ლ. გაჩეჩილაძე

ობიექტზე ორიენტირებული დაპროგრამება

(ალგორითმული ენა C++)

საქართველოს უნივერსიტეტის გამომცემლობა
თბილისი
2013

განხილულია ალგორითმულ ენა C++-ზე ობიექტურ-ორიენტირებადი დაპროგრამების ძირითადი პრინციპები და მათი გამოსახვის სხვადასხვა საშუალებანი, კერძოდ, მომხმარებლის მიერ შექმნილი სტრუქტურები და კლასები, ასევე კლასების რეალიზაციის დაფარვის შესაძლებლობები, კომპოზიციის ელემენტები, ეგმ-ის მეხსიერების დინამიური დაყოფის საშუალებები, მემკვიდრეობითობის საკითხები საბაზო და წარმოებულ კლასებში, ვირტუალური ფუნქციები და პოლიმორფიზმი, მონაცემთა ნაკადებთან სამუშაოდ გამოყენებული ფუნქციები და მანიპულატორები და სხვ.

თეორიულ მასალასთან ერთად, მოცემულია შესაბამისი ამოცანები და მაგალითები.

სახელმძღვანელო განკუთვნილია საქართველოს უნივერსიტეტის მათემატიკის და ინფორმატიკის სკოლის სტუდენტებისათვის.

რეცენზენტი:
პროფესორი მ. ავალიშვილი

დაიბეჭდა საქართველოში, თბილისში
გამომცემლობა „საქართველოს უნივერსიტეტის გამომცემლობა“

კოსტავას ქ. 77ა
თბილისი 0175, საქართველო

ISBN 978-99940-50-10-9

შესავალი

ტერმინი "ელექტრონული გამომთვლელი მანქანა" გულისხმობს კომპიუტერის ტექნიკური (Hardware) და პროგრამული (Software) უზრუნველყოფის ერთობლიობას. ამიტომ კომპიუტერული მეცნიერების განვითარება ამ ორი ძირითადი მიმართულებით მიმდინარეობს. იგი ითვალისწინებს ტექნიკური მოწყობილობების სრულყოფას და ინფორმაციის პროგრამული დამუშავების მიზნით ახალი რთული სტრუქტურების შექმნას.

დიალოგი ეგმ-სა და ადამიანს შორის ხორციელდება დაპროგრამების ენების საშუალებით, რომლებიც თავის მხრივ დაყოფილია დაბალი და მაღალი დონის დაპროგრამების ენებად.

დაპროგრამების დაბალი დონე მოიცავს მანქანურ და ასემბლერის ენებს. მაღალი დონის დაპროგრამების ენებზე კი პროგრამული უზრუნველყოფის შემუშავება პროგრამისტებისთვის გაცილებით მოსახერხებელია.

წინამდებარე სახელმძღვანელოში განხილული ალგორითმული ენა C++ მიეკუთვნება მაღალი დონის და ამავდროულად ობიექტზე ორიენტირებული დაპროგრამების ენების რიცხვს, რომელიც მონაცემებს (ატრიბუტებს) და ფუნქციებს (ქცევის ვარიანტებს) აერთიანებს ე.წ. ობიექტებში. მათ კი ახასიათებს ინფორმაციის დაფარვის თვისება. ეს ნიშნავს, რომ მართალია, ობიექტებს შეუძლიათ ერთმანეთს დაუკავშირდნენ კარგად განსაზღვრული ინტერფეისის საშუალებით, მაგრამ, როგორც წესი, მათთვის უცნობია, თუ როგორ ხდება სხვა ობიექტების რეალიზება. მაშასადამე, რეალიზა-

ციის დეტალები თავად ობიექტებშია დაფარული. კარგი პროგრამული უზრუნველყოფის შესამუშავებლად კი ინფორმაციის დაფარვა მნიშვნელოვან საკითხს წარმოადგენს.

პროცედურულ-ორიენტირებად ენებზე მომუშავე პროგრამისტები განსაკუთრებულ ყურადღებას უთმობენ ფუნქციებს, რომელთა დაჯგუფებაც ხდება პროგრამის შექმნის მიზნით. ამასთან, მართალია, მონაცემები ასევე მნიშვნელოვანია, მაგრამ ისინი არსებობენ, პირველ რიგში, ფუნქციების სარეალიზაციოდ. C++ში განსაკუთრებული ადგილი უჭირავს ***კლასებს*** – მომხმარებლის მიერ შექმნილ მონაცემთა ახალ ტიპებს. აქედან გამომდინარე, აქ ყურადღების ცენტრში ექცევა არა ფუნქციები, არამედ ამ კლასების ბაზაზე შექმნილი ობიექტები.

აღნიშნულ სახელმძღვანელოში განხილულია ალგორითმულ ენა C++-ზე ობიექტურ-ორიენტირებადი დაპროგრამების ძირითადი პრინციპები და მათი გამოსახვის სხვადასხვა საშუალებანი, კერძოდ, მომხმარებლის მიერ შექმნილი სტრუქტურები, კლასები და მათი რეალიზაციის დაფარვის შესაძლებლობები, კომპოზიციის ელემენტები, მემკვიდრეობითობის საკითხები საბაზო და წარმოებულ კლასებში და სხვ.

წინამდებარე სახელმძღვანელო განმარტავს პოლიმორფიზმისა და ვირტუალური ფუნქციების გამოყენების უპირატესობებს პროგრამული უზრუნველყოფის შემუშავების პროცესში, რაც განაპირობებს გაფართოებული შესაძლებლობების მქონე სისტემების რეალიზებას.

თავი I

ობიექტზე ორიენტირებული დაპროგრამების თეორიული საფუძვლები

1.1. სტრუქტურები

სტრუქტურა წარმოადგენს მომხმარებლის მიერ შედგენილ მონაცემთა ტიპს, რომელიც თავის მხრივ იყენებს სხვა მონაცემების უკვე არსებულ ტიპებს. განვიხილოთ სტრუქტურის აღწერის ზოგადი სახე

```
struct  Time
{
int hour;          //0-23
int minute;      // 0-59
int second;     //  0-59
};
```

კოდური სიტყვა **struct** იწყებს სტრუქტურის განსაზღვრას, ხოლო იდენტიფიკატორი **Time** წარმოადგენს სტრუქტურის სახელს. იგი გამოიყენება მოცემული ტიპის სტრუქტურის ცვლადების აღწერისას. აღნიშნულ მაგალითში **Time** ახალი ტიპის სახელია. სტრუქტურის განსაზღვრისას ფიგურულ ფრჩხილებს შორის აღწერილი მონაცემები ***სტრუქტურის ელემენტებს*** წარმოადგენს. ერთსა და იმავე სტრუქტურის ელემენტებს უნდა ჰქონდეს ერთმანეთისგან განსხვავებული (უნიკალური) სახელები, თუმცა პროგრამაში დასაშვებია, რომ ნებისმიერი ორი სტრუქტურა ერთნაირი სახელის ელემენტებს შეიცავდეს. სტრუქტურის ყოველი განსაზღვრა აუცილებლად უნდა დასრულდეს წერტილმძიმით (;).

ჩვენ შემთხვევაში, **Time** სტრუქტურა შეიცავს ***int*** ტიპის სამ ელემენტს: **hour**, **minute** და **second** (საათი, წუთი და წამი).

სტრუქტურის ელემენტები შესაძლოა იყოს ნებისმიერი ტიპის და ასევე ყოველი სტრუქტურა შეიძლება შეიცავდეს სხვადასხვა ტიპის მრავალ ელემენტს, მაგრამ სტრუქტურას არ უნდა ჰქონდეს საკუთარი თავის ეგზემპლარი. მაშასადამე, **Time** სტრუქტურის განსაზღვრისას, მასში დაუშვებელია **Time** ტიპის მონაცემის აღწერა.

სტრუქტურის ელემენტებზე მიმართვის უფლება აქვს მხოლოდ იმავე სტრუქტურის ობიექტებს, ხოლო ამ მიზნით გამოიყენება წერტილის (.) ან ისრის (**–>**) ოპერაციები. პირველი მათგანი (წერტილის ოპერაცია) უშუალოდ მიმართავს სტრუქტურის ელემენტს სახელით, ხოლო მეორე (ისრის ოპერაცია) სტრუქტურის ელემენტზე მიმართვას უზრუნველყოფს ობიექტზე მიმთითებლის საშუალებით. თუ დავუშვებთ, რომ მონაცემი ***Object*** წარმოადგენს **Time** ტიპის ობიექტს, მაშინ იგი ამავე სტრუქტურის ნებისმიერ ელემენტს მიმართავს წერტილის ოპერაციით, ე.ი.

Time *Object*;

***Object*.hour;**

ანალოგიურად, თუ დავუშვებთ, რომ ***timePtr** წარმოადგენს მიმთითებელს **Time** ტიპის ობიექტზე, მართებული იქნება შემდეგი ჩანაწერი:

Time ****timePtr*;

***timePtr*–>hour;**

გამოსახულება ***timePtr*–>hour** ეკვივალენტურია შემდეგი ჩანაწერის **(****timePtr*).hour**, სადაც ფრჩხილების გამოყენება აუცილებელია, რადგან წერტილის (.) ოპერაციას უფრო მა-

ღალი პრიორიტეტი აქვს, ვიდრე ვარსკვლავის (*) ოპერაციას.

მაგალითი 1.1. შევადგინოთ პროგრამა, რომელიც სტრუქტურის გამოყენებით დაბეჭდავს მომხმარებლის მიერ განსაზღვრულ დროს სტანდარტულ ფორმატში.

ამოხსნა

```
#include  <iostream>
struct Time
{
int hour;
int minute;
int second;
};
void PrintStandart(Time );    //prototipe
main( )
{
Time x;
x.hour = 15;
x.minute = 30;
x.second = 20;
cout << "Standard time:" << endl;
PrintStandart(x);
return 0;
}
void PrintStandart(Time t)
{
cout<<((t.hour = =0 || t.hour = =12)? 12: t.hour%12)
   << ":"<<(t.minute<10 ? "0":"")<<t.minute
   << ":"<<(t.second<10 ? "0":"")<<t.second
   << ":"<<(t.hour<12 ? "AM":"PM");
   }
```

შედეგები

```
Standard time:
3:30:20:PM
```

მაგალითი 1.2. შევადგინოთ პროგრამა, რომელიც სტრუქტურის გამოყენებით ***int*** **A[10]** მასივის ელემენტებს ერთიდან 55-მდე რიცხვით დიაპაზონში მიანიჭებს მთელი ტიპის შემთხვევით მნიშვნელობებს და გამოთვლის მასივის ლუწი მნიშვნელობისა და კენტინდექსიანი ელემენტების საშუალო არითმეტიკულს.

ამოხსნა

```
#include  <iostream>
#include  <iomanip>
#include  <stdlib>
struct Array
{
int A[10];
void function1( );      //prototipe
void function2( );      //prototipe
};
void Array :: function1( )
{
for(int i=0; i<10; i++)
 {
A[i] = 1+rand( )%55;
cout << setw(5) << A[i];
 }
}
void Array :: function2( )
{
float s = 0;
int n = 0;
for(int i=0; i<10; i++)
if(A[i]%2 = =0 && i%2 = =1)
{
s += A[i];
n ++;
}
s = s/n;
```

```
cout << "\n Average=" << s << endl;
}
main( )
{
Array x;
cout << "Current Array:" << endl;
x.function1( );
cout << "\n Total Result:" << endl;
x.function2( );
return 0;
}
```

შედეგები

Current Array:

17 21 38 46 52 23 51 36 14 52

Total Result:

Average = 44.6667

აღნიშნულ პროგრამაში გამოყენებულია ორი ორწერტილის (**::**) ბინარული ოპერაცია, რაც იმაზე მიუთითებს, რომ ჩვენ მიერ შექმნილი **function1( )** და **function2( )** ფუნქციები მიეკუთვნება **Array** სახელის მქონე სტრუქტურას. თავად ფუნქციების აღწერა კი განხორციელდა სტრუქტურის განსაზღვრის შემდეგ, რაც დაპროგრამების კარგ სტილად ითვლება. ამასთან, **function1( )** და **function2( )** ფუნქციებს **Array** სტრუქტურის, ***ფუნქცია-ელემენტები*** ეწოდება.

1.2. კლასები

კლასების საშუალებით პროგრამისტებს შეუძლიათ იმ ობიექტების მოდელირება, რომელთაც აქვთ ატრიბუტები (მონაცემი-ელემენტები) და ქცევის ვარიანტები (ფუნქცია-ელემენტები). ტიპები, რომლებიც შეიცავს მონაცემ-ელემენტებსა და ფუნქცია-ელემენტებს, როგორც წესი, C++-ში განისაზღვრება კოდური სიტყვით **class**. ფუნქცია-ელემენტებს დაპროგრამების მრავალ ობიექტურ-ორიენტირებად ენაში, ***მეთოდებს*** უწოდებენ. მათი გამოძახება სრულდება იმ შეტყობინების პასუხად, რომელიც ეგზავნება ობიექტს. თავად შეტყობინება კი შეესაბამება ფუნქცია-ელემენტის გამოძახებას.

როდესაც ესა თუ ის კლასი განსაზღვრულია, მისი სახელი (იდენტიფიკატორი) შეიძლება გამოყენებულ იქნეს ამავე კლასის ობიექტის აღსაწერად.

განვიხილოთ მარტივი მაგალითი. დავუშვათ, საჭიროა მოვძებნოთ ორი მთელი დადებითი ***a*** და ***b*** რიცხვის უმცირესი საერთო ჯერადი კლასის შესაძლებლობების გამოყენებით. ბუნებრივია, კლასის განსაზღვრა იწყება კოდური სიტყვით **class**, რასაც მოყვება მისი სახელი, ჩვენ შემთხვევაში LCM (the least common multiple). კლასის განსაზღვრის ტანი თავსდება ღია და დახურულ ფიგურულ ფრჩხილებს შორის ({}), ხოლო განსაზღვრის პროცესი სრულდება წერტილ-მძიმით (;). ჩვენ მიერ შექმნილი LCM კლასი შეიცავს ***int*** (მთელი) ტიპის ორ ელემენტს: ***a***-ს და ***b***-ს, ხოლო კლასის განსაზღვრას აქვს შემდეგი სახე:

```
class LCM
{
public:
LCM( );                          //constructor
void setvalues(int, int);    //prototipe
void function( );              //prototipe
private:
int a;
int b;
};
```

აღნიშნულ კლასში გამოყენებულია ორი ახალი ჭდე, ესენია: **public** და **private**, რომელთაც ელემენტებზე მიმართვის სპეციფიკატორები ეწოდება. ნებისმიერი მონაცემ-ელემენტი ან ფუნქცია-ელემენტი, რომელიც განსაზღვრულია **public** სპეციფიკატორის შემდეგ მომდევნო სპეციფიკატორამდე პროგრამაში, ხელმისაწვდომია ამავე კლასის ნებისმიერი ობიექტისათვის, ხოლო ყველა იმ მონაცემ-ელემენტზე ან ფუნქცია-ელემენტზე მიმართვა, რომელიც განსაზღვრულია **private** სპეციფიკატორის შემდეგ მომდევნო სპეციფიკატორამდე, შეუძლია მხოლოდ ამავე კლასის ფუნქცია-ელემენტებს. ყოველი სპეციფიკატორი აუცილებლად უნდა დასრულდეს ორწერტილით (:) და კლასის განსაზღვრის დროს შესაძლოა ნებისმიერი რაოდენობითა და თანმიმდევრობით იქნეს გამოყენებული. დაპროგრამების კარგ სტილად ითვლება, როდესაც ელემენტებზე მიმართვის ყოველი სპეციფიკატორი გამოიყენება ერთხელ, რადგან იგი პროგრამის წაკითხვას გაცილებით ამარტივებს. ამასთან, სასურვლია პირველ რიგში გამოვიყენოთ **public** სპეციფიკატორის ელემენტები, რადგან ისინი ხელმისაწვდომია კლასის ყველა ობიექტისათვის. ის მონაცემები, რომლებიც განისაზღვრები-

ან **public** სპეციფიკატორის ქვეშ მომდევნო სპეციფიკატორამდე პროგრამაში, ქმნის კლასის მომსახურების ***ღია ინტერფეისს***. მათ, როგორც წესი, იყენებენ კლასის ***კლიენტები*** (მომხმარებლები). **private** სპეციფიკატორის შემდეგ განსაზღვრული მონაცემები კი შეადგენს კლასის ***დახურულ ელემენტებს***. ამ თვალსაზრისით ცხადია, რომ კლასის ***რეალიზაცია*** (**implementation**) დაფარულია მისი კლიენტებისაგან. აღნიშნული სახით ინფორმაციის დაფარვა კი ხელს უწყობს კლიენტების მიერ კლასის აღქმის პროცესს.

ჩვენ შემთხვევაში **LCM** კლასის ღია ფუნქცია-ელემენტებია: **LCM( ), void setvalues(int, int)** და **void function( )** ფუნქციები, ხოლო დახურული მონაცემ-ელემენტებია ***a*** და ***b*** ცვლადები. ამავე კლასში გამოყენებულია ფუნქცია-ელემენტი, რომელსაც იგივე სახელი აქვს, რაც მოცემულ კლასს (**LCM**). მას კლასის ***კონსტრუქტორი*** ეწოდება. იგი ახდენს კლასის მონაცემი-ელემენტების ინიციალებას (ანიჭებს მათ საწყის ნულოვან ან ნებისმიერ მნიშვნელობებს). კლასის ობიექტის შექმნის პროცესში ავტომატურად ხდება კონსტრუქტორის გამოძახება.

კლასის განსაზღვრის შემდეგ შესაძლებელია ამავე კლასის ობიექტის შექმნა შემდეგი სახით: **LCM x;**

ამ შემთხვევაში კლასის სახელი წარმოადგენს ტიპის ახალ სპეციფიკატორს.

აღსანიშნავია ის გარემოება, რომ კლასის მონაცემი-ელემენტების ინიციალება (საწყისი მნიშვნელობების მინიჭება) დაუშვებელია კლასის ტანში. ისინი საწყის მნიშვნელობებს იღებენ კლასის კონსტრუქტორის ან ამავე კლასის სხვა ფუნქციის მეშვეობით. კლასის ფუნქცია-ელემენტები კი

შეიძლება აღიწეროს როგორც კლასის ტანში, ასევე მის გარეთ. ამ უკანასკნელ შემთხვევაში აუცილებელია ფუნქცია-ელემენტის მოქმედების არის განსაზღვრის ორი ორწერტილის (**::**) ბინარული ოპერაციის გამოყენება, რაც იმაზე მიუთითებს, რომ ესა თუ ის ფუნქცია-ელემენტი ეკუთვნის კონკრეტულ კლასს (რადგან, შესაძლოა პროგრამაში არსებობდეს ერთნაირი სახელის ფუნქცია-ელემენტის შემცველი რამდენიმე სხვადასხვა კლასი). დაპროგრამების კარგ სტილად ითვლება, როდესაც ფუნქცია-ელემენტების აღწერა წარმოებს კლასის ტანის გარეთ. ნებისმიერ შემთხვევაში ფუნქცია-ელემენტის მოქმედების არეს ის კლასი წარმოადგენს, რომელშიც იგი განსაზღვრულია.

პროგრამებში შესაძლოა შევხვდეთ კლასის სახელის მქონე ფუნქციას, რომელიც "**~**" (ტილდა) სიმბოლოთი იწყება. მას კლასის ***დესტრუქტორი*** ეწოდება (ჩვენი მაგალითი მას არ შეიცავს). იგი ასრულებს კლასის ყოველ ობიექტზე "დამამთავრებელ სამუშაოს" მანამ, სანამ ამ ობიექტებისთვის ეგმ-ის მიერ გამოყოფილ მეხსიერებას ხელახლა არ გამოიყენებს სისტემა.

სტრუქტურის ელემენტებზე მიმართვის მსგავსად, კლასის ელემენტებსაც მიმართავენ წერტილის (**.**) ან ისრის (**–>**) ოპერაციებით.

კლასებში ვხვდებით ასევე მესამე სპეციფიკატორს: **protected** (დაცული), რომელიც გამოიყენება კლასის მონაცემ-ელემენტებსა და ფუნქცია-ელემენტებზე მიმართვის საწარმოებლად. თუ კლასის განსაზღვრის დროს ელემენტებზე მიმართვის არანაირი ჭდე არ არის მითითებული, მაშინ იგულისხმება **private** სპეციფიკატორი. კლასებისგან გან-

სხვავებით, სტრუქტურებში ასეთ შემთხვევაში იგულისხმება **public** სპეციფიკატორი.

მაშასადამე, კლასის ელემენტები ავტომატურად დახურულია, ხოლო სტრუქტურისა – ღია.

ზემოთ თქმულის გათვალისწინებით, დასმული ამოცანის გადაწყვეტის პროგრამას ექნება შემდეგი სახე:

ამოხსნა

```
#include <iostream>
class LCM
{
public:
LCM( );                          //constructor
void setvalues(int, int);    //prototipe
void function( );              //prototipe
private:
int a;
int b;
};
 LCM :: LCM( )
 {
 a = b = 0;
 }
 void LCM :: setvalues(int x, int y)
 {
 a = x;
 b = y;
 }
 void LCM :: function( )
 {
 int m, n;
 long s;
 m = a;
 n = b;
 s = m*n;
 while(a!=b)
```

```
{
if(a>b)
a = a - b;
else
b = b - a;
}
cout << "\n Result=" << s/a << endl;
}
main( )
{
LCM  x;
int p, z;
cout << "a = ";
cin >> p;
cout << "b = ";
cin >> z;
x.setvalues(p,z);
x.function( );
return 0;
}
```

თუ დავუშვებთ, რომ **a=15** და **b=25**, მივიღებთ შემდეგ რეზულტატს:

a = 15

b = 25

Result = 75

მაგალითი 1.3. შევადგინოთ პროგრამა, რომელიც კლასის გამოყენებით ***int*** **B[10]** მასივის ელემენტებს ამავე კლასის კონსტრუქტორის საშუალებით მიანიჭებს მთელი ტიპის შემთხვევით მნიშვნელობებს ერთიდან 100-მდე რიცხვით დიაპაზონში და გამოთვლის მასივის ელემენტების კვადრატების ჯამს, ნამრავლსა და საშუალო არითმეტიკულს.