 if (rear == -1)
  printf("\nQueue is empty");
 else {
  int i;
  printf("\nQueue elements:\n");
  for (i = front; i <= rear; i++)
    printf("%d  ", items[i]);
 }
 printf("\n");
}
```



სურ. 138

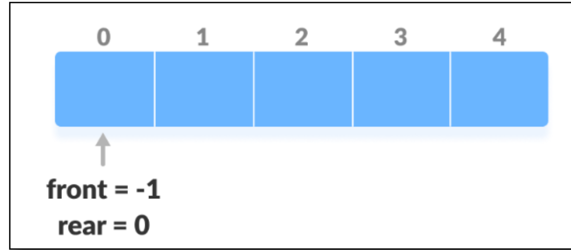
11.3 ორმხრივი რიგი (დეკი)

დეკი ანუ **ორმხრივი რიგი** - რიგის სახეობაა, რომელშიც ელემენტების ჩასმა და ამოშლა შესაძლებელია როგორც რიგის დასაწყისიდან, ასევე ბოლოდან. ეს ნიშნავს, რომ ამ ტიპის რიგი **არ ემორჩილება FIFO** (პირველი შემოსული, პირველი გადის) პრინციპს.

ოპერაციები დეკზე შემდეგი ალგორითმის მიხედვით განვახორციელოთ:

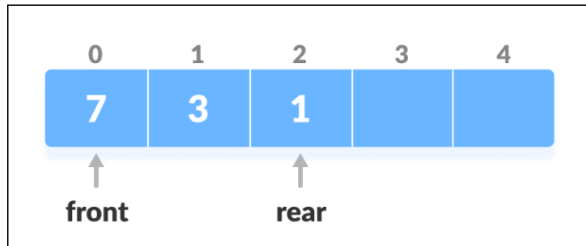
1. გამოვაცხადოთ n სიგრძის მასივი (deque);

2. გამოვაცხადოთ ორი მიმთითებელი და განათავსოთ ისინი რიგის დასაწყისში. შემდეგ მათ მივანიჭოთ მნიშვნელობები: front = -1, rear = 0 (იხილეთ 139-ე სურათი).



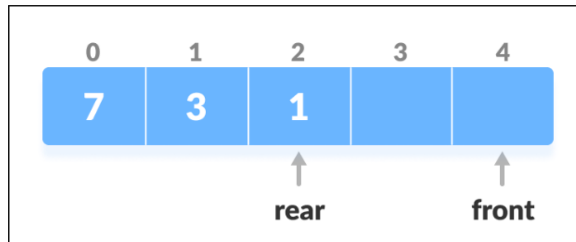
სურ. 139

3. **ელემენტის მოთავსება რიგის დასაწყისში**: ვამოწმებთ front მიმთითებლის პოზიციას (იხილეთ 140-ე სურათი).



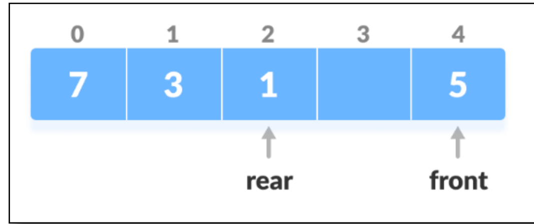
სურ. 140

- თუ front < 1, მას ხელმეორედ ვაცხადებთ: front = n-1 (ბოლო ინდექსი. იხილეთ 141-ე სურათი).



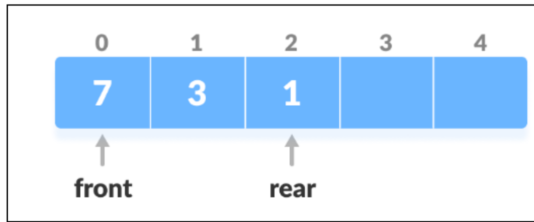
სურ. 141

- წინააღმდეგ შემთხვევაში front მიმთითებლის მნიშვნელობას ვზრდით 1-ით.
- ვამატებთ ახალ მნიშვნელობას array[front]-ში (ამ შემთხვევაში 5-ს. იხილეთ 142-ე სურათი).



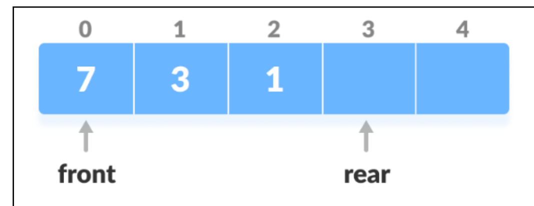
სურ. 142

4. **ელემენტის ჩამატება რიგის ბოლოში**: ვამოწმებთ, ხომ არ არის რიგი შევსებული (იხილეთ 143-ე სურათი).



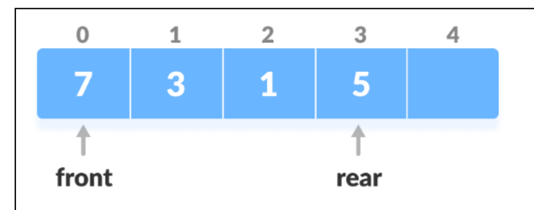
სურ. 143

- თუ რიგი სავსეა (შევსებულია), ხელახლა ვაცხადებთ rear მიმთითებელს: **rear=0;**
- წინააღმდეგ შემთხვევაში, rear მიმთითებლის მნიშვნელობას ვზრდით 1-ით (იხილეთ 144-ე სურათი).



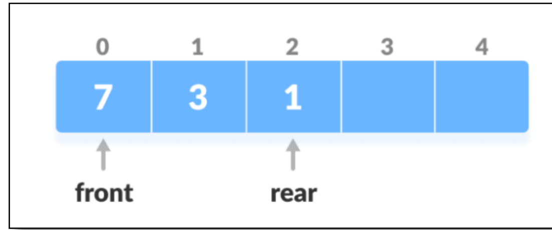
სურ. 144

- ვათავსებთ (ვამატებთ) რიგში ახალ მნიშვნელობას (იხილეთ 145-ე სურათი).



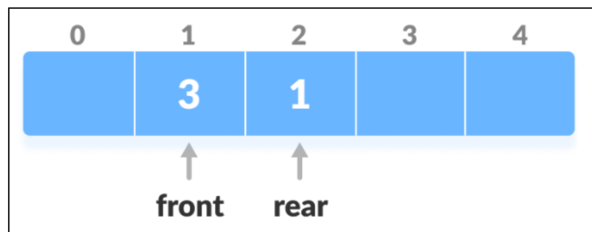
სურ. 145

5. **ელემენტის წაშლა რიგის დასაწყისიდან:** ვამოწმებთ, ცარიელია თუ არა რიგი (იხილეთ 146-ე სურათი).



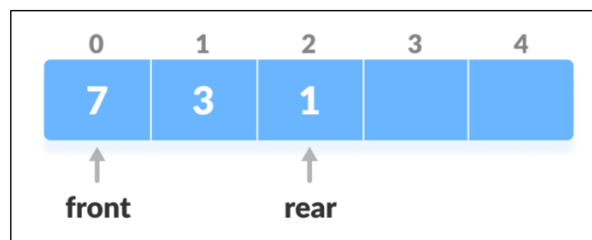
სურ. 146

- თუ ორმხრივი რიგი ცარიელია (front = -1), წაშლა ვერ მოხერხდება;
- თუ რიგში ერთი ელემენტია (front = rear), მიმთითებლებს მივანიჭოთ -1-ის ტოლი მნიშვნელობები: front = -1 და rear = -1;
- თუ front მიმთითებელი რიგის ბოლოშია (front = n – 1), მივანიჭოთ მას 0-ის ტოლი მნიშვნელობა: front =0, წინააღმდეგ შემთხვევაში - front = front +1 (იხილეთ 147-ე სურათი).



სურ. 147

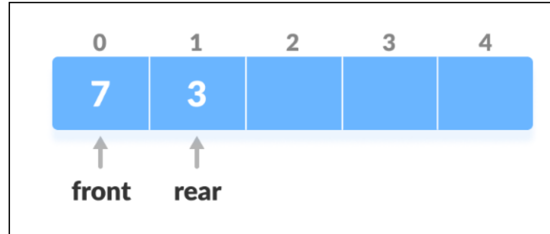
6. **ელემენტის წაშლა რიგის ბოლოდან:** ვამოწმებთ, ცარიელია თუ არა რიგი (იხილეთ 148-ე სურათი).



სურ. 148

- თუ ორმხრივი რიგი ცარიელია (front = -1), წაშლა ვერ მოხერხდება;
- თუ რიგში ერთი ელემენტია (front = rear), მიმთითებლებს მივანიჭოთ -1-ის ტოლი მნიშვნელობები: front = -1 და rear = -1; წინააღმდეგ შემთხვევაში შევასრულოთ შემდეგი ბიჯები:

- თუ rear მიმთითებელი რიგის ბოლოშია (rear = 0), მივანიჭოთ მას n-1-ის ტოლი მნიშვნელობა: front=n-1, წინააღმდეგ შემთხვევაში - rear = rear -1 (იხილეთ 149-ე სურათი).



სურ. 149

7. **რიგის სიცარიელეზე შემოწმება:** თუ front=-1, მაშინ რიგი ცარიელია.
8. **შემოწმება, სავსეა თუ არა რიგი**: თუ front = 0 და rear = n - 1 ან front = rear + 1, რიგი სავსეა.

ზოგადად, ორმხრივი რიგის რამდენიმე ტიპი არსებობს:

- **ორმხრივი რიგი ჩასმის შეზღუდვით**. ამ ტიპის რიგში (დეკში) ელემენტების ჩასმა რიგის მხოლოდ ერთი ბოლოდან არის შესაძლებელი, ხოლო წაშლა - კვლავ ორივე ბოლოდანაა ხელმისაწვდომი.
- **ორმხრივი რიგი წაშლის შეზღუდვით.** ამ ტიპის რიგში (დეკში) ელემენტების წაშლა რიგის მხოლოდ ერთი ბოლოდან არის შესაძლებელი, ხოლო ჩასმა (ჩამატება) კვლავ ორივე ბოლოდანაა ხელმისაწვდომი.

ზემოაღნიშნული ალგორითმის გათვალისწინებით მოვახდინოთ დეკის პროგრამული რეალიზება. შესრულების შედეგები 150-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
#define MAX 10
void addFront(int *, int, int *, int *);
void addRear(int *, int, int *, int *);
int delFront(int *, int *, int *);
int delRear(int *, int *, int *);
void display(int *);
int count(int *);
int main() {
 int arr[MAX];
 int front, rear, i, n;
 front = rear = -1;
 for (i = 0; i < MAX; i++)
```

```
    arr[i] = 0;
  addRear(arr, 5, &front, &rear);
  addFront(arr, 12, &front, &rear);
  addRear(arr, 11, &front, &rear);
  addFront(arr, 5, &front, &rear);
  addRear(arr, 6, &front, &rear);
  addFront(arr, 8, &front, &rear);
  printf("\nElements in a deque: ");
  display(arr);
  i = delFront(arr, &front, &rear);
  printf("\nelement removed: %d", i);
printf("\nQueue after removing an element: ");
  display(arr);
  addRear(arr, 16, &front, &rear);
  addRear(arr, 7, &front, &rear);
  printf("\nsequence after element insertion: ");
  display(arr);
  i = delRear(arr, &front, &rear);
  printf("\nelement removed: %d", i);
  printf("\nQueue after removing an element: ");
  display(arr);
  n = count(arr);
  printf("\nNumber of items in the queue: %d", n);
}
void addFront(int *arr, int item, int *pfront, int *prear) {
  int i, k, c;
  if (*pfront == 0 && *prear == MAX - 1) {
    printf("\nThe queue is full.\n");
    return;
  }
  if (*pfront == -1) {
    *pfront = *prear = 0;
    arr[*pfront] = item;
    return;
  }
  if (*prear != MAX - 1) {
    c = count(arr);
    k = *prear + 1;
    for (i = 1; i <= c; i++) {
      arr[k] = arr[k - 1];
      k--;
    }
    arr[k] = item;
    *pfront = k;
```

```
  (*prear)++;
 } else {
  (*pfront)--;
  arr[*pfront] = item;
       }
}
void addRear(int *arr, int item, int *pfront, int *prear) {
 int i, k;
 if (*pfront == 0 && *prear == MAX - 1) {
  printf("\nThe queue is full.\n");
  return;
 }
 if (*pfront == -1) {
  *prear = *pfront = 0;
  arr[*prear] = item;
  return;
 }
 if (*prear == MAX - 1) {
  k = *pfront - 1;
  for (i = *pfront - 1; i < *prear; i++) {
   k = i;
   if (k == MAX - 1)
    arr[k] = 0;
   else
    arr[k] = arr[i + 1];
  }
  (*prear)--;
  (*pfront)--;
 }
 (*prear)++;
 arr[*prear] = item;
 }
 int delFront(int *arr, int *pfront, int *prear) {
 int item;
 if (*pfront == -1) {
  printf("\nQueue is empty.\n");
  return 0;
 }
 item = arr[*pfront];
 arr[*pfront] = 0;

 if (*pfront == *prear)
  *pfront = *prear = -1;
 else
```

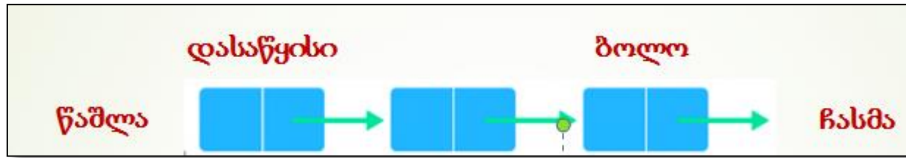
```
    (*pfront)++;
  return item;
}
int delRear(int *arr, int *pfront, int *prear) {
  int item;
  if (*pfront == -1) {
    printf("\nQueue is empty.\n");
    return 0;
  }
  item = arr[*prear];
  arr[*prear] = 0;
  (*prear)--;
  if (*prear == -1)
    *pfront = -1;
  return item;
  }
  void display(int *arr) {
  int i;
  printf("\n front:  ");
  for (i = 0; i < MAX; i++)
    printf("  %d", arr[i]);
  printf("  :the end");
}
int count(int *arr) {
  int c = 0, i;

  for (i = 0; i < MAX; i++) {
    if (arr[i] != 0)
      c++;
  }
  return c;  }
```

```
C:\Users\Pc\OneDrive\Documents\queue1234.exe
Elements in a deque:
 front:    8  5  12  5  11  6  0  0  0  0  :the end
element removed: 8
Queue after removing an element:
 front:    0  5  12  5  11  6  0  0  0  0  :the end
sequence after element insertion:
 front:    0  5  12  5  11  6  16  7  0  0  :the end
element removed: 7
Queue after removing an element:
 front:    0  5  12  5  11  6  16  0  0  0  :the end
Number of items in the queue: 6
--------------------------------
```

სურ. 150

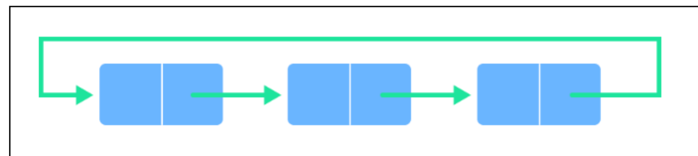
ზოგადად, რიგის რამდენიმე სახე არსებობს, კერძოდ:

- **მარტივი რიგი**, რომელიც ექვემდებარება **FIFO** პრინციპს. ელემენტი ემატება რიგის ბოლოს და იშლება მისი დასაწყისიდან (იხილეთ 151-ე სურათი).



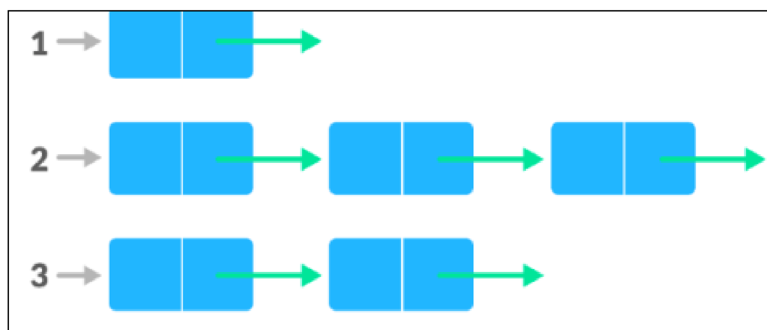
სურ. 151

- **წრიული რიგი**, რომელშიც ბოლო ელემენტი მიუთითებს პირველზე. სწორედ ეს ქმნის წრეს. წრიული რიგის მთავარი უპირატესობა მარტივ რიგთან შედარებით მდგომარეობს მეხსიერების უფრო ეფექტურ გამოყენებაში. თუ რიგში ბოლო პოზიცია დაკავებულია და პირველი ცარიელი, მაშინ ელემენტი იქ დაემატება. მარტივ რიგებში იგივეს გაკეთება შეუძლებელია (იხილეთ 152-ე სურათი).



სურ. 152

- **პრიორიტეტული რიგი**, რომელშიც თითოეულ ელემენტს თავისი პრიორიტეტია ქვს. ელემენტების მომსახურების თანმიმდევრობა დამოკიდებულია მათ პრიორიტეტზე. თუ ორ ელემენტს ერთი და იგივე პრიორიტეტი აქვს, მაშინ მათი მომსახურება რიგში მათი რიგითობის მიხედვით ხდება (იხილეთ 153-ე სურათი).



სურ. 153

სად გამოიყენება რიგი?

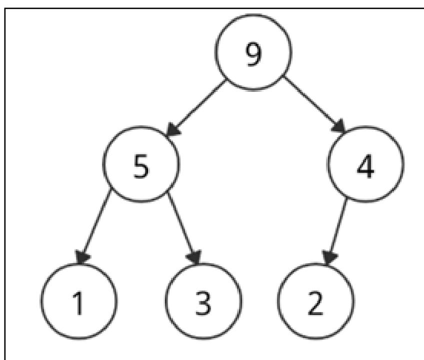
- პროცესების დაგეგმვასა და მყარი დისკის მუშაობაში;
- ორ პროცესს შორის გადატანილი მონაცემების სინქრონიზაციისთვის. მაგალითად: I/O ბუფერები, ფაილების I/O ბუფერები და ა.შ. ;
- წყვეტის მართვისთვის რეალურ დროის სისტემებში;
- ტელეფონებსა და „ქოლ-ცენტრებში“ რიგის მიხედვით კლიენტების მომსახურების მიზნით.

11.4.1 გროვა (Heap)

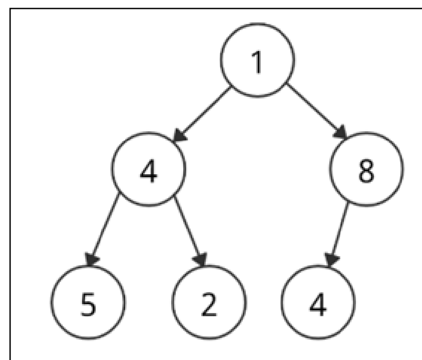
გროვა (მონაცემთა სტრუქტურა) არის სრული ორობითი ხე, რომელიც აკმაყოფილებს გროვას თვისებას: თუ კვანძი A არის B კვანძის მშობელი, მაშინ A კვანძის გასაღები ≥ B კვანძის გასაღებზე.

თუ ნებისმიერი კვანძი ყოველთვის მეტია ვიდრე შვილობილი კვანძ(ებ)ი, და ძირეული კვანძის გასაღები ყველა სხვა კვანძს შორის ყველაზე მეტია, მაშინ მას **max-heap**-ს უწოდებენ (იხილეთ 154-ე სურათი).

თუ ნებისმიერი კვანძი ყოველთვის ნაკლებია ვიდრე შვილობილი კვანძ(ებ)ი, და ძირეული კვანძის გასაღები ყველაზე ნაკლებია ყველა სხვა კვანძს შორის, მაშინ მას **min-heap**-ს უწოდებენ (იხილეთ 155-ე სურათი).



სურ. 154



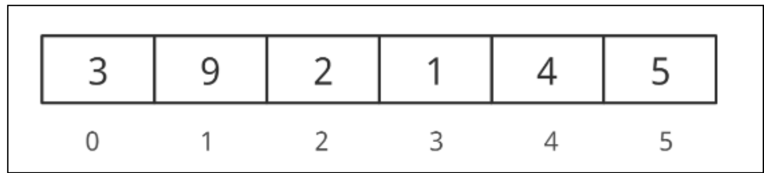
სურ. 155

მოდით, განვიხილოთ რამდენიმე ძირითადი ოპერაცია, რომელიც შეიძლება შესრულდეს გროვაზე და გავეცნოთ ამ ოპერაციების ალგორითმებს.

გროვას შექმნა (heapify)

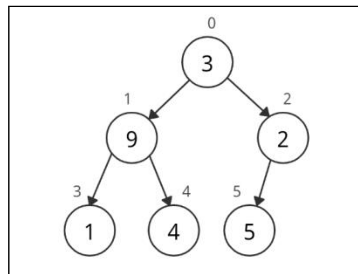
heapify არის მეთოდი, რომელიც გამოიყენება ბინარული ხისგან გროვის შესაქმნელად. მას, ასევე, ვიყენებთ **min-heap**-ის ან **max-heap**-ის შექმნისთვის.

ბიჯი 1. დავუშვათ, მოცემულია ექვსელემენტიანი შემდეგი მასივი (იხილეთ 156-ე სურათი).



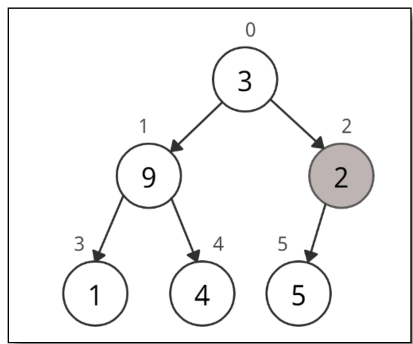
სურ. 156

ბიჯი 2. შევქმნათ სრული ორობითი ხე ამ მასივიდან (იხილეთ 157-ე სურათი).



სურ. 157

ბიჯი 3. დავიწყოთ პირველი არაფოთლოვანი კვანძით, მისი ინდექსი უდრის **n/2 – 1**-ს. ჩვენს შემთხვევაში ინდექსი 2-ის ტოლია (იხილეთ 158-ე სურათი).



სურ. 158

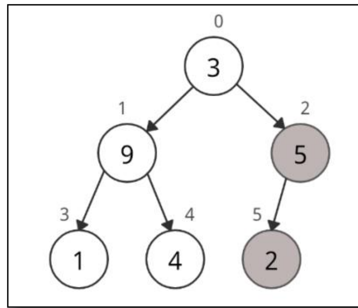
ბიჯი 4. დავუშვათ რომ, მიმდინარე ელემენტი ***i*** ინდექსით ყველაზე დიდია. ასე რომ, ***i*** არის ყველაზე დიდი კვანძის (largest) ინდექსი.

ბიჯი 5. მარცხენა შვილობილი ელემენტის ინდექსი (**leftChild**) 2 ***i*** + 1-ს ტოლია, ხოლო მარჯვენა შვილობილი ელემენტის ინდექსი (**rightChild**) - 2 ***i*** + 2-ის.

- თუ **leftChild** მეტია **currentElement**-ზე (ანუ ელემენტი ***i*** ინდექსით), მაშინ **leftChildIndex** არის უდიდესი კვანძის (largest) ინდექსი.
- თუ **rightChild** მეტია ელემენტზე ინდექსით **largest**, მაშინ **rightChildIndex** არის ყველაზე დიდი კვანძის ინდექსი (largest).

ბიჯი 6. ვცვლით **largest** და **currentElement** მნიშვნელობებს (იხილეთ 159-ე სურათი).

ბიჯი 7. ვიმეორებთ 3÷7 ბიჯებს მანამ, სანამ ქვეხეები ასევე არ იქცევა გროვებად.



სურ. 159

ფსევდოკოდის სახით ალგორითმი შემდეგია (იხილეთ 160-ე სურათი):

Heapify(array, size, i)

largest ცვლადს მივანიჭოთ i მნიშვნელობა

leftChild = 2i + 1

rightChild = 2i + 2

თუ leftChild > *array*[largest]

largest მნიშვნელობა მივანიჭოთ leftChildIndex-ს

თუ rightChild > *array*[largest]

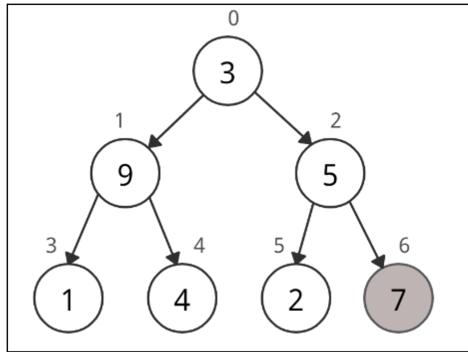
largest მნიშვნელობა მივანიჭოთ rightChildIndex-ს

შევცვალოთ მნიშვნელობები *array*[i] და *array*[largest]

სურ. 160

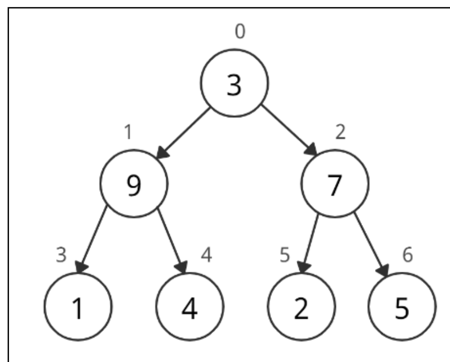
განვიხილოთ max-გროვაში **ელემენტის დამატების ალგორითმი:**

ბიჯი 1. ჩავამატოთ ელემენტი ხის ბოლოს (იხილეთ 161-ე სურათი).



სურ. 161

ბიჯი 2. heapify მეთოდი გამოვიყენოთ ხის მიმართ (იხილეთ 162-ე სურათი).



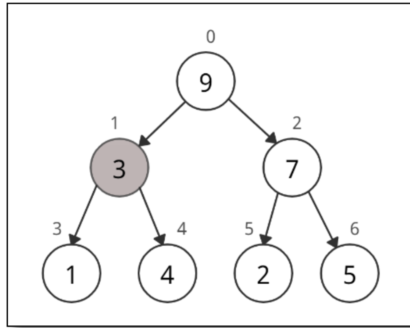
სურ. 162

- თუ არ არსებობს კვანძი, შევქმნათ ახალი. წინააღმდეგ შემთხვევაში, თუ კვანძი უკვე არსებობს, ჩავსვათ ის ხის ბოლოში. heapify მეთოდი გამოვიყენოთ მასივის მიმართ.

min-გროვას შემთხვევაში იგივე პროცედურა სრულდება რაც max-გროვაში, ოღონდ მშობელი კვანძი ყოველთვის ნაკლები უნდა იყოს ახალ კვანძზე.

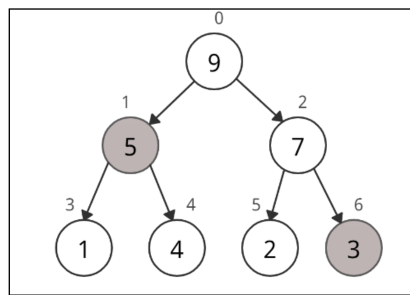
გროვადან **ელემენტის წასაშლელად** შემდეგ ალგორითმს ვიყენებთ:

ბიჯი 1. ვირჩევთ გროვადან წასაშლელ ელემენტს (იხილეთ 163-ე სურათი).



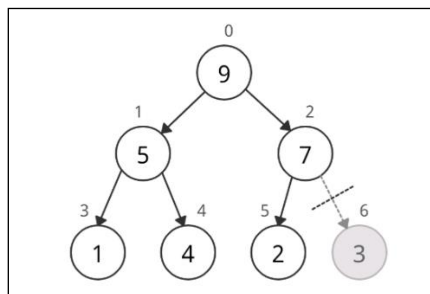
სურ. 163

ბიჯი 2. შერჩეულ და ბოლო ელემენტებს ვუცვლით ადგილებს (იხილეთ 164-ე სურათი).



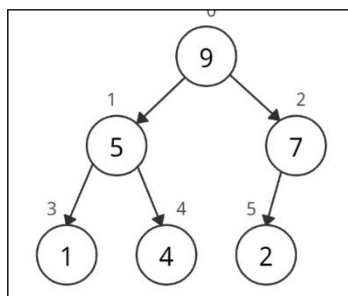
სურ. 164

ბიჯი 3. გროვადან ვშლით ბოლო ელემენტს (იხილეთ 165-ე სურათი).



სურ. 165

ბიჯი 4. heapify მეთოდს ვიყენებთ ხის მიმართ (იხილეთ 166-ე სურათი).



სურ. 166

max-გროვას შემთხვევაში:

- **თუ** წასაშლელი კვანძი არის მარცხენა კვანძი, მას ვშლით. წინააღმდეგ შემთხვევაში, წასაშლელ და უკიდურეს მარცხენა კვანძებს ვუცვლით ადგილებს და ვშლით წასაშლელ კვანძს. **heapify** მეთოდი გამოვიყენოთ მასივის მიმართ.

min-გროვას შემთხვევაში იგივე პროცედურა სრულდება რაც max-გროვაში, ოღონდ შვილობილი კვანძი მეტი უნდა იყოს მიმდინარე კვანძზე.

peek ოპერაცია აბრუნებს მაქსიმალურ ელემენტს max-heap-დან და მინიმალურ ელემენტს min-heap-დან კვანძის წაშლის გარეშე. max-heap-ისთვის და min-heap-ისთვის მას ერთიდაიგივე გამოყენება აქვს:

გროვის გამოყენება ხდება:

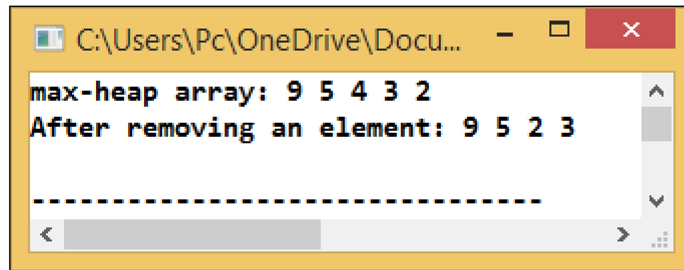
- პრიორიტეტული რიგების რეალიზაციის დროს;
- დეიქსტრას ალგორითმში;
- პირამიდულ დახარისხებაში.

ქვემოთ წარმოდგენილია გროვას რეალიზების პროგრამული კოდი, რომლის შესრულების შედეგები 167-ე სურათზეა ნაჩვენები.

```
#include <stdio.h>
int size = 0;
void swap(int *a, int *b)
{
  int temp = *b;
  *b = *a;
  *a = temp; }
void heapify(int array[], int size, int i) {
  if (size == 1)
  {
    printf("One element in the heap");  }
  else
  {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < size && array[l] > array[largest])
      largest = l;
    if (r < size && array[r] > array[largest])
      largest = r;
```

```
    if (largest != i)
    {
      swap(&array[i], &array[largest]);
      heapify(array, size, largest);
    }
  }
}
void insert(int array[], int newNum)
{
  int i;
  if (size == 0)
  {
    array[0] = newNum;
    size += 1;
  }
  else
  {
    array[size] = newNum;
    size += 1;
    for (i = size / 2 - 1; i >= 0; i--)
    {
      heapify(array, size, i);
    }
  }
}
void deleteRoot(int array[], int num)
{
  int i;
  for (i = 0; i < size; i++)
  {
    if (num == array[i])
      break;
  }
  swap(&array[i], &array[size - 1]);
  size -= 1;
  for (i = size / 2 - 1; i >= 0; i--)
  {
    heapify(array, size, i);
  }
}
void printArray(int array[], int size)
{
  int i;
  for (i = 0; i < size; ++i)
```

```
   printf("%d ", array[i]);
  printf("\n");
}
int main()
{
  int array[10];
  insert(array, 3);
  insert(array, 4);
  insert(array, 9);
  insert(array, 5);
  insert(array, 2);
  printf("max-heap array: ");
  printArray(array, size);
  deleteRoot(array, 4);
  printf("After removing an element: ");
  printArray(array, size);
}
```



სურ. 167

11.4.2 გროვას დახარისხების ალგორითმი და მისი პროგრამული რეალიზება

C დაპროგრამების ენაზე შევადგინოთ მთელრიცხვა მასივის დახარისხების პროგრამა გროვას დახარისხების (**heap sort**) ალგორითმის გამოყენებით და კონ-სოლზე წარმოვადგინოთ დახარისხებული მასივი.

აქ ყურადსაღებია რამოდენიმე ოპერაცია, რომელიც გროვას დახარისხების პროცესში უნდა გამოვიყენოთ. ვინაიდან საქმე გვაქვს მთელი რიცხვების მასივთან, რომელიც უნდა დახარისხდეს, ჩვენ გამოვიყენებთ მასივზე დაფუძნებულ ორო-ბითი გროვის წარმოდგენას. ამ წარმოდგენაში, თუ მშობელი კვანძი ინახება i ინდექსში, მაშინ მისი მარცხენა შვილობილი კვანძის ინდექსი ექნება $2*i+1$-ის ტოლი, ხოლო მისი მარჯვენა კვანძის ინდექსი - $2*i+2$-ის ტოლი.

გროვის დახარისხების ალგორითმი შემდეგია:

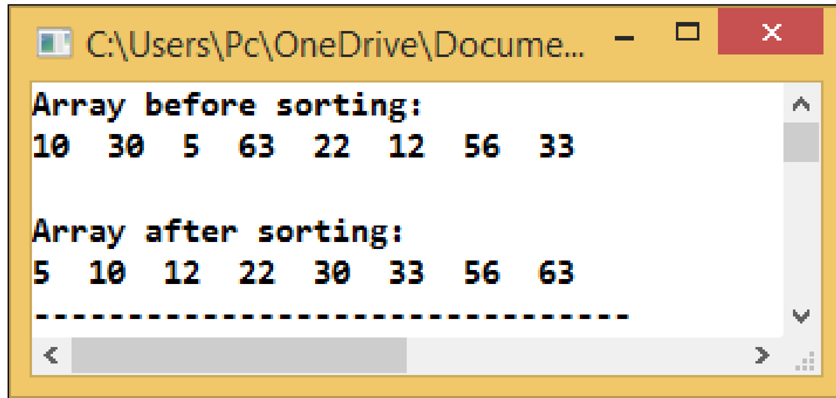
1. შევქმნათ ორობითი გროვა;
2. შევასრულოთ იტერაციები ორობითი გროვის მასივის შუიდან დასაწყისამდე;
3. ყოველ იტერაციაზე შევცვალოთ ფესვური კვანძი ბოლო ფოთლოვანი კვანძით;
4. წავშალოთ ბოლო კვანძი დახარისხებული მასივის ბოლოს მისი დაბრუნებით;
5. ხელახლა შევასრულოთ heapify ოპერაცია და გავიმეოროთ იტერაციები მე-2 ბიჯიდან;
6. დასასრული.

განხილული ალგორითმის შესაბამის პროგრამულ კოდს ქვემოთ ნაჩვენები სახე აქვს, ხოლო პროგრამის შესრულების შედეგები 168-ე სურათზეა წარმოდგენილი.

```
#include<stdio.h>
// function prototyping
void heapify(int*,int, int);
void heapsort(int*, int);
void print_array(int*, int);
int main()
{
   int arr[] = { 10, 30, 5, 63, 22, 12, 56, 33 };
   int n = sizeof(arr) / sizeof(arr[0]);
   printf("\nArray before sorting:\n");
   print_array(arr, n);
   heapsort(arr, n);
   printf("\n\nArray after sorting:\n");
   print_array(arr, n);
   return 0;
}
/* sorts the given array of n size */
void heapsort(int* arr, int n)
{
    int i;
   // build the binary max heap
   for (i = n / 2 - 1; i >= 0; i--)
   {
      heapify(arr, n, i);
```

```
    }
    // sort the max heap
    for (i = n - 1; i >= 0; i--)
    {
        // swap the root node and the last leaf node
        int temp = arr[i];
        arr[i] = arr[0];
        arr[0] = temp;
        // again heapify the max heap from the root
        heapify(arr, i, 0);
    }
}
/* heapify the subtree with root i */
void heapify(int* arr, int n, int i)
{
    // store largest as the root element
    int largest = i;
    int left = 2 * i + 1;
    int right  = 2 * i + 2;
    // now check whether the right and left right is larger than the root or not
    if (left < n && arr[left] > arr[largest])
    {
        largest = left;
    }
    if (right < n && arr[right] > arr[largest])
    {
        largest = right;
    }
// if the root is smaller than the children then swap it with the largest children's value
    if (largest != i)
    {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // again heapify that side of the heap where the //root has gone
        heapify(arr, n, largest);
    }  }
/* printf the array */
void print_array(int* arr, int n)
{
  int i;
    for (i = 0; i < n; i++)
    {
```

```
        printf("%d  ", arr[i]);
    }
}
```



სურ. 168

მოდით, ახლა დეტალურად განვიხილოთ გროვას დახარისხების ალგორითმი. აქ ჩვენ ძირითადად ორი ტიპის ოპერაციებს ვასრულებთ:

- მაქსიმალური გროვის აგება მასივის საფუძველზე.
- გავზარდოთ გროვა ყოველ ჯერზე, როდესაც ვცვლით ფესვებისა და ფოთლის კვანძებს.

ავიღოთ დაუხარისხებელი მასივის მაგალითი 8 ელემენტით (იხილეთ 169-ე სურათი).

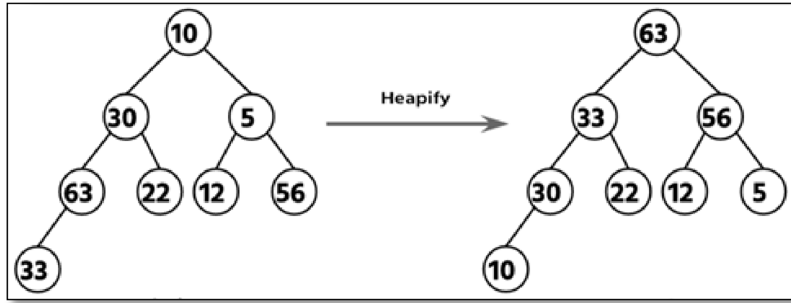
10	30	5	63	22	12	56	33

სურ. 169

ბიჯი 1: შევქმნათ მაქსიმალური გროვა მოცემული მასივიდან:

{10, 30, 5, 63, 22, 12, 56, 33}.

თავდაპირველად, ჩვენ უნდა შევქმნათ მაქსიმალური გროვა ელემენტების მოცემული მასივიდან (იხილეთ 170-ე სურათი).



მოცემული მასივის გროვა

max- გროვა

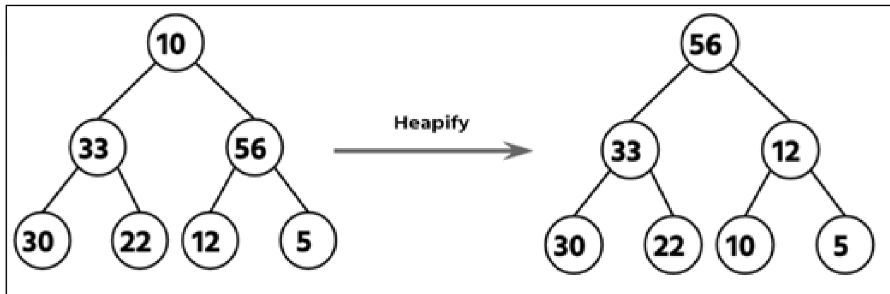
სურ. 170

მოცემული მასივი შეიცვლება და ამ ეტაპზე ის მიიღებს 171-ე სურათზე ნაჩვენებ სახეს.

63	56	33	12	30	22	10	5

სურ. 171

ბიჯი 2: წავშალოთ **root** კვანძი (63) და შევასრულოთ **heapify** ოპერაცია. ამ კვანძის წასაშლელად, ჩვენ უნდა შევცვალოთ ეს კვანძი ფოთლოვანი კვანძით 10 და წავშალოთ ფოთლის კვანძი. გაცვლის შემდეგ კვლავ უნდა ჩავატაროთ **heapify** ოპერაცია (იხილეთ 172-ე სურათი).



გროვა 63-ის წაშლის შემდეგ

max- გროვა

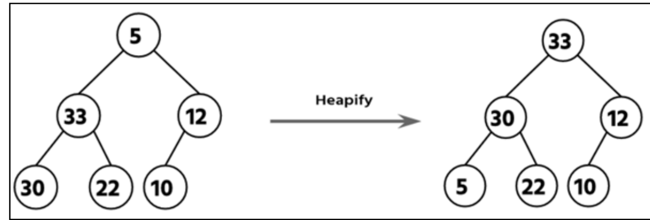
სურ. 172

მასივი შეიცვალა და 63 ინახება სიის ბოლოს (იხილეთ 173-ე სურათი).

56	33	12	30	22	10	5	63

სურ. 173

ბიჯი 3: წავშალოთ **root** კვანძი (56) და შევასრულოთ **heapify** ოპერაცია. ისევ უნდა წავშალოთ ძირეული ელემენტი 56. ამ ელემენტის წასაშლელად ფესვი უნდა შევცვალოთ ფოთლოვან კვანძთან, რომელსაც აქვს 5-ის ტოლი მნიშვნელობა. წაშლის შემდეგ უნდა შევასრულოთ **heapify** ოპე-რაცია (იხილეთ 174-ე სურათი).



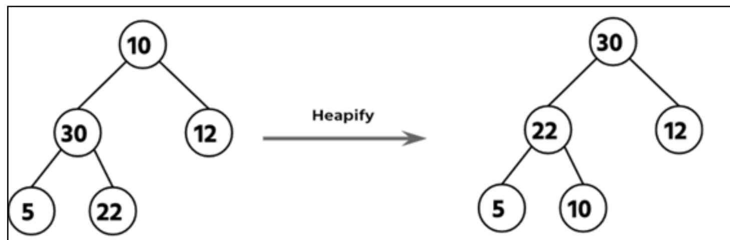
გროვა 56-ის წაშლის შემდეგ სურ. 174 **max- გროვა**

ფესვის კვაძიისა და ფოლოვანი კვანძის გაცვლის შემდეგ, ელემენტების მასივი იცვლება (იხილეთ 175-ე სურათი).

სურ. 175

ბიჯი 4: წავშალოთ **root** კვანძი (33) და შეასრულეთ **heapify** ოპერაცია.

ახლა, ჩვენ უნდა წავშალოთ ძირეული კვანძი 33. მის წასაშლელად, უნდა შევასრულოთ ჩანაცვლების ოპერაცია ძირეულ კვანძსა და ფოთლოვან კვანძს (10) შორის (იხილეთ 176-ე სურათი).



გროვა 33-ის წაშლის შემდეგ **max- გროვა**

სურ. 176

მასივი ელემენტის წაშლის შემდეგ მიიღებს 177-ე სურათზე ნაჩვენებ სახეს.

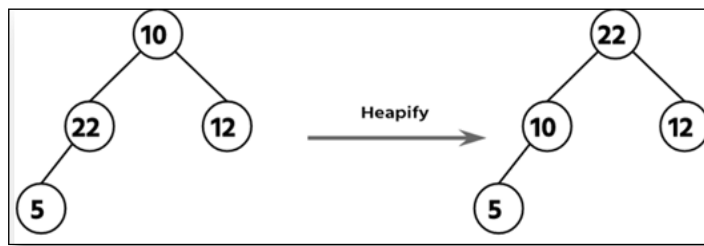
30	22	12	5	10	33	56	63

სურ. 177

ბიჯი 5: წავშალოთ **root** კვანძი (30) და შევასრულოთ **heapify** ოპერაცია.

ამ ეტაპზე, უნდა წავშალოთ ძირეული კვანძი 30-ს ტოლი მნიშვნელობით. ძირეული კვანძის წასაშლელად, საჭიროა, შევასრულოთ **swap** (გაცვლის) ოპერაცია ძირეულ კვანძსა და ფოთლოვან კვანძს შორის, რომელსაც აქვს 10-ის ტოლი მნიშვნელობა (იხილეთ 178-ე სურათი).

ძირეული კვანძის წაშლისა და შეცვლის შემდეგ, მასივი მიიღებს 179-ე სურათზე ნაჩვენებ სახეს.



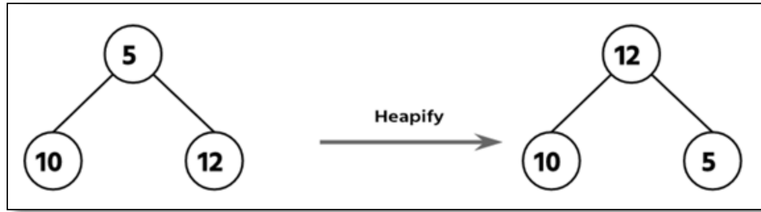
გროვა 30-ის წაშლის შემდეგ **max- გროვა**

სურ. 178

22	10	12	5	30	33	56	63

სურ. 179

ბიჯი 6: წავშალოთ ფესვის კვანძი (22) და ფოლოვანი კვანძი. შევასრულოთ ჩანაცვლების ოპერაცია. ამ ეტაპზე წავშალოთ **root** კვანძი. ძირეული კვანძის წასაშლელად, უნდა შევცვალოთ ძირეული კვანძი ბოლო ფოლოვანი კვანძთან (5). ფოლოვანი კვანძის შეცვლის შემდეგ, ჩვენ უნდა გავზარდოთ გროვა (იხილეთ 180-ე სურათი).



გროვა 22-ის წაშლის შემდეგ **max- გროვა**

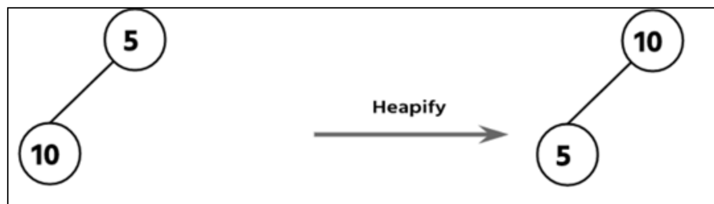
სურ. 180

ახლა მასივი იცვლება ფესვისა და ფოთლოვანი კვანძის წაშლის შემდეგ (იხილეთ 181-ე სურათი).

სურ. 181

ბიჯი 7: წავშალოთ ფესვის კვანძი (12) და შევასრულოთ ჩანაცვლების ოპერაცია.

ამ ეტაპზე წავშალოთ **root** კვანძი. ძირეული კვანძის წასაშლელად, უნდა შევცვალოთ ძირეული კვანძი ბოლო ფოთლოვან კვანძთან (5). ფოთლოვანი კვანძის შეცვლის შემდეგ, საჭიროა გავზარდოთ მაქსიმალური გროვა (იხილეთ 182-ე სურათი).



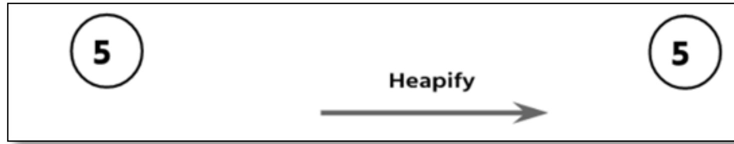
გროვა 12-ის წაშლის შემდეგ სურ. 182 **max- გროვა**

მასივი იცვლება გაცვლის ოპერაციის შემდეგ და ის 183-ე სურათზე ნაჩვენებ სახეს იღებს.

სურ. 183

ბიჯი 8: წავშალოთ ფესვის კვანძი (10) და შევასრულოთ **heapify** ოპერაცია.

აღნიშნულ საფეხურზე საჭიროა, წავშალოთ ძირეული კვანძი 10. ძირეული კვანძის წასაშლელად უნდა ჩავატაროთ გაცვლის ოპერაცია ორ ძირეულ კვანძსა და ბოლო კვანძს (5) შორის. გაცვლის შემდეგ შევასრულოთ **heapify** ოპერაცია (იხილეთ 184-ე სურათი).



გროვა 10-ის წაშლის შემდეგ **max- გროვა**

სურ. 184

root (ფესვის) კვანძის წაშლის შემდეგ, მასივი იცვლება და იღებს 185-ე სურათზე ნაჩვენებ სახეს.

5	10	12	22	30	33	56	63

სურ. 185

ბიჯი 9: წავშალოთ ფესვური კვანძი.

ამ ეტაპზე საჭიროა, წავშლოთ **root** კვანძი. ვინაიდან მხოლოდ ერთი კვანძია დარჩენილი, რომლის წაშლის შემდეგ გროვა ცარიელი იქნება. საბოლოოდ, დახარისხებულ მასივს 186-ე სურათზე ნაჩვენები სახე აქვს.

5	10	12	22	30	33	56	63

სურ. 186

ლიტერატურა:

1. Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, Charles E. Leiserson, Introduction to Algorithms, The MIT Press (3rd Edition), 2009.
2. Brian W. Kernighan, Dennis M. Ritchie, C Programming Language, 2nd Edition, Prentice Hall, 1988.
3. გია სურგულაძე, ლია პეტრიაშვილი. დაპროგრამების საფუძვლები (C–ენის ბაზაზე), ნაწ. I და II, საგამომც.სახლი „ტექნიკური უნივერსიტეტი“, 2005. https://gtu.ge/katedrebi/kat94/pdf/C-1.pdf; https://gtu.ge/katedrebi/kat94/pdf/C-2.pdf
4. გია სურგულაძე. კომპიუტერული პროგრამირების მეთოდები და მეთოდოლოგიები (SP, OOP, VP, Agile, UML). სტუ. „IT-კონსალტინგ ცენტრი“. თბ., 2019. -200 გვ. https://gtu.ge/book/Surg_ProgMethod_2019.pdf

Lela Gachechiladze, Nana Kurkumuli. Programming in C language.

© „IT-Consulting scientific center" of GTU, Tbilisi, 2024

ISBN 978-9941-8-7281-5

იბეჭდება ავტორთა მიერ
წარმოდგენილი სახით

გადაეცა წარმოებას 15.10.2024. ოფსეტური ქაღალდის ზომა 60X84 1/16. პირობითი ნაბეჭდი თაბახი 13,5. ტირაჟი 50 ეგზ.

სტუ-ის „IT კონსალტინგის სამეცნიერო ცენტრი",
თბილისი, მ.კოსტავას 77