თავი 1: შესავალი

- C++-ის ისტორია
- რა არის პროგრამა?
- კომპიუტერში მთელი რიცხვების წარმოდგენა
- C++-ის ერთი ნაწილი, რაც თანაკვეთაშია მათემატიკასთან
- როგორი უნდა იყოს იდეალური პროგრამა

C++-ის ისტორია

პროგრამირების ენა C შეიქმნა 1972 წელს პროგრამისტ დენის რიჩის (Dennis Ritchie) მიერ. თუმცა, თუ მოვინდომებთ პროგრამაში მიმდინარე დროის დაბეჭდვას, C დაბეჭდავს 1970 წლის პირველი იანვრიდან გასულ დროს წამებში, რაც მიჩნეულია C- ის ერის დასაწყისად. ახალ ენას ასეთი სახელი ავტორმა უწოდა იმის გამო, რომ ენას რომელსაც იგი მანამდე იყენებდა, ერქვა B.

1979 წელს ბიარნ სტრაუსტრუპმა (Bjarne Stroustrup) დაიწყო ამ ენის გაუმჯობესებული ვერსიის შექმნა (Bell Labs-ში). ახალი ენის თავდაპირველი სახელწოდება იყო „C კლასებით“, 1983 -სი მას C++ ეწოდა.

რა არის პროგრამა?

პროგრამა შედგება ორი მთავარი ნაწილისგან: მონაცემებისა და ინსტრუქციებისგან.

უმცირესი მონაცემი, რომლის აღქმაც შეუძლია კომპიუტერს, არის 0 ან 1, ან უფრო ზუსტად "-" ან "+", რისი ჩაწერა და წაკითხვაც მას შეუძლია ფიზიკური მეხსიერების უმცირეს ერთეულში. ეს იმის გამო, რომ ჯერ-ჯერობით ელექტრონულ სქემებს მხოლოდ ორ მდგრად მდგომარეობაში შეუძლიათ ყოფნა. მონაცემთა ამ უმცირეს ერთეულს ეწოდება ბიტი (ინგლისური ტერმინი bit წარმოადგენს ”binary digit” -ის ანუ ორობითი ციფრის შემოკლებას). ნებისმიერი მონაცემი, რომელიც მუშავდება კომპიუტერის მიერ, წარმოიდგინება ნულებისა და ერთების კომბინაციით და მის დასამახსოვრებლად საჭიროა ბიტების გარკვეული რაოდენობა. შემდეგ, შედარებით მარტივი ტიპის მონაცემებისგან შესაძლებელია უფრო რთული კონსტრუქციების შექმნა, მათგან კიდევ უფრო რთულის და დახვეწილის და ა.შ.

C++-ში ჩაშენებულია რამდენიმე მარტივი საბაზო ტიპი მონაცემებისთვის. თითოეული ტიპისთვის გამოყოფილია ბიტების მკაცრად დადგენილი რაოდენობა. ამათგან აიგება ყველა დანარჩენი. ზოგადად, მონაცემები და ინსტრუქციები ანალოგიურია მათემატიკის სიმრავლეებისა და ფუნქციებისა. ამიტომ, ვიდრე C++ -ის მონაცემთა ტიპებსა და ფუნქციებზე ვისაუბრებთ, ჯერ უნდა გავარკვიოთ რა ცოდნა შეგვიძლია გამოვიყენოთ მათემატიკიდან. მანამდე მოკლედ შევეხოთ კომპიუტერში რიცხვების წარმოდგენის საკითხს.

კომპიუტერში მთელი რიცხვების წარმოდგენა

სიმარტივისთვის განვიხილოთ მხოლოდ მთელი რიცხვების წარმოდგენის საკითხი. იმისათვის, რომ კომპიუტერმა მთელ რიცხვებზე არითმეტიკული ოპერაციები განახორციელოს, მათ ტოლი რაოდენობის ბიტები უნდა ჰქონდეს გამოყოფილი. მათემატიკაში არის ერთი სიმრავლე მთელი რიცხვებისა, ხოლო C++ -ში არსებობს რამდენიმე ასეთი სიმრავლე, თითოეული სხვადასხვა დიაპაზონს მოიცავს, კონკრეტულ ამოცანაში შეგვიძლია ისე შევარჩიოთ გამოყენებული მთელი რიცხვები, რომ მათთვის რაც შეიძლება ნაკლები მეხსიერების (მაშასადამე პროგრამული დროის) დათმობა გახდეს საჭირო. ყველაზე მცირე დიაპაზონის მთელი რიცხვები არის 0 და 1. თუ ამოცანაში ასეთი რიცხვებია საჭირო, მაშინ პროგრამაში ვიყენებთ ეგრეთ წოდებულ ბულის ტიპის ცვლადებს (**`bool`**), შემდეგ არის ძალიან მოკლე მთელი რიცხვების სიმრავლე (**`char`**), თითოეული ცვლადი ამ სიმრავლიდან იკავებს 8 ბიტს, ანუ ერთ ბაიტს. **`char`** ტიპი (სიმრავლე)

ორობითი	ათობითი
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	−1
1110	−2
1101	−3
1100	−4
1011	−5
1010	−6
1001	−7
1000	−8

კონკრეტული მიზნით შეიქმნა, **ASCII** კოდების შესანახად. ამჟამად ეს სისტემა კვლავ ინტენსიურად გამოიყენება, თუმცა უკვე მოძველებულად ითვლება. შემდეგი არის მოკლე მთელი რიცხვების სიმრავლე (**short**), თითოეული ცვლადი ამ სიმრავლიდან იკავებს 16 ბიტს, ანუ 2 ბაიტს, შემდეგ

int, long, long long.

ზოგადად, ვთქვათ გამოყოფილია N ბიტი. მათ შორის ერთი (უფროსი) ბიტი გვიჩვენებს რიცხვის ნიშანს: თუ ნიშანთვისების ბიტში დგას 0 – რიცხვი დადებითია ან ნულია, თუ კი ეს ბიტი შეიცავს 1 –ს – რიცხვი უარყოფითია (უარყოფითი რიცხვებისთვის, ნიშანთვისების ბიტი მოქმედებს აგრეთვე რიცხვის სიდიდეზეც). არაუარყოფითი რიცხვები წარმოიდგინება, ჩვეულებრივ, ორობითი ჩანაწერით დიაპაზონში 0-იდან $2^{N-1}-1$ -მდე. უარყოფითი რიცხვის ორობით ჩანაწერს მივიღებთ თუ 2^N -ს დავაკლებთ ამ უარყოფითის აბსოლუტურ მნიშვნელობას. ამ სისტემას ეწოდება two's complement system.

ახლა ეს სისტემა ყველაზე გავრცელებულია. მართალია მთავარი თანრიგი განსაზღვრავს ნიშანს, მაგრამ ეს არ ნიშნავს რომ მთელი რიცხვი და მისი მოპირდაპირე რიცხვი მხოლოდ ამ თანრიგით განსხვავდებიან. მაგალითად, შეგვიძლია ვნახოთ ყველა 4 ბიტიანი მთელი რიცხვი ზედა ცხრილში ან რამდენიმე 8 ბიტიანი მთელი რიცხვი შემდეგ ცხრილში:

მთავარი ბიტი									
0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**=**	**127**
0	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**=**	**126**
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**=**	**2**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**=**	**1**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**=**	**0**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**=**	**−1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**=**	**−2**
1	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**=**	**−127**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**=**	**−128**

8-ბიტიანი მთელები

უფრო კონკრეტულად, შეგიძლიათ იხილოთ http://en.wikipedia.org/wiki/Two's_complement.

C++-ის ერთი ნაწილი, რაც თანაკვეთაშია მათემატიკასთან

ნებისმიერ ენას ბევრი საერთო აქვს მათემატიკასთან, რადგან გარკვეული აზრით მათემატიკა თვითონ არის ენა. ეს მსგავსება განსაკუთრებით საგრძნობი გახდა C -ის შექმნის (და მის საფუძველზე ახალი ენების განვითარების) შემდეგ, რადგან ამ ენაში ყველა პროგრამა ფუნქციების კრებულია, ხოლო მონაცემები წარმოადგენენ სპეციფიკურ სიმრავლეებს, რომლებიც განუყოფელია მათზე განსაზღვრული ოპერაციებისგან. მართალია, მათემატიკური და C++ -ში გამოყენებული აღნიშვნები საგრძნობლად განსხვავდება, მაგრამ თუ მათემატიკის ელემენტების მცოდნე ადამიანი ერთხელ გაერკვევა განსხვავებებში და მსგავსებებში, მაშინვე აღმოაჩენს, რომ გარკვეული საბაზო ცოდნა უკვე გააჩნია C++ -ში.

სიმრავლეები. ჯერ კიდევ სასკოლო პროგრამებში ჩვენ ვხვდებით ნატურალური რიცხვების N , მთელი რიცხვების Z , რაციონალური რიცხვების Q და ნამდვილი რიცხვების R სიმრავლეებს. C++ ენაში "ჩაშენებულია" ორი მათგანი: მთელი რიცხვების და ნამდვილი რიცხვების (ვუწოდებთ ნამდვილ რიცხვებს, თუმცა რეალურად მისი ელემენტები არის სასრული ათწილადები) სიმრავლეები. თითოეული მათგანის რამდენიმე ვერსიაა შემოთავაზებული, რაც საშუალებას აძლევს პროგრამისტს მაქსიმალურად დაზოგოს კომპიუტერული მეხსიერება და დრო. თითოეულ რეალიზაციას აქვს თავისი დიაპაზონი, თანრიგების რაოდენობა (რასაც სისტემა გამოყოფს ასეთი რიცხვების ჩაწერისთვის) და მათემატიკური ოპერატორები, რომლებიც ხელახლა გადაიტვირთება თითოეული რეალიზაციისთვის. მათემატიკური ოპერატორების გადატვირთვა ნიშანვს, რომ ერთი და იგივე ოპერატორი სხავასხვა სიმრავლეზე აღინიშნება ერთნაირად, მაგრამ მოქმედებს განსხვავებული წესით. ამის გაკეთება აუცილებელია, რადგან, შესაძლოა ორი მთელი რიცხვის შეკრების შედეგი დამოკიდებული იყოს იმაზე, თუ როგორაა რეალიზებული ისინი (ანუ რომელი სიმრავლის ელემენტებად განვიხილავთ მათ). მაგალითად, თუ a არის **char** ტიპის ცვლადი (ერთბაიტიანი წარმოდგენით) რომლის მნიშვნელობაა 127, მაშინ შეკრების ოპერატორი, გადატვირთული ამ სიმრავლეზე, a+a -ის შედეგად მოგვცემს -2-ს. ხოლო a++ -ის შედეგად -128-ს. თუ a იქნებოდა **int** ტიპის (ან სხვა რაიმე ტიპის, რომელიც ერთ ბაიტზე მეტს გამოყოფს რიცხვის შესანახად), მაშინ შეკრების შედეგები იქნებოდა ისეთი, რასაც მიჩვეული ვართ. სხვა მაგალითი არის გაყოფის ოპერაცია "/". თუ ორ მთელ რიცხვს გავყოფთ ერთმანეთზე, გაყოფის შედეგიც მთელია (წილადი ნაწილი იგნორირდება), თუ ნამდვილ რიცხვებს გავყოფთ ერთმანეთზე იგივე (ოღონდ სხვა ტიპზე გადატვირთული ოპერატორის) საშუალებით, მაშინ შედეგი იქნება ისეთი, როგორსაც მიჩვეული ვართ.

როგორც ვხედავთ, C++ -ში მოცემული ორი მთელი რიცხვისთვის ერთი და იგივე არითმეტიკული ოპერატორების მოქმედების შედეგი შესაძლოა განსხვავდებოდეს ერთმანეთისაგან, რადგან მთელ რიცხვთა აბსტრაქტულ სიმრავლეს შეესაბამება რამდენიმე რეალიზაცია. იგივე მიზეზის გამო მათ ერთი და იგივე სახელი ვერ ერქმევათ (იგივე შეგვიძლია ვთქვათ ნამდვილ რიცხვებზეც). ამ ენაში სიმრავლის და საერთოდ ნებისმიერი რამის სახელი ისეა შერჩეული, რომ მაქსიმალურად შეესაბამებოდეს შინაარსს (სახელსა და შინაარსს შორის შინაგანი კავშირის შენარჩუნება, სხვათა შორის პროგრამირების კარგი სტილის ერთ-ერთი რეკომენდაციაა): **bool**, **char**, **short**, **int**, **long**, **long long**. იგივეა ნამდვილი რიცხვებისთვის, რომლის სხვადასხვა რეალიზაციებიდან შეგვიძლია აღვნიშნოთ **float**, **double**, **long double**.

მათემატიკაში, ბევრი გავრცელებული სიმრავლე განიმარტება სხვა, უფრო მარტივი სიმრავლეების საფუძველზე. ანალოგიური ვითარებაა C++ -ში, სადაც არსებობს უკვე განმარტებული ტიპებიდან ახალი ტიპების შექმნის მექანიზმი, მაგალითად კლასების გამოყენებით.

სიმრავლის ელემენტები. უახლოეს რამდენიმე მაგალითში ვიგულისხმოთ, რომ მთელ რიცხვთა სიმრავლის წარმოდგენისთვის ვსარგებლობთ 32 ბიტით, ანუ ვიყენებთ **int** ტიპს, ნამდვილი რიცხვების წარმოდგენისთვის ვიყენებთ **float** ტიპს. კიდევ ერთხელ დავაზუსტოთ, რომ ტიპი არის სიმრავლე და მასზე განსაზღვრული ოპერატორები (არა მხოლოდ არითმეტიკული, მაგალითად, სხვადასხვა ტიპზე ძალიან გავრცელებულია მინიჭების "=" და შედარების '==" ოპერატორები) . განვიხილოთ კარგად ცნობილი მათემატიკური ჩანაწერი: $x \in R$ და $j \in Z$. C++ -ში მიკუთვნებისთვის არ ხდება სპეციალური კვანტორის გამოყენება და ვწერთ უბრალოდ:

```
int j;
float x;                                                    (1)
```

რომელია უფრო მოხერხებული? ალბათ ბოლოს შემოღებული, რადგან მისი განზოგადება უფრო ადვილია. მაგალითად, თუ გვინდა C++ -ში ჩავწეროთ ფაქტი "ნამდვილი x ცვლადის მნიშვნელობა არის 12.95-ის ტოლი", (1) სტრიქონს შეცვლით სულ ოდნავ:

```
float x = 12.95;          // C ენაში მიღებული  ინიციალიზაციის ფორმა      (2)
```

ან

```
float x(12.95);           // C++ ენში მიღებული  ინიციალიზაციის ფორმა
```

თუმცა შეგვიძლია მათემატიკური ჩანაწერის ორი ($x \in R,\ x = 12.95$) წინადადების ანალოგიურად გამოვიყენოთ ორი შეტყობინება (`statement`):

```
float x;
x = 12.95;
```

პროგრამის ფრაგმენტებთან დაკავშირებით ხშირად გამოიყენება ტერმინი შეტყობინება და არა წინადადება, რადგან პროგრამული კოდი განკუთვნილია კომპილერისთვის, (2) შეტყობინების შედეგს წარმოადგენს ის, რომ მეხსიერებაში x ცვლადისთვის დაიჯავშნება მონაკვეთი (იმდენი ბიტი, რამდენიც ზოგადად გამოიყოფა კონკრეტული სისტემის მიერ **float** ტიპის ცვლადისთვის), და ამავე დროს ამ მონაკვეთში ჩაიწერება 12.95. ადამიანისთვის (2) წარმოადგენს ცვლადის აღწერას, ხოლო პროგრამისთვის განაცხადს კონკრეტული ცვლადისთვის საჭირო მეხსიერებაზე.

სავარჯიშო: როგორ ჩავწეროთ, რომ

1. x არის გრძელი ნამდვილი რიცხვი?
2. y არის ნამდვილი რიცხვი მნიშვნელობით 27238.32?
3. k არის მოკლე მთელი რიცხვი მნიშვნელობით 456?
4. k არის ძალიან მოკლე მთელი რიცხვი მნიშვნელობით 45?
5. t არის ბულის ტიპის რიცხვი ჭეშმარიტი მნიშვნელობით ?

ფუნქციები. ვნახოთ როგორ ხდება ფუნქციებთან დაკავშირებული ჩანაწერების კონვერტირება C++ -დან მათემატიკაში და პირიქით. მათემატიკური ჩანაწერი $f(x): R \to Z$ ნიშნავს რომ $f(x) \in Z$ ანუ $\text{int } f(x)$; და $x \in R$ ანუ $\text{float } x$. ამ ორი ფაქტის გაერთიანება გვაძლევს შესაბამის C++ –ის ჩანაწერს: $\text{int } f(\text{float } x)$;

სავარჯიშო: რას ნიშნავს:

6. $function(z): Z \to Z$?
7. $F(y): Z \to R$?
8. $F(x, y): Z^2 \to R$?
9. $\text{int } T(\text{float } a)$?

გარდა მსგავსებისა, მნიშვნელოვანია განსხვავების ცოდნა. მათემატიკაში, თუ ორ ფუნქციას აქვს ტოლი განსაზღვრის და მნიშვნელობათა სიმრავლეები და ისინი ერთი და იმავე

არგუმენტისთვის იღებენ ტოლ მნიშვნელობებს, მაშინ თვითონაც ითვლებიან ტოლად. პროგრამირებაში, გასათვალისწინებელია ალგორითმი, რომლითაც ხდება ამ ფუნქციების მნიშვნელობების გამოთვლა. თუ განსხვავებულია ალგორითმი, ფუნქციებიც განსხვავებულად ითვლებიან. მაგალითად, ფუნქციები $4x$ და $x+x+x+x$, განსხვავდებიან თავისი შესრულების სისწრაფით.

როგორი უნდა იყოს იდეალური პროგრამა?

არსებობს კრიტერიუმები, რომლებსაც უნდა აკმაყოფილებდეს იდეალური პროგრამა. ზოგჯერ მათი ერდროულად შესრულება შეუძლებელია, ზოგჯერ განზრახ ხდება საჭირო რომელიმე მათგანის დარღვევა, მაგრამ ზოგადად ითვლება, რომ იდეალური პროგრამა უნდა იყოს:

- მრავალჯერადი, გამოყენების თვალსაზრისით;
- ადვილად გაუმჯობესებადი გადაკეთების თვალსაზრისით და მარტივი ექსპლოატაციაში;
- საიმედოდ დაწერილი (მაგალითად, ნაკლებად დამოკიდებული კონკრეტულ სისტემაზე);
- ადვილად გასარჩევი;
- კარგად დოკუმენტირებული (ანუ ახლდეს ყველა საჭირო კომენტარი და ახსნა-განმარტება).

იდეალური პროგრამების შესაქმნელად აუცილებელია იმ გამოცდილების გამოყენება, რაც დაგროვდა საუკეთესო პროგრამისტების მიერ და რაც კონდენსირებულია სტილების და პარადიგმების სახით. ჩვენს კურსში ყურადღებას გავამახვილებთ რამდენიმე მომენტზე, რასაც პროგრამირების კარგი სტილი გვირჩევს: კომენტარების გამოყენებაზე და პროგრამის ფრაგმენტების შეწევაზე.

C++ –ში კოდის მრავალჯერადი განმეორების იდეა ეფუძნება ფუნქციების და კლასების გამოყენებას. სტანდარტული ბიბლიოთეკის სახით ენა თავაზობს პროგრამისტებს უამრავ ფუნქციას და კლასს. კერძოდ, C++ –ის ყველა პროგრამა იყენებს სტანდარტულ შეტანა–გამოტანის კლასებს.

უმარტივესი C++ –პროგრამის სტრუქტურა შემდეგია:

```
#include <iostream>    // მიმართვა შეტანა–გამოტანის
using namespace std;   // სტანდარტულ კლასებზე

// პროგრამის მთავარი ფუნქცია
int main()
{
      // მონაცემებზე განაცხადი
      // და შესრულებადი ინსტრუქციები
}
```

მაგალითად, ქვემოთ მოყვანილი პროგრამა იპოვის და დაბეჭდავს ორი მთელი რიცხვის ჯამს:

```
#include <iostream>
using namespace std;

int main()
{
   int number1(9), number2(-4), sum;
   sum = number1 + number2;
   cout<<"Sum = "<<sum<<endl;
   return 0;
}
```

თავი 2: პროგრამირების კარგი სტილი

- რას წარმოადგენს სტილი
- კომენტარები
- პროგრამული კოდის კომენტირება. ცვლადის სახელი როგორც კომენტარის ფორმა
- კოდის ფორმატირება წანაცვლების საშუალებით
- სიცხადე და სიმარტივე
- როგორი უნდა იყოს იდეალური პროგრამა
- პროგრამები, ამოცანის დასმიდან შესრულებამდე

რას წარმოადგენს სტილი

სტილი დაპროგრამების მნიშვნელოვანი ნაწილია, რადგან მასში კონცენტრირებულია უმდიდრესი გამოცდილება, რაც დააგროვეს საუკეთესო პროგრამისტებმა. ეს ცოდნა გვიცავს უამრავი ძნელად წარმოსადგენი შეცდომისა და დროის უაზრო ხარჯვისგან. ამიტომ ვიწყებთ კარგი სტილის გაცნობას თავიდანვე. დაპროგრამების კარგი სტილის გამოყენების გარეშე მარტივი და ადვილად წასაკითხი პროგრამის შექმნა თითქმის წარმოუდგენელია.

გავრცელებული შეხედულების საწინააღმდეგოდ, სინამდვილეში პროგრამისტი დროის უმეტეს ნაწილს ხარჯავს არა პროგრამების დაწერაზე, არამედ არსებული პროგრამების გამართვაზე, ექსპლუატაციაზე და გაუმჯობესებაზე. რაც დრო გადის, ტიპურ გამოყენებით პროგრამებში სტრიქონთა საშუალო რაოდენობა სულ უფრო იზრდება. მაგალითად, 1980–იდან 1990 წლამდე საშუალო ზომის გამოყენებით ამოცანაში სტრიქონების რაოდენობა გაიზარდა 23 000-დან 1 200 000-მდე. შესაბამისად, რთულდება კარგი სტილის დაცვით დაწერილი პროგრამული კოდის გარჩევაც კი. მაგალითად, ერთ–ერთ კონფერენციაზე მენეჯერთა 74%-მა განაცხადა, რომ მათ უხდებათ ისეთ სისტემებთან მუშაობა, რომელთა ექსპლუატაცია მხოლოდ კონკრეტულ ადამიანებს შეუძლიათ, რადგან მათ გარდა ვერავინ ვერაფერს არკვევს. პროგრამული პროდუქტების (software) უმეტესობა ეფუძნება უკვე არსებულ პროგრამულ პროდუქტებს. ამიტომ არსებითი მნიშვნელობა აქვს იმას, რომ შექმნილი პროგრამა იყოს მაქსიმალურად გასაგები ნებისმიერი მომხმარებლისათვის.

ზოგიერთი პროგრამისტი თვლის, რომ პროგრამის დანიშნულებაა მხოლოდ კომპიუტერის უზრუნველყოფა ინსტრუქციების კომპაქტური კრებულით. ასეთ პროგრამებს აქვს ორი ნაკლი:

- გასამართად ძნელია, რადგან დროის გარკვეული პერიოდის შემდეგ ავტორსაც უჭირს მისი გაგება.
- ძნელია პროგრამის მოდიფიცირება და გაუმჯობესება, რადგან პროგრამისტს სჭირდება საკმაო დრო იმის გასარკვევად, თუ რას აკეთებს პროგრამა.

კომენტარები

იდეალურ შემთხვევაში, პროგრამა ემსახურება ორ მიზანს:

- უზრუნველყოს კომპიუტერი ინსტრუქციების კრებულით;
- უზრუნველყოს პროგრამისტი ნათელი, გასაგები ენით დაწერილი აღწერებით იმის შესახებ, თუ რას აკეთებს პროგრამა.

საბედნიეროდ, ორივე ამ მიზნის მიღწევა შესაძლებელია ერთდროულად. ამის საშუალებას იძლევა **კომენტარები**, რომლებიც უკეთდება ინსტრუქციებს აუცილებლობის

შემთხვევაში. კომენტარი არ კომპილდება, ამიტომ გავლენას არ ახდენს პროგრამის შესრულებაზე, იგი საჭიროა პროგრამისტებისთვის და არა კომპიუტერისთვის.

პროგრამა აუცილებლად უნდა შეიცავდეს კომენტარებს. დანერგილი პროგრამა, რომელიც არ შეიცავს კომენტარებს, შენელებული მოქმედების ნაღმს წააგავს, რომელიც თავის წამს ელის. ადრე თუ გვიან ვიღაც აღმოაჩენს შეცდომას ამ პროგრამაში, ან კიდევ ვინმე შეეცდება მის გადაკეთებას და გაუმჯობესებას, კომენტარების არარსებობა კი მის სამუშაოს ძალზე გაართულებს.

C++ -ის სტანდარტულ ვერსიაში (ANSI/**ISO C++ Standard**) შესაძლებელია ორი სახის კომენტარის გამოყენება:

- C ენაში მიღებული კომენტარები, რომლებიც მოთავსებულია /* და */-ს შორის;
- ე.წ. ორმაგსლეშიანი კომენტარი, რომელიც ერთსტრიქონიანია და ვრცელდება ორმაგი სლეშის (//) მარჯვნივ.

C++ -ის ზოგიერთ გაფართოებაში, მაგალითად C++/CLI-ში, რომელსაც იყენებს Visual Studio, შესაძლებელია ე.წ. სამსლეშიანი კომენტარების გამოყენება, რაც უკავშირდება ინტრგრირებული XML დოკუმენტაციის გამოყენებას და მდგომარეობს შემდეგში: პროგრამული პროდუქტის შემქმნელი ვალდებულია შექმნას კარგად კომენტირემული პროგრამული კოდი, და შექმნას აგრეთვე დოკუმენტაცია, სადაც დაწვრილებით იქნება აღწერილი ამ პროდუქტთან დაკავშირებული ყველა ასპექტი. დოკუმენტაციის მომზადება ითვლება ყველაზე მომაბეზრებელ და დამღლელ ეტაპად პროგრამული პროდუქტის მომზადების პროცესში. სამმაგსლეშიანი კომენტარები საშუალებას იძლევა, რომ კომპილირების პროცესში კომენტარებისგან შეიქმნას პროდუქტის დოკუმენტაცია XML-ის სხვადასხვა ფორმატში. ამ ლექციაში ამ საკითხს არ შევეხებით მეტად, რადგან ეს C++ -ის სტანდარტულ ვერსიას ნაკლებად ეხება.

იმისთვის, რომ შევქმნათ პროგრამა, ნათლად უნდა წარმოვიდგინოთ, თუ რის გაკეთება გვინდა და ჩავწეროთ მარტივად და გასაგებად. შემდეგ ეს ყველაფერი შეიძლება კიდევ ერთხელ შემოწმდეს და "გადაითარგმნოს" კომპიუტერულ პროგრამად, ხოლო ჩვენი ჩანაწერები გამოვიყენოთ კომენტარებად. რადგან პროგრამა სხვადასხვა ნაწილისგან შედგება, კომენტარებიც სხვადასხვანაირია.

როგორც წესი, პროგრამას ყოველთვის უკეთდება საწყისი კომენტარების ბლოკი, რომელიც რამდენიმე პუნქტისგან შედგება. ზოგ შემთხვევაში ყველა მათგანის მოყვანა არც არის აუცილებელი. ეს პუნქტებია:

- **სათაური.** პირველი კომენტარი უნდა შეიცავდეს პროგრამის დასახელებას. აქვე შეგიძლიათ ჩაურთოთ მოკლე აღწერა იმისა, თუ რას აკეთებს პროგრამა. თქვენ შეიძლება გქონდეთ ყველაზე საუკეთესო პროგრამა, რომელიც მსოფლიო მნიშვნელობის ამოცანებს წყვეტს, მაგრამ პროგრამა გამოუსადეგარი იქნება, თუ არავის ეცოდინება, რას აკეთებს იგი.
- **ავტორი.** მონაცემები თქვენს შესახებ. თუ ვინმე დააპირებს თქვენი პროგრამის შეცვლას, შესაძლებლობა ექნება ინფორმაციისათვის ან დახმარებისათვის თქვენ მოგმართოთ.
- **მიზანი.** რისთვის დაწერეთ ეს პროგრამა?
- **გამოყენება.** ამ განყოფილებაში მოკლედ აღწერეთ, როგორ უნდა მართონ პროგრამა. იდეალურ შემთხვევაში ყველა პროგრამას უნდა ახლდეს დოკუმენტთა კრებული, რომელშიც აღწერილი იქნება, თუ როგორ გამოვიყენოთ იგი.
- **საცნობარო ინფორმაცია.** არსებული პროგრამების სხვადასხვა ფრაგმენტის თქვენს მიერ გამოყენება (კოპირება) წარმოადგენს დაპროგრამების გავრცელებულ ფორმას, და სრულიად კანონიერ ფორმასაც, თუ თქვენ არ არღვევთ ამ დროს საავტორო უფლებებს. ამ პუნქტში უნდა მიუთითოთ ავტორი ნებისმიერი ნაშრომისა, რომელითაც თქვენ ისარგებლეთ (რომლის ფრაგმენტების კოპირებაც მოახდინეთ).

- **ფაილების ფორმატი.** ჩამოთვალეთ ის ფაილები, რომლებსაც თქვენი პროგრამა კითხულობს ან ქმნის და აღწერეთ მოკლედ მათი ფორმატი.
- **შეზღუდვები.** ჩამოთვალეთ ნებისმიერი შეზღუდვები ან დასაშვები საზღვრები, მაგ. ასეთი ტიპისა "პროგრამა არ ამოწმებს შეცდომებს მონაცემებში".
- **ცვლილებების ქრონოლოგია.** ეს განყოფილება შეიცავს სიას, სადაც ნაჩვენებია ვინ შეცვალა პროგრამა, როდის და რა ცვლილებები იქნა მასში შეტანილი.
- **შეცდომების გასწორება.** თუ პროგრამა აღმოაჩენს შეცდომას, აღწერეთ რას მოიმოქმედებს ამ დროს იგი. მაგალითად, დაუყოვნებლივ შეწყვეტს მუშაობას, თუ მიგვითითებს შეცდომის შესწორების გზას.
- **შენიშვნები.** აქ შეიტანეთ სპეციალური კომენტარები ან სხვა ინფორმაცია, რომელიც ზემოთ არ გქონდათ მოყვანილი.

კომენტარებში ახსნილი უნდა იყოს ყველაფერი, რაც პროგრამისტმა უნდა იცოდეს პროგრამის შესახებ, მაგრამ ზედმეტი არაფერი. დარწმუნებული უნდა იყოთ, რომ ყველაფერი, რაც შეგაქვთ კომენტარებში, საჭიროა.

ტექსტურ რედაქტორებში (მაგალითად, Word) ჩვეულებრივი ტექსტის ბეჭდვის დროს ჩვენ შეგვიძლია შევცვალოთ შრიფტის ზომა, **გავამუქოთ,** *დავხაროთ,* რათა ტექსტის სხვადასხვა ნაწილი გამოვყოთ. პროგრამის ტექსტის, ანუ კოდის შემთხვევაში კი უნდა შემოვიფარგლოთ მხოლოდ ფიქსირებული ზომის შრიფტით. თუმცა, მაინც შეგვიძლია გარკვეული ეფექტების შექმნა კომენტარების წერის დროს. მაგალითად:

```
//////////////////////////////////////////////////////
//              გაფრთხილება: ეს არის გამაფრთხილებელი          //
//                        გზავნილის მაგალითი, რომელიც          //
//                        პროგრამისტის ყურადღებას მიიქცევს          //
//////////////////////////////////////////////////////
```

ან:

```
//>>>>>>>>>>>>>>>>       სექციის სათაური      <<<<<<<<<<<<<<<<<< //
```

ან უბრალოდ

```
int counter;          // მთვლელი
```

ნებისმიერ პროგრამაში, ფაქტიური კოდი (ანუ კოდი კომენტარების გარეშე) შედგება ორი ნაწილისაგან: ცვლადები და შესრულებადი ინსტრუქციები. ცვლადები გამოიყენება იმისთვის, რომ შევინახოთ პროგრამის მიერ გამოყენებული მონაცემები, ხოლო ინსტრუქციები მიუთითებს კომპიუტერს, თუ რა მოქმედებები უნდა ჩაატაროს მონაცემებზე.

პროგრამული კოდის კომენტირება. ცვლადის სახელი როგორც კომენტარის ფორმა

სერიოზულ ამოცანებში, როგორც წესი, პროგრამული კოდის წერას იწყებენ ამოცანის საფუძვლიანი გააზრების შემდეგ, როდესაც უკვე მოფიქრებულია საკმაოდ დეტალური გეგმა. გეგმის პირველ, ყველაზე უხეშ ვარიანტს ჯერ ჩვეულებრივ სალაპარაკო ენაზე წერენ, ხოლო რაც უფრო იხვეწება იგი, მით უფრო ხშირად გამოერევა მასში მათემატიკური აღნიშვნები და პროგრამირების ენისთვის ჩვეული კონსტრუქციები და შეტყობინებები. ასეთ სინთეზს (სალაპარაკო ენისა პროგრამირების ენასთან) ფსევდოკოდს უწოდებენ. როცა ფსევდოკოდი იმდენად დეტალური გახდება, რომ მხოლოდ მისი "გადათარგმნა" არის საჭირო პროგრამირების ენაზე, მაშინ იწყება პროგრამული კოდის წერა. მაგრამ, როგორც ზემოთ აღვნიშნეთ, ადრე თუ გვიან საჭირო ხდება პროგრამის გადაკეთება და გაუმჯობესება, ან კიდევ

სადმე სხვაგან დანერგვა. ისევ საჭირო რომ არ შეიქმნას მთელი სამუშაოს თავიდან გამეორება, ამ მომენტისთვის აუცილებელია იმ გეგმის და ფსევდოკოდის შენარჩუნება, რომლის საფუძველზეც შეიქმნა პროგრამა. როგორც წესი, ფსევდოკოდი ილექება კომენტარებში, კომპილერი მას არ აღიქვამს, მაგრამ კვალიფიციურ პროგრამისტს კომენტარებიდან შეუძლია აღადგინოს ყველა ის სირთულე და პრობლემა, რაც გადაიჭრა პროგრამის წერის პროცესში.

მაქსიმალური სიცხადის მიღწევის მიზნით, ძალიან გავრცელებულია ცვლადების სათაურებში მათი შინაარსის ასახვა - კარგად შერჩეული სახელი გარკვეული აზრით უნდა ითავსებდეს კომენტარის ფუნქციას. განვიხილოთ რამდენიმე მაგალითი.

პროგრამირებაში, **ცვლადი** არის ადგილი კომპიუტერის მეხსიერებაში, გამოყოფილი რაიმე სიდიდის (მნიშვნელობის) შესანახად. ამ ადგილს C++ აიგივებს ცვლადის სახელთან. სახელი შეიძლება იყოს ნებისმიერი სიგრძის და ისე უნდა შეირჩეს, რომ მისი შინაარსი გასაგები იყოს (სინამდვილეში სიგრძის ზღვარი არსებობს, მაგრამ იგი დიდია და რეალურად ამ შეზღუდვას ვერ ვგრძნობთ). საჭიროა ყველა ცვლადი, რომელთაც პროგრამაში ვიყენებთ, წინასწარ ჩამოვთვალოთ ანუ აღვწეროთ. ეს ჩამონათვალი კომპილერისთვის წარმოადგენს განაცხადს მეხსიერების გამოყოფაზე. ცვლადებზე განაცხადის საკითხი დაწვრილებით იქნება განხილული ერთ-ერთ შემდეგ ლექციაში. შემდეგი განაცხადი, რომლის მათემატიკური ჩანაწერია $p,q,r \in Z$, C++-ის აცნობებს, რომ ჩვენ ვაპირებთ სამი `p`, `q` და `r` მთელი (**`int`**) რიცხვის გამოყენებას:

```
int p,q,r;
```

მაგრამ სრულიად გაუგებარია, რისთვის? ეს სამი ცვლადი ნებისმიერ რამეს შეიძლება აღნიშნავდეს. შემოკლებულ ჩანაწერებს თავი უნდა ავარიდოთ. ზედმეტი შემოკლება ძნელად გასაგებს ხდის აღნიშვნებს. ამისგან განსხვავებით, შემდეგი განაცხადი:

```
int account_number;
int balance_owed;
```

ცვლადების სახელების საშუალებით აშკარად მიგვანიშნებს, რომ საქმე გვაქვს საბუღალტრო აღრიცხვის პროგრამასთან. მაგრამ ჩვენ უფრო მეტი ინფორმაციის მიღებაც შეგვიძლია თუ კომენტარებსაც გამოვიყენებთ. მაგალითად:

```
int  account_number;  // საბანკო ანგარიშის ნომერი
int  balance_owed;    // დავალიანება (ლარებში)
```

ვწერთ რა კომენტარებს ყოველი განაცხადის შემდეგ, ჩვენ სინამდვილეში ვქმნით მინი ლექსიკონს, სადაც განვსაზღვრავთ ყოველი ცვლადის სახელის მნიშვნელობას. ძალზე მნიშვნელოვანია პროგრამაში გამოყენებული ერთეულების მითითება (კმ, სთ, $ და ა.შ.), განსაკუთრებით მაშინ, თუ გვიწევს პროგრამის გადაკეთება, ან მისი ფორმატის შეცვლა.

ზოგადად, პროგრამისტმა ის ფაქტორები უნდა გაითვალისწინოს, რაც კარგ პროგრამას ახასიათებს. მაგალითად, იდეალურ შემთხვევაში პროგრამა უნდა იყოს ადვილად აღსაქმელი, კომპაქტური, სწრაფად უნდა სრულდებოდეს, უნდა საჭიროებდეს მინიმალურ მეხსიერებას და ა. შ. სამწუხაროდ, რეალურ სიტუაციებში ყველა მიზანი ერთდროულად ვერ მიიღწევა და უნდა ვიცოდეთ, თუ რა არის ძირითადი. ნებისმიერ შემთხვევაში, პროგრამა უნდა იყოს რაც შეიძლება ადვილად აღსაქმელი, თუნდაც ამისთვის, გონიერების ფარგლებში, სხვა მიზნების მიღწევის გზაზე ნაწილობრივი კომპრომისების გაკეთება გახდეს საჭირო. პროგრამები თავისი ბუნებით ძალიან რთულია. ყველაფერი, რასაც გააკეთებთ ამ სირთულის შესამცირებლად, გაუმჯობესებს თქვენს პროგრამას. უნდა გვახსოვდეს, რომ რთულ პროგრამას შესაძლოა გაუთვალისწინებელი ეფექტები ახლდეს (ზოგიერთ ჩრდილოვან ეფექტებს მოგვიანებით განვიხილავთ).

კომპიუტერისთვის სულერთია, თუ პროგრამის როგორ ვერსიას შექმნის პროგრამისტი: ადვილად გასარჩევს თუ რთულს და ჩახლართულს. კარგი კომპილერი ორივე ვერსიას

შეუსაბამებს მანქანურ კოდს. გასაგებად დაწერილი კოდით მხოლოდ პროგრამისტი ისარგებლებს.

კოდის ფორმატირება წანაცვლების საშუალებით

იმისათვის, რომ პროგრამა ადვილად გასაგები იყოს, პროგრამისტების უმრავლესობა წანაცვლებულად წერს პროგრამის გარკვეულ ნაწილებს. C++ -ის პროგრამების გაფორმების ზოგადი წესია წავანაცვლოთ ერთი დონით ყოველი ახალი შიგა ბლოკი. ერთ დონეზე შეიძლება მოთავსდეს მხოლოდ ერთმანეთისაგან დამოუკიდებელი ბლოკები. ძალიან მოკლედ, ბლოკს ვუწოდებთ პროგრამული კოდის ფრაგმენტს, რომელიც მოთავსებულია ფიგურულ ფრჩხილებში. განვიხილოთ მაგალითი, რომელშიც სამი განსხვავებული დონე შეგვხვდება.

```
//////////////////////////////////////////////////////////
//  ავტორი:
//  პროგრამა: პროგრამა გამოიცნობს  უდრის თუ არა რიცხვი 5-ს
//////////////////////////////////////////////////////////

#include <iostream>
using namespace std;
int main()
{
   int a(5);                //განაცხადი მთელ რიცხვზე და ინიციალიზაცია 5-ით
   if( a == 5 )
   {
      cout<<"Hello,\n";     // ეკრანზე გამოჩნდება გზავნილი Hello,
      cout<<"a = 5\n\n";    // და შემდეგ სტრიქონში გზავნილი a = 5
   }
   else
   {
     cout<<"I'm Program\n";         // ეკრანზე დაიბეჭდება I'm Program და
     cout<<"I know, a isn't 5\n\n";  // შემდეგ სტრიქონში: I know, a isn’t 5
   }
   return 0;
}
```

კომენტარებიდან გასაგებია, თუ რას აკეთებს პროგრამა. ვნახოთ, თუ როგორ აკეთებს ამას.

კომენტარის ბლოკი განმარტავს პროგრამის დანიშნულებას და შეიცავს ცნობას ავტორის შესახებ. შემდეგი სტრიქონები: `#include <iostream>` და `using namespace std;` ნიშნავს მიმართვას სტანდარტული ბიბლიოთეკის შეტანა გამოტანის კლასებზე (ჩვენ პროგრამას სხვა ინფორმაცია არ სჭირდება სტანდარტული ბიბლიოთეკიდან). ბიბლიოთეკების ჩართვა აუცილებელია, რათა გამოვიყენოთ უკვე არსებული პროგრამების ფრაგმენტები. ეს სტრიქონები მოთავსებულია ყველაზე მარცხნივ, პირველ დონეზე, ანუ ყოველგვარი წანაცვლების გარეშე.

მომდევნო სტრიქონში იგივე დონეზე მოთავსებულია მთავარი ფუნქცია `main()`. რადგან იგი ყველა პროგრამაში გვხვდება, ამიტომ მას არ ვუკეთებთ კომენტარს. მის შიგნით მოთავსებული ნაწილი მთლიანად მას ეკუთვნის და ის მეორე დონეა. ანალოგიურად, მეორე დონის ინსტრუქციის **if**-ის შიგნით მოთავსებული ინსტრუქციების შესრულების საკითხი მთლიანად **if** -ის შედეგზეა დამოკიდებული და ამიტომ წანაცვლების მესამე დონეა.

`main()`-ის ტანში პირველი სტრიქონია `a` მთელი რიცხვის აღწერა:

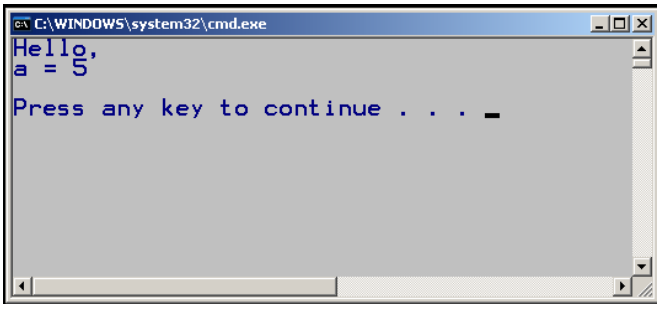
```
    int a(5);
```

რაც აცნობებს კომპილერს, რომ პროგრამაში მონაწილეობს მთელი რიცხვი, სახელით a, რომელიც ეკუთვნის მთელი რიცხვების **int** სიმრავლეს. **int** რეზერვირებული სახელია, ამ ენაში მისი სხვა მიზნით გამოყენება აკრძალულია. a ცვლადზე ფრჩხილებში მიდგმული კონსტრუქცია a(5) მიუთითებს კომპილერს, რომ მან ჩაწეროს მეხსიერებაში მითითებული მნიშვნელობა.

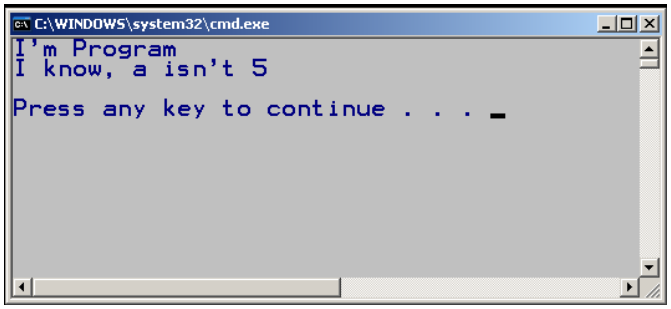
შემდეგ რამდენიმე სტრიქონს იკავებს **განშტოების** შეტყობინება **if**(a == 5) შესაძლო შედეგებითურთ. მისი შესრულების წესი მარტივია: შემოწმდება პირობა a==5 (უუდრის?) და თუ პირობა ჭეშმარიტია (სწორია), შესრულდება პირველ ფიგურულ ფრჩხილებში მოთავსებული ყველა შეტყობინება (ანუ, თუ a ტოლია 5-ის, შესრულდება cout<<"Hello,\n"; და cout<<"a = 5\n\n";), ხოლო თუ a არ უდრის 5-ს (ე. ი. პირობა მცდარია) – მაშინ შესრულდება **else** სიტყვის შემდეგ ფიგურულ ფრჩხილებში მოთავსებული ყველა შეტყობინება.

ყურადღება მივაქციოთ ჩანაწერს a == 5. ეს არის შედარება ტოლობაზე - ტოლია თუ არა? ასეთი შედარება C++-ში ჩაიწერება ზედიზედ ორი ტოლობის ნიშნით. არავითარ შემთხვევაში არ შეიძლება " == " ნიშნის შეცვლა " = "-ით ან " = = "-ით.

პროგრამაში a -ს თავიდან მივანიჭეთ 5, შედარება a == 5 ჭეშმარიტია და პროგრამა გამოიტანს შედეგს:



შევცვალოთ main()-ის მეორე სტრიქონი ასე: int a(9); რაც მიანიჭებს a-ს მნიშვნელობას 9. პირობა a==5 იქნება მცდარი, და პროგრამა გამოიტანს შედეგს



არსებობს წანაცვლების ორი ძირითადი სტილი. პირველი — მოკლე ფორმაა:

```
int main()
{
   int a =9;
   if( a == 5 ) {
      cout<<"Hello,\n";
      cout<<"a = 5\n\n";
   }
   else {
     cout<<"I'm Program\n";
     cout<<"I know, a isn't 5\n\n";
   }
   return 0;
}
```

მეორე სტილი ფიგურულ ფრჩხილებს ცალკე სტრიქონებზე სვამს:

```
int main()
{
   int a =9;
   if( a == 5 )
  {
      cout<<"Hello,\n";
      cout<<"a = 5\n\n";
   }
   else
   {
     cout<<"I'm Program\n";
     cout<<"I know, a isn't 5\n\n";
   }
   return 0;
}
```

ორივე ფორმატი ხშირად გამოიყენება.

წანაცვლებაში, გამოტოვებული სიმბოლოების (space) რაოდენობა პროგრამისტზეა დამოკიდებული. მიღებულია ორი, ოთხი ან რვა space-ით წანაცვლება. გამოკვლევებმა აჩვენეს, რომ ოთხი ინტერვალით წანაცვლების შემთხვევაში პროგრამა უკეთ იკითხება.

ზოგიერთი რედაქტორი, UNIX Emacs editor,Borland C++ და Microsoft Visual C++ internal editors შეიცავს საშუალებებს, რაც ავტომატურად ახდენს თქვენს პროგრამაში კოდის ტექსტის წანაცვლებას.

სიცხადე და სიმარტივე

პროგრამა უნდა იკითხებოდეს, როგორც ტექნიკური დოკუმენტი. ის დაყოფილი უნდა იყოს სექციებად და პარაგრაფებად. პარაგრაფი უნდა დაიწყოთ შესაბამისი კომენტარით და გამოყოთ ეს კომენტარი სხვა პარაგრაფისგან თავისუფალი სტრიქონით. მაგალითად, ვხსნით ამოცანას: სიბრტყეზე მოცემულია ორი წერტილი A და B. A -ს კოორდინატებია x1 და y1, B -სი – x2 და y2. ჩვენ გვინდა A და B წერტილებს მნიშვნელობები გავუცვალოთ.

```
/* არცთუ საუკეთესო პროგრამა */
temp = x1;
x1 = x2;
x2 = temp;
temp = y1;
y1 = y2;
y2 = temp;
```

უკეთესი ვერსია იქნება:

```
/*
* ადგილი გავუცვალოთ (swap) A და B-ს
*/

/* X კოორდინატების გაცვლა */
temp = x1;
x1 = x2;
x2 = temp;

/* Y კოორდინატების გაცვლა */
temp = y1;
y1 = y2;
```

```
y2 = temp;
```

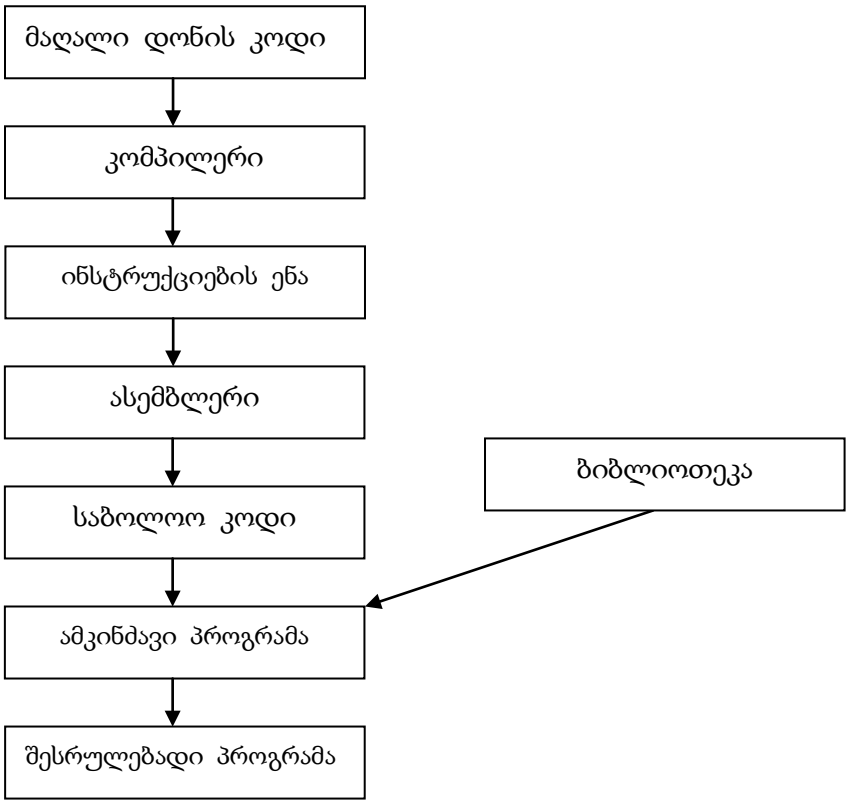
პროგრამა მარტივი უნდა იყოს. ზოგადი წესები ასეთია:

- ეცადეთ, თქვენს პროგრამას არ ჰქონდეს რთული ლოგიკა. დაყავით ცალკეულ პროცედურებად და შეამცირეთ სირთულის ხარისხი.
- გრძელ ინსტრუქციებს თავი აარიდეთ. უმჯობესია გრძელი წინადადება რამდენიმე მოკლეთი შეცვალოთ (ისევე, როგორც სალაპარაკო ენაში). პროგრამა ასე უფრო ადვილი აღსაქმელი იქნება.

და ბოლოს, ყველაზე მნიშვნელოვანი წესი: ეცადეთ თქვენი პროგრამა იყოს რაც შეიძლება მარტივი და ნათელი, თუნდაც თქვენ მოგიხდეთ ზოგიერთი აქ მოყვანილი წესის დარღვევა. მიზანი — ნათელი და გასაგები პროგრამის დაწერაა და ეს წესები იმისთვისაა, რომ დაგეხმაროთ ამ მიზნის მიღწევაში.

პროგრამები, ამოცანის დასმიდან შესრულებამდე

მაღალი დონის პროგრამები C++ ენაზე იწერება კლავიატურაზე მოთავსებული სიმბოლოების გამოყენებით. რადგან კომპიუტერი რეალურად ასრულებს (უშვებს) მხოლოდ დაბალი დონის, მანქანურ კოდებში დაწერილ პროგრამას, ამიტომ, C++ –ის პროგრამები განიცდის მთელ რიგ გარდაქმნებს, ვიდრე მათი შესრულება (გაშვება) მოხდება.



პროგრამაზე მუშაობა იწყება ამოცანის დასმით ჩვეულებრივ სალაპარაკო ენაზე, რაც შეიძლება იყოს ზეპირი ან წერილობითი, შესაძლოა დიაგრამების, ცხრილების, გრაფიკების, ფორმულების ან სხვა რაიმე მოდელის გამოყენებით. მას შემდეგ, რაც პროგრამისტი მოიფიქრებს ამოცანის გადაჭრის გზას, ანუ ალგორითმს, იგი ტექსტური რედაქტორის დახმარებით ქმნის **საწყის ფაილს**, რომელიც შეიცავს **საწყის კოდს** (source code). საწყის კოდს პროგრამა სახელად **კომპილერი** გადაიყვანს საბოლოო ფაილში (object file). შინაარსობრივად ესაა ფაილი, რომლის შექმნაც წარმოადგენდა ჩვენს მიზანს. შემდეგ კიდევ ერთი პროგრამა (linker) ერთად აკინძავს საბოლოო ფაილს და ყველა იმ ფუნქციას, რომლებსაც ის იძახებს სტანდარტული თუ სხვა სახის ბიბლიოთეკებიდან. შედეგად მიიღება შესრულებადი პროგრამა (executable program) მანქანური ენის ინსტრუქციების კრებული. ამჟამად, პროგრამები ამოცანის დასმიდან შესრულებამე ყველა ფაზას გადიან ე.წ. ინტეგრებულ გარემოში (IDE), რომელიც შეიცავს ტექსტურ რედაქტორს, სხვადასხვა მენიუს, მართვის ღილაკებს და ა.შ.. ჩვენ შემთხვევაში ესაა ***Visual Studio***. ასეთ გარემოს ბევრი ღირსება გააჩნია. ერთ-ერთი ისაა, რომ პროგრამისტს არ სჭირდება იზრუნოს საწყისი კოდის გარდაქმნებზე და საჭირო მომენტში საჭირო პროგრამები (კომპილერი და ამკინძავი) თვითონ იწყებენ მუშაობას.

თავი 3: ზოგადი ცნობები ცვლადების შესახებ

- მოკლედ პროგრამის ძირითადი ელემენტების შესახებ
- ცვლადი: მეხსიერება, სახელი, ზომა
- ცვლადის ინიციალიზაციის ორი მარტივი გზა, მინიჭების შეტყობინება და რეზერვირებული სიტყვა **`const`**
- მარტივი გამოსახულებები
- **`bool`** ტიპი, **`if-else`** კონსტრუქცია

მოკლედ პროგრამის ძირითადი ელემენტების შესახებ

პროგრამის ძირითადი ელემენტები არის თავსართი (სათაურების) ფაილები, განაცხადები ცვლადებზე, ფუნქციები და შესრულებადი შეტყობინებები.

თავსართი ფაილების ჩართვა ხდება პრეპროცესორის `#include` დირექტივით. როდესაც ბიბლიოთეკა ჩაშენებულია C++-ში, მაშინ თავსართი ფაილის სახელი, როგორც წესი, თავსდება მეტობის და ნაკლებობის ნიშნებისგან გაკეთებულ ფრჩხილებში. მაგალითად:

```
#include<iostream>
```

როდესაც ისეთ ბიბლიოთეკას ვრთავთ, რომელიც არაა ჩაშენებული C++-ში, მაშინ თავსართი ფაილის სახელი, როგორც წესი, თავსდება ორმაგ ბრჭყალებში. მაგალითად:

```
#include "my_math.h"
```

თავსართი ფაილების სახელები მჭიდრო კავშირშია მათ შინაარსთან. მაგალითად, `iostream` ნიშნავს შეტანა-გამოტანის (`input-output`) ნაკადების (`stream`) თავსართ (`header`) ფაილს.

ცვლადებზე განაცხადი კეთდება მათთვის მეხსიერების გარკვეული მონაკვეთის "დაჯავშვნის" მიზნით, რომელშიც შეინახება ამ ცვლადის მიმდინარე მნიშვნელობები. უნდა გავითვალისწინოთ, რომ თუ განაცხადის გაკეთების მომენტში არ მივანიჭებთ ცვლადს გარკვეულ მნიშვნელობას (ანუ არ მოვახდენთ მის **ინიციალიზაციას),** სისტემა მის მნიშვნელობად აღიქვამს იმას, რაც ამ მომენტისთვის წერია მეხსიერების ამ მონაკვეთში - C++ ენაში მეხსიერების მონაკვეთის გამოყოფა არ ნიშნავს მის დამუშავებას ან გასუფთავებას.

ჩვეულებრივი სალაპარაკო ენის წინადადებებს C++-ში შეესაბამება შეტყობინებები (`statements`). და რატომ აქაც წინადადებები არა? იმიტომ, რომ წინადადებები განკუთვნილია ადამიანებისთვის, ხოლო შეტყობინებები კომპილერისთვის, რომელსაც, თავის მხრივ, შეტყობინებები გადაჰყავს მანქანურ ინსტრუქციებში (იმედი გვაქვს, ასეთ განმარტებას რასიზმის გამოვლინებად არ ჩათვლით). სასვენი ნიშნების საქმე C++-ში შედარებით მარტივადაა, აქ ყოველი შეტყობინება მთავრდება წერტილ-მძიმით, რაც ნიშნავს ამ შეტყობინების დასასრულს, დამთავრებას. საზოგადოდ, შეგვიძლია ერთ სტრიქონში რამდენიმე შეტყობინების ჩაწერა, მაგრამ ეს არაა კარგი სტილი, პროგრამა ადვილად წასაკითხი რომ იყოს, სჯობს ერთ სტრიქონში მხოლოდ ერთი შეტყობინება დავწეროთ.

მეორეს მხრივ, ზოგიერთი სხვა ენისგან განსხვავებით, C++ –ში ხშირად გვხვდება შეტყობინებები, რომლებიც იკავებენ რამდენიმე სტრიქონს.

ცვლადი: მეხსიერება, სახელი, ზომა

განაცხადები კეთდება როგორც ცვლადებზე, ასევე ფუნქციებზე. ერთის მხრივ, განაცხადი წარმოადგენს მოცემულ ამოცანაში გამოყენებული ცვლადების და ფუნქციების აღწერას. მაგალითად, თუ რაიმე გამოყენებითი ამოცანის მათემატიკურ მოდელს ვაპროგრამებთ და პროგრამაში უნდა გამოვიყენოთ მათემატიკურ მოდელში აღწერილი ფუნქცია $f(x): R \to Z$ და ცვლადი $j \in Z$, მაშინ ისინი უნდა აღვწეროთ პროგრამაშიც C++ ენის სინტაქსის დაცვით:

```
int f (float x);
int j;
```

მეორეს მხრივ, რაც ადამიანისთვის წარმოადგენს ფუნქციის და ცვლადის აღწერას, იგივე ჩანაწერი კომპილერისთვის წარმოადგენს განაცხადს მეხსიერების მონაკვეთზე, რომელშიც მოთავსდება მოცემული ფუნქციის და ცვლადის შესახებ ყველა საჭირო ინფორმაცია. მეხსიერების მოცემული მონაკვეთები გაიგივდება შესაბამის ფუნქციასთან და ცვლადთან, ანუ დაიჯავშნება მათთვის. პროგრამის იმ ბლოკის დასრულებამდე, რომელშიც ეს განაცხადებია გაკეთებული, მეხსიერების ეს ნაწილი სისტემის მიერ სხვა მიზნით არ იქნება გამოყენებული.

მოცემულ მარტივ მაგალითში ფუნქციის და ცვლადის სახელად თითო ასო არის გამოყენებული, რაც არაა შესაბამისობაში პროგრამირების კარგ სტილთან (შინაარსს ვერ გამოხატავს). უმეტეს შემთხვევაში სახელები საკმაოდ გრძელია და ხშირად შედგება რამდენიმე ერთმანეთთან შეერთებული სახელისგან. ასეთი სახელებისთვის, პროგრამირების კარგი სტილი იძლევა შემდეგ რეკომენდაციებს:

- სახელები შევაერთოთ ზედა რეგისტრის ტირეთი (ან ტირეებით, თუ რამდენიმეა), მაგალითად `balance_owed`. ამ სტილს ძირითადად C -პროგრამისტები იყენებენ;
- ყოველი ახალი სახელი დავიწყოთ მთავრული ასოთი, მაგალითად `balanceOwed, bankBill.` ზოგჯერ პირველი ასოც მთავრულია. ეს საკმაოდ მოხერხებულია და ამიტომ ბევრი პროგრამისტი იყენებს ასეთ სტილს, რომელსაც გასაგები მიზეზების გამო პროგრამისტების სლენგზე ეწოდება “camel” (აქლემი).

რა ტიპისაც არ უნდა იყოს ცვლადი, მისი სახელის შედგენისას უნდა გავითვალისწინოთ შემდეგი შეზღუდვები:

ცვლადების სახელებში გამოიყენება ასოები, ციფრები და ზედა რეგისტრის ტირეს სიმბოლო (”_”). სახელის ციფრით დაწყება არ შეიძლება. საზოგადოდ, ზედა რეგისტრის ტირეთი იწყება სისტემური სახელები. სტანდარტული ბიბლიოთეკების ფუნქციებში ხშირად გამოიყენება ამ სიმბოლოთი დაწყებული სახელები, ამიტომ არ არის რეკომენდებული ცვლადის სახელის ზედა რეგისტრის ტირეთი დაწყება. რადგან C++ განასხვავებს ზედა და ქვედა რეგისტრის სიმბოლოებს, ამიტომ კომპილერისთვის `Gia, gia, GIA` სხვადასხვა სახელებია.

C++ ენის სტანდარტული კონსტრუქციები ეფუძნება რამდენიმე ხშირად გამოყენებად სახელს, მაგალითად, **`int, while, for, float`** და სხვა, რომელთაც C++ ენის რეზერვირებული (მომსახურე) სიტყვები ეწოდება. მათი გამოყენება არ შეიძლება ცვლადების სახელებად.

მაგალითად, ცვლადის სახელი **შეიძლება იყოს:**

```
sashualo              /* ყველა გაზომვის საშუალო მნიშვნელობა */
pi                    /* პი რიცხვის მნიშვნელობა მეათასედის სიზუსტით */
studentebis_raoden    /* ჯგუფში სტუდენტების რაოდენობა */
```

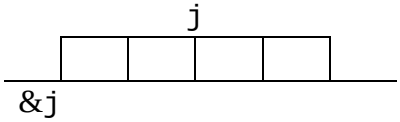
ცვლადის სახელი **არ შეიძლება** იყოს შემდეგი:

```
3_gazomva             // იწყება ციფრით
fasi$                 // შეიცავს სიმბოლო $-ს
studentebis raoden    // შეიცავს ჰარს
int                   // რეზერვირებლი სიტყვაა
```

თავი უნდა ავარიდოთ ერთ პროგრამაში ცვლადებისთვის ერთმანეთის მსგავსი სახელების გამოყენებას, მაგალითად:

```
total                 // ერთ ჯერზე შეტანილი ცვლადების ჯამი
totals                // ყველა ცვლადის ჯამი
```

სხვადასხვა ტიპის ცვლადებს არა მარტო სახელების შედგენის წესები აქვთ საერთო. ნებისმიერი ტიპის ცვლადს აქვს მისამართი და სიგრძე. ზოგადად, განაცხადის გაკეთების მომენტში ცვლადის სახელი გაიგივდება მისთვის გამოყოფილ მეხსიერების მონაკვეთთან. თვითონ მეხსიერება შეიძლება გავაიგივოთ სხივთან, რომელიც დაყოფილია ერთბაიტიან მონაკვეთებად, და ეს მონაკვეთები დანომრილია ერთმანეთის მომდევნო არაუარყოფითი რიცხვებით. მაგალითად, თუ `j` არის მთელი ტიპის ცვლადის სახელი, მაშინ ამ ცვლადის მისამართს, ანუ `j`-სთან გაიგივებული მონაკვეთის დასაწყისს გავიგებთ C++ -ის ოპერატორ ”&”-ის გამოყენებით, - `&j` წარმოადგენს `j` ცვლადის მისამართს.



ნახაზზე `j`-ის მონაკვეთი გაყოფილია ოთხ ნაწილად იმის მისათითებლად, რომ მთელი რიცხვების წარმოდგენისთვის გამოყენებულია 32-ბიტი. თუმცა ეს დამოკიდებულია სისტემაზე. ზოგადად, ცვლადისთვის საჭირო მეხსიერების სიდიდეს გავიგებთ C++ -ის ოპერატორ **`sizeof`** -ის საშუალებით. მაგალითად, თუ დავბეჭდავთ **`sizeof`**(`j`) სიდიდეს, დაიბეჭდება სისტემის მიერ `j` ცვლადისთვის გამოყოფილი ბაიტების რაოდენობა. იგივე ოპერატორი შეგვიძლია გამოვიყენოთ უშუალოდ ტიპებისთვის. მაგალითად, დავბეჭდოთ და ვნახოთ რისი ტოლია **`sizeof`**(**`int`**), **`sizeof`**(**`double`**) და ა.შ.. თუ ვიცით რამდენი ბაიტია გამოყოფილი ცვლადისთვის, შეგვიძლია გამოვთვალოთ რა დიაპაზონის და რა სიზუსტის მიღწევა შეიძლება ამ ტიპის საშუალებით. მაგალითად, თუ **`int`** ტიპის ცვლადისთვის ოთხი ბაიტია გამოყოფილი, მაშინ ყველაზე დიდი მთელი (**`int`**) რიცხვი არის $2^{31}-1$ (გავიხსენოთ რიცხვების წარმოდგენის `two's complement` სისტემა). მაგრამ ამის ანგარიშის ან დამახსოვრების აუცილებლობა არ არის, რადგან

```
#include <cmath>
```

დირექტივა საშუალებას გვაძლევს გამოვიყენოთ ამ ბიბლიოთეკაში აღწერილი მუდმივი სიდიდეები. მაგალითად, თუ ჩართულია ეს ბიბლიოთეკა, კომპილერისთვის `INT_MAX` მუდმივი არის ზუსტად უდიდესი **`int`** მთელი რიცხვი, რომლის წარმოდგენაც შესაძლებელია **`sizeof`**(**`int`**) რაოდენობა ბაიტებში. ანგარიში ჩვენ არ გვჭირდება. თუ რამდენიმე წლის შემდეგ **`sizeof`**(**`int`**) გაიზრდება, შესაბამისად გაიზრდება `INT_MAX`-იც.

ყურადღება უნდა მივაქციოთ განსხვავებას ”&” და ”&&” ოპერატორებს შორის. პირველი მათგანი არის მისამართის აღების ოპერატორი, მეორე - ლოგიკური ”და” ოპერატორი.

ცვლადის ინიციალიზაციის ორი მარტივი გზა, მინიჭების შეტყობინება და რეზერვირებული სიტყვა `const`

C++ ენა საშუალებას გვაძლევს, რომ განაცხადის გაკეთების მომენტში ყოველ ცვლადს (ტიპის მიუხედავად) მივცეთ საწყისი მნიშვნელობა ან გულისხმობის პრინციპით (იგივე უპარამეტრო კონსტრუქტორი), ან მინიჭების შეტყობინებით (ასლის გაკეთების კონსტრუქტორი). ვნახოთ რამდენიმე მარტივი მაგალითი:

```
int answer(55);
int position(( 1 + 2 ) * 4);
double x(12.563);
double height(x);    //როდესაც x ცვლადი უკვე არის ინიციალიზებული
```

და ანალოგიურად სხვა ტიპის ცვლადებისთვის. თუ ცვლადი მუდმივი არაა, მისთვის შესაძლებელია მნიშვნელობის შეცვლა პროგრამის განხორციელების პროცესში, მინიჭების შეტყობინებით. მაგალითად, განვიხილოთ პროგრამის ფრაგმენტი:

```
int size(3*5);          // size მონაწილეობს მომდევნო ორ გამოსახულებაში
int size_2;             // ცვლადი გაორმაგებული size-ისთვის
int size_3;             // ცვლადი გასამმაგებული size-ისთვის
  ...
size _2 = 2 * size;
size _3 = 3 * size;
  ...
```

ცხადია, შეგვეძლო ცვლადებისთვის დაგვერქმია სხვა სახელები და გაგვეკეთებინა იგივე.

ერთი ზოგადი გზა იმისათვის რომ C++ -ში ჩაშენებული რომელიმე ტიპის ცვლადის ინიციალიზება მოვახდინოთ განაცხადის გაკეთების შემდეგ, ან უბრალოდ შევცვალოთ მისი მიმდინარე მნიშვნელობა სხვა მნიშვნელობით, არის კლავიატურიდან (ან ფაილიდან) შემოტანილი ახალი მნიშვნელობის მინიჭება მისთვის..

მონაცემების შეტანა კლავიატურიდან C++ -ში სრულდება `cin` -ის საშუალებით (`cin` – `console input`). მაგალითად,

```
int number;

cin >> number;
```

ფრაგმენტის პირველი სტრიქონი წარმოადგენს მთელ number ცვლადზე განაცხადს, ხოლო მეორე სტრიქონი არის შეტანის შეტყობინება. მასში გამოიყენება cin და შეტანის ოპერატორი >>. შეიძლება ითქვას, რომ ჩვენ შემთხვევაში კლავიატურის ნაკადიდან ამოიღება მთელი რიცხვი და მიენიჭება >> ოპერატორის მარჯვნივ მდგომ number -ს.

```
string name;

cin >> name;
```

ფრაგმენტის შესრულების დროს კი კლავიატურაზე უნდა ავკრიფოთ სახელი (მაგალითად, Giorgi) და დავაჭიროთ Enter –ს. ამის შემდეგ სტრიქონის ტიპის name ცვლადს მიენიჭება მნიშვნელობა Giorgi (name ცვლადის შესაბამის მეხსიერებაში ჩაიწერება სტრიქონი Giorgi).

ზოგჯერ საჭირო ხდება ისეთი ცვლადების გამოყენება, რომლის მნიშვნელობები არ უნდა შეიცვალოს პროგრამის განხორციელების პროცესში. იმისათვის რომ თავი დავიზღვიოთ უნებლიე შეცდომისგან, ვიქცევით შემდეგნაირად. ვთქვათ, ხშირად გვჭირდება პროგრამაში π მუდმივის გამოყენება, რომლის მნიშვნელობაც არის დაახლოებით 3.14. შემოვიტანოთ ცვლადი და გავაკეთოთ განაცხადი და მოვახდინოთ ინიციალიზაცია:

```
const double PI(3.14);
```

ამის შემდეგ `PI` ცვლადს ჩვეულებრივ ვიყენებთ პროგრამაში და მისი მნიშვნელობა ყოველთვის არის 3.14. თუ ჩვენ შეგვეშალა და სადმე დავწერეთ

```
PI = 55.17;
```

კომპილერი მიგვითითებს შეცდომაზე. ამგვარად, რეზერვირებული სიტყვა **`const`** არის თავის დაზღვევის საშუალება როდესაც ვმუშაობთ მუდმივ სიდიდეებთან. მისი გამოყენება აგრეთვე შეიძლება სხვა საბაზო ტიპებისთვის, რომლებსაც ენა გვთავაზობს გამზადებული სახით.

თუ მივაქციეთ ყურადღება, მუდმივი (კონსტანტური) ცვლადის სახელი ჩვენ მთავრული (დიდი) ასოებით ავკრიფეთ. ესაც პროგრამირების კარგი სტილის რეკომენდაციაა: პროგრამისტი, რომელიც კითხულობს სხვის დაწერილ პროგრამას მისი გადაკეთების მიზნით, მთავრული ასოებით აკრეფილი ცვლადის სახელის დანახვისას ხვდება, რომ ეს მუდმივი ცვლადია და მისი მნიშვნელობა ერთხელ და სამუდამოდ არის განსაზღრული. თუ მისი

მნიშვნელობის შეცვლაა საჭირო, უნდა მოიძებნოს ინიციალიზაციის სტრიქონი და იქ გაკეთდეს ცვლილება.

მინიჭების შეტყობინების ზოგადი სახეა:

ცვლადი = *გამოსახულება;*

ოპერატორი " = " გამოიყენება მინიჭებისთვის. ეს შეტყობინება ამბობს: გამოთვალე გამოსახულება და მიანიჭე ცვლადს ამ გამოსახულების მნიშვნელობა.

ვიდრე ვისაუბრებთ გამოსახულებების შესახებ, რაც აუცილებელია მინიჭების შეტყობინების ზოგადი სახის გამო, კიდევ ერთხელ გავამახვილოთ ყურადღება " = " ოპერატორის ერთ სახასიათო თავისებურებაზე. შემდეგ ფრაგმენტში

```
int counter;        //  მთვლელი
int value = 44;     //  სიდიდე მთვლელის საწყისი მნიშვნელობისთვის
  ...
counter = value;
  ...
```

გვაქვს ორი მინიჭება. პირველი კეთდება განაცხადის გაკეთების მომენტში, ანუ სხვა სიტყვებით ცვლადი `value` ინიციალიზდება მნიშვნელობით 44. მეორე მინიჭება ცვლადს `counter` ანიჭებს იგივე მნიშვნელობას, რაც ჰქონდა ცვლადს `value`, თანაც ამ უკანასკნელის მნიშვნელობა არ იცვლება. სხვა სიტყვებით, ამ შემთხვევაში " = " ოპერატორი აკეთებს მარჯვენა მხარეში მდგომი ცვლადის ასლს მარცხენა მხარეში მდგომ ცვლადში. ეს ანალოგია ასლის გადაღებასთან დაგვეხმარება რომ გავიგოთ ე.წ. მინიჭების კონსტრუქტორის შინაარსი ისეთი ტიპის ცვლადებისთვის, რომლებსაც არა აქვთ რიცხვითი მნიშვნელობები (მაგალითად, სხვადასხვა კლასის ობიექტები).

ყურადღება უნდა მივაქციოთ განსხვავებას "=" და "==" ოპერატორებს შორის. პირველი მათგანი არის მინიჭების ოპერატორი, მეორე - შედარების ოპერატორი.

მარტივი გამოსახულებები

გამოსახულება ჩვენს კურსში იგივეს ნიშნავს რაც მათემატიკის სასკოლო კურსში: გამოსახულება შედგება (სხვადასხვა ტიპის) რიცხვების, ცვლადების, ფრჩხილების და არითმეტიკული მოქმედებისგან. შემდეგ ცხრილში ჩამოთვლილია C++ ენაში გამოყენებული 5 მარტივი ოპერატორი:

მარტივი ოპერატორების ცხრილი:

ოპერატორი	მოქმედება
*	გამრავლება
/	გაყოფა
+	შეკრება
-	გამოკლება
%	ნაშთი (მთელი გაყოფისას მიღებული)

შესრულების თვალსაზრისით, *, / და % ოპერატორებს აქვს მეტი პრიორიტეტი (უფრო მაღალი რიგი და სრულდება უფრო ადრე) + და — ოპერატორებთან შედარებით. მრგვალი ფრჩხილები გამოიყენება როგორც წევრების დასაჯგუფებლად, ასევე შესრულების რიგის შესაცვლელად. მაგალითად: (1+2)*3 = 9, მაშინ, როცა 1+2*3 = 7, ხოლო (4+5)%2 = 1.

შევადგინოთ პროგრამა, რომელიც გამოთვლის (1+2)*3 გამოსახულების მნიშვნელობას.

```
int main()
{
    ( 1 + 2 ) * 3;
    return (0);
}
```

ამ პროგრამის მიხედვით, კომპიუტერი გამოთვლის გამოსახულების მნიშვნელობას, მაგრამ ჩვენ ვერ ვნახავთ შედეგს ეკრანზე. ესაა ე.წ. "ნულოვანი ეფექტის" მაგალითი, როდესაც გვაქვს კორექტული პროგრამა, რომელიც სრულიად უსარგებლოა რადგან ვერ ვხედავთ მის შედეგს.

bool ტიპი

ზოგადად, ტიპი ნიშნავს გარკვეულ სიმრავლეს, ამ სიმრავლეზე განსაზღვრულ ოპერაციებთან ერთად. როდესაც ცვლადზე ვაკეთებთ განაცხადს, აუცილებლად მივუთითებთ ცვლადის ტიპს. ეს საკმარისია, რომ კომპილერმა გამოყოს ადგილი ცვლადისთვის და მერე, რა ვითარებაშიც არ უნდა შეხვდეს ეს ცვლადი, კომპილერმა იცის რომელი ოპერაცია როგორ უნდა შეასრულოს, რომ შედეგი კორექტულად და სწრაფად მიიღოს.

ბულის ტიპის ცვლადმა ან მუდმივმა შესაძლოა მიიღოს მხოლოდ ორი მნიშვნელობა: 0 (იგივე **false**) და 1 (იგივე **true**). ამ რიცხვებისთვის გადატვირთულია არითმეტიკული ოპერატორები ისე, რომ შედეგი არ გავიდეს ბულის სიმრავლიდან. ბულის რიცხვების შეკრება სრულდება ისევე როგორც ჩვეულებრივი რიცხვებისთვის,

გამოსახულება	შედეგი
1 + 1	0
1 + 0	1
0 + 1	1
0 + 0	0

ხოლო სხვა ფუნქციები, რაც ვახსენეთ ზემოთ (**sizeof**(), &) აგრეთვე გადატვირთულია ამ ტიპზე.

თითოეული ტიპი უნიკალურია, ამიტომ თითოეულ ტიპთან დაკავშირებულია სპეციფიკური ფუნქციები (ოპერატორები). განვიხილოთ ბულის ტიპის სპეციფიკა.

C++ ენაში ნებისმიერი გამოსახულება (შედარების, ლოგიკური, არითმეტიკული) ინტერპრეტირდება როგორც **bool** ტიპის მნიშვნელობა. ნებისმიერი ჭეშმარიტი შედარების გამოსახულება (მაგალითად 8 <= 124) C++ –ში გაიგივებულია 1-იანთან, ხოლო მცდარი (მაგალითად 453 < -3) გაიგივებულია ნულთან. იგივე ეხება ლოგიკურ გამოსახულებასაც. მაგალითად, 5 > 3 && 123 <= 3736 გამოსახულების მნიშვნელობაა 1 (**true**), ხოლო 5 > 3 && 123 > 3736 გამოსახულების შედეგია 0 (**false**). არითმეტიკული გამოსახულების გამოთვლის არანულოვანი შედეგი (–100, 2.456, 9) გაიგივდება **bool** ტიპის მნიშვნელობასთან 1 (**true**), ხოლო ნულოვანი შედეგი – მნიშვნელობასთან 0 (**false**).

ეს ძალიან მნიშვნელოვანია პროგრამის შემადგენელი ინსტრუქციების მიმდევრობის მართვისთვის. ისეთი ფუნდამენტური კონსტრუქციები, როგორიცაა **if-else** და განმეორების შეტყობინებები (ციკლები), არგუმენტებად ღებულობენ ბულის ტიპის გამოსახულებებს. ამჟამად განვიხილოთ პირველი მათგანი, რომლისთვისაც ბულის ტიპის ცვლადების ცოდნა საკმარისია.

if შეტყობინების ზოგადი ფორმა ასეთია:

if (*პირობა*)

შეტყობინება;

თუ პირობა ჭეშმარიტია (ნულისაგან განსხვავებულია), მაშინ *შეტყობინება* სრულდება, ხოლო თუ მცდარია (0-ის ტოლია) - არ სრულდება. მაგალითად

№	პროგრამის ფრაგმენტი	შედეგი

1	`cout << "AAA\n";`
	`if(5 > 3)  cout << "BBB\n";`
	`cout << "EEE\n";`
		AAA
		BBB
		EEE

2	`cout << "AAA\n";`
	`if(15 <= 3)  cout << "BBB\n";`
	`cout << "EEE\n";`
		AAA
		EEE

აქ ">" და "<=" წარმოადგენენ შედარების ოპერატორებს. შედარების ოპერატორების სრული სია მოყვანილია შემდეგ ცხრილში.

ოპერატორი	მნიშვნელობა
<=	ნაკლებია ან ტოლი
<	ნაკლებია
>	მეტია
>=	მეტია ან ტოლი
==	უდრის
! =	არ უდრის

ყურადღება უნდა მივაქციოთ, რომ ოპერატორი == შედგება ორი = ნიშნისგან და არ უნდა ავურიოთ მინიჭების შეტყობინებაში.

რამდენიმე შეტყობინების დაჯგუფება შეიძლება მათი ფიგურულ ფრჩხილებში ჩასმით. მაგალითად,

```
cout << "AAA\n";
if(5 > 3)
{
    cout << "BBB\n";
    cout << "DDD\n";
}
cout << "EEE\n";
```

ფრაგმენტის შესრულების შემდეგ დაიბეჭდება

```
AAA
BBB
DDD
EEE
```

ყველა ის შეტყობინება, რომლის შესრულება დამოკიდებულია **if**-ის პირობის შესრულებაზე, წარმოადგენს ერთ ბლოკს და შეწეულია ერთნაირად.

if შეტყობინების სრული ფორმა ასეთია:

if (*პირობა*)
    შეტყობინება1;
else
    შეტყობინება2;

თუ პირობა ჭეშმარიტია (ნულისაგან განსხვავებულია), მაშინ სრულდება *შეტყობინება1*, ხოლო თუ მცდარია (0-ის ტოლია), მაშინ- *შეტყობინება2*. სხვა სიტყვებით, *შეტყობინება1* და *შეტყობინება2* ერთმანეთის ალტერნატივებია და მხოლოდ ერთი მათგანი სრულდება. მაგალითად:

№	პროგრამის ფრაგმენტი	შედეგი
1	`cout << "1111\n";` `if(15 <= 3)  cout << "2222\n";`	1111 3333

	else cout << "3333\n"; cout << "4444\n";	4444
2	**int** a =11; cout << "1111\n"; **if**(a >= 5) cout << "2222\n"; **else** cout << "3333\n"; cout << "4444\n";	1111 2222 4444
3	**if**(1) cout << "yes"; **else** cout << "no";	yes
4	**int** k =7; **if**(k > 0) cout << k << " > 0\n"; **else** cout << k <<" <= 0\n";	7 > 0
5	**int** x =8; **if**(x % 2 == 0) cout << x <<" is even\n"; **else** cout << x << " is odd\n";	8 is even

პირობა შესაძლოა შედარებით რთული სახისაც იყოს. მაგალითად:

№	პროგრამის ფრაგმენტი	შედეგი
1	**if**(5 > 3 && 123 <= 3736) cout << "1111\n"; **else** cout << "2222\n";	1111
2	**int** x =8; **if**(x <= 3736 || 5 > 333) cout << "1111\n"; **else** cout << "2222\n";	1111
3	**if**(5 > 333 || 123 > 3736) cout << "1111\n"; **else** cout << "2222\n";	2222
4	**if**(14) cout << "1111\n"; **else** cout << "2222\n";	1111
5	**if**(2*5 - 1.2) cout << "1111\n"; **else** cout << "2222\n";	1111
6	**if**(2*5 - 10) cout << "1111\n"; **else** cout << "2222\n";	2222

საინტერესოა თუ როგორ ხდება კომპილერის მიერ && და || ოპერატორების გამოყენებით შედგენილი პირობის შემოწმება. მოკლედ რომ ვთქვათ, თუ აუცილებელი არაა, C++ არ განიხილავს ამ ოპერატორების ორივე ოპერანდს. მაგალითად, ბოლო ცხრილის მეორე მაგალითში ||-ის მარჯვნივ მდგარი პირობა აღარ შემოწმდება, რადგან მის მარცხნივ მდგარი პირობა 1-ის ტოლია და ამიტომ ლოგიკური შეკრებაც ერთის ტოლია. იგივე ცხრილის მესამე მაგალითში ორივე მხარის პირობების შემოწმება გახდა აუცილებელი რადგან პირველი პირობა არ შესრულდა (თუმცა ლოგიკური შეკრების შედეგი მაინც ნულის ტოლი დარჩა). აქვე, პირველ მაგალითში, აგრეთვე ორივე მხარის ჭეშმარიტობა შესრულდა. პირველი პირობა რომ ყოფილიყო 3>5 სახის, მაშინ მარჯვნივ აღარ გადავიდოდით, რადგან 0 && x არის 0 ნებისმიერი x -ისთვის.

საკმაოდ ხშირად გვხვდება ჩალაგებული **if**-**else**-ები. ამ დროს მთავარია გვახსოვდეს, რომ **else** ეკუთვნის მის წინ მდგომ პირველივე **if**-ს. მაგალითად, ვთქვათ პროგრამაში უკვე შეყვანილია ორი არანულოვანი ნამდვილი რიცხვი (x და y) და ჩვენს ამოცანას წარმოადგენს გავარკვიოთ და დავბეჭდოთ თუ სიბრტყის რომელ საკოორდინატო მეოთხედში არის მოთავსებული (x,y) წერტილი. ამ მიზნით შეგვიძლია გამოვიყენოთ შემდეგი კოდი:

```
if (x > 0)
{
    if ( y > 0)
       cout << "პირველი მეოთხედი\n";
```

```
        else
           cout << "მეოთხე მეოთხედი\n";
    }
    else
    {
         if ( y > 0)
             cout << "მეორე მეოთხედი\n";
        else
            cout << "მესამე მეოთხედი\n";
    }
```

თუმცა, რადგან **else** ეკუთვნის მის წინ მდგომ პირველივე **if**-ს, ამიტომ იგივე იქნებოდა თუ დავწერდით:

```
    if (x > 0)
         if ( y > 0)
            cout << "პირველი მეოთხედი\n";
        else
           cout << "მეოთხე მეოთხედი\n";
    else
         if ( y > 0)
             cout << "მეორე მეოთხედი\n";
        else
            cout << "მესამე მეოთხედი\n";
```

ზოგადად, ფრჩხილების გამოყენება ყოველთვის უფრო უსაფრთხოა.

თავი 4. მთელი რიცხვები

- ტიპი **char**
- მთელი რიცხვების ტიპები
- **switch** შეტყობინება

ტიპი char

ტიპი **char** აგრეთვე განეკუთვნება მთელ ტიპებს. ამ ტიპის რიცხვი იკავებს მეხსიერებაში ერთ ბაიტს და მოთავსებულია დიაპაზონში (-128)-იდან 127-მდე, ბოლოების ჩათვლით. გულისხმობის პრინციპით **char** ნიშნიანი (**signed**) ტიპია. **unsigned char** ფარავს დიაპაზონს 0–დან 255–მდე.

char ხშირად მოიხსენიება, როგორც სიმბოლური ტიპი. აქ წინააღმდეგობა არ არის, რადგან ყოველ სიმბოლოს კომპიუტერის მეხსიერებაში შეესაბამება მოკლე მთელი რიცხვი – მისი კოდი. სიმბოლური წარმოდგენა გვჭირდება, როდესაც სიმბოლური ინფორმაცია შეგვაქვს კლავიატურიდან ან გამოგვაქვს ეკრანზე (ან პრინტერზე), ხოლო კომპიუტერში სიმბოლოები შენახულია მათი კოდების საშუალებით. სიმბოლოს გარდაქმნა კოდში და პირიქით ხდება ავტომატურად. თითოეული სიმბოლოს კოდი მოყვანილია ASCII (American Standard Code for Information Interchange) ცხრილში, რომელიც კოდების სტანდარტული ცხრილია და მოქმედებს მთელს მსოფლიოში. მაგალითად, სიმბოლო-ციფრი '0' წარმოდგენილია კომპიუტერში ორობითი რიცხვით 00110000, რაც შეესაბამება ათობით მნიშვნელობას 48, ე.ი. სიმბოლო '0'-ის კოდია 48 (ASCII ცხრილის მიხედვით).

ამრიგად, **char** ტიპი პროგრამაში გამოიყენება ორ შემთხვევაში: თუ ვმუშაობთ ძალიან მოკლე მთელ რიცხვებთან ან თუ ვმუშაობთ სიმბოლოებთან. ჩვენ შეგვიძლია დავბეჭდოთ ამ ტიპის ცვლადი ორივენაირად. მაგალითად, შემდეგი ფრაგმენტი

```
char ch('F');
    cout << int(ch) << endl;
    cout << ch << endl;
```

დაბეჭდავს ჯერ კოდს (70), ხოლო შემდეგ თვითონ F სიმბოლოს. იგივე შედეგს მივიღებთ, თუ განაცხადს გავაკეთებთ

```
    char ch(70);
```

სახით.

C++-ს სიმბოლოებისთვის გააჩნია შეტანა-გამოტანის სპეციალური სტანდარტული ფუნქციებიც: getchar და putchar, ორივე ფუნქცია მოითხოვს #include <cstdio> დირექტივას.
getchar –ს აქვს სახე:

```
    getchar()
```

ფუნქცია "კითხულობს" თითო სიმბოლოს კლავიატურიდან და აბრუნებს მის კოდს. მაგალითად,

```
    char ch;
    ch = getchar();
```

ფრაგმენტის შესრულების შემდეგ ch ცვლადის მნიშვნელობა იქნება getchar-ით კლავიატურიდან წაკითხული სიმბოლო (მისი კოდი).

putchar ფუნქცია გეჭდავს სიმბოლოს ეკრანზე. მისი სახეა:

```
    putchar(სიმბოლო)
```

მაგალითად,

```
    char ch ='Z';
    putchar(ch);
```

ფრაგმენტი დაბეჭდავს ეკრანზე Z სიმბოლოს.

getchar ფუნქციის გამოყენებისას გვმართებს სიფრთხილე, რადგან ფუნქცია არ ახდენს ჰარების იგნორირებას. საკმარისია 'A'–ს ნაცვლად ავკრიბოთ ჰარი და 'A', რომ მივიღოთ შემდეგი შედეგი: სიმბოლოდ ჩაითვლება ჰარი , ხოლო ასო 'A' დარჩება ბუფერში.

ანალოგიური პრობლემები ჩნდება, როდესაც getchar –ით შესაყვანი გვაქვს სიმბოლო, ხოლო პროგრამამ ამ მომენტისთვის უკვე მოახდინა რაიმე სხვა მონაცემის შეყვანა. მაშინ, ბოლო შეყვანის მანიშნებელი '\n' სიმბოლო, რომელიც ბუფერში არის დარჩენილი, ავტომატურად წავა შესატანი სიმბოლოს ნაცვლად.

C++ –ში საერთოდ ძნელია შეცდომების აღმოჩენა და გასწორება, მათ შორის ამ ტიპის შეცდომებისაც. ამ შემთხვევაში, გამოსავალს წარმოადგენს სიმბოლოს შემტანი getchar –ის წინ კიდევ ერთი getchar ფუნქციის გამოყენება, რომელიც "მოაშორებს" ზედმეტ სიმბოლოს. ამ ტიპის მაგალითები განხილულია ჩვენი კურსის პრაქტიკულ და ლაბორატორიულ სავარჯიშოებში.

getchar და putchar ფუნქციები ძირითადად უნდა გამოვიყენოთ მაშინ, როდესაც პროგრამა ამუშავებს მხოლოდ სიმბოლურ ინფორმაციას.

getchar ფუნქციის მსგავსია getch. მისი პროტოტიპი მოცემულია conio.h ფაილში. მას, განსხვავებით getchar-ისგან, კლავიატურის ღილაკზე თითის დაჭერისთანავე შეჰყავს სიმბოლო. რომელიმე ღილაკზე ხელის დაჭერამდე getch -ის მოქმედება ანალოგიურია system("PAUSE") ფუნქციის მოქმედებისა, ამიტომ ხშირად იგი ამ მიზნით გამოიყენება. ეს ფუნქციები საერთოა C და C++ –ისთვის. C –ში ჩნდება ეკრანის გაჩერების პრობლემა, როდესაც მონაცემები შეგვაქვს ფაილიდან. ასეთ შემთხვევებში system("PAUSE") –ის ნაცვლად ვიყენებთ getch ფუნქციას, როგორც ნაჩვენებია ქვემოთ.

```
#include <iostream>
#include <cstdio>
#include <conio.h>
using namespace std;
int main()
{
    char p;
    cout<<"Enter a symbol : ";
    p = getchar();                    // სიმბოლოს წაკითხვა
    cout<<"The symbol is : ";
    putchar(p);                       // სიმბოლოს ბეჭდვა
    getch();                          // ეკრანის გაჩერება
    cout<<endl;
    return 0;
}
```

შესრულების შედეგი შეიძლება იყოს:

```
Enter a symbol : B
The symbol is : B
Press any key to continue . .
```

მთელი რიცხვების ტიპები

C++ ფართო არჩევანს სთავაზობს პროგრამისტს მთელი რიცხვების გამოყენებისათვის. ყველაზე ხშირად გამოიყენება **int** ტიპი. მისი ზომის გარკვევა შეიძლება **sizeof** ფუნქციის საშუალებით. შეგვიძლია გამოვიყენოთ აგრეთვე **long int** და **short int** ტიპები, მაგრამ ამის აუცილებლობა ნაკლებად იგრძნობა ხოლმე. ერთერთი ფაქტი, რაც უნდა გვახსოვდეს, ისაა რომ შეგვიძლია **long int**–ის ნაცვლად ვწეროთ **long**, ხოლო **short int**–ის ნაცვლად ვწეროთ **short.** ნებისმიერ შემთხვევაში:

თუ რაიმე ამოცანაში გვჭორდება მხოლოდ არაუარყოფითი რიცხვები, მაშინ შეგვიძლია განაცხადის გაკეთების მომენტში გამოვიყენოთ რეზერვირებული სიტყვა **unsigned**, რაც საშუალებას გვაძლევს გავაორმაგოთ დადებითი მთელი რიცხვების დიაპაზონი უარყოფითი რიცხვების გამორიცხვის ხარჯზე.

შემდეგი საილუსტრაციო მაგალითი გვიჩვენებს რამდენ ბაიტს იკავებს და რა უდიდეს და უმცირეს მნიშვნელობებს ღებულობს ზოგიერთი მთელი ტიპი:

```
#include<iostream>
using namespace std;

int main()
{
   int x = INT_MIN, y = INT_MAX;
   unsigned int ux = UINT_MAX;
   long int lx = LONG_MIN, ly = LONG_MAX;
   unsigned long int lu = ULONG_MAX;
   short int sx = SHRT_MIN, sy = SHRT_MAX;
   unsigned short int su = USHRT_MAX;
   cout<<"int ikavebs "<<sizeof(int)<<" baits"<<endl;
   cout<<"misi diapazonia  " << x <<"\t:  " << y <<endl;
   cout<<"unsigned int-is diapazonia   0 : "<< ux << endl;
   cout<<"long int ikavebs " <<sizeof(long) <<" baits"<< endl;
   cout<<"misi diapazonia  "<< lx <<"\t:  " << ly <<endl;
   cout<<"unsigned long-is diapazonia   0 : " << lu << endl;
   cout<<"short int ikavebs " <<sizeof(short)<<" baits"<< endl;
   cout<<"misi diapazonia:  " << sx <<" :  "<< sy << endl;
   cout<<"unsigned short int-is diapazonia    0 : "<< su << endl;
   system("pause");
   return 0;
}
```

პროგრამის შესრულების შედეგია:

```
int ikavebs 4 baits
misi diapazonia  -2147483648    :  2147483647
unsigned int-is diapazonia   0 : 4294967295
long int ikavebs 4 baits
misi diapazonia  -2147483648    :  2147483647
unsigned long-is diapazonia   0 : 4294967295
short int ikavebs 2 baits
misi diapazonia:  -32768 :  32767
unsigned short int-is diapazonia    0 : 65535
Press any key to continue . . .
```

switch შეტყობინება

switch შეტყობინება ხშირად გამოიყენება მთელი რიგი **if-else** შეტყობინებების სანაცვლოდ. იგი მჭიდროდაა დაკავშირებული მთელ რიცხვებთან, რადგან გამოსახულება, რომელიც წარმოადგენს **switch** -ის არგუმენტს (ანუ რომლის საფუძველზეც უნდა გადაწყვიტოს ამ შეტყოობინებამ თუ რა მიმდევრობით გაგრძელდეს პროგრამის შესრულება) აუცილებლად არის მთელი (ზოგჯერ ამბობენ, რომ არის რომელიმე ჩამოთვლადი ტიპის). მისი ზოგადი სახე ასეთია:

switch (გამოსახულება)
{
    case კონსტანტა1:
        შეტყობინება;
        . . .
        break;
    case კონსტანტა2:
        შეტყობინება;
        . . .

```
        // სრულდება მომდევნო შეტყობინებაც
    default:
        შეტყობინება;
        ...
        break;
    case კონსტანტა3:
        შეტყობინება;
        ...
        break;
}
```

switch შეტყობინება ითვლის გამოსახულებას, რომელიც შეიძლება იყოს მთელი, სიმბოლური ან ჩამონათვალის ტიპის და ირჩევს შესასრულებლად იმ შტოს, რომლის კონსტანტაც გამოსახულების მნიშვნელობის ტოლია. თუ გამოსახულების მნიშვნელობა არ დაემთხვევა არცერთ კონსტანტას, მაშინ შესრულდება **default** შტო. თუ ეს შტო არ არის განსაზრვრული **switch** შეტყობინებაში, მაშინ ის არაფერს არ გააკეთებს.

კონსტანტების განმეორება დაუშვებელია. შტოების თანმიმდევრობა შეიძლება სხვადასხვა გვარი იყოს. ხშირად, თუ განსაკუთრებული შემთხვევა არაა, **default** შტო ბოლო შემთხვევას წარმოადგენს. რეკომენდირებულია რომ იგი ყოველთვის ჩასვათ **switch** შეტყობინებაში.

შემდეგი მაგალითი შეიცავს **if-else** შეტყობინებების მთელ რიგს:

```
if (operat == '+' ){
   result += value;
}
else
   if (operat == '-' ){
      result -= value;
   }
   else
      if (operat == '*' ){
         result *= value;
      }
      else
        if (operat == '/' ){
           if ( value == 0 ){
              cout <<" Error: Divide by zero \n " ;
              cout <<" operation ignored \n ";
            }
            else
              result /= value;
      }
      else
         cout <<" Unknown operator" << operat << endl;
```

სადაც `operat` სიმბოლური ტიპის ცვლადია და აღნიშნავს ოპერაციის ნიშანს, `value` და `result` – მთელი ტიპის ცვლადებია და აღნიშნავენ შესაბამისად რიცხვს, რომელზეც სრულდება ოპერაცია და შედეგს.

კოდის ეს ნაწილი შეიძლება გადავწეროთ **switch** შეტყობინების საშუალებით, რომელშიც სხვადასხვა **case** შეტყობინებას ვიყენებთ სათანადო ოპერაციის შესასრულებლად. **default** შტო ზრუნავს ყველა იმ შემთხვევაზე, რომლებიც გათვალისწინებული არ გვაქვს.

აღვნიშნოთ, რომ **switch** შეტყობინების გამოყენებით ჩვენი პროგრამა, რომელიც უმარტივეს კალკულატორს ქმნის, არ გამარტივებულა, ის გახდა უფრო გასაგები და აქვს შემდეგი სახე:

```
#include<iostream>
using namespace std;
int main()
{
   int result = 0;  // გამოთვლის შედეგი
   char operat;     // ოპერაციის ნიშანი
   int value;       // ოპერაციაში მონაწილე რიცხვი
   while (1) {
      cout<<"Result: "<<result<<endl;
      cout<<"Enter operator and number: ";
      cin >> operat >> value;
      if((operat == 'q') || (operat =='Q'))
         break;
      switch (operat){
        case '+':
           result += value;
           break;
        case '-':
           result -= value;
           break;
        case '*':
           result *= value;
           break;
        case '/':
           if ( value ==0 ){
              cout<<" Error: Divide by zero \n ";
              cout<<" operation ignored \n ";
           }
           else
              result /= value;
           break;
        default:
           cout<<" Unknown operator "<<operat<<endl;
           break;
     } // of switch
  } // of while
  system("pause");
  return 0;
}
```

switch –ის შიგნით გამოყენებული **break** შეტყობინება წყვეტს (ანუ ამთავრებს) **switch** შეტყობინებას. თუ არ წერია **break**, მაშინ შესრულდება რიგის მიხედვით შემდეგი შეტყობინებები. მაგალითად,

```
int control=0;
/* ასეთ დაპროგრამებას კარგი არ ეთქმის */
   switch (control) {
     case 0:
        cout<<"Morning\n";
     case 1:
        cout<<"Noon\n";
        break;
     case 2:
        cout<<"Evening\n";
   }
```

იმ შემთხვევაში, თუ `control == 0`, პროგრამა დაბეჭდავს:

```
Morning
Noon
```

case 0: შემთხვევა არ დამთავრდა **break** შეტყობინებით, ამიტომ morning-ის დაბეჭდვის შემდეგ შესრულდა შემდეგი შეტყობინებაც (**case** 1:), შედეგად დაიბეჭდა noon.
პრობლემა სინტაქსში მდგომარეობს. როცა პროგრამისტს ავიწყდება **break** შეტყობინებით ერთ-ერთი **case** –ის დამთავრება, მაშინ შესრულდება მომდევნო შეტყობინებები, ამ შემთხვევაში **case** 1. როცა პროგრამისტი გამიზნულად გამოტოვებს **break** შეტყობინებას ორი ან მეტი **case**–ის გაერთიანების მიზნით, მაშინ აუცილებლად უნდა გააკეთოს შესაბამისი კომენტარი /* სრულდება მომდევნო შეტყობინებაც */. ანუ, განხილულ მაგალითში **case** 0: მიიღებს შემდეგ სახეს:

```
case 0:
   cout<<"Morning\n";
   /* სრულდება მომდევნო შეტყობინებაც */
```

ვინაიდან **case** 2: ბოლო შტოა, ამიტომ აქ უკვე შეიძლება **break** შეტყობინების გამოტოვება.
და ბოლოს, ისმის კითხვა, რა ხდება, როცა control = 5? ვინაიდან **switch** -ს არ აქვს შტო **default**, ამიტომ ის უბრალოდ გაატარებს ამ შემთხვევას და არაფერი არ შესრულდება.
პროგრამისტმა ყოველთვის უნდა ჩართოს **default** შტო, ვინაიდან ცვლადმა control, გარდა მნიშვნელობებისა 0,1,2, შესაძლოა სხვა მნიშვნელობებიც მიიღოს. ყოველივე ზემოთქმულიდან გამომდინარე, მაგალითი მიიღებს შემდეგ სახეს:

```
int control=5;
  /* უკეთესი ვერსია */
  switch (control) {
    case 0:
       cout<<"Morning\n";
          /* სრულდება მომდევნო შეტყობინებაც */
    case 1:
       cout<<"Noon\n";
       break;
    case 2:
       cout<<"Evening\n";
        break;
    default:
          cout<<"Internal error, control value "
               <<control<<" impossible\n";
       break;
  }
```

მაშინაც კი, როცა ნამდვილად ვიცით, რომ **default** არ არის აუცილებელი, მაინც უნდა ჩავრთოთ **switch** -ში, თუნდაც ასე:

```
default:
/*საერთოდ  არაფერი სრულდება */
break;
```

თავი 5: ნამდვილი რიცხვები და განმეორების შეტყობინება

- ნამდვილი რიცხვები
- შემოკლებული ოპერატორები
- ჩრდილოვანი ეფექტები
- `++x` ოუ `x++`
- განმეორების(`looping`) შეტყობინება
- **`while`** შეტყობინება
- **`continue`** შეტყობინება
- **`break`** შეტყობინება
- გვერდითი ეფექტები

ნამდვილი რიცხვები

ნამდვილმა რიცხვმა შეიძლება მთელი მნიშვნელობაც მიიღოს. მაგრამ თუ ერთ-ერთი ნამდვილი ტიპის (მაგ. **`float, double`**) ცვლადი მიიღებს მთელ მნიშვნელობას, კომპიუტერში მისი წარმოდგენა მაინც განსხვავებულია იგივე რიცხვითი სიდიდის მქონე მთელი ტიპის ცვლადის წარმოდგენისგან. ნამდვილი რიცხვების წარმოდგენის სპეციფიკის გამო, პროგრამირებაში ისინი უფრო ხშირად მოიხსენიება როგორც **მცოცავწერტილიანი** რიცხვები. მაგალითად, 5.5, 8.3, -12.6 არის მცოცავწერტილიანი რიცხვები. მცოცავწერტილიანი რიცხვების და მთელი რიცხვების გასარჩევად C++ იყენებს ათობით წერტილს. ასე, 5.0 არის მცოცავწერტილიანი რიცხვი, მაშინ როცა 5 არის მთელი. მცოცავწერტილიანი რიცხვი აუცილებლად შეიცავს ათობით წერტილს. მაგალითად: 3.14, 0.7, 5.47 და ა.შ.

თუმცა დაშვებულია ციფრი 0-ის გამოტოვება წერტილის წინ ან შემდეგ, მაინც რეკომენდებულია მათი სრული სახით ჩაწერა. მაგალითად, .51-ის ნაცვლად უმჯობესია ვწეროთ 0.51.

C++-ში რიცხვი შეიძლება შეიცავდეს ექსპონენტას. მაგალითად, `1.2e34` (ანუ `1.2 x` 10^{34}).

ჩვენ ძირითადად გამოვიყენებთ ორმაგი სიზუსტის მცოცავწერტილიანი ტიპის რიცხვებს. ეს ყველაზე გავრცელებული ტიპია და ასე აღიწერება:

`double` *`ცვლადი;`* *`// კომენტარი`*

მცოცავწერტილიანი რიცხვების დიაპაზონი დამოკიდებულია კომპიუტერზე.

`double` ტიპზე გადატვირთულია შეტანა გამოტანის ოპერატორები (<<, >>), არითმეტიკული ოპერატორების ნაწილი (ნაშთის აღების ოპერატორი % აღარ გვაქვს), მისამართის აღების, ზომის განსაზღვრის და ბევრი სხვა ოპერატორი. ეს ძალიან ამარტივებს კოდს, მაგრამ უნდა გვახსოვდეს, რომ ერთი და იგივე ოპერატორები სხვადასხვა ტიპის მონაცემებზე სხვადასხვანაირად მოქმედებენ (ანუ სხვადასხვა ალგორითმის საფუძველზე არიან აგებული). მაგალითად განვიხილოთ **რიცხვების გაყოფა.**

მთელი რიცხვების გაყოფა განსხვავდება მცოცავწერტილიანი რიცხვების გაყოფისგან. მთელი რიცხვების გაყოფისას მიიღება მთელი რიცხვი, წილადი ნაწილი კი იგნორირდება. ასე, რომ 19/10 იგივეა რაც 1.

როცა გასაყოფი და გამყოფი, ან ერთ-ერთი მათგანი მცოცავწერტილიანია, მაშინ შედეგიც მცოცავწერტილიანი რიცხვია. მაგალითად, 19.0/10.0 არის 1.9 (ისევე როგორც 19/10.0 და 19.0/10). რამდენიმე მაგალითი მოტანილია შემდეგ ცხრილში:

გამოსახულება	შედეგი	შედეგის ტიპი
1 + 2	3	მთელი
1.0 + 2	3.0	მცოცავწერტილიანი
19 / 10	1	მთელი
19 / 10.0	1.9	მცოცავწერტილიანი

მე –2 და მე –4 მაგალითში C++ ავტომატურად გარდაქმნის მთელს მცოცავწერტილიან მნიშვნელობად. იგი ანალოგიურ გარდაქმნას ასრულებს, როდესაც ხდება მთელი ტიპის ცვლადზე მცოცავწერტილიანი მნიშვნელობის მინიჭება.

მაგალითად, განვიხილოთ პროგრამის ფრაგმენტი:

```
int main()
{
    int i;                    // მთელი ტიპის ცვლადი სახელით i
    float x;                  // მცოცავწერტილიანი ცვლადი სახელით x
    x = 1.0 / 2.0;            // მიანიჭებს ცვლად x - ს მნიშვნელობას 0.5
    i= 1 / 3;                 // მიანიჭებს ცვლად i - ს მნიშვნელობას 0
    x = (1 / 2) + (1 / 2);    // მიანიჭებს ცვლად x -ს მნიშვნელობას 0.0
    x = 3.0 / 2.0;            // მიანიჭებს ცვლად x-ს მნიშვნელობას 1.5
    i = x;                    // მიანიჭებს ცვლად i-ს მნიშვნელობას 1
    return (0);
}
```

შევნიშნოთ, რომ 1/2 არის მთელი ტიპის გამოსახულება და მისი მნიშვნელობაა 0.

შემოკლებული ოპერატორები

C++–ში არის არაერთი ოპერატორი, რომლის შესრულება ნიშნავს ორი სხვა ოპერატორის ან ინსტრუქციის შესრულებას. როგორც წესი, ამ ოპერატორების უმრავლესობა აგრეთვე გადატვირთულია სხვადასხვა საბაზო ტიპზე. უფრო მეტიც, მომხმარებლის მიერ განსაზღვრულ სხვადასხვა ტიპისთვის (კლასისთვის) ხშირად ძალიან მოსახერხებელია შემოკლებული და სხვა ოპერატორების (ნაწილის მაინც) გადატვირთვა.

მაგალითად, ძალიან ხშირად არის საჭირო რომელიმე ცვლადის, მთელის ან ნამდვილის, (ვთქვათ, counter) მნიშვნელობის ერთით გაზრდა. თუ ამას მინიჭების შეტყობინებით გავაკეთებთ, გვექნება:

```
counter = counter +1;
```

იგივე შედეგს მოგვცემს **++** (მომატების ოპერატორი ანუ ინკრიმენტი) ოპერატორის გამოყენება:

```
++counter;
```

ანალოგიური მოქმედებისაა -- (მოკლების ოპერატორი ანუ დენკრიმენტი) ოპერატორი, რომელიც ერთით ამცირებს ცვლადის მნიშვნელობას. თუ გვინდა არა ერთით, არამედ 3–ით გავზარდოთ ცვლადის მნიშვნელობა, უნდა გამოვიყენოთ += ოპერატორი, ანუ შემდეგი ორი შეტყობინება აკეთებს ერთი და იგივე რამეს:

```
counter = counter + 3;
counter += 3;
```

შემდეგ ცხრილში მოყვანილია რამდენიმე მაგალითი:

ოპერატორები	შეტყობინება	ეკვივალენტური შეტყობინება
+=	x += 2;	x = x + 2;
-=	x -= 2;	x = x - 2;
*=	x *= 2;	x = x * 2;
/=	x /= 2;	x = x / 2;
%=	x %= 2;	x = x % 2;

ჩრდილოვანი ეფექტები

სამწუხაროდ, C++ არ კრძალავს **ჩრდილოვანი ეფექტების** გამოყენებას. ჩრდილოვანი ეფექტი ნიშნავს ოპერაციას, რომელიც სრულდება შეტყობინების ძირითადი ოპერაციასთან ერთად, დამატებით. მაგალითად, შემდეგი კოდი სრულიად კორექტულია:

```
size = 5;
result = ++size;
```

პირველი შეტყობინება მიანიჭებს `size`–ს მნიშვნელობას 5, მეორე მიანიჭებს `result` ცვლადს `size` –ის მნიშვნელობას (ესაა ძირითადი ოპერაცია), მაგრამ დამატებით კიდევ, გაზრდის `size` –ს (ჩრდილოვანი ეფექტი). აქ საკმაოდ ფაქიზი საკითხია იმის გარკვევა, თუ რა მიმდევრობით შესრულდება ეს ოპერაციები, მაგრამ ჩვენ ამის გარკვევას არ დავიწყებთ და არც ჩრდილოვან ეფექტებზე გავაგრძელებთ მსჯელობას, რადგან პროგრამირების კარგი სტილი გულისხმობს, რომ მაქსიმალურად უნდა ავარიდოთ თავი ჩრდილოვანი ეფექტების გამოყენებას.

`++x` თუ `x++`

მომატების ოპერატორის ეს ორი ფორმა ერთნაირად ეფექტურია C++ –ში, და თუ ჩრდილოვან ეფექტებს თავს ავარიდებთ, შედეგიც ერთნაირი აქვთ. მაგრამ პირველი, ე.წ. პრეფიქსული ფორმა `++x` შედარებით ეფექტურია და ამიტომ სჯობს თავიდანვე მას მივანიჭოთ უპირატესობა. ზოგადად, მათი მოქმედების მექანიზმი ასეთია: თუ გამოსახულებაში მონაწილეობს `x++`, მაშინ ჯერ გამოსახულების მნიშვნელობა გამოითვლება და მერე `x` –ის მნიშვნელობას მოემატება ერთი. ამიტომ, შემდეგი ფრაგმენტის შედეგად

```
n = 5;
x = n++;
```

`x = 5`. ამისგან განსხვავებით, შემდეგი ფრაგმენტის შედეგად:

```
n = 5;
x = ++n;
```

გვექნება `x = 6`.

განმეორების (looping) შეტყობინება

განმეორების შეტყობინება საშუალებას იძლევა გავიმეოროთ პროგრამული კოდის ნაწილი რამდენჯერმე ან მანამდე, ვიდრე სრულდება რაიმე პირობა. მაგალითად, განმეორების შეტყობინებები გამოიყენება, როცა ვითვლით დოკუმენტში სიტყვების რაოდენობას, ან სტუდენტთა შორის ფრიადოსნების რაოდენობას.

განმეორების შეტყობინებებია: **`while`**, **`for`** და **`do-while`**.

`while` შეტყობინება

`while` შეტყობინების ზოგადი ფორმა ასეთია:

`while` (*პირობა*)

შეტყობინება;

პროგრამა შეასრულებს **`while`** - ში არსებულ შეტყობინებას რამდენჯერმე, მანამ სანამ *პირობა* გახდება მცდარი (0). თუ პირობა თავიდანვე მცდარია, მაშინ ეს შეტყობინება საერთოდ არ შესრულდება. შესაძლოა, **`while`** –ის პირობის ჭეშმარიტების შემთხვევაში საჭირო გახდეს არა ერთის, არამედ რამოდენიმე შეტყობინების შესრულება. მაშინ შესასრულებელი შეტყობინებები ჩაისმება { } ფრჩხილებში, ანუ ბლოკში. **`while`** –ის ამ ნაწილს ეწოდება ტანი.

განვიხილოთ მაგალითი. ვთქვათ სტუდენტთა ჯგუფმა ჩააბარა გამოცდა. ჯგუფში 6 სტუდენტია. მათი გვარები და შეფასებები შეგვყავს კლავიატურიდან. ვიპოვოთ იმ სტუდენტთა რაოდენობა, რომლებმაც ჩააბარეს გამოცდა (ანუ მიიღეს 51 ქულა ან უფრო მეტი).

შევადგინოთ ალგორითმის ფსევდოკოდი:

1. სტუდენტის გვარის და სტუდენტის ქულის შესანახად გვჭირდება ორი ცვლადი, მაგალითად `name` და `grade`. ქულის შემოწმების შემდეგ ამ ცვლადებს გამოვიყენებთ სხვა სტუდენტისთვის.
2. **`while`** - ის პირობისთვის გვჭირდება მთვლელი (`stud_count`). ვიდრე ეს ცვლადი (0-იდან დაწყებული) 6 -ზე ნაკლებია, **`while`** შეტყობინება რიგრიგობით შეიყვანს სტუდენტების მონაცემებს `name` და `grade` ცვლადებში და შეამოწმებს ქულას.
3. თუ ქულა 51 -ზე მეტია, პროგრამამ ერთით უნდა გაზარდოს კიდევ ერთი ცვლადის (`passed_count`) მნიშვნელობა, რომელიც წარმოადგენს გამოცდაჩაბარებული სტუდენტების მთვლელს და რომლის მნიშვნელობაც თავიდან უნდა იყოს ნულის ტოლი.

ცხადია, ბოლოს პასუხი უნდა დავბეჭდოთ.

C++ -ის შესაბამის პროგრამას აქვს სახე:

```
//////////////////////////////////////////////////////
// პროგრამა grade.cpp
// რომელიც 6 სტუდენტის მონაცემებს მიიღებს კლავიატურიდან
// და დაბეჭდავს გამოცდაჩაბარებულების რაოდენობას
//////////////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int stud_count = 0;     // სტუდენტების მთვლელი
    string name;            // ცვლადი გვარისთვის
    int grade;              // ცვლადი ქულისთვის
    int passed_count = 0;   // გამოცდაჩაბარებული  სტუდენტების მთვლელი

    // ვიდრე განხილული სტუდენტების რაოდენობა 6-ს გადააჭარბებს
    while (stud_count < 6)
    {
       // მორიგი სტუდენტის მონაცემი
       cout <<"Enter name"<<endl;
       cin >> name ;

       cout <<"Enter grade"<<endl;
       cin >>grade;

       if(grade > 50)
           ++passed_count;   // გავზარდოთ გამოცდაჩაბარებულების მთვლელი
       stud_count++;         // გავზარდოთ სტუდენტების მთვლელი
    }
    // დავბეჭდოთ გასულების რაოდენობა
    cout <<"number of passed students is " << passed_count <<endl;
    system("PAUSE");
    return 0;
}
```

მაგალითად:

```
Enter name
Arevadze
Enter grade
67
Enter name
Buadze
Enter grade
40
Enter name
Jalali
Enter grade
75
Enter name
Iashvili
Enter grade
100
Enter name
Lejava
Enter grade
50
Enter name
Nikoladze
Enter grade
51
number of passed students is 4
Press any key to continue . . .
```

აქვე განვიხილოთ ამ პროგრამის ერთი განზოგადება, რომელიც საშუალებას გვაძლევს ამოვხსნათ იგივე ამოცანა იმ შემთხვევაშიც, თუ ჯგუფებში სტუდენტთა რაოდენობა ფიქსირებული არაა (სხვადასხვა ჯგუფისთვის სხვადასხვა შეიძლება იყოს). განმეორების შეტყობინება შეიტანს მონაცემებს მანამდე, ვიდრე ჩვენ არ ავკრეფთ კლავიატურიდან კლავიშთა **Ctrl+z** მიმდევრობას, რომელსაც **ფაილის დასასრული** ეწოდება და რომელიც ეკრანზე გამოჩნდება როგორც **^Z.**

იმისათვის, რომ კარნახი ყოველი შესაყვანი სტრიქონის წინ გამოვიდეს, ჩვენ მისი ორჯერ დაწერა მოგვიწევს (სცადეთ დამოუკიდებლად, რომ გააკეთოთ როგორმე სხვანაირად და მიხვდებით რატომ ვწერთ ასე).

```
//////////////////////////////////////////////////
// პროგრამა grade1.cpp
// რომელიც სტუდენტების მონაცემებს მიიღებს კლავიატურიდან
// და დაბეჭდავს  გამოცდაჩაბარებულების რაოდენობას
//////////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;            // ცვლადი გვარისთვის
    int grade;              // ცვლადი ქულისთვის
    int passed_count = 0;   // გამოცდაჩაბარებული  სტუდენტების მთვლელი
    cout <<"Enter name and grade, separated by space"<<endl;
    while (cin >> name >>grade)
    {
       if(grade > 50)
           ++passed_count;
       cout <<"Enter name and grade, separated by space"<<endl;
    }
```

```
      // დავბეჭდოთ გასულების რაოდენობა
    cout <<"number of passed students is " << passed_count <<endl;
    system("PAUSE");
    return 0;
}
```

`continue` შეტყობინება

`continue` შეტყობინება წყვეტს განმეორების შეტყობინების **მიმდინარე ბიჯს**, გამოტოვებს **`continue`**-ს მომდევნო შეტყობინებებს და თავიდან იწყებს განმეორების შეტყობინების **მომდევნო ბიჯის** შესრულებას.

ვთქვათ, ისევ იგივე ამოცანას ვხსნით, მხოლოდ ახლა ცალ-ცალკე დავბეჭდოთ გამოცდაჩაბარებული, ჩაჭრილი და განმეორებით გამსვლელი სტუდენტების გვარები.

შესაბამის C++-პროგრამას აქვს სახე:

```
//////////////////////////////////////////////////////
// continue შეტყობინების გააზრება
// პროგრამა ითვლის და ბეჭდავს გამოცდაჩაბარებული, ჩაჭრილი
// და განმეორებით გამსვლელი სტუდენტების რაოდენობებს
//////////////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;

int main()
{
    int stud_count(0);
    int passed_count(0);
    int failed_count(0);
    string name;
    int grade;

    cout <<"Enter name and grade, separated by space"<<endl;
    while (cin >> name >>grade)
    {
       stud_count++;
       if(grade < 41)
       {
          ++failed_count;
          continue;
       }
       if(grade > 50)
          ++passed_count;

       cout <<"Enter name and grade, separated by space"<<endl;
    }
    cout <<"number of passed students is " << passed_count <<endl;
    cout <<"of failed students is " << failed_count <<endl;
    cout <<"meored gasulTa raodenobaa "
         << stud_count - passed_count - failed_count <<endl;
    system("PAUSE");
    return 0;
}
```

პროგრამის მუშაობის შედეგს ჩვენს მონაცემბზე აქვს სახე:

```
number of passed students is 4
of failed students is 1
meored gasulTa raodenobaa 1
```

შევნიშნოთ, რომ ჩვენ შევამცირეთ კომენტარების რაოდენობა, რადგან პროგრამა საკმაოდ მარტივია.

შემდეგი პროგრამა ბეჭდავს სტუდენტების გვარებს და მათ მიერ მიღებულ შეფასებებს, ქულებიდან გამომდინარე.

```
//////////////////////////////////////////////
// continue შეტყობინების გააზრება
// პროგრამა ბეჭდავს სტუდენტების გვარებს და მათ მიერ
// მიღებულ შეფასებებს, ქულებიდან გამომდინარე.
//////////////////////////////////////////////
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string name;
    int grade;
    cout <<"Enter name and grade, separated by space,or Ctrl+z"<<endl;
    while (cin >> name >>grade)
    {
       if(90 < grade ) {
            cout << name << '\t' << 'A'<<endl;
            continue;
       }
       if(80 < grade  && grade < 91 ) {
            cout << name << '\t' << 'B'<<endl;
            continue;
       }
       if(70 < grade  && grade < 81 ) {
            cout << name << '\t' << 'C'<<endl;
            continue;
       }
       if(60 < grade  && grade < 71 ) {
            cout << name << '\t' << 'D'<<endl;
            continue;
       }
       if(50 < grade  && grade < 61 )  {
            cout << name << '\t' << 'E'<<endl;
            continue;
       }
       if(40 < grade  && grade < 51 )  {
            cout << name << '\t' <<"Try again"<<endl;
            continue;
       }
       if(grade < 41 )  {
            cout << name << '\t'<< "failed"<<endl;
            continue;
      }
    }
    system("PAUSE");
    return 0;
}
```

შედეგი ნაჩვენებია შემდეგ სურათზე:

```
Enter name and grade, separated by space,or Ctrl+z
Arevadze 67
Arevadze        D
Buadze 40
Buadze  failed
Jalali 76
Jalali  C
Iashvili 100
Iashvili        A
Lejava 50
Lejava  Try again
Nikoladze 51
Nikoladze       E
^Z
Press any key to continue . . .
```

break შეტყობინება

while-ში გაერთიანებული შეტყობინებების განმეორება მთავრდება, როცა მისი პირობა ხდება მცდარი (0). თუმცა განმეორების შეტყობინება შეგვიძლია შევწყვიტოთ ნებისმიერ ბიჯზე **break** (შეწყვეტა) შეტყობინების გამოყენებით.

შემდეგი მარტივი პროგრამა წარმოადგენს ამ შეტყობინების გამოყენების ნიმუშს.

```
//////////////////////////////////////////////////
// პროგრამა ითვლის [0,99] შუალედში მოთავსებული
// შემთხვევითი რიცხვების ჯამს. შეკრება წყდება,
// როგორც კი შემთხვევითი რიცხვი არის 13-ის ჯერადი.
//////////////////////////////////////////////////
#include <iostream>
using namespace std;
int main()
{
    int s = 0;
    int i;

    while ( true )
    {
        i = rand()%100;
        if (i%13 == 0) break;
         s += i ;
    }
    cout<<"s = " << s <<endl;
    system("PAUSE");
    return 0;
}
```

შემდეგი ამოცანა აგრძელებს "info.txt" ფაილში ჩაწერილი სტუდენტების მონაცემების დამუშავების თემას. ამჯერად ჩვენ გვინდა დავბეჭდოთ ერთ-ერთი ისეთი სტუდენტის მონაცემები, რომელსაც მიღებული აქვს უმაღლესი შეფასება.

```
//////////////////////////////////////////////
// break შეტყობინების გააზრება
// პროგრამა ბეჭდავს სტუდენტის მონაცემებს, რომელსაც
// მიღებული აქვს უმაღლესი შეფასება.
//////////////////////////////////////////////
```

```cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
    string name;            // სტუდენტის გვარი
    int grade;              // ქულა
    int max_grade = 0;      // მაქსიმალური ქულა
    string max_student;     // მაქსიმალური ქულის მქონის გვარი

    ifstream fin("info.txt"); // ფაილის გახსნა მონაცემების წასაკითხად

    // ვიდრე ფაილში არის წასაკითხი სტრიქონები
    while ( fin >> name >>grade )
    {
        // 100 ქულაზე მეტი არ იქნება, ამიტომ აქ განმეორება შეწყდება
       if( 100 == grade )
       {
          cout<<"students "<< name
              <<" aqvs umaglesi Sefaseba 100 qula " << endl;
          break;
       }
       // თუ 100 ქულა არაა, ჩვეულებრივად ვეძებთ მაქსიმუმს
       if(grade > max_grade)
       {
          max_grade = grade;
          max_student = name;
       }
    }

     fin.close();   // ფაილის დახურვა

    /* თუ 100 ქულიანი არ შეგვხვდა, grade < 100  იქნება,
       ამიტომ მაქსიმალური ქულის მქონის მონაცემებს აქ ვბეჭდავთ */
    if( grade < 100 )
       cout << "students " << max_student << " aqvs umaglesi Sefaseba "
            << max_grade <<" qula " << endl;

    system("pause");
    return 0;
}
```

შედეგი:

info.txt ფაილი	გამოტანის ეკრანი
Arevadze 67 Buadze 40 Jalali 76 Iashvili 100 Lejava 50 Nikoladze 51	students Iashvili aqvs umaglesi Sefaseba 100 qula

გვერდითი ეფექტები

C++ საშუალებას გვაძლევს გამოვიყენოთ მინიჭების შეტყობინება თითქმის ყველგან. მაგალითად, მინიჭების შეტყობინება შეგიძლიათ ჩასვათ სხვა მინიჭების შეტყობინების შიგნით:

```
/* არ დააპროგრამოთ ასე  */
        average = total_value / (number_of_entries = last - first);
```

ეს ექვივალენტურია შემდეგი ორი შეტყობინებებისა:

```
/* დააპროგრამეთ ასე */
        number_of_entries = last - first;
        average = total_value / number_of_entries;
```

პირველი ვერსია “მალავს” `number_of_entries`–ის მინიჭებას გამოსახულების შიგნით. პროგრამა კი ნათელი და მარტივი უნდა იყოს.

C++ იმის საშუალებასაც აძლევს პროგრამისტს, რომ მან მინიჭების შეტყობინება ჩასვას **`while`**–ის პირობაში. მაგალითად:

```
/* არ დააპროგრამოთ ასე */
      while ((current_number = last_number + old_number) < 100)
          cout << "Term " << current_number << endl;
```

ერიდეთ ასეთნაირად დაპროგრამებას. მიაქციეთ ყურადღება, რამდენად კარგად ჩანს ლოგიკა ქვემოთ მოყვანილ ვერსიაში:

```
/* დააპროგრამეთ ასე */
      while (1) {
          current_number = last_number + old_number;
          if (current_number >= 100)
             break;
          cout << "Term " << current_number << endl;
      }
```

თავი 6: ფუნქციები, სტრიქონები (`string`) და ვექტორები (`vector`)

- ფუნქციები
- ობიექტები და მეთოდები
- სტრიქონები (`string`) და ვექტორები (`vector`) - ზოგადი მიმოხილვა
- სტრიქონის (`string`) ობიექტის კონსტრუირება
- მოქმედება სტრიქონებზე

ფუნქციები

ფუნქცია არის პროგრამული კოდის ბლოკი (ანუ ფიგურულ ფრჩხილებს შორის მოთავსებული ფრაგმენტი), რომელსაც მინიჭებული აქვს სახელი. სახელის საშუალებით ხდება ფუნქციის გამოძახება იმდენჯერ, რამდენჯერაც საჭიროა.

პროგრამული კოდის ნებისმიერ ფრაგმენტს ვერ ვაქცევთ ფუნქციად, რადგან აუცილებელია სინტაქსის გარკვეული წესების დაცვა.

მეორე მხრივ, პროგრამული კოდის ნებისმიერი ფრაგმენტი არც უნდა ვაქციოთ ფუნქციად, რადგან ფუნქციის შესაქმნელად აუცილებელია მოტივის არსებობა. აზრი არა აქვს ისეთი ფუნქციის შექმნას, რომელსაც არასოდეს, ან მხოლოდ ერთხელ გამოვიყენებთ. პროგრამული უზრუნველყოფის ინჟინერიაში (კურიკულუმით ეს საგანი გათვალისწინებულია მეშვიდე სემესტრში) არსებობს რამდენიმე პრინციპი, რომელთა გათვალისწინება ხდება ფუნქციის შექმნის დროს. ერთი ძირითადი პრინციპი არის **მრავალჯერ გამოყენებადობა**, ანუ **reusability.** იდეალურ შემთხვევაში, ახალი ფუნქცია თავსდება რომელიმე ბიბლიოთეკაში, რათა მისაწვდომი იყოს მომხმარებლებისთვის. პროგრამული კოდის კარგი ბიბლიოთეკა ბევრია, ზოგი მათგანი არის ღია ლიცენზიით, ზოგი კომერციულით.

სხვა გარემოებები, რაც უნდა გავითვალისწინოთ ფუნქციის შექმნისას, არის: კარგი სტილი, კოდის გარჩევის და გადაკეთების სიადვილე, თვითონ ფუნქციის ზომა (მიზანშეწონილად არ ითვლება ძალიან გრძელი ფუნქციების შექმნა) და სხვა.

ფუნქცია ინდენად ფუნდამენტური ცნებაა პროგრამირებაში, რომ მუდმივად ხდება მისი განვითარება პროგრამირების ენებში, შესაბამისად, ნებისმიერი პროგრამისტი ერთი მხრივ მუდმივად იძენს გამოცდილებას, მეორე მხრივ კი მუდმივად ითვისებს ახალ-ახალ საშუალებებს, რასაც ახალი კომპილერები სთავაზობს. ამიტომ დავიწყოთ უმარტივესით, რაც უკვე ვიცით ოდნავ სხვა ტერმინებში.

ჩვენ უკვე ვისაუბრეთ აღნიშვნებზე. მაგალითად, მათემატიკური $h : R \to Z$ ჩანაწერი C++ ენაში იღებს (კომპილერისთვის) განაცხადის სახეს:

```
int h(double);
```

თუმცა, რეალურ პროგრამირებაში არავინ იყენებს ერთსიმბოლოიან სახელს ფუნქციისთვის, რადგან ეს ეწინააღმდეგება კარგ სტილს. როგორც ვხედავთ ფუნქციის აღწერა გადადის განაცხადში (პროგრამირებაშიც ვიყენებთ ტერმინს აღწერა, იგივე დანიშნულებით რაც მათემატიკაში). ახლა ვნახოთ როგორ სახეს მიიღებს ფუნქციის განსაზღვრა. მაგალითად, თუ $f : R \to R$ წარმოადგენს კვადრატულ სამწევრს $f(x) = 5x^2 + 11x - 2$, ამ ფუნქციაზე განაცხადი და მისი განსაზღვრა ერთად C++ ენაში იღებს სახეს:

```
double f(double x)
{
    return 5*x*x+11*x-2;
}
```

და მას ეწოდება ფუნქციის განხორციელება ანუ **იმპლემენტაცია.**

მაგალითი ძალიან მარტივია, მაგრამ საკმაოდ ზოგადიც: თუ ფუნქცია ითვლის რაიმე გამოსახულების მნიშვნელობას, შეგვიძლია მექანიკურად დავიმახსოვროთ, რომ ტოლობის ნიშნის როლს ასრულებს რეზერვირებული სიტყვა **return**. რეკომენდებულია, რომ განაცხადი ფუნქციაზე, ანუ ფუნქციის **პროტოტიპი** მოთავსებული იყოს **main()** ფუნქციის წინ, ხოლო განაცხადი და განსაზღვრა ერთად - **main()**-ის შემდეგ. ან, რაც უფრო პროფესიონალურია, პროტოტიპები მოთავსებული იყოს ცალკე header ფაილში, ხოლო იმპლემენტაციები - იგივე სახელის მქონე .cpp ფაილში.

განხილული მაგალითი ძალიან მარტივი და არაბუნებრივი იყო, რადგან კონკრეტულად ამ სახის ფუნქციაზე მოთხოვნილება ალბათ არავის გაუჩნდება სხვა დროს. განვიხილოთ სხვა, შედარებით რეალისტური ამოცანა. შევქმნათ ფუნქცია, რომელიც ყოველი ნამდვილი რიცხვისთვის გამოთვლის მის ნიშანს. თუ ფუნქციას დავარქმევთ sign სახელს, მათემატიკურად მისი აღწერა ასეთი იქნება:

$$sign : R \to Z, \qquad sign(x) = \begin{cases} 0, & თუ \quad x = 0, \\ 1. & თუ \quad x > 0, \\ -1, & თუ \quad x < 0. \end{cases}$$

ფუნქციის პროტოტიპი არის:

```
int sign(double);
```

ხოლო იმპლემენტაციაა:

```
int sign (double x)
{
    if(x > 0)
        return 1;
    if(x < 0)
        return -1;
    return 0;
}
```

ყურადღება მივაქციოთ რამდენიმე გარემოებას:

- მათემატიკურ აღწერაში სულ ერთია რომელი შემთხვევა იქნება პირველი, რომელი მეორე და რომელი მესამე. მაგრამ განხორციელებაში (იმპლემენტაციაში) პირველ რიგში უნდა გავითვალისწინოთ ყველაზე ალბათური შემთხვევა, მერე ნაკლებ ალბათური და ა. შ.
- ფუნქციების შიგნით **if-else** კონსტრუქცია ძალიან სპეციფიკურად მუშაობს, რადგან ფუნქციის შიგნით პირველივე **return** ამთავრებს ფუნქციას. ამიტომ, თუ შესრულდა ისეთი **if** -ის პირობა, რასაც მოყვება **return**-ოპერატორი, მაშინ **else** -ის გამოყენება უკვე ზედმეტია, რადგან ფუნქცია დაასრულებს მუშაობას. ამიტომ აქვს ჩვენ მაგალითს ასეთი სახე.

ახლა განვიხილოთ მარტივი არამათემატიკური ფუნქციის მაგალითი, რომელსაც არგუმენტი არ სჭირდება. ვთქვათ, გვინდა გვქონდეს ფუნქცია, რომელიც საჭიროების შემთხვევაში დაბეჭდავს გზავნილს შეცდომის შესახებ. მის იმპლემენტაციას შესაძლოა ჰქონდეს ასეთი სახე:

```
void errorMessage (void)
{
    std::cout << "Error!" << std::endl;
}
```

მოკლედ შევეხოთ ამ ფუნქციების გამოყენების საკითხს. ვთქვათ, ამ სამივე ფუნქციის პროტოტიპი მოყვანილია **main()** ფუნქციის წინ, ხოლო იმპლემენტაციები მის შემდეგ. მაშინ:

```
    std::cout << f(1.5) << std::endl;
```

დაბეჭდავს კვადრატული სამწევრის მნიშვნელობას როცა არგუმენტის მნიშვნელობაა `1.5`. იგივეს იზამს შემდეგი ფრაგმენტი:

```
double x = 1.5;
std::cout << f(x) << std::endl;
```

შემდეგი ფრაგმენტი:

```
double x = -71.5;
std::cout << sign(x) <<s td::endl;
```

დაბეჭდავს `-71.5`-ის ნიშანს, ანუ `-1` -ს.

ბოლოს, შეტყობინება

```
errorMessage ();
```

დაბეჭდავს მარტივ გზავნილს: `Error!`

დავიმახსოვროთ რამდენიმე მნიშვნელოვანი ფაქტი:

1. ფუნქციის შიგნით სხვა ფუნქციის განსაზღვრა აკრძალულია, თუმცა ფუნქციის შიგნით შეგვიძლია გამოვიძახოთ უკვე არსებული სხვა ფუნქციები;
2. ერთადერთი ფუნქცია, რომელსაც არ იძახებს რომელიმე სხვა ფუნქცია, არის **`main()`.** მისი სახელიც მიგვითითებს, რომ მისგან ხდება სხვა ფუნქციების გაშვება (რომლებმაც შეიძლება კიდევ სხვა ფუნქციები გაუშვან და ა.შ.);
3. თუ **`main()`** ფუნქციის წინ მოვათავსებთ რამდენიმე ფუნქციის იმპლემენტაციას, მაშინ მოგვიწევს ანგარიში გავუწიოთ, რომ რომელიმე ფუნქციის იმპლემენტაციაში არ იყოს ისეთი ფუნქციის გამოძახება, რომელიც მის წინ არაა იმპლემენტირებული.

ფუნქციების შესწავლა გრძელდება მთელი სიცოცხლის განმავლობაში, უფრო კონკრეტულად, ყველა მომდევნო მეცადინეობაზე მეტ-ნაკლები ინტრენსივობით შევეხებით ამ თემას.

ობიექტები და მეთოდები

ობიექტის განმარტება ძალიან გავს ცვლადისას. ობიექტი არის ადგილი კომპიუტერის მეხსიერებაში, რომელიც განკუთვნილია გარკვეული ტიპის მონაცემების შესანახად. მეხსიერება გამოიყოფა განაცხადის გაკეთების მომენტში და ობიექტი იგივდება იმ სახელთან, რომელიც განაცხადშია მითითებული. განსხვავება ცვლადთან იმაში მდგომარეობს, რომ ერთი და იგივე ტიპის ობიექტებს აქვთ შესაძლებლობა, რომ გამოიძახონ სპეციფიკური ფუნქციები, რომლებიც მხოლოდ ამ ტიპის ობიექტებზეა განსაზღვრული, ან ამ ტიპის სპეციფიკის გათვალისწინებით არის იმპლემენტირებული.

განსხვავდება გამოძახების ფორმატიც. მაგალითად, ფუნქცია **`sizeof()`** განსაზღვრულია ნებისმიერი ტიპის `x` ცვლადისა და ობიექტისთვის, და მისი გამოძახება ხდება **`sizeof`**`(x)` სახით. ამისგან განსხვავებით, რამდენიმე ტიპის ობიექტისთვის არსებობს ფუნქცია **`size()`**, რომელიც ზომავს ამ ტიპის `x` ობიექტს რაღაც აზრით (მაგალითად, რამდენი რიცხვი წერია ვექტორში). მაგრამ, რადგან როგორც `x` ობიექტი, ასევე **`size()`** ფუნქცია ეკუთვნის ერთი და იგივე ტიპს, ამიტომ `x` ობიექტისთვის **`size()`** ფუნქციის გამოძახება მოხდება **`x.size()`** სახით.

C++ ენაში ჩაშენებული ტიპების გარდა გათვალისწინებულია ახალი ტიპების შექმნა, რაც კეთდება **კლასის** საშუალებით. ამ საკითხს დიდ დროს დავუთმობთ შემდეგში. ამჯერად საკმარისია დავიმახსოვროთ, რომ ტიპის კონკრეტულ ეგზემპლარს, რომლისთვისაც გამოიყოფა მეხსიერებაში ეგზემპლარის, სახელთან გაიგივებული ადგილი, ეწოდება **ობიექტი**, ხოლო სპეციალურად ამ კონკრეტული ტიპისთვის შექმნილ ფუნქციას ეწოდება **მეთოდი**.

ობიექტის ცნება ძალიან მნიშვნელოვანია ობიექტზე ორიენტირებულ პროგრამირებაში და მას თანდათან უფრო მეტი მნიშვნელობა ენიჭება. მაგალითად, C++ ენის იმ გაფართოებულ დიალექტში, რომელსაც იყენებს `Visual Studio`, და რომელსაც ჰქვია `CLR/C++`, ყველაფერი - მუდმივები, მარტივი და შედგენილი ტიპის ცვლადები და ა. შ. წარმოადგენენ ობიექტებს,

რადგან ყველა ტიპს გააჩნია სტანდარტული ფუნქციების გარკვეული მარაგი. ამ ფუნქციების ნაწილს აქვს ერთი და იგივე სახელი, მაგრამ მორგებული არის ამ კონკრეტულ ტიპზე თავისი იმპლემენტაციით. მაგალითად, ამ ენაში თუ გვინდა რიცხვი `5.5` გარდავქმნათ სტრიქონად, უნდა გამოვიძახოთ შესაბამისი მეთოდი და დავწეროთ: `(5.5).ToString()`. C++ ენის სტანდარტული ვერსია იცავს ბალანს, მარტივი ტიპებისთვის მეთოდები განსაზღვრული (ყოველ შემთხვევაში ცხადი სახით) არ არის.

სტრიქონი (`string`) და ვექტორი (`vector`) – ზოგადი მიმოხილვა.

სტრიქონი (`string`) წარმოადგენს მონაცემთა ერთ-ერთ მნიშვნელოვან ტიპს C++ ენაში, რომელიც შექმნილია როგორც კლასი. ამიტომ ხშირად უპირატესობას მივანიჭებთ დაკონკრეტებას და ვიტყვით "`string` კლასი", "`string` ტიპი"-ს ნაცვლად.

`string` კლასი შეიცავს ბევრ ფუნქციას (მეთოდს), რაც საშუალებას გვაძლევს ვიმუშაოთ სტრიქონებთან იმის მსგავსად, როგორც ვმუშაობთ ენის ჩაშენებულ (**`int`, `double`, `char`**) ტიპებთან. მაგალითად, შეგვიძლია გამოვიყენოთ სტრიქონებთან ზოგიერთი ოპერატორი: `=` (მინიჭების), `==` (ტოლობაზე შედარების) , `>` (მეტობის), `+` (შერწყმის ან კონკატენაციის), `+=` (კონკატენაციისა და მინიჭების) და სხვა.

`string` ტიპის მონაცემი `string` კლასის ობიექტს წარმოადგენს. ამიტომ პროგრამაში შექმნილი ყოველი სტრიქონისთვის მისაწვდომია კლასში განსაზღვრული ფუნქციები. მაგალითად,

```
string S ("This is a test string");
```

განაცხადის შემდეგ პროგრამაში შეგვიძლია გავარკვიოთ `S` სტრიქონის სიგრძე `size()`ფუნქციის გამოძახებით: `S.size()`. იგივე მოქმედების არის ფუნქცია `length()`. პროგრამის ორივე შეტყობინება

```
cout << S.size() << endl;
cout << S.length() << endl;
```

დაბეჭდავს S სტრიქონის მიმდინარე სიგრძეს (21 –ს).

სტრიქონისთვის მეხსიერების განაწილება ხდება დინამიკურად მისი შექმნის მომენტში. მაგალითად, `Visual C++` გარემოში ცარიელ სტრიქონს თავიდანვე გამოეყოფა 15 ბაიტი, ხოლო თუ სტრიქონის სიგრძეს გავზრდით (მაგალითად სხვა სტრიქონის დამატების - კონკატენაციის ხარჯზე), მეხსიერება ორმაგდება იმდენჯერ, რამდენჯერაც საჭიროა.

სრტიქონის სიგრძე შეიძლება შეიცვალოს პროგრამის განმავლობაში. მაგალითად, ფრაგმენტი

```
string K;
K = "First string";
cout << K.size() << endl;
K += " and Second string";
cout << K.size() << endl;
K = "Short";
cout << K.size() << endl;
```

დაბეჭდავს

```
12
30
5
Press any key to continue . . .
```

სტრიქონის თითოეულ სიმბოლოზე მიმართვა (წვდომა) ხდება ე.წ. ინდექსის მეშვეობით, რომელიც არის კვადრატულ ფრჩხილებში ჩაწერილი არაუარყოფითი მთელი რიცხვი. ინდექსის მნიშვნელობის ათვლა იწყება ნულიდან. მაგალითად, K სტრიქონისთვის `K[0]` არის სიმბოლო `'S'`, `K[1]` – სიმბოლო `'h'`, `K[2]` –`'o'`, `K[3]` –`'r'` და `K[4]` – სიმბოლო `'t'`. ე.ი. K

სტრიქონის ბოლო სიმბოლოს ინდექსი არის `K.size()-1` –ის ტოლი, ხოლო თვითონ ბოლო სიმბოლო იქნება `K[K.size()-1]`.

ინგლისურენოვან ლიტერატურაში, ინდექსის ნაცვლად ხშირად იყენებენ ტერმინს "`offset`"-გადაწევა. თუ `offset` ნულია, ეს ნიშნავს რომ გვაინტერესებს პირველივე სიმბოლო, რითიც სტრიქონი იწყება. თუ `offset` ერთია, ე.ი. ერთის შემდეგზეა ლაპარაკი და ა.შ.

ინდექსების სწორად შერჩევა პროგრამისტის პასუხისმგებლობაა. საზღვრიდან გასულმა ინდექსმა შეიძლება გამოიწვიოს გაუთვალისწინებელი შედეგი. მაგალითად, წინა ფრაგმენტში რომ დავამატოთ

```
K [10] = 'A';
```

შეტყობინება, პროგრამის შესრულება ავარიულად შეწყდება, ხოლო `Debug` რეჯიმში შეცდომა ასე აიხსნება – `vector subscript out of range`.

სიმბოლოებზე წვდომისათვის `string` კლასი ასევე უზრუნველყოფს ფუნქციას `at` შემდეგი ფორმატით: `at( < ინდექსი > ).` მაგალითად, იგივე K სტრიქონისთვის

```
cout << K.at(4) << endl;
```

შეტყობინება დაბეჭდავს სიმბოლოს `t`, ხოლო

```
K.at(1) = 'p';
cout << K << endl;
```

ფრაგმენტი დაბეჭდავს გარდაქმნილ K სტრიქონს – `Sport`.

უნდა გვახსოვდეს, რომ პროგრამაში, რომელიც იყენებს `string` კლასს აუცილებელია `#include <string>` ბრძანების ჩართვა.

C++ –ის სტანდარტული ბიბლიოთეკის კიდევ ერთი მნიშვნელოვანი კლასია `vector`.

ვექტორი (`vector`) არის მეხსიერებაში თანმიმდევრობით განთავსებული ერთი და იგივე ტიპის მონაცემების (ერთი და იგივე სიგრძის მონაკვეთების) სიმრავლე. პროგრამაში შეგვიძლია შევქმნათ მთელი რიცხვების ვექტორი, ნამდვილი რიცხვების ვექტორი, სტრიქონების ვექტორი და ა.შ. ვექტორში განთავსებულ მონაცემებს მისი ელემენტები ეწოდება.

```
vector<int> V;
```

განაცხადის საფუძველზე იქმნება ცარიელი ვექტორი, რომლის ზომას გავიგებთ `size()` ფუნქციის გამოყენებით. მაგალითად,

```
cout << V.size() << endl;
```

ჩვენ შემთხვევაში დაბეჭდავს 0 –ს.

ვექტორის შევსება ხდება დინამიკურად, მონაცემის დამატებით მის ბოლოში. ელემენტის დამატება სრულდება `push_back( <ელემენტი> )` ფუნქციის მეშვეობით:

```
V.push_back(100);
```

შეტყობინება ჩასვავს ვექტორში მთელ რიცხვს 100, და ვექტორის ზომა გახდება 1 –ის ტოლი.

```
vector<double> Vec(2);
```

განაცხადი შექმნის ნამდვილ რიცხვთა 2 ელემენტიან ვექტორს და ჩასვავს გამოყოფილ მეხსიერებაში ნულებს (გაანულებს ელემენტებს).

მსგავსად სტრიქონისა, ვექტორის ელემენტებზე მიმართვისთვის ვიყენებთ ინდექსებს ან `at` ფუნქციას. მაგალითად, შემდეგი პროგრამის

```
#include <iostream>
#include <vector>
```

```
using namespace std;
int main(){
  vector<double> Vec(2);
  cout<<"size ="<<Vec.size()<<endl;
  cout<<"first element is: "<<Vec.at(0)<<endl;
  cout<<"second element is: "<<Vec.at(1)<<endl;
  Vec[0] = 0.025;
  Vec[1] = -87.65;
  cout<<"\nAfter assignment\nsize ="<<Vec.size()<<endl;
  cout<<"first element is: "<<Vec[0] <<endl;
  cout<<"second element is: "<<Vec[1]<<endl;
  return 0;
}
```

შესრულების შედეგია

```
size =2
first element is: 0
second element is: 0

After assignment
size =2
first element is: 0.025
second element is: -87.65
Press any key to continue . . .
```

მივაქციოთ ყურადღება `#include<vector>` ბრძანებას. პროგრამაში, რომელიც იყენებს `vector`–ს მისი ჩართვა აუცილებელია.

განვიხილოთ კიდევ ერთი მაგალითი, სადაც შევქმნით სტრიქონების ვექტორს, მასში ჩავწერთ კლავიატურიდან შემოტანილ რამოდენიმე სიტყვას, შემდეგ დავალაგებთ სიტყვებს ზრდადობით და ისე დავბეჭდავთ:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;
int main(){
  vector<string> Sentence;
  string word;
  while(cin>>word)
     Sentence.push_back(word);
  cout << "Number of words = " << Sentence.size() << endl;
  sort(Sentence.begin(), Sentence.end());
  int i =0;
  while(i < Sentence.size()){
     cout<<Sentence[i]<<" ";
     ++i;
  }
  cout << endl;
  return 0;
}
```

პროგრამის შედეგია

```
Paris London Washington Berlin
^Z
Number of words = 4
Berlin London Paris Washington
Press any key to continue . . .
```

სიტყვების დალაგებისთვის პროგრამაში ვისარგებლეთ `sort` ალგორითმით, ამისთვის დაგვჭირდა `#include<algorithm>` ბრძანების ჩართვა.

სტრიქონის (string) ობიექტის კონსტრუირება

ჩვენ ვნახეთ როგორ ხდებოდა რამდენიმე მარტივი ტიპის ცვლადის ინიციალიზება მინიჭების ოპერატორით ან კონსტრუქტორით. მარტივ ტიპებთან შედარებით, კლასს აქვს მრავალფეროვანი საშუალებები იმისთვის, რომ თავისი ახლადშექმნილი ობიექტების მნიშვნელობები განსაზღვროს. ეს კეთდება სპეციალური ფუნქციების, კონსტრუქტორების საშუალებით, რომლების გამოძახებაც ხდება ავტომატურად, განაცხადის გაკეთების მომენტში: თუ ახალშექმნილი ობიექტის სახელს მივუწერთ ფრჩხილებს (და არგუმენტსაც), ავტომატურად ამუშავდება ფუნქცია კონსტრუქტორი, რომელიც არგუმენტში მითითებულ მნიშვნელობას ჩაწერს ახალ ობიექტში.

კონკრეტულ მაგალითებზე, განვიხილოთ string კლასის რამდენიმე კონსტრუქტორი. ზოგიერთ მათგანს, რომლებიც შედარებით მოძველებულია და განკუთვნილია C ენის საშუალებებთან თავსებადობისთვის, არ განვიხილავთ.

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    // კონსტრუქტორი არგუმენტების გარეშე:
    string s1;
    s1 = " short line for testing";
    cout << "s1  is: " << s1 << endl;
    // ასლის კონსტრუქტორი
    string s2 (s1);
    cout << "s2  is: " << s2 << endl;
    // პირველი არგუმენტი სტრიქონია,
    // მეორე არგუმენტი წარმოადგენს სიმბოლოების რაოდენობას
    string s3 (s1,7);
    cout << "s3  is: " << s3 << endl;
    // კონსტრუქტორის არგუმენტები:
    // 1 - სტრიქონი
    // 2 - საწყისი პოზიცია
    // 3 - სიმბოლოების რაოდენობა
    string s4 (s1,7,4);
    cout << "s4  is: " << s4 << endl;
    // კონსტრუქტორის არგუმენტები:
    // 1 - სიმბოლოების რაოდენობა
    // 2 - თვითონ ჩასასმელი სიმბოლო
    string s5 (15,'*');
    cout << "s5  is: " << s5 << endl;
    // კონსტრუქტორის არგუმენტები:
    // 1 – begin საწყისი იტერატორი
    // 2 - end იტერატორი
    string s6 (s1.begin(),s1.end()-11);
    cout << "s6  is: " << s6 << endl;
    system("PAUSE");
    return 0;
}
```

OUTPUT:

```
s1  is:  short line for testing
s2  is:  short line for testing
s3  is: line for testing
s4  is: line
s5  is: ***************
s6  is:  short line
Press any key to continue . . .
```

თანამედროვე ტენდენციით, უპარამეტრო კონსტრუქტორების გამოყენება მხოლოდ იმ შემთხვევაშია მიზანშეწონილი, თუ იგი რაიმე მნიშვნელობას ჩაწერს ობიექტში გულისხმობის პრინციპით. ამიტომ,

```
string s1;
s1 = " short line for testing";
```

ფრაგმენტის გამოყენების ნაცვლად უმჯობესია თუ ინიციალიზაციას გავაკეთებთ მინიჭების შეტყობინებით:

```
string s1 = " short line for testing";
```

ან პარამეტრიანი კონსტრუქტორის გამოყენებით:

```
string s1 (" short line for testing");
```

უშუალოდ იტერატორებს ჩვენს კურსში არ განვიხილავთ, მაგრამ განვიხილავთ მასთან ახლოს მდგომ ცნებას - პოინტერს, რაც გარკვეულ წარმოდგენას მოგვცემს იტერატორის მოქმედების მექანიზმზე. გარკვეულ წარმოდგენას ამ საკითხზე ჩვენ შევიქმნით არაერთი მაგალითით, რაც განხილული იქნება ჩვენს კურსში.

მოქმედება სტრიქონზე

განვიხილოთ სტრიქონზე მოქმედების რამდენიმე საილუსტრაციო მაგალითი. უფრო სრულად, ეს მასალა შეგიძლიათ იხილოთ ელექტრონული სწავლების სისტემაში, ნაკვეთში „დამატებითი ინფორმაცია“.

1. სტრიქონების შეტანა და ბეჭდვა. **=, +, +=** ოპერატორების გამოყენება

```
#include <iostream>
#include <string>
using namespace std;
int main(){
   string Name, lastName;
   cout << "\nEnter your name and lastname\n";
   cin >> Name >> lastName;
   string a = "My name";  // სტრიქონის ინიციალიზება
   string b(" is ");      // სტრიქონის ინიციალიზება
   string fullName;
   fullName = a +  b;     // სტრიქონების შერწყმა და მინიჭება
   // სტრიქონთან += ოპერარის გამოყენება
   fullName += Name + " " + lastName;
   cout << fullName + "!" << endl;
   system("pause");
   return 0;
}
```

პროგრამა დაბეჭდავს

```
Enter your name and lastname
Luka Toreli
My name is Luka Toreli!
Press any key to continue . . .
```

2. როდის გამოვიყენოთ სტრიქონის კლავიატურიდან შეტანის ფუნქცია `getline` ?

```
#include <iostream>
#include <string>
using namespace std;
int main(){
   cout << "Enter text\n";
   string words;
```

```
    cin >> words; // კლავიატურიდან შეიტანეთ ტექსტი : I'm successful student
    cout << "\nEntered text is\n";
    cout << words << endl;
    system("pause");
    return 0;
}
```

პროგრამის შესრულების შედეგია:

```
Enter text
I'm successful student

Entered text is
I'm
Press any key to continue . . .
```

შეტანის შეტყობინება `cin >> words;` შეცვალეთ ფუნქციით `getline(cin, words);`

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    cout << "Enter text\n";
    string words;
    getline(cin, words);// შეიტანეთ ტექსტი : I'm successful student
    cout << "\nEntered text is\n";
    cout << words << endl;
    system("pause");
    return 0;
}
```

ახლა პროგრამის შესრულების შედეგია:

```
Enter text
I'm successful student

Entered text is
I'm successful student
Press any key to continue . . .
```

გააკეთეთ სწორი დასკვნები.

3. `append` ფუნქციის გამოყენება

```
#include <iostream>
#include <string>
using namespace std;
int main (){
     string str = "Nobody is perfect";
     string s   = ""; // ცარიელი სტრიქონი
     // s სტრიქონის ბოლოში str სტრიქონის 6 სიმბოლოს
     // დამატება, დაწყებული 0 პოზიციიდან
     s.append(str,0,6);
     cout << "s is : " << s << endl;
     // str-ის ქვესტრიქონის დამატება: მე -6 პოზიციიდან მის ბოლომდე
      // (s -ს დაემატება ' is perfect' )
     s.append(str.begin()+6, str.end());
     cout << "s is : " << s << endl;
     // სამი '!' დამატება
     s.append(3,'!');
     cout << "s is : " << s << endl;
     return 0;
}
```

გამოტანის ეკრანი:

```
s is : Nobody
s is : Nobody is perfect
s is : Nobody is perfect!!!
Press any key to continue . . .
```

4. ფუნქციები empty და erase

ა)

```
#include <iostream>
#include <string>
using namespace std;
int main (){
    string str = "*******";
    // სანამ str არ გახდება ცარიელი
    while ( ! str.empty() )
    {
      cout << str << endl;
    // წავშალოთ str -ის ბოლო სიმბოლო
      str.erase(str.end()-1);
    }
    cout << endl;
    return 0;
}
```

გამოტანის ეკრანი:

```
*******
******
*****
****
***
**
*
Press any key to continue . . .
```

ბ)

```
#include <iostream>
#include <string>
using namespace std;
int main (){
    string str, s;
    char ch = 'A';
    while (ch <= 'Z'){
       str.append(1,ch);
     ch++;
    }
    s = str;
    cout << "str is: " << str << endl;
    cout << "s   is: " << str << endl;

    // წავშალოთ str სტრიქონის პირველი 13 სიმბოლო
    str.erase(0,13);
    cout << "Erased range fron str : " << str << endl;

    // წავშალოთ str სტრიქონის ბოლო 13 სიმბოლო
    //( დაწყებული მე-14 სიმბოლოდან )
    str = s.erase(13,13);
    cout << "Erased range from s   : " << str << endl;
    // წავშალოთ s -ის ერთი სიმბოლო (ინდექსით 1)
    cout << endl << "Erase one, second character from s\n";
    s.erase(s.begin()+1);
    cout << "s     is: " << s << endl;
    // წავშალოთ s -იდან სიმბოლოების ჯგუფი
    // (პირველი ოთხი სიმბოლო)
    s.erase(s.begin(),s.begin()+4);
    cout << "s     is: " << s << endl;
    return 0;
}
```

გამოტანის ეკრანი:

```
str is: ABCDEFGHIJKLMNOPQRSTUVWXYZ
s   is: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Erased range fron str : NOPQRSTUVWXYZ
Erased range from s   : ABCDEFGHIJKLM

Erase one, second character from s
s     is: ACDEFGHIJKLM
s     is: FGHIJKLM
Press any key to continue . . .
```

6. ფუნქცია `find`.

```
#include <iostream>
#include <string>
using namespace std;
int main (){
    string  str("C++ is best language");
    int pos1, pos2;
    // str სტრიქონში ვეძებთ სიტყვას "best"
    pos1 = str.find ("best");
    cout << "Word best is found on position " << pos1+1
         << endl;

    pos2 = str.find ("best",pos1+1);
    cout << "Word best is found on position " << pos2+1
         << endl;

    // ვეძებთ სიმბოლო 'g' -ს პირველ შესვლას
    pos1 = str.find('g');
    cout << "First character 'g' found on position "
         << pos1 << endl;
    return 0;
}
```

გამოტანის ეკრანი:

```
Word best is found on position 8
Word best is found on position 0
First character 'g' found on position 15
Press any key to continue . . .
```

7. ალგორითმი `reverse`.

```
#include <iostream>
#include <string>
using namespace std;
int main (){
    string str = "Programming Language";
    reverse(str.begin(), str.end());
    cout << str << endl;
    system("pause");
    return 0;
}
```

გამოტანის ეკრანი:

```
egaugnaL gnimmargorP
Press any key to continue . . .
```

თავი 7: განმეორების **for** შეტყობინება. ფუნქციები **rand**(), **time**() და **srand**(). მოქმედება ვექტორზე

- **for** შეტყობინება
- **for, break** და **continue**
- **rand, time** და **srand** ფუნქციები
- მოქმედება ვექტორზე
- **do – while** შეტყობინება

for შეტყობინება

for წარმოადგენს განმეორების ერთ-ერთ შეტყობინებას. განმეორებათა რიცხვს თვითონ შეტყობინების სახე განსაზღვრავს. **for** შეტყობინებების ზოგადი სახე ასეთია:

```
for ( ინიციალიზება; პირობა; კორექცია )                    (1)
    შეტყობინება;
```

ინიციალიზების ნაწილში ხდება გამოყენებული ცვლადისთვის (ცვლადებისთვის) საწყისი მნიშვნელობის მინიჭება. ეს ნაწილი სრულდება მხოლოდ ერთხელ **for** –ის შესრულების დაწყებისთანავე. *პირობა* განსაზღვრავს, შესრულდეს თუ არა შეტყობინება (**for** –ის ტანი). ტანის შესრულების შემდეგ სრულდება *კორექციის* ნაწილი, სადაც ხდება ცვლადებისთვის მნიშვნელობების შეცვლა.

(1) ექვივალენტურია შემდეგის:

```
ინიციალიზება;
while (პირობის შემოწმება ) {
    შეტყობინება;
    კორექცია;
}
```

მაგალითად, შემდეგი ფრაგმენტი [15, 25] შუალედის რიცხვების შესაკრებად იყენებს განმეორების **while** შეტყობინებას.

```
int total = 0;        // ყველა რიცხვის ჯამი
int number = 15;      // რიცხვის მიმდინარე მნიშვნელობა
while(number <= 25 ){
     total += number;
     ++number;
}
cout << "The grand total is "<< total << endl;
```

გადავწეროთ ეს ფრაგმენტი განმეორების **for** შეტყობინების გამოყენებით

```
int total = 0;  // ყველა რიცხვის ჯამი
int number;     // რიცხვის მიმდინარე მნიშვნელობა
for(number = 15; number <= 25; ++number)
    total += number;
cout << "The grand total is "<< total << endl;
```

ან უფრო მოკლედაც

```
int total = 0;  // ყველა რიცხვის ჯამი
for(int number = 15; number <= 25; ++number)
    total += number;
cout << "The grand total is "<< total << endl;
```

for არ შესრულდება არცერთხელ, თუ *პირობა* თავიდანვე მცდარია, ხოლო თუ *პირობა* ყოველთვის ჭეშმარიტია, განმეორება არ დასრულდება. მაგალითად

```
for( int n=15; n>9 && 10-2*5; n++ )
   puts("ar shesruldeba");
```

```
for( int n=10; 0 || -100; n++ )
   puts("ar dasruldeba");
```

for, break და continue

for შეტყობინების განმეორება მთავრდება, როდესაც მისი პირობა ხდება მცდარი (0). თუმცა შეგვიძლია შევწყვიტოთ **for** –ის შესრულება ნებისმიერ ბიჯზე **break** შეტყობინების გამოყენებით.

continue შეტყობინება წყვეტს **for** –ის **მიმდინარე ბიჯს**, გამოტოვებს **continue**-ს მომდევნო შეტყობინებებს და თავიდან იწყებს **for** –ის **მომდევნო ბიჯის** შესრულებას.

break და **continue** შეტყობინებების გამოყენება **for** –თან ნაჩვენებია ქვემოთ მოცემულ მაგალითებში:

პროგრამის ფრაგმენტი	შედეგი

`int s = 30;`
`for(int n =7; true; --n){`
`  s -= n;`
`  if(n == 4) break;`
`}`
`cout<<"s = "<<s<<endl;`
	`/* s -ის მნიშვნელობა მცირდება 7 -ით, 6-ით, 5-ით და ბოლოს 4 -ით */`
	`s = 8`

`int n =5, p =1;`
`for( ; n<=11; n++){`
`  if( n%2 == 0) continue;`
`  p *=n;`
`}`
`cout<<"p = "<<p<<endl;`
	`/* p -ს მიენიჭება 5, 7, 9 და 11 (კენტი) რიცხვების ნამრავლი */`
	`p = 3465`

შემდეგი პროგრამა შეკრებს შემთხვევით რიცხვებს, ვიდრე ჯამი არ გახდება 10 –ის ჯერადი, ოღონდ ჯამში არ გაითვალისწინებს 13-ზე დამთავრებულ რიცხვებს:

```
#include <iostream>
using namespace std;

int main (){
    int a;       // შემთხვევითი რიცხვი
    int s =0;    // რიცხვების ჯამი

    for (  ; ;  )  // უსასრულო განმეორება, იგივეა რაც  while(true)
    {
       a = rand();
       if(a%100 == 13)
          continue;
       s += a;
       if(s%10 == 0)
          break;
    }
    cout<< "s = " << s <<'\n';
    return 0;
}
```

პროგრამის შესრულების შედეგია:

```
s = 554810
Press any key to continue . . .
```

ფუნქციები `rand()`,`time()` და `srand()`

C++ პროგრამაში შემთხვევითობის ელემენტის შეტანა შეიძლება `rand()` ფუნქციის გამოყენებით. `rand()` ფუნქციაზე განაცხადს შეიცავს `stdlib.h` თავსართი ფაილი. ფუნქცია აგენერირებს (ქმნის) მთელ ფსევდოშემთხვევით რიცხვს `[0, RAND_MAX]` შუალედიდან. მუდმივი `RAND_MAX` ასევე განსაზღვრულია `stdlib.h` ფაილში როგორც

```
#define RAND_MAX 0x7fff
```

სადაც 16-ობითი მნიშვნელობა `0x7fff` ათობითი `32 767`-ის ტოლია.

ფრაგმენტი

```
int i, k;
for(i = 1; i <= 100; i++) {
   k = rand();
   cout << k << '\t';
   if ( i%10 == 0) cout << endl;
}
```

10 სვეტად დაგვიბეჭდავს 100 შემთხვევით მთელ რიცხვს.

(როგორ უნდა შეიცვალოს `for(i = 0; i < 100; i++)` შეტყობინების ტანი, რომ მივიღოთ იგივე შედეგი?)

აქ `k = rand();` შეტყობინების შესრულება მიანიჭებს `k` ცვლადს რომელიღაცა მთელ რიცხვს `[0, RAND_MAX]` დიაპაზონიდან..

შეგვიძლია მოვახდინოთ შემთხვევითი `k` რიცხვის გენერირება `[0,m]` შეალედიდან, ამისათვის უნდა გამოვიყენოთ `rand()` ფუნქცია შემდეგი სახით: `k = rand()% (m+1);`

თუ კი შემთხვევითი მთელი `k` რიცხვის მიღება გვინდა `[n,m]` დიაპაზონიდან, უნდა დავწეროთ `k = rand()%(m+1-n) + n;`
მაგალითად,

```
#include <iostream>
using namespace std;
int main(){
    int i, k, n, m;
    puts("ShemoitaneT diapazonis sazgvrebi : ");
    cin >> n >> m;
    for(i =1; i <= 10; i++) {
        k = rand()%(m+1-n) + n;
        cout << k << ' ';
    }
    cout << endl;
    return 0;
}
```

პროგრამის შესრულების შესაძლო შედეგია

```
ShemoitaneT diapazonis sazgvrebi :
-5 30
0 30 29 -1 12 23 25 13 29 15
Press any key to continue . . .
```

პროგრამის რამდენჯერმე გაშვება გვიჩვენებს, რომ ყოველთვის ვღებულობთ ზუსტად იგივე რიცხვთა მიმდევრობას. საქმე ისაა, რომ `rand()` ფუნქცია ახდენს ე.წ. ფსევდოშემთხვევითი რიცხვების გენერირებას, მაგრამ `rand()` ფუნქციის ალგორითმი არის დეტერმინირებული იმ აზრით, რომ ამ ფუნქციის ბევრჯერ გამოყენება გვაძლევს ფსევდოშემთხვევითი რიცხვების ერთი და იგივე სერიას:

41 18467 6334 26500 19169 15724 11478 29358 26962 24464 . . . (2)

`rand()` ფუნქციის ამ თვისებით სარგებლობენ პროგრამის გამართვის პროცესში: იმის დასადგენად თუ როგორ აისახება შედეგებზე პროგრამაში შეტანილი ცვლილებები აუცილებელია პროგრამა შესრულდეს ერთი და იმავე მონაცემებზე.

გამართვის პროცესის დასრულების შემდეგ უნდა მოხდეს პროგრამის მოდიფიცირება, რომ სხვადასხვა დროს მისი გაშვებისას მიიღებოდეს შემთხვევით რიცხვთა სხვადასხვა მიმდევრობა. ამას *რანდომიზაციას* უწოდებენ. რანდომიზაციას უზრუნველყოფს სტანდარტული ბიბლიოთეკის `srand()` ფუნქცია შემდეგი ფორმატით:

`srand(` *პარამეტრი* `),`

სადაც *პარამეტრი* **`unsigned`** ტიპის ცვლადი ან მუდმივია. *პარამეტრის* სხვადასხვა მნიშვნელობისათვის იგივე პროგრამაში გამოყენებული `rand()` მოახდენს განსხვავებული ფსევდოშემთხვევითი რიცხვების მიმდევრობების გენერირებას.

`srand` აღწერილია `stdlib.h` ფაილში.

პარამეტრის სახით, ჩვეულებრივ, იყენებენ `time(NULL)`ფუნქციის შედეგს:

```
srand( time(NULL) );
```

`time(NULL)`-ის შედეგი წარმოადგენს მიმდინარე კალენდარულ დროს, გაზომილს წამებში. დროის ათვლა მიმდინარეობს 1970 წლის 1 იანვრიდან.

საინტერესოა ფუნქციის სათაურიც. გავრცელებული ვერსიით, `srand` ნიშნავს შემთხვევითობის დათესვას (`seed rand`), თუმცა, იგი შეიძლება ნიშნავდეს წანაცვლებულ შემთხვევითობას (`shifted rand`) , რადგან რა არგუმენტითაც არ უნდა გავუშვათ ფუნქცია `srand()`, `rand()` ფუნქციის მომდევნო შედეგები მაინც მოგვცემს (2) მიმდევრობას, დაწყებულს რაღაც მომენტიდან (მაგალითად, მე-100-იდან დაწყებული, ან 2 345-ედან და ა.შ.).

`time` ფუნქციით სარგებლობისათვის საჭიროა `#include <ctime>` ბრძანების ჩართვა პროგრამაში.

შემდეგი პროგრამა ქმნის 15 შემთხვევით რიცხვს [-7, 15] შუალედიდან და ბეჭდავს 5 სვეტად.

```
#include <iostream>
#include <ctime>
using namespace std;
int main(){
   int i, k;
   srand( time(NULL) );
   for( i=1; i<=10; i++){
      k = rand()% 23 - 7;
      cout << k << '\t';
      if(i % 5 == 0) cout << endl;
   }
   system("PAUSE");
   return 0;
}
```

პროგრამის შესრულების შედეგები:

ტესტი # 1					ტესტი # 2				
-7	5	1	15	14	10	-6	-3	2	13
13	5	-3	2	5	8	8	-3	-4	9
14	-2	2	9	11	9	-2	2	4	-2
Press any key to continue . . .					Press any key to continue . . .				

შემთხვევითი რიცხვების გენერირება ხშირად გამოიყენება ვექტორებთან მუშაობის დროს – ვექტორის შემთხვევითი მნიშვნელობებით შესავსებად. ეს ხერხი დიდი ზომის ვექტორებთან მომუშავე პროგრამების გამართვის კარგი საშუალებაა.

მოქმედება ვექტორზე

ვექტორში განსათავსებელი საწყისი მონაცემები პროგრამას შეიძლება მიეწოდებოდეს კლავიატურიდან ან ფაილიდან. ვექტორის ხიბლი ის არის, რომ არაა აუცილებელი წინასწარ ვიცოდეთ მასში ჩასაწერი მონაცემების რაოდენობა.

მაგალითად, შემდეგი პროგრამა შეიტანს კლავიატურიდან სტუდენტების გვარებს და ჩაწერს სათანადო ტიპის ვექტორში. შემდეგ დაალაგებს გვარებს ანბანის მიხედვით და ისე დაბეჭდავს. გვარების შეტანა უნდა დავასრულოთ `Ctrl+z` კლავიშების კომბინაციას აკრეფით:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;
int main(){
    vector<string> Surnames;
    string name;
    cout << "Enter Students' Surnames :\n";
    while( cin >> name )
       Surnames.push_back(name);
    sort(Surnames.begin(), Surnames.end());
    cout << "In alphabetical order:\n";
    for(int i=0; i < Surnames.size(); ++i)
        cout << Surnames[i] << endl;
    system("PAUSE");
    return 0;
}
```

პროგრამის შესრულების შედეგია

```
Enter Students' Surnames :
Kiladze  Dondua  Iashvili  Aroshidze  Shengelia
^Z
In alphabetical order:
Aroshidze
Dondua
Iashvili
Kiladze
Shengelia
Press any key to continue . . .
```

პროგრამაში გამოვიყენეთ ალგორითმი `sort(Surnames.begin(),Surnames.end());` რომელმაც დაალაგა `Surnames` ვექტორის ელემენტები ზრდადობით. `sort` ალგორითმის გამოყენებისტვის პროგრამაში უნდა ჩავრთოთ `#include <algorithm>` ბრძანება.

შემდეგ პროგრამას შეჰყავს კლავიატურიდან ნამდვილი რიცხვები (მანამ, სანამ არ შევიტანთ სიმბოლოს ან `Ctrl+z` კლავიშების კომბინაციას) და ათავსებს სათანადო ტიპის ვექტორში. შემდეგ პოულობს და ბეჭდავს ვექტორის უდიდესი და უმცირესი ელემენტების ჯამს.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main(){
```

```
    vector<double> V;
    double real;
    cout << "Enter real numbers, "
            "at the end enter symbol" << endl;
    while( cin >> real )
       V.push_back( real );
    sort( V.rbegin(), V.rend() );
    double sum;
    sum = V[0] + V[ V.size()- 1 ];
    cout<<"Sum of the maximum and minimum is "
        << sum <<endl;
    system("PAUSE");
    return 0;
}
```

პროგრამის შესრულების შედეგია

```
Enter real numbers, at the end enter symbol
1.2 0.3 12.25 -11.25 3.0 20.5 -10 -5.5 0 2.125
Sum of the maximum and minimum is 9.25
Press any key to continue . . .
```

აქ `sort` ალგორითმი გამოყენებული გვაქვს `sort(V.rbegin(),V.rend());` სახით, რაც უზრუნველყოფს V ვექტორის ელემენტების დალაგებას კლებადობით. ამაში შეგვიძლია დავრწმუნდეთ, თუ დავბეჭდავთ ვექტორს დალაგების შემდეგ:

```
for(int i=0; i < V.size(); i++)
    cout << V[i] << ' ';
```

`vector` კლასში განსაზღვრულია ბევრი სასარგებლო ფუნქცია და ასევე შექმნილია ბევრი სტანდარტული ალგორითმი, რომელიც მუშაობს ვექტორთან. განვიხილოთ რამდენიმე მათგანი:

1. ალგორითმი `count`.

პროგრამა ითვლის, რამდენჯერ გვხვდება ვექტორში რიცხვი 23. ვექტორში 20 შემთხვევითი რიცხვია [10, 30] შუალედიდან.

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main(){
   int number;
   vector<int> V;
   const int N = 20;
   for(int counter = 0;counter < 20;counter++)
      V.push_back( rand()%21 + 10 );
   for(int i=0; i<V.size(); ++i)
       cout<<V[i]<<' ';
   cout<<endl;
   int quantity;
   quantity = count( V.begin(), V.end(), 23);
   cout<<quantity<<endl;
   system("PAUSE");
   return 0;}
```

პროგრამის შესრულების შედეგია

```
30 18 23 29 27 26 22 10 29 30 24 15 23 16 17 18 23 24 28 28
3
Press any key to continue . . .
```

2. ალგორითმი count_if.

პროგრამა ითვლის რამდენი უარყოფითი რიცხვია number.dat ფაილში.

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;
bool isNegative(double x)  // ფუნქცია–პრედიკატზე განაცხადი
{                          // და მისი განსაზღვრა
   return x < 0;
}
int main(){
   double number;
   vector<double> Vec;
   ifstream fin("number.dat");
   while( fin >> number )
      Vec.push_back( number );
   int quantity;
   // აქ გამოიყენება ფუნქცია isNegative
   quantity = count_if( Vec.begin(), Vec.end(), isNegative);
   cout<<"In file is "<<quantity
       <<" negative numbers"<<endl;
   system("PAUSE");
   return 0;
}
```

პროგრამის შესრულების შედეგია

numbers.dat ფაილი	გამოტანის ეკრანი
10 -2 3 -34 -91 67 8 19 -100 511 0 -13	In file is 5 negative numbers Press any key to continue . . .

3. ალგორითმი insert.

data.txt ფაილი შეიცავს სიტყვებს. ჩავამატოთ მის დასაწყისში სიტყვა "header"

```
#include <iostream>
#include <fstream>
#include <vector>
#include <string>
using namespace std;
int main(){
   vector<string> Words;
   string w;
   ifstream fin("data.txt");
   while( fin >> w )
      Words.push_back( w );
   Words.insert(Words.begin(),"header");
   fin.close();
   ofstream fout("data.txt");
   for( int i=0; i < Words.size(); i++ )
       fout << Words[i] <<' ';
   return 0;
}
```

პროგრამის შესრულების შედეგია

data.txt ფაილი	data.txt ფაილი პროგრამის შესრულების შემდეგ
do for while int double	header do for while int double

4. ფუნქცია pop_back().

ვექტორში ნამდვილი რიცხვებია. წაშალეთ ვექტორის ბოლო ელემენტი. რიცხვები ვექტორში შეიტანეთ reals.info ფაილიდან.

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
void printVector(vector<double> );
int main(){
   vector<double> D;
   double number;
   ifstream ifs("reals.info");
   while( ifs >> number )
      D.push_back( number );
   D.pop_back();
   printVector(D);
   system("PAUSE");
   return 0;
}
void printVector(vector<double> x){
   for(int i=0; i<x.size(); i++)
       cout<<x[i]<<' ';
   cout<<endl;
}
```

პროგრამის შესრულების შედეგია

reals.info ფაილი	გამოტანის ეკრანი
-0.01 5.25 43.75 1.5 -0.823	-0.01 5.25 43.75 1.5 Press any key to continue . . .

5. ალგორითმი erase.

reals.dat ფაილი შეიცავს ნამდვილ რიცხვებს. ჩაწერეთ რიცხვები სათანადო სპეციფიკაციის ვექტორში და შემდეგ წაშალეთ კლავიატურიდან შემოტანილი ნამდვილი რიცხვის ტოლი ვექტორის ელემენტი. თუ ასეთი ელემენტი ვექტორში არ არის, დაბეჭდეთ შესაბამისი გზავნილი.

```
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
void printVector(vector<double> );
int main(){
   vector<double> v;
   double number;
   ifstream fin("reals.dat");
   while(fin >> number)
      v.push_back(number);
   printVector(v);
```

```
    double del;
    cout<<"Enter number to delete: ";
    cin>>del;
    int i;
    for(i=0; i<v.size(); i++)
       if(v[i] == del) break;
    if(i == v.size())
       cout<<"vector not contains this number\n";
    else{
       v.erase( v.begin()+ i );
       printVector(v);
    }
    system("PAUSE");
    return 0;
}
void printVector(vector<double> x){
    for(int i=0; i<x.size(); i++)
        cout<<x[i]<<' ';
    cout<<endl;
}
```

პროგრამის შესრულების შედეგია

reals.dat ფაილი	გამოტანის ეკრანი
1.2 -0.03 2.5 0.056 12.45	1.2 -0.03 2.5 0.056 12.45 Enter number to delete: 2.5 1.2 -0.03 0.056 12.45 Press any key to continue . . .

დამატებითი ინფორმაცია `string` და `vector` კლასების ფუნქციების და ალგორითმების შესახებ შეგიძლიათ იხილოთ ინტერნეტ–მისამართზე http://cplusplus.com/

განმეორების შეტყობინება **do – while**

do – while განმეორების შეტყობინების ზოგადი ფორმაა:

```
do
{
   შეტყობინება;
}
while (პირობა);
```

do – while შეტყობინება განსხვავებით **while** შეტყობინებისგან იწყებს განმეორებას უპირობოთ, შეასრულებს განმეორების ერთ ბიჯს მაინც, ხოლო დანარჩენი ბიჯების შესრულება უკვე დამოკიდებულია პირობაზე: თუ პირობა ჭეშმარიტია, განმეორება გრძელდება, თუ კი მცდარია – წყდება. **do – while** -თან **break;** და **continue;** შეტყობინებების გამოყენების წესი იგივეა, რაც **while** –თან და **for**–თან.

როდის სჯობს **do – while** შეტყობინების გამოყენება? ამას ამოცანის პირობა გვიკარნახებს – თუ დარწმუნებული ვართ, რომ განმეორება ერთხელ მაინც უნდა შესრულდეს, შეგვიძლია გავაფორმოთ იგი **do – while** სახით.

თავი 8: ცვლადების ხილვადობის არე და ფუნქციები. ფუნქციის პარამეტრები.

- ხილვადობის არე და მეხსიერების კლასი, სტეკი
- **void** ტიპის ფუნქცია, უპარამეტრო ფუნქცია
- ფუნქციისთვის არგუმენტის გადაცემა მნიშვნელობით (`Pass-by-value`)
- არგუმენტის გადაცემა მისამართით (`Pass-by-reference` , `Pass-by-const reference`)

ხილვადობის არე და მეხსიერების კლასი, სტეკი

თითოეულ ცვლადს ორი ატრიბუტი აქვს: ხილვადობის არე და მეხსიერების კლასი. ცვლადის ხილვადობის არე არის პროგრამის ის ნაწილი, სადაც ცვლადის სახელი (იდენტიფიკატორი) შეიძლება გამოვიყენოთ ცვლადის მნიშვნელობაზე წვდომისათვის. თუ ცვლადის მნიშვნელობა მისაწვდომია პროგრამის ნებისმიერ ნაწილში, მაშინ მას **გლობალური** ეწოდება. მისგან განსხვავებით, **ლოკალური** ცვლადის ხილვადობის არე არის ის ბლოკი, რომელშიც კეთდება მასზე განაცხადი და ამ ბლოკის გარეთ ეს ცვლადი არაა ხელმისაწვდომი. ბლოკი არის კოდის ნაწილი, რომელიც შემოსაზღვრულია ფიგურული ფრჩხილებით (**{}**). შემდეგ პროგრამაში ნაჩვენებია განსხვავება ლოკალურსა და გლობალურ ცვლადებს შორის.

```
#include <iostream>
using namespace std;
int global;      // გლობალური ცვლადი
int main(){
   int local;    // ლოკალური ცვლადი
   global = 10; // აქ გლობალურ ცვლადზე წვდომა გვაქვს
   local = 19;  // ასევე ლოკალურ ცვლადზეც

   {// გაიხსნა ბლოკი
         // განაცხადი ბლოკის ლოკალურ ცვლადზე
         int block_local = 100;
         // აქ წვდომა გვაქვს სამივე ცვლადზე
         cout<< global + local + block_local<<endl;
   }// ბლოკი დაიხურა

   // აქ შეგვიძლია მივმართოთ global და local ცვლადებს
   cout<< global + local<<endl;
   cout<< block_local<<endl; // შეცდომა! აქ ბლოკში აღწერილ ცვლადზე
                             // წვდომა აღარა გვაქვს
   return 0;
}
```

ჩვენ შეგვიძლია რომელიმე შიდა ბლოკში შემოვიღოთ იგივე სახელის მქონე ლოკალური ცვლადი. ამ ბლოკის შიგნით ამ სახელით მხოლოდ ლოკალური ცვლადი იქნება ხილვადი. ამ ბლოკის გარეთ კი იგივე სახელით კვლავ გლობალური ცვლადი ხდება ხილვადი.

გაუგებრობის თავიდან ასაცილებლად რეკომენდებულია, რომ სხვადასხვა ხილვადობის ცვლადებს ერთიდაიგივე სახელი არ დავარქვათ.

მეხსიერების კლასის მიხედვით ცვლადები შეიძლება იყოს ორიდან ერთი: მუდმივად არსებული ან დროებითი. გლობალური ცვლადები ყოველთვის მუდმივად არსებული ცვლადებია. ისინი იქმნება და ინიციალდება პროგრამის შესრულების დაწყებამდე. დროებითი ცვლადები მეხსიერების იმ ნაწილში თავსდება, რომელსაც სტეკი (`stack`) ეწოდება. მრავალი დროებითი ცვლადის გამოყენებამ შეიძლება "სტეკის გადავსების" შეცდომა (`stack overflow`) გამოიწვიოს.

სტეკში დროებითი ცვლადის მიერ დაკავებული ადგილი თავისუფლდება შესაბამისი ბლოკის დასრულების შემდეგ. ბლოკში ყოველი შესვლისას დროებით ცვლადს სტეკში ხელახლა გამოეყოფა ადგილი.

სტეკის გადავსება სხვადასხვა მიზეზით შეიძლება მოხდეს. ერთ შემთხვევაში მას იწვევს მრავალი დროებითი ცვლადი ან დიდი დროებითი ვექტორი, სხვა შემთხვევაში შესაძლოა რომელიმე ფუნქცია დაუსრულებლად იძახებდეს თავის თავს (მაგალითად, ასე ხდება ცუდად ორგანიზებული რეკურსიის შემთხვევაში).

დროებითი ცვლადები გვხვდება როგორც ბლოკებში, ასევე ფუნქციებში. განსაკუთრებით საინტერესო სიტუაციები იქმნება, როდესაც გვხვდება ერთმანეთში ჩაწყობილი ბლოკები, ან, რაც უფრო გავრცელებული შემთხვევაა, ერთი ფუნქცია იძახებს მეორეს, ის სხვას და ა. შ. ამ შემთხვევაში სტეკში ხდება ახალ-ახალი ინფორმაციის ჩატვირთვა (push). როცა რომელიმე ბლოკი დაიხურება ან რომელიმე ფუნქცია დაასრულებს მუშაობას, სტეკიდან "გაქრება" (pop) შესაბამისი ცვლადები და აგრეთვე ზოგიერთი სხვა ინფორმაცია, რასაც მთლიანობაში უწოდებენ სტეკის კადრს.

თავისი მუშაობის პრინციპის გამო, სტეკს ხშირად ადარებენ ერთმანეთზე დადგმული ერთი ზომის თეფშების წყებას, რომელშიც ახალი თეფშის დამატება შესაძლებელია ზემოდან, და რომელიმე თეფშის აღებაც მხოლოდ იმ შემთხვევაში შეიძლება, თუ ის ზემოდან დევს. სტეკშიც ბოლოს დამატებული ინფორმაცია თავსდება სტეკის ბოლოში და წაშლაც ხდება ბოლოდან. მუშაობის ასეთ პრინციპს ეწოდება LIFO(last-in,first-out) - ანუ ბოლოს მოსული პირველი მიდის.

სტეკის სიგრძე დამოკიდებულია ოპერაციულ სისტემაზე და კომპილერზე, რომელსაც თქვენ იყენებთ.

ლოკალური ცვლადი არის დროებითი გარდა იმ შემთხვევისა, როცა მის განაცხადში მითითებულია **static**.

თუ **static** გამოიყენება გლობალურ ცვლადზე განაცხადში, მაშინ მას აქვს სრულიად სხვა დანიშნულება. ის მიუთითებს, რომ ცვლადი არსებობს მხოლოდ მიმდინარე ფაილში.

შემდეგი მაგალითი გვიჩვენებს განსხვავებას მუდმივად არსებულ და დროებით ლოკალურ ცვლადებს შორის. სიმარტივისთვის, temporary არის დროებითი ცვლადი, ხოლო permanent არის მუდმივად არსებული. ორივე ცვლადზე განაცხადი კეთდება **for** შეტყობინების ბლოკის დასაწყისში. **for** შეტყობინების ტანის ყოველი განმეორებისას ხდება temporary ცვლადის ხელახალი შექმნა და ინიციალიზება, ხოლო სტატიკური ცვლადი permanent იქმნება და მისი ინიციალიზება ხდება მხოლოდ ერთხელ **for** -ის პირველი ბიჯის დროს.

```
#include <iostream>
using namespace std;
int main(){
   int counter;      // განმეორების მთვლელი
   for( counter=0; counter<3; ++counter ){
     int temporary = 1;          // დროებითი ცვლადი
     static int permanent = 1;  // მუდმივი ცვლადი
     cout<<"Temprary "<< temporary
         <<" Permanent "<< permanent<<endl;
     ++temporary;
     ++permanent;
   }
   return 0;
}
```

პროგრამის შედეგია:

```
Temprary 1 Permanent 1
Temprary 1 Permanent 2
Temprary 1 Permanent 3
Press any key to continue . . .
```

დროებით ცვლადებს ზოგჯერ ავტომატურ ცვლადებს უწოდებენ, რადგან მათთვის მეხსიერების გამოყოფა ხდება ავტომატურად. სპეციფიკატორი **auto** შეიძლება გამოყენებული იყოს იმის მაჩვენებლად, რომ ცვლადი დროებითია. პრაქტიკაში ეს სპეციფიკატორი არ გამოიყენება.

შემდეგი ცხრილით მოცემულია განაცხადის გაკეთების სხვადასხვა შემთხვევები

განაცხადი გაკეთებულია	მოქმედების არე	კლასი	ინიციალიზება
ყველა ბლოკის გარეთ	გლობალური	მუდმივი	ერთხელ
static, ყველა ბლოკის გარეთ	გლობალური	მუდმივი	ერთხელ
ბლოკის შიგნით	ლოკალური	დროებითი	ბლოკში ყოველი შესვლისას
static, ბლოკის შიგნით	ლოკალური	მუდმივი	ერთხელ

ისევ ფუნქციის შესახებ

აქ თავი მოვუყაროთ ყველა იმ ცოდნას ფუნქციის შესახებ, რაც უკვე გაგვაჩნია.

ფუნქციის იმპლემენტაციის ფორმატი შემდეგია:

<*დასაბრუნებელი ტიპი*> ფუნქციის სახელი (<*პარამეტრების სია*>)
{
განაცხადი ცვლადებზე
შეტყობინებები
}

<დასაბრუნებელი ტიპი> განმარტავს ფუნქციით შექმნილი მნიშვნელობის ტიპს. რადგანაც ფუნქცია განკუთვნილია კონკრეტული ამოცანის ამოსახსნელად, მისი შესრულების შედეგი შეიძლება იყოს რიცხვი, სიმბოლო, სტრიქონი და ა.შ. სწორედ ეს არის ფუნქციის დასაბრუნებელი მნიშვნელობა. როდესაც საქმე მათემატიკურ ფუნქციებს ეხება, დასაბრუნებელი ტიპი იგივეა რაც ფუნქციის მნიშვნელობათა სიმრავლე.

ფუნქციის სახელის საშუალებით კეთდება ფუნქციაზე განაცხადი (მისი პროტოტიპის მოცემა), ფუნქციის განსაზღვრა (შესასრულებელი მოქმედებების აღწერა) და ფუნქციაზე მიმართვა (მისი გამოძახება). კომპილერისთვის ნებისმიერ ობიექტზე განაცხადი ნიშნავს, რომ ეს ობიექტი გაიგივდება მეხსიერების გარკვეულ მონაკვეთთან. ფუნქციების შემთხვევაში, ფუნქციის სახელი გაიგივდება ამ მონაკვეთის დასაწყისთან, რაც ძალიან მოსახერხებელია თუ დავაპირებთ ფუნქციის არგუმენტად სხვა ფუნქციის გამოყენებას. ეს მიდგომა ძალიან განვითარებულია ბევრ თანამედროვე პროგრამირების ენაში. მაგალითად, შემდეგი მარტივი ფუნქციისთვის შეგვიძლია ვნახოთ, რომ მისი მისამართი მართლაც იგივეა რაც მისი სახელი:

```
int f(int x){
    return x*x;
}
int main(){
   cout<<int(&f)<<endl;
   cout<<int(f)<<endl;
   return 0;
}
```

<პარამეტრების სია> წარმოადგენს ფუნქციის პარამეტრების ჩამონათვალს მათი ტიპების მითითებით. ჩამონათვალში პარამეტრები გამოიყოფა მძიმით, ამასთან ტიპი უნდა მიეთითებოდეს ყოველ პარამეტრს. პარამეტრების საშუალებით ფუნქციას გადაეცემა მისთვის საჭირო საწყისი მონაცემები (`main`-იდან ან სხვა ფუნქციიდან). დასაშვებია, რომ პარამეტრების სია იყოს ცარიელი. მაგალითად, მათემატიკური ფუნქციებისთვის, პარამეტრი იგივეა რაც არგუმენტი და თანამედროვე ტენდენციით პროგრამირებაშიც სულ უფრო ხშირად ხდება ტერმინ არგუმენტის გამოყენება პარამეტრის ნაცვლად.

ფიგურულ ფრჩხილებში მოთავსებულია ამოსახსნელი ამოცანის რეალიზებისთვის საჭირო ცვლადებზე განაცხადები და შესრულებადი შეტყობინებები - ფუნქციის ტანი.

ფუნქციის ტანში აღწერილი ცვლადები და ფუნქციის პარამეტრები წარმოადგენენ მის *ლოკალურ* ცვლადებს, რომლებიც ხილვადი (ცნობილი, წვდომადი) არიან მხოლოდ ფუნქციაში.

ვთქვათ, პროგრამაში გვჭირდება ფუნქცია, რომელიც ითვლის ორი მთელი რიცხვის საშუალო არითმეტიკულს. ასეთ ფუნქციაზე განაცხადს ექნება სახე

```
double Average(int x, int y);
```

სადაც

`double` - ფუნქციის ტიპია, ანუ მისი დასაბრუნებელი მნიშვნელობის ტიპი;

`Average` - ფუნქციის სახელია;

`x` და `y` - ფუნქციის პარამეტრებია.

ფუნქციაზე განაცხადს ასევე პროტოტიპი ეწოდება. პროტოტიპის შემდეგ უნდა დავსვათ წერტილ-მძიმე (`;`), წინააღმდეგ შემთხვევაში მივიღებთ კომპილაციის შეცდომას.

პროტოტიპში პარამეტრების სახელების მითითება აუცილებელი არ არის - კომპილერი ამ სახელებს უგულვებელყოფს. მაგალითად,

```
double Average(int , int );
```

პროტოტიპის სწორი ჩანაწერია.

ფუნქციის გააქტიურება, ანუ მასში დაპროგრამებული კოდის შესრულების დაწყება, ხდება მისი გამოძახების გზით. მაგალითად, `Average` ფუნქციის გამოძახების ფრაგმენტი შესაძლოა ასე გამოიყურებოდეს:

```
int a = 287, b = -143;
double answer = Average(a, b);
```

სადაც

`a` და `b` -ს ფუნქციის *არგუმენტებია.*

პროტოტიპის საშუალებით კომპილერი ამოწმებს, რამდენად სწორია ფუნქციაზე მიმართვა. ფუნქციის გამოძახების დროს მისი არგუმენტები (`a` და `b`) უნდა შეესაბამებოდეს პროტოტიპში აღწერილ პარამეტრებს (`x` და `y`): არგუმენტების რაოდენობა, ტიპები და მათი რიგი უნდა ემთხვეოდეს პარამეტრების რიცხვს, თითოეულის ტიპსა და რიგს. წინააღმდეგ შემთხვევაში ფიქსირდება კომპილაციის შეცდომა.

ასევე უნდა ემთხვეოდეს პროტოტიპში მითითებული ფუნქციის ტიპი და `answer` ცვლადის ტიპი. შეუსაბამობის შემთხვევაში კომპილერი ავტომატურად გარდაქმნის ფუნქციის დასაბრუნებელი მნიშვნელობის ტიპს `answer` ცვლადის ტიპზე (რამაც შეიძლება გამოიწვიოს ინფორმაციის დაკარგვა) ან მივიღებთ კომპილაციის შეცდომას: `there is no acceptable conversion`.

სინტაქსური შეცდომა ფიქსირდება მაშინაც, როდესაც შეუსაბამობაა ფუნქციის პროტოტიპსა და ფუნქციის იმპლემენტაციის სათაურს შორის. ორივეში უნდა ემთხვევოდეს როგორც დასაბრუნებელი ტიპი, ისე პარამეტრების რიცხვი და მათი ტიპები.

`Average` ფუნქციის იმპლემენტაციის შესაძლო სახეა:

```
double Average(int x, int y){
   double t; //  x, y და t - ფუნქციის ლოკალური ცვლადებია
   t = (x + y)/ 2. ;
   return t;
}
```

`Average` ფუნქციის ტანის ბოლო ინსტრუქცია

```
return t;
```

წარმოადგენს `return` შეტყობინებას.

თუ ფუნქციამ უნდა დააბრუნოს შედეგი, მაშინ მის ტანში სრულდება შეტყობინება

```
return ( გამოსახულება );
```

რომელშიც ჯერ გამოითვლება გამოსახულების მნიშვნელობა და შემდეგ `return` დააბრუნებს მას ფუნქციაზე მიმართვის წერტილში.

ფუნქცია შეიძლება შეიცავდეს ერთზე მეტ `return` –ს. მაგალითად, როგორც ადრე განხილულ ფუნქციაში `sign`, რომელიც ყოველ ნამდვილ რიცხვს შეუსაბამებდა მის ნიშანს:

```
int sign (double x){
   if( x > 0 )
       return 1;
   if( x < 0 )
       return -1;
   return 0;
}
```

ფუნქციაში სამი `return` შეტყობინებაა, რომლიდანაც სრულდება ერთ–ერთი.

აღვნიშნოთ, რომ `Average` ფუნქცია შეიძლება ჩაიწეროს უფრო მოკლედაც, ლოკალური `t` ცვლადის შემოღების გარეშე:

```
double Average(int x, int y){
   return (x + y)/ 2. ;
}
```

`void` ტიპის ფუნქცია

C++ -ში, განხვავებით მათემატიკისგან, აქტიურად გამოიყენება ისეთი ფუნქციები, რომლებიც არ აბრუნებენ მნიშვნელობას (მაგალითად, ფუნქციის დანიშნულებაა გზავნილის ბეჭდვა, ხატვა, ვექტორში მონაცემების მოთავსება, ვექტორის ელემენტების ბეჭდვა და სხვა). ასეთი ფუნქციები `void` ტიპისაა. მაგალითად, შემდეგი ფუნქცია განკუთვნილია მთელ რიცხვთა ვექტორის ელემენტების დასაბეჭდად:

```
void printVector(vector<int> x)
{
   for(int i=0; i< x.size(); i++)
     cout<<x[i]<<'\t';
   cout<<endl;
}
```

შემდეგ პროგრამაში ნაჩვენებია `printVector` ფუნქციის გამოყენება (გამოძახება). გზად, ნაჩვენებია თუ როგორ ჩავამატოთ რამდენიმე ერთნაირი ელემენტი მოცემულ ადგილზე.

```
#include <iostream>
#include <vector>
using namespace std;
void printVector( vector<int> );
int main()
{
   vector<int> V;
   for(int count=1; count<=10; count++)
      V.push_back( rand() );
   printVector( V );
   V.insert(V.begin() + V.size()/2, 2, 300);
   printVector( V );
   return 0;
}
void printVector( vector<int> x )
{
    for(int i=0; i<x.size(); i++)
       cout<<x[i]<<' ';
    cout<<endl;
}
```

პროგრამის შედეგია:

```
41 18467 6334 26500 19169 15724 11478 29358 26962 24464
41 18467 6334 26500 19169 300 300 15724 11478 29358 26962 24464
Press any key to continue . . .
```

`void` ტიპის ფუნქციის ტანში `return` შეტყობინება ან არ გამოიყენება, ან (თუ საჭიროა ფუნქციის შესრულების შეწყვეტა რაიმე პირობის მიხედვით) გამოიყენება ფორმატით `return;`

ორივე შემთხვევაში ფუნქციის დასაბრუნებელი მნიშვნელობა განსაზღვრული არ არის.

უპარამეტრო ფუნქცია

განხვავებით მათემატიკისგან, უამრავ შემთხვევაში C++ -ის ფუნქცია არ საჭიროებს საწყის მონაცემებს (მას არ გადაეცემა არგუმენტები). ამ ფაქტის აღსანიშნავად ენის სინტაქსი ითვალისწინებს ორ შესაძლებლობას:

- პარამეტრების ცარიელი სია.
  მაგალითად,

  ```
  float Calculation();
  ```

  განაცხადი აღნიშნავს, რომ ფუნქცია `Calculation` დააბრუნებს ნამდვილ რიცხვს, მას არა აქვს პარამეტრები და გამოძახების დროს არ გადაეცემა არგუმენტები.

- მომსახურე `void` სიტყვის გამოყენება.
  `Calculation` ფუნქციაზე ზემოთ მოყვანილის ტოლძალოვანი განაცხადია

  ```
  float Calculation(void);
  ```

  სადაც მრგვალ ფრჩხილებში ჩაწერილი `void` ნიშნავს პარამეტრების ცარიელ სიას.

ქვემოთ მოყვანილ პროგრამაში ფუნქცია `Calculation` ითვლის [217, 426] შუალედში 13-ის ჯერადი რიცხვების საშუალო არითმეტიკულს.

```
#include <iostream>
using namespace std;
float Calculation();
int main()
{
```

```
    float k;
    k = Calculation();
    cout<<k<<endl;
    return 0;
}
float Calculation(){
    int s =0, k =0;
    for(int n = 426/13*13; n >= 217; n -= 13){
        s += n;
        k++;
    }
    return (float)s/k;
}
```

პროგრამა დაბეჭდავს:

```
318.5
Press any key to continue . . .
```

ფუნქციისთვის არგუმენტის გადაცემა მნიშვნელობით (**Pass-by-value**)

არგუმენტის გადაცემის უმარტივესი ხერხია გადავცეთ ფუნქციას არგუმენტის ასლი. ამ შემთხვევაში ფუნქციის გამოძახების დროს მის პარამეტრს ენიჭება იმ მნიშვნელობის ასლი, რომელსაც არგუმენტად მივიჩნებთ. არგუმენტის გადაცემის ამ ხერხს უწოდებენ მნიშვნელობით გადაცემას – `Pass-by-value`.

შემდეგ მაგალითში განისაზღვრება სამ რიცხვს შორის უმცირესის პოვნის ფუნქცია `min3Int`. პროგრამაში შეგვაქვს 3 მთელი რიცხვი. შემდეგ ეს რიცხვები გადაეცემა ფუნქციას `min3Int`, რომელიც დაადგენს მათ შორის უმცირესს. **`return`** შეტყობინება დაუბრუნებს `main`-ს უმცირესის მნიშვნელობას, და ის დაიბეჭდება.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int min3Int(int, int, int); // ფუნქციის პროტოტიპი
int main()
{
    int a, b, c;
    cout<<"ShemoiteneT 3 mTeli ricxvi: ";
    cin>>a>>b>>c;
    cout<<"Minimaluri udris "
        << min3Int(a,b,c) <<endl;
    return 0;
}
// minimum ფუნქციის განსაზღვრა
int min3Int(int x, int y, int z){
    int min = x;
    if (y < min) min = y;
    if (z < min) min = z;
    return min;
}
```

პროგრამის შესრულების შედეგია:

```
ShemoiteneT 3 mTeli ricxvi: 23 7 69
Minimaluri udris 7
Press any key to continue . . .
```

ფუნქციის პროტოტიპი **int** min3Int(**int**, **int**, **int**); გვიჩვენებს, რომ ფუნქცია "ელოდება" 3 მთელ არგუმენტს და დააბრუნებს მთელს.

ფუნქციის გამოძახება ჩვენ მაგალითში ხდება გამოტანის შეტყობინებაში

```
cout<<"Minimaluri udris "<< min3Int(a,b,c)<<endl;
```

და გვიჩვენებს, რომ ფუნქციაში გათვალისწინებული მოქმედებები უნდა შესრულდეს a, b და c არგუმენტებზე – მთელ რიცხვებზე. min3Int(a,b,c) -ის შესრულება შემდეგი ფრაგმენტის შესრულების ეკვივალენტურია:

```
int x = a;  int y = b;  int z = c;
int min =x;
if (y < min) min =y;
if (z < min) min =z;
return min;
```

ე.ი. x პარამეტრს მიენიჭება a არგუმენტის მნიშვნელობა, y პარამეტრს – b-ს მნიშვნელობა, z პარამეტრს – c-ს მნიშვნელობა (ანუ ფუნქციის ლოკალური ცვლადები "დაიჭერენ" მათთვის გამოძახების დროს გადაგზავნილ მნიშვნელობებს), ხოლო ფუნქციის შეტყობინებები, ფაქტობრივად, შესრულდება a, b და c მნიშვნელობებისთვის. უნდა გვახსოვდეს, რომ პარამეტრებსა და არგუმენტებს შორის შესაბამისობა დგინდება მათი რიგის მიხედვით: პირველ პარამეტრს (x -ს) შეესაბამება პირველი არგუმენტი (a), მეორე პარამეტრს (y -ს) - მეორე არგუმენტი (b), ხოლო მესამე პარამეტრს (z -ს) - მესამე არგუმენტი (c).

არგუმენტის გადაცემა მისამართით (Pass-by-reference)

ფუნქციაში არგუმენტების გადაცემის 2 ხერხი არსებობს: არგუმენტების გადაცემა მნიშვნელობით და არგუმენტების გადაცემა მისამართით.

ფუნქციის გამოძახების დროს მის პარამეტრებს და ლოკალურ ცვლადებს გამოეყოფათ ადგილი ოპერატიული მეხსიერების სპეციალურ არეში – სტეკში. როდესაც არგუმენტი გადაეცემა ფუნქციას მნიშვნელობით, ფუნქციის პარამეტრს ენიჭება ამ არგუმენტის ასლი. ამიტომ, თუ ფუნქციის ტანში მისი პარამეტრი იცვლის მნიშვნელობას, შესაბამისი არგუმენტი არ იცვლება.

ხშირად საჭიროა ფუნქციაში გადაცემული არგუმენტის შეცვლა. მაგალითად, როდესაც:

- ფუნქციიდან ერთზე მეტი მნიშვნელობის დაბრუნება გვჭირდება. მაშინ ერთ მნიშვნელობას დააბრუნებს **return** შეტყობინება, დანარჩენი მნიშვნელობების დაბრუნება კი უნდა მოხერხდეს პარამეტრების მეშვეობით.
- პროგრამაში გვჭირდება ფუნქციის მიერ არგუმენტის შეცვლილი მნიშვნელობა.
- ფუნქციას დასამუშავებლად გადაეცემა მონაცემთა დიდი ობიექტი. ასეთი არგუმენტის გადაცემა მნიშვნელობით (მისი კოპირება პარამეტრში) მოითხოვს გარკვეულ დროს, რაც ანელებს პროგრამის შესრულებას.

მსგავს შემთხვევებში ფუნქციას უნდა გადაეცეს არა არგუმენტის ასლი, არამედ მისი მისამართი. მაშინ ყველა ცვლილება, რომელსაც ფუნქციაში განიცდის პარამეტრი, სინამდვილეში ხორციელდება არგუმენტზე. ასეთ პარამეტრებს ეწოდებათ ცვლადი პარამეტრები.

არგუმენტის მისამართით გადაცემის ერთი ხერხია შესაბამისი პარამეტრი აღვწერით ე.წ. reference–ის სახით. ტერმინი reference ხშირად ითარგმნება როგორც მითითება, მეტსახელი ან ფსევდონიმით მიმართვა.

reference არის პროგრამული ობიექტის ფსევდონიმი (alias), ანუ, მარტივად რომ ვთქვათ, იმ პროგრამული ობიექტის მეტსახელი, რომელსაც იგი მიუთითებს.

მაგალითად,

```
int j = 10, k;
```

```
int &i = j;     // განაცხადი  i მითითებაზე (ფსევდონიმზე)
```

განაცხადი აღნიშნავს, რომ `i` ირიბად მიუთითებს `j`-ს, ანუ *მთელი ტიპის* `i` ცვლადის მისამართი იგივეა, რაც მთელი ტიპის `j` ცვლადისა (ე.ი. `i` და `j` - ერთი და იგივე მეხსიერების სახელებია). ცხადია, რომ მითითებაზე (ფსევდონიმზე) განაცხადისთანავე აუცილებლად უნდა მოხდეს მისი ინიციალიზება.

შემდეგი ფრაგმენტი

```
int j = 10, k;
int &i = j;     // განაცხადი  i მითითებაზე (ფსევდონიმზე)
cout<<j<<" "<< i;    //  დაიბეჭდება  10 10
k = 121;
i = k;
//  i-ს მიენიჭა k-ს მნიშვნელობა, ე.ი.  j-ც გახდება k-ს ტოლი
cout<<j<<" "<<i;     // დაიბეჭდება  121  121
i++;
cout<<j<<" "<<i;     // დაიბეჭდება  122  122
```

გვარწმუნებს, რომ მითითება `i` არის `j` ცვლადის (რომელსაც `i` მიუთითებს) მეტსახელი.

ფუნქციის `reference`-პარამეტრი წარმოადგენს შესაბამისი არგუმენტის ფსევდონიმს. მაგალითად,

```
void f(int & i);
```

ფუნქციის პროტოტიპში `i` არის `int` ტიპის `reference`-პარამეტრი (ამბობენ აგრეთვე, რომ `i` არის მითითება `int` ტიპზე).

ვთქვათ, `k` – მთელი ტიპის ცვლადია. `f` ფუნქციის

```
f(k);
```

გამოძახების დროს სრულდება მინიჭება

```
int &i = k
```

ანუ `i` ხდება `k` ცვლადის მეტსახელი. ამიტომ `i` პარამეტრზე მიმართვა ფუნქციის ტანში ფაქტობრივად ნიშნავს `k` არგუმენტზე მიმართვას, და `i` –ს შეცვლა იგივე `k` –ს შეცვლას ნიშნავს.

არგუმენტის ასეთ გადაცემას უწოდებენ `Pass-by-reference`.

შემდეგ საილუსტრაციო მაგალითში ნაჩვენებია `reference`-პარამეტრის გამოყენება

```
#include <iostream>
using namespace std;
void f(int &);
int main(){
   int value = 1;
   cout<<"value–s sawyisi mnishvneloba: "
       <<value<<'\n';
   f(value);     //value–ს გადავცემთ მითითებით
   cout<<"value–s axali mnishvneloba: "
       <<value<<'\n';
   return 0;
}
void f(int &i){
   i = 10;    // არგუმენტის შეცვლა
}
```

პროგრამის შესრულების შედეგია:

```
value–s sawyisi mnishvneloba: 1
value–s axali mnishvneloba: 10
Press any key to continue . . .
```

შემდეგ მაგალითში reference–პარამეტრს გამოვიყენებთ ვექტორის კლავიატურიდან შევსების ფუნქციაში, ხოლო ვექტორის ელემენტების ბეჭდვის ფუნქციაში – const reference–პარამეტრს:

```
#include <iostream>
#include <vector>
using namespace std;
void fillVector(vector<int>& );
void printVector(const vector<int>& );
int main(){
   vector<int> A;
   fillVector(A);
   cout<<"Vectori:\n";
   printVector(A);
   return 0;
}
void fillVector(vector<int>& x){
   int number;
   while( cin>>number )
      x.push_back(number);
}
void printVector(const vector<int>& x){
   for(int i=0; i<x.size(); i++)
       cout<<x[i]<<' ';
   cout<<endl;
}
```

პროგრამის შესრულების შედეგია:

```
3 -7 19 231 0 25 -87 ^Z
Vectori:
3 -7 19 231 0 25 -87
Press any key to continue . . .
```

```
void fillVector(vector<int>& x);
```

ფუნქციის პარამეტრი წარმოადგენს reference–პარამეტრს. ეს აუცილებელია, რადგან ფუნქციის დანიშნულებაა ჩაწეროს ვექტორში კლავიატურიდან შემოსული მთელი რიცხვები, ანუ ფუნქციის

```
fillVector(A);
```

გამოძახების შედეგად A ვექტორი უნდა შეიცვალოს.

ფუნქცია printVector ბეჭდავს ვექტორს. რადგან A ვექტორის ზომა შეიძლება იყოს დიდი, ამ ფუნქციის პარამეტრი მიზანშეწონილია გამოვიყენოთ reference სახით, მიუხედავად იმისა რომ ვექტორის შეცვლას არ ვგეგმავთ. ასეთ შემთხვევებში უნდა ვიყოთ ფრთხილად – თუ ფუნქციის ტანში შემთხვევით შეიცვლება x ვექტორის ელემენტი (ელემენტები), ეს შეგვიცვლის A ვექტორს. ენაში გვაქვს დაცვის მექანიზმი, რომელიც მსგავსი შეცდომებისგან გვიცავს – const reference პარამეტრის გამოყენება:

```
void printVector(const vector<int>& x);
```

ახლა თუ ნებით ან უნებლიეთ შევეცდებით a ვექტორის შეცვლას, კონპილერი გამოიტანს შეცდომის შესახებ გზავნილს: 'x' : you cannot assign to a variable that is const.

თავი 9: პოინტერი. ფუნქციის არგუმენტი – პოინტერი. ფუნქციის პარამეტრების ინიციალიზება (გულისხმობის პრინციპით). ფუნქციების გადატვირთვა.

- მისამართები. მოქმედებები მისამართებზე.
- პოინტერი, მუდმივი პოინტერი.
- პოინტერის გამოყენება ფუნქციის არგუმენტად (Pass-by-pointer)
- ფუნქციის არგუმენტების მნიშვნელობა გაჩუმებით
- ფუნქციების გადატვირთვა
- ფუნქციის არგუმენტი – ფუნქცია

მისამართები. მოქმედებები მისამართებზე

C++ უზრუნველყოფს ძირითადი ტიპის მონაცემებთან (მთელი, სიმბოლური, ათწილადი) მუშაობას. პროგრამისტს თვითონაც შეუძლია მონაცემთა მრავალფეროვანი ტიპების შექმნა. ტიპები ერთმანეთისგან განსხვავდება არა მხოლოდ მეხსიერებაში მათი წარმოდგენით, არამედ იმ მოქმედებების (ოპერაციები, ფუნქციები) ერთობლიობითაც, რაც ამ ტიპის მონაცემებზეა დაშვებული.

ერთ-ერთი ძირითადი ოპერაცია არის მონაცემის მისამართის განსაზღვრა. თუ x არის რაიმე ტიპის მონაცემი, მაშინ &x არის მისი მისამართი მეხსიერებაში. იმის სანახავად, თუ რას წარმოადგენს მისამართი, საკმარისია იგი ამოვბეჭდოთ როგორც ათობითი ან თექვსმეტობითი რიცხვი.

&x–ის რიცხვითი მნიშვნელობა წარმოადგენს მეხსიერებაში იმ მონაკვეთის დასაწყისს, სადაც მოთავსებულია x–ის მნიშვნელობა. აღსანიშნავია, რომ &x–ის საშუალებით შეგვიძლია ისიც გავიგოთ თუ სად მთავრდება x–ის მნიშვნელობის შემცველი მონაკვეთი.

კარგად რომ გავერკვეთ მისამართების არითმეტიკაში, მეხსიერება უნდა წარმოვიდგინოთ როგორც დიდი მონაკვეთი, რომელიც დაყოფილია ბაიტებად და წარმოადგენს რიცხვითი ღერძის დისკრეტულ ანალოგს. მართლაც, &x იძლევა ათვლის წერტილს (სადაც &x, იგივეა რაც &x+0), ანუ საწყისი ბაიტის მისამართს, ხოლო ბაიტების ის რაოდენობა, რაც საჭიროა x მონაცემის მეხსიერებაში მოსათავსებლად წარმოადგენს ერთეულს. თუ ერთი და იგივე ტიპის მონაცემები მეხსიერებაში მიმდევრობით არის მოთავსებული, მაშინ &x+1 წარმოადგენს x-ის მომდევნო მონაცემის მისამართს და ა. შ. შეგვიძლია მეხსიერებაში ნავიგაცია &x-ის ორივე მხარეს "ერთეულის" ჯერადი მისამართებით.

ვთქვათ, გვაქვს განაცხადი

```
double x = 5.009;
```
(1)

და &x არის 1245012 –ის ტოლი, ხოლო **sizeof**(x) არის 8. ეს სიდიდეები დამოკიდებულია სისტემაზე, ხოლო მისამართის მნიშვნელობა, საზოგადოდ, დამოკიდებულია იმაზეც თუ კონკრეტულ სიტუაციაში რამდენად არის დატვირთული კომპიუტერი და პროგრამის სხვადასხვა გაშვებაზე შეიძლება სხვადასხვა მნიშვნელობა მოგვცეს. განსხვავებული იქნება ათვლის წერტილი, ხოლო მასშტაბი უცვლელი.

```
cout<<int(&x + 1);
```

შეტყობინების გამოყენებით მარტივად შეგვიძლია შევამოწმოთ, რომ &x+1 არის არა 1245013, არამედ 1245020, რადგან **double** ტიპის ცვლადისთვის ერთეულს წარმოადგენს 8 ბაიტი, რაც დაემატა მისამართს. ანალოგიურად, &x+3 არის 1245036 და ა. შ. ნავიგაცია შეიძლება ორივე მიმართულებით. მაგალითად, &x-186 არის 1243524.

მისამართებზე მოქმედებები მეხსიერების სხვადასხვა მონაკვეთებზე წვდომის საშუალებას იძლევა. ცხადია &x მისამართზე მოთავსებულია x–ის მნიშვნელობა, ანუ 5.0. x–ის მნიშვნელობის მოპოვება მისამართის საშუალებით შეიძლება ასე: (&x)[0]. ანალოგიურად, იმისათვის რომ გავიგოთ თუ რა მნიშვნელობა არის მოთავსებული &x-186 მისამართზე, ვწერთ (&x)[-186].

&x-ზე მოთავსებული მნიშვნელობის განსაზღვრისთვის C++ -ში გათვალისწინებულია ირიბი მნიშვნელობის აღების, ანუ განმისამართების ოპერაცია *(&x), იგივე *&x. მაგალითად, *(&x+17) იგივეა, რაც (&x)[17].

ყოველ ცვლადს, მაგალითად (1) განაცხადით განსაზღვრულს, გააჩნია სამი მახასიათებელი: აუცილებლად სახელი და მისამართი, და მნიშვნელობა. ამისგან განსხვავებით, მის მისამართს ანუ &x–ს გააჩნია სახელი (x–ის მისამართი), მნიშვნელობა (ჩვენს მაგალითში 2293620) და ირიბი მნიშვნელობა (ჩვენს მაგალითში 5.0). როგორც ვხედავთ, ცვლადის მისამართს არ გააჩნია თავისი მისამართი, თუმცა ხშირად ჩნდება მისამართების დამახსოვრების აუცილებლობა მასზე განთავსებული მნიშვნელობის აღდგენის მიზნით. ჩვენთვის აქამდე ცნობილი მონაცემთა ტიპების საფუძველზე ამის გაკეთება შეუძლებელია.
მაგალითად, ავიღოთ

```
int  p;
```

და დავიმახსოვროთ

```
p = &x;        // შეცდომა
```

ფორმალურად p ტოლია x–ის მისამართის, მაგრამ მისი საშუალებით ჩვენ ვეღარ აღვადგენთ &x -ის ირიბ მნიშვნელობას ანუ x–ს, რადგან მთელი ტიპის ცვლადზე ირიბი მნიშვნელობის აღების ოპერაცია (*p) განსაზღვრული არაა.

მიმთითებელი ანუ პოინტერი

მისამართებთან სამუშაოდ C++ ენაში შემოტანილია მისამართის ტიპი. მისამართის ტიპის ცვლადს პოინტერი (მიმთითებელი) ეწოდება. პოინტერის დანიშნულებაა ცვლადის მისამართის შენახვა. მასზე განაცხადს აქვს სახე:

*ტიპი *პოინტერი;*

მაგალითად, შეტყობინება

```
float *f_ptr;                                    (2)
```

ნიშნავს, რომ f_ptr –ის მნიშვნელობა შესაძლებელია იყოს მხოლოდ **float** ტიპის ცვლადის მისამართი, ხოლო მისი ირიბი მნიშვნელობა იქნება **float** ტიპისა. ზოგადად, პოინტერი არის ცვლადი, რომელსაც გააჩნია ოთხი მახასიათებელი: სახელი, მისამართი (ესენი აუცილებლად), აგრეთვე მნიშვნელობა და ირიბი მნიშვნელობა. (2) შეტყობინებით შემოტანილ პოინტერს ბოლო ორი ჯერ-ჯერობით არ გააჩნია.

შემდეგი კოდის ფრაგმენტი:

```
double *f_ptr, x = 13.45;
f_ptr = &x;
```

ნიშნავს შემდეგს: f_ptr –ის ირიბი მნიშვნელობა არის **double** ტიპის, x -ის მნიშვნელობაც არის **double** ტიპის, f_ptr –ის მნიშვნელობა x -ის მისამართია, ხოლო ირიბი მნიშვნელობა არის 13.45, ანუ *f_ptr ტოლია 13.45-ის.
ან, მაგალითად,

```
int a = 9;
int *a_ptr = &a;
```

ნიშნავს, რომ a –ს მნიშვნელობა და a_ptr –ის ირიბი მნიშვნელობა *a_ptr, ორივე უდრის 9-ს. უფრო მეტიც, ერთის შეცვლა იწვევს იგივე მნიშვნელობით მეორის შეცვლას. მაგალითად, *a_ptr = 34; შეტყობინების შემდეგ a –ს მნიშვნელობა იქნება 34.

პოინტერების ერთ-ერთ თავისებურებას ის წარმოადგენს, რომ ყველა ტიპის პოინტერი მეხსიერებაში, ჩვეულებრივ, 4 ბაიტს იკავებს. ეს იძლევა საშუალებას, რომ განვსაზღვროთ **void** ტიპის პოინტერიც, რომელსაც შემდეგ შეგვიძლია მივანიჭოთ სხვა ტიპის პოინტერების მნიშვნელობები.

არარსებული მისამართის აღსანიშნავად გამოიყენება მუდმივი NULL. მაგალითად, შეტყობინება

```
f_ptr = NULL;
```

ნიშნავს, რომ ამ პოინტერს არა აქვს კონკრეტული მნიშვნელობა. თუ გამოვბეჭდავთ, არარსებული მისამართი დაიბეჭდება როგორც 0. NULL მუდმივი განსაკუთრებით სასარგებლოა დინამიკურ მონაცემთა სტრუქტურებთან მუშაობისას.

მუდმივი პოინტერი

მუდმივ პოინტერზე განაცხადის გაკეთება, სხვა ტიპებისგან განსხვავებით, მეტ დაკვირვებას მოითხოვს. თუ ჩვენ, მაგალითად, გავაკეთებთ ასეთ განაცხადს:

```
const int number = 5;
```

იგი აცნობებს C++ –ს, რომ number ცვლადი არის მუდმივი, ასე რომ

```
number = 10;    // შეცდომა
```

არის შეცდომა. ფრაგმენტი

```
char symbol = 'M';
const char *ch_ptr = &symbol;
```

C++ –ისთვის არ ნიშნავს, რომ ცვლადი ch_ptr არის მუდმივი, იგი აცნობებს C++ –ს რომ ამ ცვლადის ირიბი მნიშვნელობა არის მუდმივი, ანუ ირიბ მნიშვნელობას, რომელზეც მიუთითებს პოინტერი, ვერ შევცვლით, მაგალითად,

```
*ch_ptr = 'X';
```

შეცდომაა. მაგრამ შეგვიძლია შევცვალოთ პოინტერის მნიშვნელობა.

იმისათვის რომ C++ –ისთვის თვითონ პოინტერი იყოს მუდმივი, უნდა გავაკეთოთ, მაგალითად, ასეთი განაცხადები:

```
char c = 'M';
char * const c_ptr = &c;
```

ან

```
double y = 0.00357;
double * const d_ptr = &y;
```

რაც ნიშნავს, რომ პოინტერი მუდმივია, მისი ირიბი მნიშვნელობა კი არა. მაშასადამე სწორია

```
*c_ptr = 'B';      // სწორია, რადგან *c_ptr არის სიმბოლო
*d_ptr = 6.1;      // სწორია, რადგან *d_ptr არის რიცხვი
```

მაგრამ არასწორია

```
c_ptr += 3;        // არასწორია, რადგან c_ptr მუდმივია
d_ptr = &y - 1;    // არასწორია, რადგან d_ptr მუდმივია
```

დასასრულ, C++ საშუალებას გვაძლევს ერთდროულად გამოვაცხადოთ მუდმივებად როგორც პოინტერი, ასევე მისი ირიბი მნიშვნელობა. მაგალითად:

```
int K = 719;
const int *const ptr = &K;
```

პოინტერის გამოყენება ფუნქციის არგუმენტად (**Pass-by-pointer**)

ჩვენ განვიხილეთ ფუნქციისთვის არგუმენტების გადაცემის ორი გზა: არგუმენტების ასლების გადაცემა (Pass-by-value), და არგუმენტების გადაცემა მისამართებთან ერთად (Pass-by-reference).

პირველი მიდგომა ყველაზე ადრინდელია. დაკავშირებულია დროისა და მეხსიერების ხარჯთან, ამ დროს ფუნქციას შეუძლია რომ მხოლოდ ერთადერთი მნიშვნელობა დააბრუნოს **return** შეტყობინების გამოყენებით. ფუნქციის გამოძახებისას არგუმენტების ასლების გადაცემა ჰგავს "ცალმხრივ მოძრაობას" - პარამეტრების შეცვლილი მნიშვნელობები უკან აღარ ბრუნდება, სამაგიეროდ, ესაა საკმაოდ კარგი გზა საიმედო კოდის შექმნისთვის.

მეორე მიდგომა ძალიან თანამედროვეა, იგი აქტიურად გამოიყენება პროგრამირების სხვა თანამედროვე ენებშიც და მას დიდ დროს დავუთმობთ შემდეგში.

C++ -ში არსებობს საშუალება, რომ ფუნქციას გადავცეთ არა არგუმენტის ასლი, არამედ მისი მისამართი. რა ცვლადის მისამართსაც გადავაწვდით, იმ ცვლადის დაბრუნება აღარაა საჭირო, რადგან რეალურ მეხსიერებაში ფუნქციის მოქმედების შედეგად ჩაიწერება საჭირო მნიშვნელობა. ამ მიზნის მისაღწევად (ე.ი. მისამართების გადასაცემად) უნდა ვისარგებლოთ პოინტერით.

მაგალითად, გვაქვს ფუნქცია, რომელიც ზრდის count ცვლადს ერთით, ხოლო პროგრამაში ეს ფუნქცია რამდენჯერმე გამოიძახება. გავიგოთ, რამდენჯერ მოხდა ფუნქციის გამოძახება.

```
#include <iostream>
using namespace std;
void inc_count(int *count_ptr)
{
    (*count_ptr)++;
}
int main()
{
    int count(0);                   // განმეორებათა რაოდენობა
    while (count < 10)
        inc_count(&count);
    cout<<"count =  "<<count<<endl;
    system("PAUSE");
    return (0);
}
```

როგორც ფუნქციის სათაური გვიჩვენებს, ფუნქციას გადაეცემა მთელი ტიპის რიცხვის მისამართი count_ptr, რომლის ირიბი მნიშვნელობა არის მთელი. მართლაც, main –ში ამ ფუნქციის ყოველი გამოძახება ხდება სახით: inc_count(&count); შედეგად ფუნქციის ლოკალური ცვლადი "დაიჭერს" მისამართს ანუ count_ptr = &count; და არგუმენტის ამ მნიშვნელობისთვის ფუნქციის ტანი (*count_ptr)++; განხორციელდება სახით:

(*&count)++; ანუ count++;

როგორც ვხედავთ, ყოველ გამოძახების შედეგად იზრდება იმ ცვლადის მნიშვნელობა, რომლის მისამართსაც ფუნქცია მიიღებს არგუნეტად.

ვთქვათ, ჩვენი ამოცანის პირობაა: დაწერეთ პროგრამა, რომელიც გამოითვლის კლავიატურიდან შემოტანილი რადიუსის მქონე წრის ფართობსა და წრეწირის სიგრძეს. ორივე სიდიდის მისაღებად დაწერეთ და პროგრამაში გამოიყენეთ ერთი ფუნქცია.

```
#include <iostream>
using namespace std;

const double PI = 3.14159;

double Circle(double r, double * sAddress);

int main()
{
   double radius, area, length;
   cout<<"Enter circle's radius  ";
   cin >> radius;
   length = Circle( radius, &area );
   cout<<"Area of Circle = "<< area
      <<"\nLength of Circle = "<< length << endl;
   return 0;
}

double Circle(double r, double * sAddress)
{
   * sAddress = PI * r * r;
   return 2 * PI * r;
}
```

პროგრამის შედეგი:

```
Enter circle's radius  1
Area of Circle = 3.14159
Length of Circle = 6.28318
Press any key to continue . . .
```

`Circle` ფუნქცია **`return`** –ის საშუალებით დააბრუნებს წრეწირის სიგრძეს, და ეს სიდიდე მიენიჭება `main`–ის `length` ცვლადს, ხოლო პარამეტრი –პოინტერის საშუალებით დააბრუნებს წრის ფართობს შემდეგნაირად: გამოძახების დროს გვექნება `sAddress=&area;` ამიტომ ფუნქციის ტანში პირველი შეტყობინება შესრულდება რეალური ცვლადებისთვის შემდეგი სახით:

```
*&area = PI * radius * radius;
```

რაც ნიშნავს, რომ `area` ცვლადის მისამართზე ჩაიწერება (ანუ`area` ცვლადს მიენიჭება) წრის ფართობის მნიშვნელობა.

ფუნქციის არგუმენტების მნიშვნელობა გულისხმობის პრინციპით

C++ -ის სინტაქსის მიხედვით ფუნქციის პარამეტრებს შეიძლება მივანიჭოთ საწყისი მნიშვნელობები, ანუ მოვახდინოთ პარამეტრების ინიციალიზება ფუნქციაზე განაცხადის დროს.

ფუნქციის გამოძახებისას, ჩვეულებრივ, ფუნქციის არგუმენტებისა და პარამეტრების რიცხვი უნდა ემთხვევოდეს. თუ ფუნქციის პროტოტიპში პარამეტრებს მინიჭებული აქვთ საწყისი მნიშვნელობები, ფუნქციის არგუმენტების რაოდენობა შეიძლება იყოს პარამეტრების რაოდენობაზე ნაკლები. ამ შემთხვევაში გამოტოვებული არგუმენტის ნაცვლად ფუნქცია გამოიყენებს მისი შესაბამისი პარამეტრის საწყის მნიშვნელობას. ასეთ არგუმენტებს ეწოდებათ *ნაგულისხმევი არგუმენტები.*

მაგალითად, თუ ფუნქციის პროტოტიპია

```
void f ( int x = 3, int y = 7 );
```

მაშინ მის გამოძცებას შესაძლოა ჰქონდეს შემდეგი სახე:

```
f( 10, –67);
```

ან

```
f( );   // გამოიძახება f( 3, 7 );
```

შესაძლებელია პარამეტრების ნაწილობრივი ინიციალიზება. ამ შემთხვევაში ინიციალიზება უნდა იწყებოდეს *მარჯვნიდან, დაწყებული ბოლო პარამეტრით*. მაგალითად,

```
void func(int a, int b=2, float c=3.75);
```

სინტაქსურად სწორი პროტოტიპია. ამ ფუნქციის გამოძახებას შესაძლოა ჰქონდეს სახე:

```
func(10);    // გამოიძახება f ( 10, 2, 3.75  );
func(5,7);   // გამოიძახება f ( 5, 7, 3.75 );
func(1,3,12.7);
```

ხოლო გამოძახება

```
func();
```

გამოიწვევს კომპილაციის შეცდომას.

ფუნქციის პროტოტიპში დასაშვებია პარამეტრების სახელების გამოტოვება. მაგალითად, სწორი იქნება პროტოტიპი

```
void func1(int, int=2, float=3.75);
```

უნდა გავითვალისწინოთ, რომ `func1` ფუნქციის განსაზღვრისას პარამეტრების სახელების მითითება *აუცილებელია*, ხოლო პარამეტრების საწყისი მნიშვნელობების გამეორება *შეცდომაა*. მაგალითად, `func1` ფუნქციის განსაზღვრას შესაძლოა ჰქონდეს სახე

```
void func1(int x, int y, float z){
    // შესრულებადი შეტყობინებები
}
```

გავითვალისწინოთ, რომ თუ პროტოტიპი და ინტერპრეტაცია განცალკევებულია, მაშინ ნაგულისხმევი არგუმენტების მითითება მხოლოდ პროტოტიპშია საჭირო.

შემდეგ საილუსტრაციო მაგალითში შევქმნით ფუნქციას `Sum`, რომელიც გამოძახების შესაბამისად გამოითვლის ორი ნამდვილი რიცხვის ან სამი ნამდვილი რიცხვის ჯამს.

```
#include <iostream>
using namespace std;
double Sum(double, double, double = 0.);
int main(){
   double a, b, c;
   cout<<"Enter 3 real numbers:  ";
   cin >> a >> b >> c;
   cout<<"Two numbers sum is "
        << Sum( a, b)<<endl
        <<"Three numbers sum is "
        <<Sum( a, b, c )<<endl;
   return 0;
}
double Sum(double x, double y, double z)
{
     return x + y + z;
}
```

პროგრამა დაბეჭდავს:

```
Enter 3 real numbers:  1.2 3.4 5.6
Two numbers sum is 4.6
Three numbers sum is 10.2
Press any key to continue . . .
```

ფუნქციების გადატვირთვა

C++ -ში შესაძლებელია ერთი და იგივე სახელი დავარქვათ რამდენივე ფუნქციას იმ პირობით, რომ ასეთი ფუნქციების პარამეტრების სია იქნება განსხვავებული: განსხვავებული უნდა იყოს ან პარამეტრების რიცხვი, ან მათი ტიპი, ან პარამეტრების რიცხვიც და ტიპიც. ფუნქციების ამ თვისებას ეწოდება *ფუნქციათა გადატვირთვა* (function overloading). გადატვირთული (overloaded) ფუნქციების გამოძახების დროს C++ -ის კომპილერი აანალიზებს არგუმენტების რაოდენობას, მათ ტიპსა და რიგითობას და ისე ადგენს შესასრულებელი ფუნქციის შესაბამის ეკზემპლარს. უნდა აღვნიშნოთ, რომ ფუნქციების დასაბრუნებელ მნიშვნელობას კომპილერი "ყურადღებას არ აქცევს". ე.ი. თუ ფუნქციების პროტოტიპები განსხვავდება მხოლოდ დასაბრუნებელი მნიშვნელობებით, ასეთი ფუნქციების გადატვირთვა არ შეიძლება.

შემდეგ მაგალითში გადატვირთულია ფუნქცია Minimal, რომლის სამი ვერსია განსხვავებული მოქმედებისაა: პირველი პოულობს სამი სიმბოლოდან უმცირესს, მეორე ადგენს უმცირესს მთელ რიცხვთა ვექტორში, ხოლო მესამე აბრუნებს უმცირესს ორ ნამდვილ რიცხვს შორის.

```
#include <iostream>
#include <vector>
using namespace std;
char Minimal(char, char, char );
int Minimal(vector<int> );
double Minimal(double, double );
int main(){
   char a, b, c;
   cout<<"ShemoitaneT 3 simbolo ";
   cin >> a >> b >> c;
   cout<<"ShemoitaneT 2 namdvili ricxvi ";
   double d, t;
   cin >> d >> t;
   vector <int> V;
   int n;
   cout<<"Shemoitanet ramdenime mTeli ricxvi  ";
   while( cin >> n)
        V.push_back(n);
   cout<<"3 simbolos shoris unciresi = "
        <<Minimal(a, b, c)<<endl;
   cout<<"2 namdvil ricxvs Soris umciresi = "
        <<Minimal( d, t )<<endl;
   cout<<"Veqtoris umciresi elementi = "
        <<Minimal( V )<<endl;
   return 0;
}
char Minimal(char x, char y, char u){
     char min = x;
     if( y < min ) min = y;
     if( u < min ) min = u;
     return min;
}
int Minimal(vector<int> x){
   int min = x[0];
   for(int i=0; i<x.size(); ++i)
       if( x[i] < min )
         min = x[i];
   return min;
}
double Minimal(double a, double b){
     return a < b ? a : b;
```

```
}
```
პროგრამა დაბეჭდავს:

```
ShemoitaneT 3 simbolo a A z
ShemoitaneT 2 namdvili ricxvi 23.78 10.25
Shemoitanet ramdenime mTeli ricxvi  100 -180 435 -200 793
^Z
3 simbolos Soris unciresi = A
2 namdvil ricxvs Soris umciresi = 10.25
Veqtoris umciresi elementi = -200
Press any key to continue . . .
```

C++ ენის საკმაოდ გავრცელებულ დიალექტში (C++/CLR), გულისხმობის პრინციპით არგუმენტების გადაცემა არ არის დაშვებული, რადგან ალტერნატივად განიხილება ფუნქციების გადატვირთვის მექანიზმი. მაგალითად, განვიხილოთ თუ რა იქნება

```
void F(int a, int b=2, float c=3.75)
{
    std::cout << a+b+c << std::endl;
}
```

ფუნქციის ალტერნატივა (C++/CLR)-ში. იგივე შედეგის მისაღწევად, ვაკეთებთ სამ ფუნქციას: ერთს სამი არგუმენტით,

```
void F(int a, int b, float c)
{
    std::cout << a+b+c << std::endl;
}
```

ერთს ორი არგუმენტით,

```
void F(int a, int b, float c)
{
    std::cout << a+b+3.75 << std::endl;
}
```

ერთს ერთი არგუმენტით,

```
void F(int a, int b, float c)
{
    std::cout << a+2+3.75 << std::endl;
}
```

ფუნქციის არგუმენტი – ფუნქცია

ფუნქციას შეიძლება არგუმენტად გადავცეთ სხვა ფუნქცია. შემდეგი პროგრამა ვექტორში ითვლის 3–ის ჯერად ელემენტებს და შემდეგ უარყოფით ელემენტებს. პროგრამაში შექმნილია **bool** ტიპის 2 ფუნქცია `isMultiple_3` და `isNegative`. პირველი ადგენს, არის თუ არა მისი მთელი არგუმენტი 3–ის ჯერადი რიცხვი, მეორე კი – არის თუ არა მისი მთელი არგუმენტი უარყოფითი რიცხვი. ასევე გვაქვს ფუნქცია `Count`, რომელსაც არგუმენტად გადავცემთ პირველს და შემდეგ მეორე ფუნქციას. ვექტორში 15 შემთხვევითი რიცხვია `[-20;40]` შუალედიდან.

```
#include <iostream>
#include <vector>
#include <ctime>
using namespace std;
bool isNegative(int );
bool isMultiple_3(int );
int Count(vector<int>, bool f(int) );
void fillVector(vector<int>& );
void printVector(const vector<int>& );
```

```
int main (){
  vector<int> A;
  fillVector( A );
  printVector( A );
  cout<<"3-is jeradebis raodenoba = "
       << Count( A, isMultiple_3 )<< endl;
  cout<<"UaryofiTebis raodenoba = "
       << Count( A, isNegative )<< endl;
  return 0;
}
bool isNegative(int x){
  return x < 0;
}
bool isMultiple_3(int x){
  return x%3 == 0;
}
int Count(vector<int> x, bool f(int) ){
   int counter =0;
   for(int i=0; i< x.size(); ++i)
      if( f( x[i]) ) counter++;
   return counter;
}
void fillVector(vector<int>& x){
      int number, count = 1;
      srand( time(NULL) );
      while( count <= 15 ){
         number = rand()%61 - 20;
         x.push_back( number );
         count++;
      }
}
void printVector(const vector<int>& x){
   for(int i=0; i<x.size(); ++i)
     cout<<x[i]<<' ';
   cout<<endl;
}
```

პროგრამის მუშაობის შედეგია:

```
22 -9 -2 -15 15 2 39 38 -20 -4 7 22 -20 3 10
3-is jeradebis raodenoba = 5
UaryofiTebis raodenoba = 6
Press any key to continue . . .
```

თავი 10: ორგანზომილებიანი ვექტორი

- ვექტორის ელემენტი – ვექტორი
- ტიპის განმსაზღვრელი **typedef**
- ორგანზომილებიანი ვექტორის გადაცემა ფუნქციაში
- მოქმედება ორგანზომილებიან ვექტორზე

ვექტორების ვექტორი (ორგანზომილებიანი ვექტორი)

პრაქტიკულ ამოცანებში ხშირად მოსახერხებელია საწყისი მონაცემები წარმოვადგინოთ ცხრილის სახით. C++ გვაძლევს საშუალებას ასეთი წარმოდგენისთვის შევქმნათ შესაბამისი მონაცემთა ტიპი – ვექტორების ვექტორი, ანუ ორგანზომილებიანი ვექტორი.

მაგალითად, განაცხადი

```
vector< vector <int> > intMatrix;
```

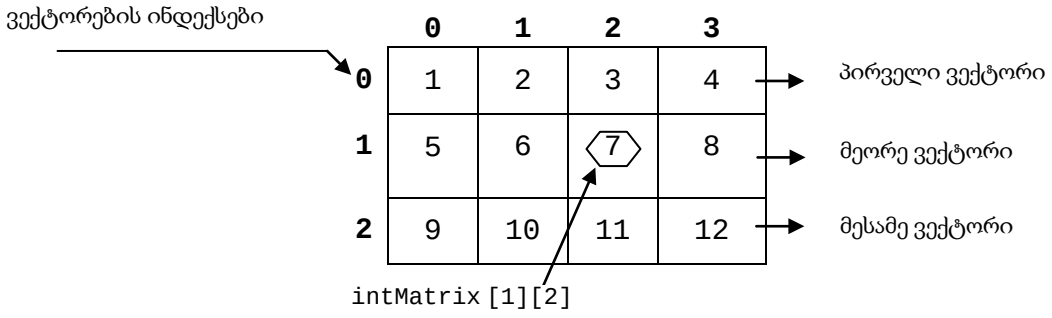
შექმნის ვექტორს `intMatrix`, რომლის ყოველი ელემენტი იქნება მთელი რიცხვების ვექტორი. ვამბობთ იქნება, რადგან ჯერჯერობით ეს ვექტორი ცარიელია. ვექტორში ელემენტების დამატება (`push_back()` მეთოდით) დიდად არ განსხვავდება ერთგანზომილებიან ვექტორში ელემენტების დამატებისგან და ამაზე ოდნავ მოგვიანებით ვილაპარაკებთ.

ვთქვათ, `intMatrix` ვექტორში უკვე ჩაწერილია გარკვეული ელემენტები. მაშინ `intMatrix` ვექტორის ელემენტებს შეიძლება მივმართოთ ინდექსის გამოყენებით: `intMatrix[0], intMatrix[1], ... , intMatrix[ intMatrix.size() – 1 ]`. ოღონდ, ეს ელემენტები არის, თავის მხრივ, მთელი რიცხვების ვექტორები და მათში ჩაწერილ რიცხვებზე წვდომა ხორციელდება ორი ინდექსის გამოყენებით: `intMatrix[i][j]`, სადაც `i` არის `intMatrix` – ში ელემენტი–ვექტორის ინდექსი, ხოლო `j` არის ამ ელემენტ–ვექტორში ელემენტის (რიცხვის) ინდექსი.

მაგალითად, `intMatrix` –ის პირველი (ინდექსით `0`) ვექტორის მესამე (ინდექსით `2`) რიცხვი არის

```
intMatrix[0][2]
```

თუ `intMatrix` ვექტორს აქვს სამი ელემენტი (თითოეული ელემენტი - ერთგანზომილებიანი ვექტორია), ხოლო მისი ყოველი ელემენტი (როგორც ერთგანზომილებიანი ვექტორი) შეიცავს ოთხ მთელ რიცხვს, მაშინ ორგანზომილებიანი ვექტორი `intMatrix` შეგვიძლია წარმოვიდგინოთ როგორც ცხრილი. `intMatrix` –ის ელემენტები ასეთი ცხრილის სტრიქონებს შეესაბამება, ხოლო რიცხვები – სტრიქონებში ელემენტებს.



თავდაპირველად ცარიელ `intMatrix` ვექტორში რიცხვები რომ ჩაიწეროს, საჭიროა შემდეგი მოქმედებების ჩატარება:

- უნდა გაუნაწილდეს (ანუ გამოეყოს) დინამიკური მეხსიერება ერთგანზომილებიან ვექტორს;
- დაემატოს იგი `intMatrix` –ს `push_back()` ფუნქციის გამოყენებით;
- შემდეგ ყოველ `intMatrix[i]` –ურში ჩავწეროთ რიცხვები.

შემდეგი პროგრამა შექმნის ორგანზომილებიან `intMatrix` ვექტორს, რომელიც შეიცავს `M (4)` ელემენტს – ვექტორს, ყოველში `N (5)` რიცხვით.

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
const int M = 4, N = 5;
int main (){
   vector< vector <int> > intMatrix;
    // intMatrix -ში ვექტორების დამატება
    for(int i=0; i<M; i++)
    {
         vector<int> row;
         intMatrix.push_back(row);
    }
    // ყოველ intMatrix[i] ვექტორში რიცხვების ჩაწერა
    for(int i=0; i<M; i++)
       for (int j=0; j<N; j++)
           intMatrix[i].push_back(i+j);
    // intMatrix ვექტორის სტრიქონ-სტრიქონ ბეჭდვა
    for(int i=0; i<intMatrix.size(); i++)
    {
        for (int j=0; j<intMatrix[i].size(); j++)
        {
           cout<<setw(4)<<intMatrix[i][j];
        }
        cout<<endl;
    }
    cout<<endl;
    return 0;
}
```

პროგრამის მუშაობის შედეგია:

```
   0   1   2   3   4
   1   2   3   4   5
   2   3   4   5   6
   3   4   5   6   7
Press any key to continue . . .
```

როდესაც ორგანზომილებიანი ვექტორი შეიცავს ერთი ზომის ვექტორებს (მათი ელემენტების რაოდენობა ერთი და იგივეა) შეგვიძლია მათთვის მეხსიერება გამოვყოთ განაცხადის გაკეთებისთანავე.

გავიხსენოთ ერთგანზომილებიანი ვექტორის შექმნის ერთ–ერთი ხერხი:

```
vector<int> X(4,100);
```

განაცხადის საფუძველზე იქმნება `X` ვექტორი, რომელსაც შექმნისთანავე გამოეყოფა მეხსიერება `4` მთელი რიცხვისთვის და თითოეული რიცხვის მეხსიერებაში ჩაიწერება `100`.

```
vector<int> X(4);
```

შემთხვევაში კი X ვექტორის 4 –ივე ელემენტს გულისხმობის პრინციპით მიენიჭება 0.

თუ ვისარგებლებთ ამ ხერხით, მაშინ

```
vector <int> row(5);
vector< vector <int> > intMatrix(4, row);
```

განაცხადი უზრუნველყოფს intMatrix ორგანზომილებიანი ვექტორისთვის 4 ვექტორი–ელემენტის შექმნას, ყოველში 5 მთელი 0–ლის ტოლი რიცხვით.

ამიტომ წინა ამოცანაში შეგვიძლია დავწეროთ

```
vector <int> row(N);
vector< vector <int> > intMatrix(M, row);
```

რაც გარანტიას გვაძლევს, რომ intMatrix –სთვის ყველა საჭირო მეხსიერება იქნება განაწილებული და მეტიც, განულებული. ამიტომ push_back ფუნქციის გამოყენება ამ შემთხვევაში საჭირო აღარ არის. უფრო მეტიც, ამ შემთხვევაში ვექტორის შევსების დროს შეგვიძლია გავითვალისწინოთ, რომ intMatrix ვექტორის ზომა არის intMatrix.size(), ხოლო მისი ყოველი სტრიქონის ზომა (ელემენტების რაოდენობა) არის intMatrix[i].size(), სადაც i ინდექსი მოთავსებულია ნულსა და intMatrix.size()-1 სიდიდეებს შორის (ბოლოების ჩათვლით). ახლა, ჩვენი პროგრამა მიიღებს შემდეგ სახეს:

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
const int M = 4, N = 5;
int main (){
    vector <int> row(N);
    vector< vector <int> > intMatrix(M, row);
   // ყოველ intMatrix[i] ვექტორში რიცხვების ჩაწერა
   for(int i=0; i<intMatrix.size(); i++)
      for (int j=0; j<intMatrix[i].size(); j++)
          intMatrix[i][j]= i+j;
   // intMatrix ვექტორის სტრიქონ-სტრიქონ ბეჭდვა
   for(int i=0; i<intMatrix.size(); i++)
   {
       for (int j=0; j<intMatrix[i].size(); j++)
       {
          cout<<setw(4)<<intMatrix[i][j];
       }
       cout<<endl;
   }
   cout<<endl;
   return 0;
}
```

ცხრილის სახით წარმოდგენასთან ანალოგიით ორგანზომილებიან ვექტორში შემავალ ვექტორებს მოიხსენიებენ როგორც მის სტრიქონებს, ხოლო ვექტორებში ერთი და იგივე ადგილას მდგომ ელემენტებს (ტოლი მეორე ინდექსით) – როგორც სვეტებს.

სხვათა შორის, შეგვიძლია სხვადასხვა სტრიქონს კიდევ დავუმატოთ სხვადასხვა რაოდენობის ელემენტები, რის შედეგადაც ჩვენი ვექტორი გახდება არამართკუთხა.

განხილული ორი მაგალითი სრულიად საწინააღმდეგო იყო ერთმანეთის იმ აზრით, რომ პირველში გავაკეთეთ სრულიად ცარიელი ვექტორი და მასში დინამიკურად ჩავამატეთ როგორც სტრიქონები, ასევე სვეტები. მეორე მაგალითში კი თვიდანვე დავაფიქსირეთ სვეტებისა და სტრიქობების რაოდენობები.

შესაძლებელია მესამე, კომპრომისული ვარიანტიც: თავიდან დავაფიქსიროთ მხოლოდ სტრიქონების რაოდენობა, ხოლო სტრიქონებში ელემენტები ჩავყაროთ `push_back()` მეთოდით. მაგალითად:

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
const int M(4), N(5);
int main (){
    vector< vector <int> > intMatrix(M);
   // ყოველ intMatrix[i] ვექტორში რიცხვების ჩაწერა
   for(int i=0; i<intMatrix.size(); i++)
      for (int j=0; j<N; j++)
          intMatrix[i].push_back(i+j);
   // intMatrix ვექტორის სტრიქონ-სტრიქონ ბეჭდვა
   for(int i=0; i<intMatrix.size(); i++)
   {
       for (int j=0; j<intMatrix[i].size(); j++)
       {
          cout<<setw(4)<<intMatrix[i][j];
       }
       cout<<endl;
   }
   cout<<endl;
   return 0;
}
```

ტიპის განმსაზღვრელი typedef

C++ შესაძლებლობას აძლევს პროგრამისტს განსაზღვროს საკუთარი ტიპი **typedef** შეტყობინების საშუალებით. **typedef** შეტყობინების ზოგადი ფორმა ასეთია:

typedef *განაცხადი ტიპზე;*

სადაც *განაცხადი ტიპზე* იგივეა, რაც განაცხადი ცვლადებზე, იმ განსხვავებით, რომ ცვლადის სახელის ნაცვლად გამოიყენება ტიპის სახელი. მაგალითად:

```
typedef int count;
```

განსაზღვრავს ახალ ტიპს `count`, რომელიც იგივეა რაც `int`. ასე, რომ განაცხადი:

```
count flag;
```

იგივეა რაც

```
int flag;
```

typedef –ის გამოყენებით ჩვენ პროგრამაში შეგვიძლია განვსაზღვროთ შემდეგი ორი ტიპი:

```
typedef vector<int> row;
typedef vector< row > intMatrix;
```

მაშინ პროგრამა მიიღებს სახეს:

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
typedef vector<int> row;
typedef vector< row > matrix;
```

```
const int M = 4, N = 5;
int main (){
   row R(N);
   matrix intMatrix(M, R);
   for(int i=0; i<M; i++)
      for (int j=0; j<N; j++)
          intMatrix[i][j]= i+j;
   for(int i=0; i<intMatrix.size(); i++)
   {
       for (int j=0; j<intMatrix[i].size(); j++)
       {
          cout<<setw(4)<<intMatrix[i][j];
       }
       cout<<endl;
   }
   cout<<endl;
   return 0;
}
```

ორგანზომილებიანი ვექტორის გადაცემა ფუნქციაში

განხილულ საილუსტრაციო მაგალითში ჩვენ ვქმნით ორგანზომილებიან ვექტორს და შემდეგ ვბეჭდავთ მას ცხრილის სახით ვექტორ–ვექტორ. ამოცანის კოდი მთლიანად მოთავსებულია `main` –ში, რაც C++ ენაში პროგრამირების კარგ სტილად არ ითვლება. დროა პროგრამა ავაგოთ C++ -ისთვის ჩვეულ სტილში: დავწეროთ ვექტორის შევსების ფუნქცია, ვექტორის ბეჭდვის ფუნქცია, ხოლო `main` –ში მოვახდინოთ ამ ფუნქციების გამოძახება:

ვექტორში რიცხვების ჩაწერის ფუნქცია ასე გამოიყურება:

```
void fillMatrix(vector< vector<int> > & a){
   for(int i=0; i<M; i++)
      for(int j=0; j<N; j++)
          a[i][j]= i+j;
}
```

ვექტორის ელემენტების ბეჭდვის ფუნქციას აქვს სახე

```
void printMatrix(const vector< vector<int> > &a){
   for(int i=0; i<a.size(); i++)
   {
       for (int j=0; j<a[i].size(); j++)
           cout<<setw(4)<<a[i][j];
       cout<<endl;
   }
   cout<<endl;
}
```

ხოლო ჩვენი პროგრამა ასე გადაიწერება:

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
void fillMatrix(vector< vector<int> > & a);
void printMatrix(const vector< vector<int> > &a);
const int M = 4, N = 5;
int main (){
   vector <int> row(N);
   vector< vector <int> > intMatrix(M, row);
```

```
   fillMatrix( intMatrix );
   printMatrix( intMatrix );
   return 0;
}
void fillMatrix(vector< vector<int> > & a){
   for(int i=0; i<M; i++)
      for (int j=0; j<N; j++)
          a[i][j]= i+j;
}
void printMatrix(const  vector< vector<int> > &a){
   for(int i=0; i<a.size(); i++)
   {
       for (int j=0; j<a[i].size(); j++)
           cout<<setw(4)<<a[i][j];
       cout<<endl;
   }
   cout<<endl;
}
```

თუ კი ვისარგებლებთ **typedef** შეტყობინებით და როგორც ადრე შემოვიღებთ ტიპის ახალ სახელს (ახალ ტიპს) ორგანზომილებიანი ვექტორის ელემენტი–ვექტორისთვის

```
typedef vector<int> row;
```

და თვითონ ორგანზომილებიანი ვექტორისთვის

```
typedef vector< row > matrix;
```

მაშინ ჩვენი პროგრამა გამარტივებული სახით ჩაიწერება

```
#include <iostream>
#include <iomanip>
#include <vector>
using namespace std;
typedef vector<int> row;
typedef vector< row > matrix;
const int M = 4, N = 5;
void fillMatrix(matrix & a);
void printMatrix(matrix a);
int main (){
   row R(N);
   matrix intMatrix(M, R);
   fillMatrix( intMatrix );
   printMatrix( intMatrix );
   return 0;
}
void fillMatrix(matrix & a)
{
   for(int i=0; i<M; i++)
      for (int j=0; j<N; j++)
          a[i][j]= i+j;
}
void printMatrix(matrix a){
   for(int i=0; i<a.size(); i++)
   {
       for (int j=0; j<a[i].size(); j++)
           cout<<setw(4)<<a[i][j];
       cout<<endl;
```

```
    }
    cout<<endl;
}
```

მოქმედება ორგანზომილებიან ვექტორზე

აუცილებელი არაა ორგანზომილებიანი ვექტორის ელემენტები – ვექტორები იყვნენ ერთი ზომისა. შეგვიძლია შემოვიღოთ ისეთი ორგანზომილებიანი ვექტორები, რომლებიც შეესაბამება სხვადასხვა ფორმის ცხრილებს, მაგალითად სამკუთხა ან ე.წ. კბილოვან ცხრილებს, რაც ორგანზომილებოანი ვექტორის უპირატესობაა.

შემდეგ მაგალითში შევქმნით სამკუთხა ორგანზომილებიან ვექტორს `triangularMatrix`:

```
#include <iostream>
#include <vector>
using namespace std;
const int M =4;
int main()
{
    vector<vector <int> > triangularMatrix;
    for(int i = 0; i < M; i++)
    {
         vector<int> row;
         triangularMatrix.push_back(row);
    }
    for(int i=0; i<M; i++)
       for (int j=0; j<= i; j++)
           triangularMatrix[i].push_back(10*i+j);

    for(int i=0; i< triangularMatrix.size(); i++)
    {
        for (int j=0; j<triangularMatrix [i].size(); j++)
        {
           cout << triangularMatrix[i][j]<<"\t";
        }
        cout<<endl;
    }

    cout<<endl;
    return 0;
}
```

პროგრამის შესრულების შედეგია:

```
0
10      11
20      21      22
30      31      32      33
Press any key to continue . . .
```

შემდეგ მაგალითში კი იქმნება ორგანზომილებიანი ვექტორი `juggedMatrix`, რომლის ელემენტები – ვექტორები შეიცავენ სხვადასხვა რაოდენობის რიცხვებს:

```
#include <iostream>
#include <vector>
using namespace std;
const int M =4;
int main()
{
    vector<vector <int> > juggedMatrix;
    for(int i = 0; i < M; i++)
    {
```

```
        vector<int> row;
        juggedMatrix.push_back(row);
    }
    int count, number;
    for(int i=0; i<M; i++){
       cout<<"ramdeni ricxvia "<<i+1<<" veqtorSi? ";
        cin>> count;
        cout<<"SemoitaneT "<<count<<" mTeli ricxvi  ";
       for(int k=0; k<count; k++)
       {
             cin >> number;
            juggedMatrix[i].push_back(number);
        }
    }
    cout<<"\nSevqmeniT organzomilebiani veqtori \n\n";
    for(int i=0; i< juggedMatrix.size(); i++)
    {
        for (int j=0; j< juggedMatrix[i].size(); j++)
        {
           cout << juggedMatrix[i][j]<<"  ";
        }
        cout<<endl;
    }
    cout<<endl;
    return 0;
}
```

პროგრამის შესრულების შედეგია:

```
ramdeni ricxvia 1 veqtorSi? 3
SemoitaneT 3 mTeli ricxvi  110 5 -67
ramdeni ricxvia 2 veqtorSi? 5
SemoitaneT 5 mTeli ricxvi  2 7 9 0 28
ramdeni ricxvia 3 veqtorSi? 2
SemoitaneT 2 mTeli ricxvi  91 -4
ramdeni ricxvia 4 veqtorSi? 1
SemoitaneT 1 mTeli ricxvi  356
SevqmqniT organzomilebiani veqtori
110  5  -67
2  7  9  0  28
91  -4
356
Press any key to continue . . .
```

განვიხილოთ ორგანზომილებიან ვექტორზე მოქმედების რამდენიმე მაგალითი. ამჯერად საქმე გვექნება ე.წ. მართკუთხა ორგანზომილებიან ვექტორებთან, რომლებიც ერთი ზომის ვექტორებს შეიცავენ.

ამოცანის პირობაში ჩანაწერი A(M x N) ნიშნავს, რომ A ორგანზომილებიანი ვექტორი შეიცავს M ვექტორს, თითოეულს N ელემენტით.

ამოცანა 1. გაცვალეთ ორგანზომილებიანი Matrix(M x N) ვექტორის პირველი და ბოლო ვექტორი (ანუ სტრიქონი).

```
#include <iostream>
#include <iomanip>
#include <vector>
```

```
using namespace std;
const int M = 3, N = 4;
void fillMatrix(vector< vector <int> > & x);
void Transform(vector< vector <int> > & x);
void printMatrix(vector< vector <int> > x);
int main()
{
   vector <int> Row(N);
   vector< vector <int> > Matrix(M, Row);
   fillMatrix(Matrix);
   Transform(Matrix);
   cout<<"\nAfter transform matrix is:\n\n";
   printMatrix(Matrix);
   return 0;
}
void fillMatrix(vector< vector <int> > & x)
{
      for(int i=0; i<x.size(); i++)
        for(int j=0; j<x[i].size(); j++)
           cin >> x[i][j];
}
void printMatrix(vector< vector <int> > x){
   for(int i=0; i< x.size(); i++)
   {
      for(int j=0; j< x[i].size(); j++)
         cout <<setw(5)<<x[i][j];
      cout<<endl;
   }
   cout<<endl;
}
void Transform(vector< vector <int> > & x)
{
   swap( x[0], x[M-1] );
}
```

პროგრამის შედეგი:

```
1 2 3 4
5 6 7 8
9 10 11 12

After transform matrix is:

    9   10   11   12
    5    6    7    8
    1    2    3    4

Press any key to continue . . .
```

ამოცანა 2. ჩაწერეთ ორგანზომილებიან V(M x N) , M = 5, N = 3 ვექტორში შემთხვევითი რიცხვები [-12,42] შუალედიდან, შემდეგ იპოვეთ ვექტორის კენტი ელემენტების ჯამი და დაბეჭდეთ.

```
#include <iostream>
#include <iomanip>
#include <ctime>
#include <vector>
using namespace std;
const int M = 5, N = 3;
void fillMatrix(vector< vector<int> > & a);
void printMatrix(vector< vector<int> > a);
bool Odd(int x);
```

```
int sumOdds(vector< vector<int> > a);
int main()
{
   vector<int> row(N);
   vector< vector<int> > V(M, row);
   fillMatrix( V );
   printMatrix( V );
   int S = sumOdds( V );
   cout<<"Sum of odd elements = "<<S<<endl;
   return 0;
}
void fillMatrix(vector< vector<int> > & a)
{
      srand(time(NULL));
      for(int i=0; i<M; i++)
         for(int j=0; j<N; j++)
            a[i][j] = rand()%55 - 12;
}
bool Odd(int x)
{
   return x%2 == 1;
}
int sumOdds(vector< vector<int> > a)
{
   int sum =0;
   for(int i=0; i<M; i++)
      for(int j=0; j<N; j++)
         if( Odd(a[i][j]) )
            sum += a[i][j];
   return sum;
}
void printMatrix(vector< vector<int> > a)
{
   for(int i=0; i<M; i++)
   {
      for(int j=0; j<N; j++)
         cout <<setw(9)<<a[i][j];
      cout<<endl;
   }
   cout<<endl;
}
```

პროგრამა დაბეჭდავს

```
       29       30       -3
       33       17       37
       26       31        0
       32       28       28
        4       40       -6

Sum of odd elements = 147
Press any key to continue . . .
```

ამოცანა 3. A(M x N) ორგანზომილებიან ვექტორში ნამდვილი რიცხვებია. დაწერეთ ფუნქცია, რომელიც დააბრუნებს ვექტორის უდიდესი და უმცირესი ელემენტების ჯამს. რიცხვები ვექტორში ჩაწერეთ `reals.in` ფაილიდან. ამისათვის ასევე დაწერეთ ფუნქცია. პროგრამაში გამოიყენეთ ორივე ფუნქცია და დაბეჭდეთ შედეგი.

```cpp
#include <iostream>
#include <fstream>
#include <iomanip>
#include <vector>
using namespace std;
const int M = 3, N = 4;
typedef vector<double> row;
typedef vector< row > matrix;
void fillMatrix(matrix & x);
void printMatrix(matrix x);
double maxminSum(matrix x);
int main(){
   row R(N);
   matrix A(M, R);
   fillMatrix(A);
   printMatrix(A);
   double sum = maxminSum(A);
   cout<<"Udidesi da umciresi ricxvebis jami = "<<sum<<endl;
   return 0;
}
void fillMatrix(matrix & x)
{
    ifstream fin("reals.in");
    for(int i=0; i<x.size(); i++)
       for(int j=0; j<x[i].size(); j++)
          fin >> x[i][j];
}
void printMatrix(matrix x){
   for(int i=0; i< x.size(); i++)
   {
      for(int j=0; j< x[i].size(); j++)
         cout <<setw(10)<<x[i][j];
      cout<<endl;
   }
   cout<<endl;
}
double maxminSum(matrix x){
   double min, max;
   min = max = x[0][0];
   for(int i=0; i<x.size(); i++)
      for(int j=0; j<x[i].size(); j++)
         if(x[i][j] > max)
            max = x[i][j];
         else if(x[i][j] < min)
                  min = x[i][j];
   return max + min;
}
```

პროგრამის შესრულების შედეგია:

reals.in ფაილი	გამოტანის ეკრანი
5.8 12.25 -33.75 0.0 250.5 150.45 10.0 -150.25 324.5 91.0 0.0 -1.75	5.8 12.25 -33.75 0 250.5 150.45 10 -150.25 324.5 91 0 -1.75 Udidesi da umciresi ricxvebis jami = 174.25 Press any key to continue . . .

თავი 11: მონაცემთა ახალი ტიპის შექმნა - კლასი

- მონაცემთა ახალი ტიპების შექმნა
- განაცხადი კლასზე
- ტერმინოლოგიური შეთანხმებები
- კონსტრუქტორები და განაცხადები ობიექტებზე
- პოინტერი კლასის ობიექტზე

მონაცემთა ახალი ტიპების შექმნა

პროფესიონალი პროგრამისტები პროგრამებს წერენ რეალური სამყაროს პრობლემების გადასაჭრელად. მაგალითად, რაიმე დაწესებულებაში დასაქმებულების მონაცემების დასამუშავებლად, რაიმე მოწყობილების მოდელის შესადგენად და ფუნქციონირების სიმულირებისთვის და ა. შ..

ასეთ შემთხვევებში ძალიან მოსახერხებელია, რომ კონკრეტული ამოცანის დაპროგრამების პროცესში ყოველი ობიექტისთვის, რომლის შესწავლას და დამუშავებასაც ვაპირებთ, შევქმნათ შესაბამისი პროგრამული ეკვივალენტი (წარმოდგენა) და განვახორციელოთ საჭირო მოქმედებები მათზე.

ჩვენ უკვე განვიხილეთ მონაცემთა უმარტივესი ტიპები (**int, double, char, bool** და სხვა), აგრეთვე შედარებით რთული ტიპები (**string, vector**). ნებისმიერი მათგანისთვის, განაცხადი ამომწურავ ინფორმაციას აძლევს კომპილერს: რა მოცულობის მეხსიერება გამოყოს ამ ტიპის ყოველი წარმომადგენლისთვის (ცვლადისთვის ან ობიექტისთვის), რა მოქმედებები არის ნებადართული ამ ტიპის წარმომადგენლებისთვის და რა საწყისი ინფორმაცია იწერება მათში (ინიციალიზაციის პროცესში).

დაწყებული **string** და **vector** ტიპებიდან, როგორც წესი, აღარ იყენებენ ტერმინს ცვლადი. იგივეა სამართლიანი მომხმარებლის მიერ შექმნილი ტიპებისთვის, რაც კლასის საშუალებით განხორციელდება. მიზეზი ძალიან მარტივია: მარტივი ტიპი ნიშნავს, რომ ამ ტიპის ცვლადი არის რაღაც განუყოფელი, მარტივი, რომელიც არ არის შედგენილი სხვა, უფრო მარტივი ტიპის ცვლადებისგან. ეს რაღაც ძალიან გავს ქიმიური ელემენტების სიტუაციას, სადაც არის უმარტივესი არაორგანული ნივთიერებები და არის შედგენილი, ორგანული ნაერთები.

string და **vector** ტიპების თითოეული წარმომადგენელი, რომელსაც უკვე ობიექტს ვუწოდებთ, შედგება უფრო მარტივი ტიპის არაერთი ცვლადისგან. მაგალითად, შემდეგი განაცხადით

```
vector<int> v(10);
```

გაკეთებული v ობიექტი შედგება 10 მთელი ტიპის ცვლადისგან. მაგრამ **string** და **vector** ტიპებიც მარტივია იმ აზრით, რომ მათი ობიექტები შედგება მხოლოდ ერთი რომელიმე ტიპის რამდენიმე ცვლადისგან (ან რამდენიმე ობიექტისგან, როგორც ვექტორების ვექტორის შემთხვევაში).

სამაგიეროდ, ამ ორ ტიპიდან თითოეულს (კლასს) უკვე აქვს შესაძლებლობა, რომ მათმა ობიექტებმა მიმართონ სპეციალურ ფუნქციებს, რომლებსაც მეთოდებს ვუწოდებთ და რომლებიც სპეციალურად ამ კლასისთვისაა შექმნილი ან იმპლემენტირებული.

C++ ენა პროგრამისტს აძლევს საშუალებას, რომ მან შექმნან პრაქტიკულად ნებისმიერი რეალური ობიექტის შესაბამისი პროგრამული ობიექტი, რომელიც შედგება სხვა, უფრო მარტივი ობიექტებისგან, ან ამავე კლასის ობიექტის მისამართებისგან, და რომელშიც გარდა ობიექტებისა, წევრებად შედის ფუნქციები (მეთოდები). ისევე როგორც ჩვენთვის ნაცნობი ტიპების შემთხვევაში, კლასის ობიექტი კომპილერს აცნობებს თუ:

- რა ზომის მეხსიერება სჭირდება ობიექტს;
- რა ინფორმაციის შენახვა შეუძლია მას;
- რა მოქმედებები შეიძლება განხორციელდეს მათზე.

განაცხადი კლასზე

როდესაც C++ ენაში ახალი კლასი იქმნება, როგორც წესი, კლასზე განაცხადი კეთდება თავსართ `.h (header)` ფაილში, ხოლო კლასის იმპლემენტაცია, ანუ სრული განსაზღვრა ხდება `.cpp` ფაილში, როგორც ეს გავაკეთეთ მე–13 ლაბორატორიულ მეცადინეობაზე.

სიმარტივისთვის, ზოგჯერ განაცხადს და იმპლემენტაციას მოვათავსებთ იმავე ფაილში. რომელშიც არის მთავარი `main()` ფუნქცია.

ცხადია, ჯერ კეთდება განაცხადი კლასზე, შემდეგ შეგვიძლია შევქმნათ ამ კლასის ობიექტები.

კლასზე განაცხადის გასაკეთებლად ვიყენებთ გასაღებ სიტყვას **`class`**, რომელსაც მოყვება კლასის (ახალი ტიპის) სახელი, ხოლო შემდეგ ფიგურული ფრჩხილების წყვილში ვათავსებთ განაცხადებს წევრ ცვლადებზე და ფუნქციებზე.

როგორც სხვა ნებისმიერი განაცხადი, კლასის განაცხადი უნდა დამთავრდეს წერტილ-მძიმით. მაგალითად:

```
class Circle
{
  public:
     double radius;
     Circle(double value);
     Circle();
     double Area();
     double Perimeter();
};
```

ყურადღება მივაქციოთ, რომ ამ კლასის ორ მეთოდს არ გააჩნია დასაბრუნებელი მნიშვნელობა: ესაა ორი კონსტრუქტორი. უფრო მეტი, ამ მეთოდების სახელი ემთხვევა კლასის სახელს. დასაბრუნებელი მნიშვნელობის მითითება არ არის საჭირო კიდევ ერთი ტიპის ფუნქციისთვის, რომლებსაც მეორე სემესტრში შევისწავლით დაწვრილებით, და რომლებსაც დესტრუქტორები ეწოდებათ.

კლასის მეთოდების იმპლემენტაციას, როდესაც ეს კლასის გარეთ ხდება (და ასეც უნდა იყოს, გარდა უმარტივესი შემთხვევებისა), აქვს თავისი წესი: იწერება კლასის სახელი, უშუალოდ მას მოყვება ოთხი წერტილი და შემდეგ მეთოდის განსაზღვრა (იმპლემენტაცია). მაგალითად:

```
Circle::Circle(double value){
    radius = value;
}
```

კლასის ობიექტზე განაცხადის გაკეთებისთვის საჭიროა კლასში დამუშავებული (ინკაფსულირებული) იყოს ისეთი კონსტრუქტორი ან კონსტრუქტორები, რომლებიც საჭიროა ობიექტების ინიციალიზაციისთვის.

ტერმინოლოგიური შეთანხმებები

C++ ენაში ყველაფერი უნდა გავაკეთოთ მხოლოდ პროგრამირების კარგი სტილის შესაბამისად. კერძოდ, ნებისმიერი სახელი მაქსიმალურად შინაარსიანი უნდა იყოს და მიგვანიშნებდეს შესაბამისი კლასის, ობიექტის თუ ცვლადის დანიშნულებაზე. მაგალითად, `Cat, Rectangle, Employee` არის კარგი სახელები კლასებისთვის. რაც შეეხება კლასის წევრ ცვლადებს, ბევრი

პროგრამისტი იყენებს მათთვის პრეფიქსს `its`. მაგალითად, `itsAge, itsWeight, itsSpeed, itsRadius`.
ამით ხაზი ესმება განსხვავებას წევრ ცვლადებსა და არაწევრ ცვლადებს შორის. სხვა პროგრამისტები იყენებენ განსხვავებულ პრეფიქსებს. ზოგი წერს `myAge, myWeight, mySpeed, myRadius`. ზოგიც უბრალოდ იყენებს ასოს `m`, ან `m_` (წევრებისთვის) და წერენ `mAge, mWeight, mSpeed, mRadius` ან `m_age, m_weight, m_speed, m_radius`. კლასების სახელებს პროგრამისტების უმეტესობა წერს მთავრული (დიდი) ლათუნური ასოთი.

ყველა გავრცელებული სტილის აღნიშვნა პრაქტიკულად შეუძლებელია. მით უმეტეს, რომ ძლიერ და ზრდად კომპანიებში მიღებული ტენდენციაა, რომ სტილებთან დაკავშირებით იქონიონ შინაური სტანდარტები. ეს აადვილებს ასეთი კომპანიის დეველოპერების საქმეს - ისინი ადვილად კითხულობენ თავისი კოლეგების დაწერილ კოდს. ეს ნიშნავს, რომ C++ ენაზე მომუშავე პროგრამისტები მზად უნდა იყონ იმუშავონ განსხვავებულ სტილებთან.

კონსტრუქტორები და განაცხადები ობიექტებზე

C++ ენის კარგი სტილის ერთ-ერთი თანამედროვე ტენდენცია გულისხმობს, რომ, თუ ეს შესაძლებელია, ცვლადის/ობიექტის ინიციალიზაცია სასურველია მოხდეს განაცხადის გაკეთბის მომენტში.

ამით აიხსნება ის დიდი ყურადღება, რაც ეთმობა კონსტრუქტორების განსაზღვრის საკითხს.

ფორმალურად, კონსტრუქტორის განსაზღვრა არაა სავალდებულო. მაგალითად, თუ ზემოთ აღწერილ წრის კლასის განაცხადს გადავაკეთებდით ასე:

```
class Circle
{
  public:
     double radius;
     double Area();
     double Perimeter();
};
```

მაშინ კომპილერი თვითონ შექმნის უპარამეტრო კონსტრუქტორს, რომელიც შექმნის განუსაზღვრელ რადიუსიან წრეს, ხოლო განაცხადის გაკეთების მომენტში ობიექტის ინიციალიზება შეუძლებელია.

მაგალითად, მთავარ ფუნქციაში, ჩანაწერი

```
Circle c;
```

ერთადერთი გზაა ამ კლასის ობიექტზე განაცხადის გაკეთებისა. თუ გვინდა განსაზღვრული მნიშვნელობა მივანიჭოთ რადიუსს, უნდა შევქმნათ შესაბამისი შეტყობინება, მაგალითად

```
c.radius = 11;
```

თუ გვინდა ინიციალიზაცია მოხდეს განაცხადის გაკეთების მომენტში, როგორც ეს კეთდებოდა მარტივი ცვლადებისთვის

```
double x(11.3);
int n(1853);
```

და ა. შ., კლასში უნდა გვქონდეს შესაბამისი კონსტრუქტორი, რომელიც (რადგან კონსტრუქტორის სახელი ემთხვევა კლასის სახელს) გააქტიურდება კლასის სახელისა და ობიექტის შემდეგ ფრჩხილების გამოჩენისთანავე. მაგალითად, თუ წრის კლასზე განაცხადს მივცემთ სახეს:

```
class Circle
{
  public:
     double radius;
     Circle(double value);
     double Area();
     double Perimeter();
};
```

და კონსტრუქტორის იმპლემენტაციას მოვახდენთ შემდეგნაირად:

```
Circle::Circle(double value)
{
    radius = value;
}
```

მაშინ განაცხადი

```
    Circle c(111);
```

ხდება ვალიდური: იგი შექმნის წრის ობიექტს, სახელად c, და ამ წრის რადიუსს გაუტოლებს 111-ს.

ყურადღება მივაქციოთ იმ გარემოებას, რომ ჩვენ უნდა შევქმნათ ყველა ის კონსტრუქტორი, რომლის გამოყენებასაც ვაპირებთ. ბოლო შემთხვევაში, იმის გამო რომ ჩვენ აღვწერეთ ცხადი (ანუ პარამეტრიანი) კონსტრუქტორი, კომპილერი აღარ ქმნის უპარამეტროს და ამიტომ ასეთი შეტყობინება:

```
    Circle x;
```

უკვე შეცდომაა. ამიტომ, თუ გვინდა ასეთი განაცხადიც გამოვიყენოთ, კლასის განაცხადში უნდა იყოს უპარამეტრო კონსტრუქტორიც, როგორც ეს იყო სულ პირველ მაგალითში.

კონსტრუქტორების თემა იმდენად მდიდარია და მნიშვნელოვანი, რომ ასეთ მარტივ მაგალითშიც შეგვიძლია განვავითაროთ და შევქმნათ განსხვავებული და შინაარსიანი კონსტრუქტორები.

დავუშვათ, გვინდა რომ პროგრამაში განაცხადი გავაკეთოთ წრის ობიექტებზე, მათგან ზოგიერთის მონაცემებს შევიყვანთ ფაილიდან, ზოგიერთის მონაცემს კლავიატურიდან, ხოლო ზოგიერთის ინიციალიზებას მოვახდენთ განაცხადის გაკეთბის მომენტში, პარამეტრიანი კონსტრუქტორით.

```
#include <iostream>
#include <fstream>
using namespace std;

class Circle
{
  public:
    double radius;
    Circle(double value);
    Circle();
    Circle(ifstream & );
    Circle(istream & );
    double Area();
    void showCircle(void);
};
Circle :: Circle(double value){
```

```
    radius = value;
}
Circle :: Circle() {radius = 0.0; }
Circle :: Circle(ifstream & x ){
   x >> radius;
}
Circle :: Circle(istream & x ){
   cout<<"ShemoitaneT wris radiusi: ";
   x >> radius;
}

double Circle :: Area(){
   return 3.1416 * radius * radius;
}
void Circle :: showCircle(){
   cout << "Radius: " << radius <<endl;
   cout << "Area: " << Area() <<endl;
   cout << endl;
}

int main()
{
   Circle C1(2.4);
   C1.showCircle();

   Circle C2;
   C2.showCircle();

   ifstream fin("Circles.txt");
   cout<<"\nInformacia radiusis Sesaxeb Semodis failidan\n\n";

   Circle A(fin);
   A.showCircle();

   Circle B(fin);
   B.showCircle();

   Circle F(cin);
   F.showCircle();

   system("pause");
   return 0;
}
```

Circles.txt	გამოტანის ეკრანი
25 2 123 15.1 9.8	Radius: 2.4 Area: 18.0956 Radius: 0 Area: 0 Informacia radiusis Sesaxeb Semodis failidan Radius: 25 Area: 1963.5

	Radius: 2 Area: 12.5664 ShemoitaneT wris radiusi: 1 Radius: 1 Area: 3.1416 Press any key to continue . . .

წვდომა კლასის წევრებზე, ინტერფეისი

ყველა წევრი, რაც წერია გასაღები სიტყვა **public**: -ის შემდეგ, არის ღია ნებიმიერი ობიექტისთვის, რაც ნიშნავს, რომ

```
Circle c(111);
```

შეტყობინების შემდეგ ნებადართულია შემდეგი სახის შეტყობინებები:

```
...
c.radius = 21;
...
cout << c.Area() << endl;
```

ასეთი რამე მხოლოდ ძალიან მცირე ზომის კლასებისთვის არის დასაშვები და ისიც არა ყოველთვის.

დიდი ზომის, რთული კლასის შემთხვევაში, მეთოდების და ველების უმეტესობა ტექნიკური ხასიათისაა, და ძირითადი მოქმედებები, რაც უნდა განხორციელდეს კლასზე, არ მოითხოვს რომ მისი ყველა წევრი იყოს ღია.

მაგალითად, ეს იმის ანალოგიურია, რომ პლეერის ინტერფეისი შედგება რამდენიმე ღილაკისგან (გადამრთველები და მარეგულირებლები) და ეს საკმარისია მისი გამოყენებისთვის, მაშინ როდესაც ამ ღილაკების გარდა იგი ბევრ სხვა დეტალს შეიცავს, რომლებზეც წვდომა არ გვაქვს. ანუ, პლეერის გამართული და კორექტული ფუნქციონირებისთვის უმჯობესია, რომ მისი შიგნეულობა დახურულია და მის მახასიათებლებზე წვდომა ხდება ღილაკების საშუალებით. ამის ანალოგიურად, კლასის გამართული ფუნქციონირებისთვის აუცილებელია, რომ მის წევრებზე წვდომა ხორციელდებოდეს არა უშუალოდ (მაგ. `c.radius = 21;`), არამედ გაშუალებულად, სპეციალურად ამისთვის შექმნილი მეთოდების საშუალებით. ღია წევრები წარმოადგენენ კლასის ინტერფეისს (იგივეს რაც ღილაკია პლეერისთვის).

ამ საკითხებს შემდეგში დიდი ყურადღება დაეთმობა ობიექტზე-ორიენტირებული პროგრამირების კურსში.

პოინტერი კლასის ობიექტზე

რაც უფრო დიდი არის კლასი, უფრო მეტი მნიშვნელობა ენიჭება მისი ობიექტების პოინტერების გამოყენებას, რადგან პოინტერი იკავებს მხოლოდ 4 ბაიტს, ხოლო ობიექტმა შეიძლება მეგაბაიტები დაიკავოს. უფრო კონკრეტულები რომ ვიყოთ, განვიხილოთ ასეთი ამოცანა. ვთქვათ ფაილში `"volcanoes.txt"` წერია სამი ვულკანის მონაცემი – სახელი, სიმაღლე და ადგილმდებარეობა. მაგალითად :

```
vezuv                   1277      italy
kluchevskaia_sopka      4750      kamchatka
fuziama                 3776       japan
```

ჩვენი ამოცანაა ვიპოვოთ ყველაზე მაღალი ვულკანი და დავბეჭდოთ მისი მონაცემები.

ამ ამოცანის ამოსახსნელად ჯერ გამოვიყენოთ იგივე გზა, რაც გვქონდა პრაქტიკულ მეცადინეობაზე, ოღონდ გავითვალისწინოთ აგრეთვე ფაილიდან ობიექტების კონსტრუირების ზემოთ აღნიშნული შესაძლებლობა. შემდეგ, იგივე ამოცანა გავაკეთოთ უკვე პოინტერის გამოყენებით და შევაფასოთ განსხვავება.

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

class Volcano
{
  public:
    string name;
    int height;
    string location;
    Volcano(ifstream & );
     void printVolcano(void);
};
Volcano :: Volcano(ifstream & x ) {
   x >> name >> height >> location;
}
void Volcano :: printVolcano(){
         cout<< name <<", its height is " << height <<" meters"<<endl
                <<"It is located in "<< location<<endl;
       }
int main()
{
   ifstream fin("volcanoes.txt");

   Volcano A(fin);
   Volcano B(fin);
   Volcano C(fin);
   Volcano maxVolcano(A);

   if(maxVolcano.height < A.height)
           maxVolcano = A;
   if(maxVolcano.height < B.height)
           maxVolcano = B;
   cout<<"Maximum height has volcano\n";
   maxVolcano.printVolcano();
   system("pause");
   return 0;
}
```

შედეგად ვღებულობთ:

```
Maximum height has volcano
kluchevskaia_sopka, its height is
4750 meters
It is located in kamchatka
Press any key to continue . .  .
```

ამ პროგრამაში, ჩვენ სამჯერ (ერთხელ შეტყობინებაში `Volcano maxVolcano(A);` ორჯერ `if` შეტყობინების ტანში) მივანიჭეთ ობიექტს სხვა ობიექტის მნიშვნელობა და განაცხადი გავაკეთეთ 4 ობიექტზე. ობიექტები მინიჭება შრომატევადი ოპერაციაა, რაც უფრო გრძელია ობიექტი, მით უფრო მეტი დრო არის საჭირო მის შესასრულებლად, და მით უფრო მეტი ადგილია საჭირო მეხსიერებაში მისი განთავსებითვის.

ახლა განვიხილოთ განსხვავებული მიდგომა, რომ განაცხადი გავაკეთოთ სამ ობიექტზე, და დავიმახსოვროთ არა ობიექტი, არამედ მისი მისამართი. მისამართიდან ობიექტის წევრებთან წვდომა ხდება ისრების საშუალებით. მთავარ ფუნქციას ექნება სახე:

```
int main()
{
   ifstream fin("volcanoes.txt");

   Volcano A(fin);
   Volcano B(fin);
   Volcano C(fin);
   Volcano* addressOfMmax(&A);

   if(addressOfMmax->height < B.height)
           addressOfMmax = &B;
   if(addressOfMmax->height < C.height)
           addressOfMmax = &C;
   cout<<"Maximum height has volcano\n";
   addressOfMmax->printVolcano();
   system("pause");
   return 0;
}
```

შედეგი იგივეა რაც ზემოთ. განსხვავება ისაა, რომ ჩვენ განაცხადი გავაკეთეთ სამ ობიექტზე და ერთ პოინტერზე (ტიპის მიუხედავად, ნებისმიერი პოინტერი მეხსიერებაში იკავებს მხოლოდ 4 ბაიტს), და სამჯერ პოინტერს მივანიჭეთ სხვა პოინტერის მნიშვნელობა. ოთხბაიტიანი ცვლადების მნიშვნელობების მინიჭება გაცილებით ეკონომიურია, ვიდრე ობიექტებისთვის მნიშვნელობების მინიჭება. მიდგომებს შორის განსხვავება განსაკუთრებით თვალსაჩინო ხდება დიდ ამოცანებში, სადაც დიდი რაოდენობა დიდი ზომის ობიექტები მონაწილეობს. ასეთ შემთხვევებში, როცა შესაძლებელია, ჯობია ვიმუშავოთ ობიექტების მისამართებთან, იმ პირობით რომ დაცული იქნება უსაფრთხოების გარკვეული წესები.