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 კომენრტარის 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, გააუმჯობესებს 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:

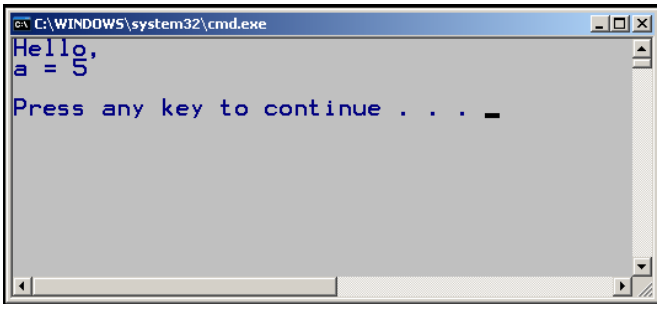
```
    int a(5);
```

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.

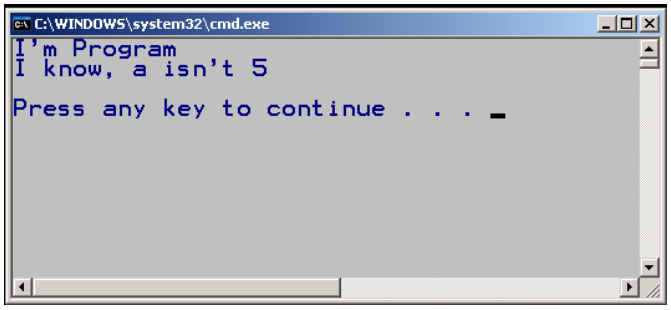
შემდეგ რამდენიმე სტრიქონს იკავებს **განშტოების** შეტყობინება **if**(a == 5) შესაძლო შედეგებითურთ. მისი შესრულების წესი მარტივია: შემოწმდება პირობა a==5 (უუდრის?) და თუ პირობა ჭეშმარიტია (სწორია), შესრულდება პირველ ფიგურულ ფრჩხილებში მოთავსებული ყველა შეტყობინება (ანუ, თუ a ტოლია 5-ის, შესრულდება cout<<"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;
```

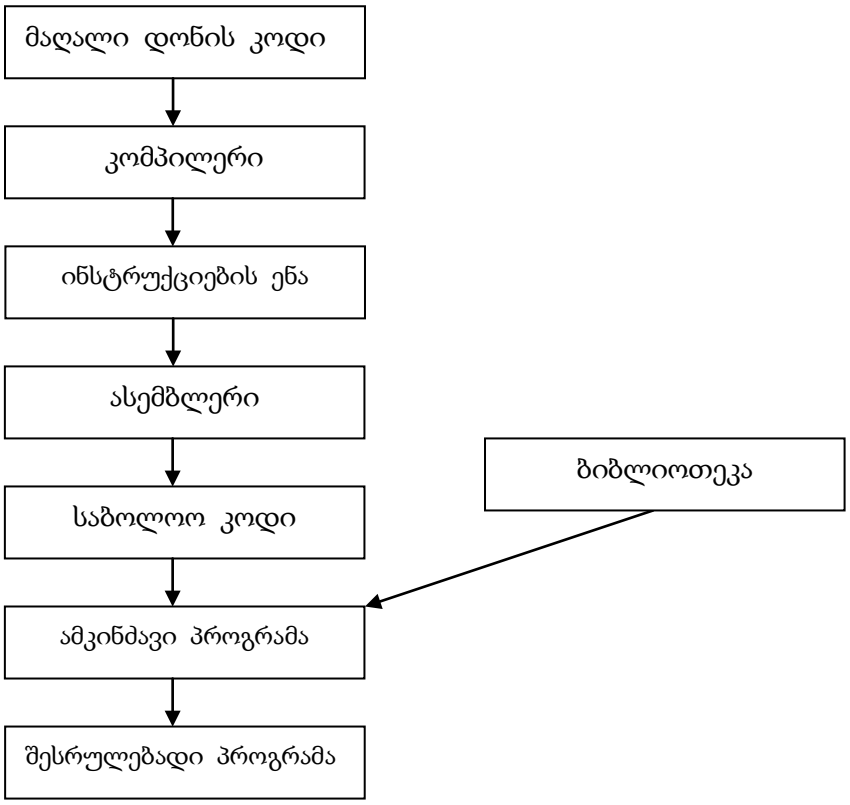
პროგრამა მარტივი უნდა იყოს. ზოგადი წესები ასეთია:

- 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.