თემა 3. დაპროგრამების ენის ძირითადი ცნებები

* დაპროგრამების ენის ელემენტები:

მონაცემები: ტიპი; ცვლადი – მისი სახელი, მნიშვნელობა, წარმოდგენა; ლიტერალი; მუდმივი; ობიექტი

* ელემენტარული მონაცემთა ტიპები

მთელი ტიპი, ჩამოთვლითი, სიმბოლური და ლოგიკური ტიპები

* გამოსახულება

პროგრამა შედგება ორი მთავარი ნაწილისგან: მონაცემებისა და ინსტრუქციებისგან. პროგრამირების თანამედროვე ხედვა ხსნის ინსტრუქციების შესრულებას როგორც მანიპულირებას მონაცემებზე. ამდენად, ენის მონაცემთა სტრუქტურირების ასპექტები უნდა განიხილებოდეს ინსტრუქციების შესწავლამდე.

ასემბლერის ენაზე დაწერილი პროგრამა აღწერს იმ მოქმედებების, რომლებიც უნდა შესრულდეს ფიზიკურ ობიექტებზე, როგორიცაა მეხსიერების უჯრედები და რეგისტრები.

ადრეულ დაპროგრამების ენებში პროგრამული ობიექტების იდენტიფიცირების ეს ტრადიცია გაგრძელდა. მაგალითად, მათში ცვლადი განიხილება როგორც მეხსიერების სიტყვა. თუმცა, ცვლადებს მიეწერება მათემატიკური ატრიბუტებიც (მაგალითად, მთელი – integer).

დაპროგრამების ძირითადი ცნებები:

***ტიპი*** არის რაიმე სახის მნიშვნელობათა სიმრავლე და მასზე განსაზღვრული ოპერაციების სიმრავლე.

მაგალითად, C -ში int ტიპი მოიაზრება როგორც მნიშვნელობათა სასრული სიმრავლე (დაახლოებით 65 000 ან 4 მილიარდი, დამოკიდებულია კომპიუტერზე) და ამ მნიშვნელობებისთვის არსებული ოპერაციების სიმრავლე (ოპერატორებით +, \*, <=, და ა.შ.). ბევრ ენას ახასიათებს მონაცემთა ახალი ტიპების შექმნის შესაძლებლობა. ე.ი., პროგრამისტი აღარ შემოიფარგლება ენაში ჩაშენებული რამდენიმე ტიპით; ნაცვლად ამისა, მას შეუძლია შექმნას საკუთარი ტიპები, რომლებიც უფრო ზუსტად შეესაბამება გადასაწყვეტ ამოცანას. უნდა გვახსოვდეს, რომ ტიპის განმარტება დამოკიდებულია ენაზე: ასე, მასივების მინიჭების ოპერაცია განსაზღვრულია Ada –ში, მაგრამ არაა C/C++ –ში, Java –სა და C# –ში. ან, მაგალითად, Fortran –ში არსებობს ხარისხში აყვანის \*\* ოპერატორი, Visual Basic –ში – იგივე შინაარსის **^** ოპერატორი, რასაც მოკლებულია ბევრი სხვა ენა. Fortran –ში მხარდაჭერილია მასივების მინიჭებაც, და, უფრო მეტიც – შეგვიძლია ჩავატაროთ ოპერაცია მთელ მასივზე და არა თითოეულ ელემენტზე მასივის გარბენით: მაგალითად, ფრაგმენტი

real a(6), b(6)

. . .

a = b\*\*2

a მასივს მიანიჭებს b მასივის ელემენტების კვადრატებს.

***მნიშვნელობა*** არის განუმარტებელი პრიმიტიული ცნება.

***ლიტერალი*** – პროგრამაში სიმბოლოთა მიმდევრობის სახით მოცემული "კონკრეტული" მნიშვნელობა. მაგალითად, 2018, -456.78, TRUE, , "Hello, All".

***წარმოდგენა*** – კომპიუტერის მეხსიერებაში მნიშვნელობის წარმოდგენა ბიტების კონკრეტული სტრიქონით (კოდით). მაგალითად, სიმბოლური 'x' მნიშვნელობის 8–ბიტიანი კოდი არის 01111000.

***ცვლადი*** – მეხსიერების დასახელებული უბანი, რომელიც შეიძლება შეიცავდეს კონკრეტული ტიპის მნიშვნელობის წარმოდგენას. მნიშვნელობა შეიძლება შეიცვალოს პროგრამის შესრულების დროს.

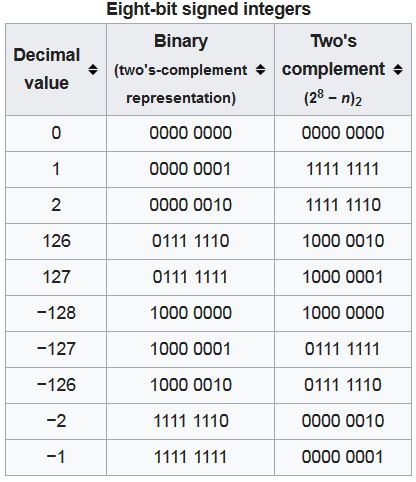
***მუდმივი*** (კონსტანტა) – მეხსიერების დასახელებული უბანი, რომელიც შეიძლება შეიცავდეს კონკრეტული ტიპის მნიშვნელობის წარმოდგენას. მნიშვნელობის შეცვლა პროგრამის შესრულების დროს იკრძალება.

***ობიექტი*** - ცვლადი ან მუდმივი.

**მთელი ტიპები**

დაპროგრამების ენებში მთელი რიცხვები წარმოდგენილია რამდენიმე სიმრავლის სახით, რომლებიც მათემატიკური მთელ რიცხვთა უსასრულო სიმრავლის სასრულ ქვესიმრავლეებს ქმნიან. განსხვავებული სიმრავლეების არსებობა განპირობებულია ბიტების რაოდენობით ტიპის წარმოდგენაში. მაგალითად, თუ მთელ რიცხვს მეხსიერებაში აქვს 8–ბიტიანი წარმოდგენა 10100011, მაშინ იგი ინტერპრეტირდება როგორც

(1 x 27)+(1 x 25)+(1 x 21)+(1 x 20) = 128 + 32 + 2 + 1 = 163

და განიხილება როგორც არაუარყოფითი მთელი. ასეთი რიცხვების შესაძლო მნიშვნელობები შეადგენენ დიაპაზონს 0..255.

ზოგადად, ვთქვათ გამოყოფილია N ბიტი. მათ შორის უფროსი - ნიშანთვისების ბიტია: მისი 0–ს ტოლი სიდიდე განსაზღვრავს დადებით რიცხვს ან ნულს, 1–ს ტოლი კი – უარყოფითს (უარყოფითი რიცხვებისთვის, ნიშანთვისების ბიტი მოქმედებს აგრეთვე რიცხვის სიდიდეზეც). არაუარყოფითი რიცხვები წარმოიდგინება, ჩვეულებრივ, ორობითი ჩანაწერით დიაპაზონში 0-იდან 2N –1 -მდე. უარყოფითი რიცხვის ორობით ჩანაწერს მივიღებთ თუ 2N -ს დავაკლებთ ამ უარყოფითის აბსოლუტურ მნიშვნელობას. ამ სისტემას ეწოდება two’s complement system (დამატება ორამდე). ახლა ეს სისტემა ყველაზე გავრცელებულია.

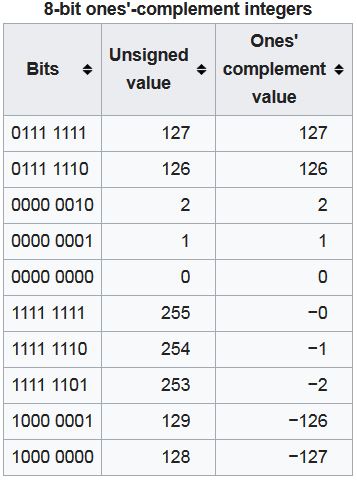
არსებობს უარყოფითი რიცხვის კოდის მიღების სხვა გზაც:

• იღებენ მოპირდაპირე დადებითი რიცხვის ინვერსიას (ლოგიკურ დამატებას)

• უმატებენ 1–იანს



უმცირესი 8–ბიტიანი რიცხვია 1000 0000 => –128. ცხადია, რომ მთელი ტიპების ცვლილების დიაპაზონი არასიმეტრიულია.

არსებობს ნიშნიანი მთელი რიცხვების წარმოდგენის ალტერნატიული სისტემა – one’s complement system (დამატება ერთამდე), რომელშიც უარყოფითი მთელი –K რიცხვის წარმოდგენა არის მისი მოპირდაპირე დადებითი K რიცხვის დამატება. ამ შემთხვევაში მნიშვნელობათა დიაპაზონი სიმეტრიულია, ხოლო 0 –ს აქვს 2 წარმოდგენა: 0000 0000 – ე.წ. დადებითი ნული და 1111 1111 – უარყოფითი ნული.

ბევრ ენაში მთელ რისხვთა ტიპებთან გამოიყენება სპეციფიკატორები: unsigned აღნიშნავს უნიშნო მთელს, ე.ი. ამ ტიპის მონაცემი განიხილება როგორც არაუარყოფითი რიცხვი; თუ unsigned სპეციფიკატორი მითითებული არაა, გულისხმობის პრინციპით რიცხვი არის ნიშნიანი, ანუ signed.

განსხვავებულ რიცხვით ტიპებს შეგვიძლია დამატებით მიუთითოთ სიგრძის სპეციფიკაცია:

short (2 ბაიტი), int (4 ბაიტი), long (4 ბაიტი), long long (8 ბაიტი)– C++ -ში;

byte (1 ბაიტი), short (2 ბაიტი), int (4 ბაიტი), long (8 ბაიტი) – Java;

sbyte, byte, short, ushort, int, uint, long, ulong – C# –ში;

Shortint (ნიშნიანი,1 ბაიტი), Byte (უნიშნო, 1 ბაიტი), Integer ((ნიშნიანი, 2 ბაიტი), Word (უნიშნო, 2 ბაიტი), Longint ((ნიშნიანი, 4 ბაიტი) – Turbo Pascal–ში.

**გადავსება**

თუ ოპერაციის შესრულების შედეგად ვღებულობთ მნიშვნელობას, რომელიც გადის დასაშვები მნიშვნელობების დიაპაზონიდან, ადგილი აქვს გადავსებას.

დავუშვათ, რომ 8–ბიტიანი ნიშნიანი მთელი ტიპის a ცვლადის მნიშვნელობაა 127, და ჩვენ ვზრდით მას ერთით:



შედეგად მივიღებთ რიცხვს –128. ამ შეცდომას ეწოდება გადავსება. გადავსებამ შეიძლება გამოიწვიოს არაადეკვატური შეცდომები. მაგალითად, a + a -ის შედეგად მივიღებთ –2-ს.

ან C –ს ინსტრუქცია

for (i = 0; i < j \* k; i++). . .

შეიძლება არ შესრულდეს, თუ j \* k –ს გამოთვლა გამოიწვევს გადავსებას.

გადავსების შეცდომა შეიძლება გამოიწვიოს ოპტიმიზატორის (კომპილერის ნაწილი) მუშაობამაც:

განვიხილოთ ერთ–ბაიტიანი მთელი რიცხვები. ვთქვათ, a, b და с ცვლადების მნიშვნელობებია 90,60 და 80, შესაბამისად.

პროგრამა ითვლის а - b + с –ს, რაც უდრის 110 –ს, თუ გავყვებით გამოთვლებს თანმიმდევრულად მარცხნიდან მარჯვნივ. თუმცა, ოპტიმიზატორს შეუძლია აირჩიოს მოქმედებების სხვა რიგი: а + с – b, რაც მოგვცემს შეცდომით შედეგს, რადგან а + с უდრის 170, ე.ი. მივიღებთ გადავსებას.

ახლა, როდესაც კომპიუტერების უმრავლესობა 32-ბიტიანი მეხსიერების სიტყვებს იყენებს, მთელრიცხვა გადავსება იშვიათია, მაგრამ უნდა ვიცოდეთ ამ შესაძლებლობის შესახებ და ფრთხილად ვიყოთ, რომ ზემოთ ნაჩვენებ ხაფანგებში არ გავებათ.

**ჩამოთვლითი ტიპი**

დავიწყოთ მაგალითით. დავუშვათ, ვწერთ გამათბობლის მართვის პროგრამას. გადამრთველის მიმდინარე პოზიციის შესანახად დაგვჭირდა ცვლადი dial (წრიული სკალა). გადამრთველს აქვს სულ ოთხი პოზიცია: Off (გამორთულია), Low (დაბალი), Medium (საშუალო) და High (მაღალი). როგორ გავაკეთოთ განაცხადი dial ცვლადზე და შემდეგ ვიმუშაოთ მასთან? პირველი, რასაც მოვიფიქრებთ პოზიციების აღსაწერად, იქნება მთელი რიცხვების გამოყენება (ვთქვათ, 1, 2, 3, 4):

int dial; /\* წრიული სკალის მიმდინარე პოზიცია \*/

if (dial < 4) dial++; /\* გავზარდოთ გათბობის დონე \*/

ცხადია, რომ ამ ხერხით დაწერილი პროგრამა იქნება აღსაქმელად რთული, დაგვჭირდება ვრცელი დოკუმენტაციის შექმნა, და, სავარაუდოდ, მასში ბევრჯერ ჩახედვა.

გაუმჯობესება მდგომარეობს მაკროსების გამოყენებაში, მაგალითად, ასე:

#define Off 1

#define Low 2

#define Medium 3

#define High 4

int dial;

if (dial < High) dial++;

თუმცა, მაინც რჩება ბევრი პრობლემა: შემდეგი მინიჭებები არ იკრძალება

dial = -1; /\* არარსებული მნიშვნელობა \*/

dial = High + 1; /\* დაუდგენელი გადავსება \*/

dial = dial \* 3; /\* უაზრო ოპერაცია \*/

მოყვანილი ფრაგმენტები C –ს კოდია.

თითქოს და C –ს typedef enum შესაძლებლობამ უნდა მოხსნას ყველა უხერხულობა:

typedef enum { Off, Low, Medium, High } Heat;

Heat dial;

dial = -1; /\* ეს შეცდომა არ არის \*/

dial = High + 1; /\* შეცდომა არ არის \*/

dial = dial \* 3; /\* შეცდომა არ არის \*/

როგორც ვხედავთ, ჩვენი მოლოდინი არ გამართლდა: არც ახლა არ იკრძალება შეცდომების დაშვება. საქმე ისაა, რომ Heat ტიპით განცხადებული ცვლადები ჯერ კიდევ მთელი რიცხვებია და არცერთი ზემოთ მოყვანილი ინსტრუქცია არ ითვლება შეცდომად (თუმცა კომპილერს შეუძლია გამოიტანოს გაფრთხილება).

პრობლემას აგვარებს ახალი ტიპების შექმნის ნებადართვა, რაც ბევრ ენაშია განხორციელებული. მაგალითად, Ada –ში

type Heat is (Off, Low, Medium, High);

Dial: Heat;

Dial := Low;

if Dial < High then Dial := Heat'Succ(Dial);

Dial := -1; -- შეცდომაა

Dial := Heat'Succ(High); -- შეცდომაა

Dial := Dial \* 3; -- შეცდომაა

C++ –ში შესაბამისი ტიპის აღწერა იწყება მომსახურე სიტყვით enum (enumeration)

enum Heat { Off, Low, Medium, High };

Heat dial;

dial = Low;

გულისხმობის პრინციპით Off -ს შეესაბამება 0, Low -ს – 1 და ა.შ. მიუხედავად ამისა, ჩამოთვლითი ტიპის ცვლადებისთვის აკრძალულია მთელი რიცხვების მინიჭება. ამრიგად, C++ –შიც

dial = -1; // შეცდომაა

dial = High + 1; // შეცდომაა

dial = dial \* 3; // შეცდომაა

მომდევნო მნიშვნელობაზე გადასვლა ხდება ცხადი გარდაქმნის გამოყენებით

dial = (Heat) (dial + 1);

ჩამოთვლითი ტიპის ცვლადებზე განსაზღვრული ოპერაციების ნაკრები შეზღუდულია. მასში შედის: მინიჭება ( :=, = ), შედარების ოპერაციები – ტოლია ( =, == ), არ უდრის ( /=, <>, != ) და <, >, >=, <= .

ჩამოთვლის კონსტანტას შეგვიძლია მივანიჭოთ მთელი მნიშვნელობა ჩვენი სურვილით. მაგალითად,

enum Heat {Off = 0, Low = 10, Medium = 15, High = 20};

თუ ერთ-ერთ კონსტანტას მინიჭებული აქვს მთელი რიცხვი, ხოლო მის შემდეგ მოცემულ კონსტანტებს - არა, რიგის მიხედვით მათ მიენიჭებათ ამ რიცხვის მომდევნო მნიშვნელობები. მაგალითად,

enum Heat {Off = 0, Low = 15, Medium, High};

განაცხადში კონსტანტას Medium –ს მიენიჭება 16, High -ს – 17.

ენაში Ada გამოიყენება შემდეგი ხერხი:

type Heat is (Off, Low, Medium, High);

for Heat use (Off => 1, Low => 2, Medium => 4, High => 8);

მონაცემთატიპი enumაკავშირებს რიცხვებს სახელებთან ერთი ძირითადი მიზნით - გააადვილოს პროგრამის კოდის გაგება.

**სიმბოლური ტიპი**

ყოველ სიმბოლოს კომპიუტერის მეხსიერებაში შეესაბამება მთელი რიცხვი – მისი კოდი. ფაკტობრივად, ჩინური და იაპონური ენების გარდა, სადაც ათასობით სიმბოლო გამოიყენება, სიმბოლოების წარმოსადგენად საკმარისია 8 ბიტი: 128 ნიშნიანი მთელი ან 256 უნიშნო მთელი მნიშვნელობა.

დაპროგრამების ენებში სიმბოლოების სიმრავლე არის დალაგებული მნიშვნელობების მოკლე თანმიმდევრობა. ე.ი. სიმბოლოები შეიძლება განისაზღვროს ჩამოთვლითი ტიპის მეშვეობით.

მაგალითად, Ada –ში არსებობს ჩაშენებული ჩამოთვლითი ტიპი:

type Character is (..., 'А', 'В',...);

და ჩამოთვლითი ტიპისთვის განმარტებული ყველა ოპერაცია გამოიყენება სიმბოლოებთანაც.

მოცემული Т ჩამოთვლითი ტიპისთვის და Т ტიპის V ცვლადისთვის Ada შეიცავს შემდეგ ფუნქციებს (მათ *ატრიბუტებს* უწოდებენ):

• T’First აბრუნებს Т –ს პირველ მნიშვნელობას;

• T’Last აბრუნებს Т –ს ბოლო მნიშვნელობას;

• T’Succ(V) აბრუნებს V ელემენტის შემდეგ მნიშვნელობას (successor – მემკვიდრე);

• T’Pred(V) აბრუნებს V ელემენტის წინა მნიშვნელობას (predecessor – წინამორბედი);

• T’Pos(V) აბრუნებს V ელემენტის პოზიციას Т სიაში;

• T’Val(I) აბრუნებს I –ური პოზიციის მნიშვნელობას.

C –ში, char მხოლოდ შეზღუდული მთელრიცხვთა ტიპია, და ყველა ქვემოთ მოყვანილი შეტყობინება დასაშვებია, რადგან char და int არსებითად იგივეა:

**char** с;

**int** i;

с = 'А' + 10; /\* char–ის int–ში გარდაქმნა და პირიქით \*/

i = 'А'; /\* char–ის int–ში გარდაქმნა \*/

с = i; /\* int–ის char–ში გარდაქმნა \*/

ერთადერთი, რითაც სიმბოლოები განსხვავდებიან ჩამოთვლებისგან ან მთელებისგან არის სპეციალური სინტაქსი ('А') სიმბოლოს მნიშვნელობისათვის და, რაც მთავარია, განსაკუთრებული სინტაქსი სიმბოლოთა მასივებისთვის – სტრიქონებისთვის.

**ლოგიკური ტიპი**

ლოგიკურ ტიპს დიდი მნიშვნელობა აქვს დაპროგრამების ენებში, რადგან:

* შედარების ოპერატორები ( =, >, < და ა.შ. ) გამოიმუშავებენ ლოგიკური ტიპის მნიშვნელობებს
* განშტოების და განმეორების შეტყობინებები ამოწმებენ ლოგიკური ტიპის პირობებს
* ბულის ალგებრის ოპერაციები (and, or, not, xor) განისაზღვრება ლოგიკური ტიპისათვის

ენას Ada აქვს ჩაშენებული ჩამოთვლითი ტიპი Boolean:

type Boolean is (False, True);

C –ში ცალკე ლოგიკური ტიპი განმარტებული არ არის, მის ნაცვლად გამოიყენება მთელი რიცხვები შემდეგი ინტერპრეტაციით: მნიშვნელობას „ჭეშმარიტი“ შეესაბამება ნებისმიერი არანულოვანი მთელი რიცხვი, ხოლო მნიშვნელობას „მცდარი“ – ნული.

მაგალითად, C –ში დასაშვებია შემდეგი ფრაგმენტის შესრულება:

int b;

double s;

. . .

if (b) {

s = 1.0;

}

სინამდვილეში, ეს არის დაპროგრამების ცუდი სტილის მაგალითი. გაცილებით ნათელია შემდეგი ფრაგმენტი, რომელიც ზემოთ მოყვანილის ეკვივალენტურია:

if (b != 0) {

s = 1.0;

}

უფრო მკაცრ ენაში Java, მეორე ფრაგმენტი კორექტულად ითვლება, პირველი კი არაკორექტულია.

C –ში დაშვებულია ლოგიკური ტიპის შემოღება, მაგალითად ისე, რომ ნებადართული იყოს ფუნქციების გამოყენება ბულის ტიპის დასაბრუნებელი მნიშვნელობით:

typedef enum {false, true} bool;

bool func (int a, float b);

if ( func (x, y) ) . . .

მაგრამ ეს ხერხი ემსახურება, რა თქმა უნდა, მხოლოდ წაკითხვადობის სიმარტივეს, და სინამდვილეში ახალ ტიპს არ ქმნის. ამიტომ შემდეგი უაზრო ინსტრუქცია

bool b = true;

b = b + 56; /\* გავზარდოთ «true» 56 –ით ??? \*/

დასაშვებად ითვლება, რამაც შეიძლება გამოიწვიოს ფარული შეცდომები.

C++ –ში bool ტიპი არის ჩაშენებული მთელრიცხვა ტიპი.

საზოგადოდ, C++ ენაში ნებისმიერი გამოსახულება (შედარების, ლოგიკური, არითმეტიკული) მოიაზრება როგორც bool ტიპის მნიშვნელობა. 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 ფაქტი ძალიან მნიშვნელოვანია პროგრამის შემადგენელი ინსტრუქციების მიმდევრობის მართვისთვის. ისეთი ფუნდამენტური კონსტრუქციები, როგორიცაა განშტოების და განმეორების შეტყობინებები (ციკლები), სრულდებიან ბულის ტიპის გამოსახულების მნიშვნელობის მიხედვით.

**გამოსახულება**

დავაზუსტოთ ცნება *გამოსახულება*.

გამოსახულება არის ცვლადების, მუდმივების, ოპერატორების, ფუნქციათა გამოძახებებისა და ფრჩხილების კომბინაცია, რომელიც განსაზღვრავს გარკვეული მნიშვნელობის გამოთვლის წესს დაპროგრამების ენის სინტაქსის შესაბამისად. მონაცემებს, რომლებზეც მოქმედებს ოპერატორი, უწოდებენ ოპერანდებს.

გამოსახულება შეიძლება იყოს მარტივი, როგორიცაა ლიტერალი (-12.25, 'U', true), ცვლადი ან ფუნქციის გამოძახება, როგორც სტანდარტულის, ისე შექმნილის მომხმარებლის მიერ.

არსებობს გამოსახულების ჩაწერის სამი ნოტაცია: ინფიქსური, პრეფიქსური და პოსტფიქსური.

ინფიქსური ნოტაცია არის გამოსახულების ჩაწერის ისეთი ფორმა, როდესაც ოპერატორი ჩაიწერება იმ ოპერანდებს შორის, რომლებზეც იგი მოქმედებს. მაგალითად, a \* b – გამრავლების ბინარული ოპერაციის ინფიქსური ჩანაწერია. პრეფიქსური ნოტაცია გულისხმობს ჩაწერის სტილს, როდესაც ოპერატორი მოთავსებულია ოპერანდების მარცხვნივ, ე.ი. \* a b, ხოლო პოსტფიქსურ ჩანაწერში ოპერატორი მოთავსე­ბულია ოპერანდების მარჯვნივ, ე.ი. a b \*.

პრეფიქსური ნოტაცია ცნობილია პოლონური ნოტაციის ან პოლონური ჩანაწერის სახელით. შესაბამისად, პოსტფიქსურ ნოტაციას შებრუნებულ პოლონურ ნოტაციას (reverse polish notation) ან შებრუნებულ პოლონურ ჩანაწერს უწოდებენ.

პრეფიქსური ნოტაცია შემოიღო პოლონელმა ლოგიკოსმა იან ლუკასევიჩმა დაახლოებით 1920 წელს ლოგიკური გამოსახულებების უფრჩხილებო ჩანაწერის შექმნაზე მუშაობისას. პოლონური ჩანაწერის ორივე სახე ფართოდ გამოიყენება კომპილერებში გამოსახულებების გარჩევისა და გამოთვლის დროს. მიუხედავად ამისა, დაპროგრამების ენების უმრავლესობა იყენებს გამოსახულების ჩაწერის ინფიქსურ ნოტაციას, რომელიც უფრო ჩვეულია ადამიანებისათვის.

ინფიქსური ნოტაციით ჩაწერილი გამოსახულების გამოთვლების რიგითობა დგინდება მასში არსებული ოპერატორების უპირატესობის (პრიორიტეტის) მიხედვით.

თითქმის ყველა დაპროგრამების ენა იზიარებს მათემატიკის სტანდარტს, რომელიც უფრო მაღალ პრიორიტეტს ანიჭებს მულტიპლიკატიურ ოპერატორებს ადიტიურ ოპერატორებთან შედარებით. მაგალითად, C/C++/C# –ში ოპერატორებს \*, /, % აქვთ უფრო მაღალი პრიორიტეტი, ვიდრე + და – ოპერატორებს. რაც შეეხება სხვა ოპერატორების პრიორიტეტებს, მათ ყოველი ენა განსაზღვრავს თავისებურად.

ორი უკიდურესობაა ენა APL, რომელიც პრიორიტეტებს საერთოდ არ განსაზღვრავს (არითმეტიკული ოპერატორებისთვისაც კი) და C, რომელშიც განსაზღვრულია პრიორიტეტების 15 დონე.

Pascal–ში ლოგიკურ ოპერატორს and აქვს უფრო მაღალი პრიორიტეტი, ვიდრე შედარების ოპერატორებს. ამიტომაც გამოსახულება

if a > b and b > c then . . .

ინტერპრეტირდება როგორც

if a > (b and b) > c then . . .

ე.ი. შეიცავს სინტაქსურ შეცდომას.

ინფიქსური გამოსახულების მნიშვნელობა ასევე დამოკიდებულია ოპერატორების ასოციატიურობაზე, კერძოდ იმაზე, თუ როგორ ჯგუფდებიან ერთნაირი პრიორიტეტის მქონე ოპერატორები: მარცხნიდან მარჯვნივ თუ მარჯვნიდან მარცხვნივ.

C –ს ფრაგმენტის

int i = 6, j = 7, k = 3;

i = i \* j / k; /\* რას უდრის შედეგი: 12 თუ 14 ? \*/

შესრულება იძლევა შედეგს 14, ვინაიდან გამოთვლა სრულდება მარცხნიდან მარჯვნივ:

i = ( i \* j )/ k;

ზოგადად, ბინარული ოპერატორები ჯგუფდებიან მარცხნიდან მარჯვნივ, იმ დროს როცა უნარული ოპერატორები – მარჯვნიდან მარცხვნივ.

მაგალითად, C –ს გამოსახულება ! ++i გამოითვლება როგორც !(++i).

პრიორიტეტებთან და ასოციატიურობასთან დაკავშირებული ყველა პრობლემა მარტივად გადაწყდება მრგვალი ფრჩხილების გამოყენებით. ამიტომ თუ შეცდომის დაშვების ოდნავი შესაძლებლობაც კი არსებობს, უნდა გამოვიყენოთ ფრჩხილები.

ზოგჯერ ფრჩხილების გამოყენება აუცილებელია: C++ –ის გამოსახულება \*this.value შეიცავს სინტაქსურ შეცდომას, რადგან ოპერატორს . მეტი პრიორიტეტი აქვს, ვიდრე ოპერატორს \* , და გამოსახულება კომპილირდება როგორც \*(this.value). ეს გამოსახულება თავიდანვე უნდა ჩაიწეროს ფრჩხილების გამოყენებით: (\*this).value.