ლექციათა კურსი

ობიექტზე ორიენტირებული დაპროგრამება 1 (C++)

ლექცია 1-2

შინაარსი:

- ფუნქციის ნაგულისხმევი არგუმენტები (ფუნქციის პარამეტრების ინიციალიზება)
- ფუნქციათა გადატვირთვა
- მაკროსები და ჩადგმადი `(inline)` ფუნქციები
- კლასი: ინკაფსულაციის პრინციპი `(Encapsulation)`. კლასის წევრებზე წვდომა `(private, public)`
- კლასას ინტერფეისის გამოყოფა იმპლემენტაციისგან
- კლასის დახურულ `(private)` წევრებზე წვდომა
- კონსტრუქტორის გადატვირთვა. კონსტრუქტორი ნაგულისხმევი არგუმენტებით.

ფუნქციის არგუმენტების მნიშვნელობები გულისხმობის პრინციპით
(ან ფუნქციის პარამეტრების ინიციალიზება)

შემდეგ მაგალითში ფუნქცია `Volume` ითვლის მართკუთხა პარალელეპიპედის მოცულობას. ფუნქციის პროტოტიპში სამივე ფორმალურ პარამეტრს მინიჭებული აქვთ საწყისი მნიშვნელობები. ეს გვალევს საშუალებას ფუნქციის გამოძახების დროს გადაეცემა მას არგუმენტების სხვადასხვა რაოდენობა.

```
#include <iostream>
using namespace std;
int Volume(int a = 2, int b = 3, int c = 4);
// a - პარალელეპიპედის ფუძის სიგრძე, b - პარალელეპიპედის ფუძის სიგანე,
// c - პარალელეპიპედის სიმაღლე
int main()
{
   cout << Volume () << endl          // volume of box 2 x 3 x 4
        << Volume (1, 2) << endl      // volume of box 1 x 2 x 4
        << Volume (1) << endl         // volume of box 1 x 3 x 4
        << Volume (3, 2, 5) << endl; // volume of box 3 x 2 x 5
}
int Volume (int a, int b, int c)
{
   return a * b * c;
}
```

```
                OUTPUT
24
8
12
30
Press any key to continue . . .
```

ფუნქციის პარამეტრების ნაწილობრივი ინიციალიზება

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

```
// ფუნქციის პროტოტიპი
  void func(int a, int b = 2, float c = 3.75);
// მისი შესაძლო გამოძახებები
  func(10);        // ok
  func(5,7);       // ok
  func(1,2,12.7); // ok
```

ოღონდ გამოძახება

```
        func(); // wrong call
```

გამოიწვევს კომპილაციის შეცდომას

```
error C2660: ' func ': function does not take 0 arguments
```

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

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

უნდა გავითვალისწინოთ, რომ **func1** ფუნქციის იმპლემენტაციაში პარამეტრების სახელების მითითება *აუცილებელია*, ხოლო პარამეტრების *საწყისი მნიშვნელობების გამეორება შეცდომაა*. ანუ **func1** ფუნქციის იმპლემენტაციაში სათაურის შესაძლო სახეა

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

ფუნქციათა გადატვირთვა

C++ –ში მხარდაჭერილია შესაძლებლობა განისაზღვროს ერთი და იგივე სახელის მქონე რამდენიმე ფუნქცია, თუ განსხვავებულია მათი სიგნატურა, ე.ი. გადასატვირთი ფუნქციების პარამეტრების სია უნდა იყოს განსხვავებული ან პარამეტრების რაოდენობით, ან მათი ტიპით, ან პარამეტრების რაოდენობითაც და ტიპითაც. ამას 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.

შემდეგ მაგალითში გადატვირთული Opposite ფუნქცია აბრუნებს თავისი ერთადერთი არგუმენტის მოპირდაპირე მნიშვნელობას.

```
#include <iostream>
using namespace std;
int Opposite (int );
double Opposite (double );
long long Opposite (long long );
int main()
{
   int i = -10; double f = 11.23;
   long long k = 5910000000LL;
   cout << "Opposite (-10): " << Opposite (i) <<'\n';
   cout << "Opposite (5910000000LL): " << Opposite (k) <<'\n';
   cout << "Opposite (11.23): " << Opposite (f)<<'\n';
}
int Opposite (int n){
    return -n;
}
```

```
double Opposite (double n){
    return -n;
}
long long Opposite (long long n){
    return -n;
}
```

პროგრამის გამოტანის ეკრანია:

```
                    OUTPUT
Opposite (-10): 10
Opposite (5910000000LL): -5910000000
Opposite (11.23): -11.23
Press any key to continue . . .
```

ვთქვათ, გვაქვს შემდეგი ფუნქციების პროტოტიპები:

```
double func(int , double , char = 50);

int func(int , double );

void func(int , double , int = 3);
```

ამ შემთხვევაში ფუნქციის გამოძახება

```
func (10, 0.0123);
```

გამოიწვევს კომპილაციის დროის შეცდომას

```
error C2668: 'func' : ambiguous call to overloaded function
    could be 'void func(int,double,int)'
    or       'int func(int,double)'
    or       'double func(int,double,char)'
```

დავუშვათ, რომ ჩვენს პირველ მაგალითში Volume ფუნქციის პროტოტიპი გამოიყურება ასე

```
int Volume(int, int, int = 1);
```

ეს შესაძლებელს ხდის ფუნქციის გამოყენებას როგორც პარალელეპიპედის ფუძის ფართობის, ისე მისი მოცულობის გამოსათვლელად, ანუ გვაქვს ფუნქციის გადატვირთვის ეფექტი.

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

int Volume(int, int, int = 1);

int main(){
     cout << "area of base = "
          << Volume(3, 2) << endl
          << "volume = "
          << Volume(3, 2, 5) << endl;
}
int Volume(int a, int b, int c){

   return a * b * c;
}
```

```
                OUTPUT
area of base = 6
volume = 30
Press any key to continue . . .
```

მაკროსები

ფუნქციის გამოძახება რთული მექანიზმია და მოითხოვს გარკვეულ დროს, რაც საბოლოო ჯამში იწვევს პროგრამის შესრულების დროის ზრდას. ფუნქციაზე მიმართვის დროის შემცირების მიზნით C/C++-ში მხარდაჭერილია ე.წ. პარამეტრებიანი მაკროგანმარტებები (მაკროსები), რომლებიც განისაზღვრება პრეპროცესორის **#define** დირექტივით. მაგალითად,

```
#define  MIN(a,b) ((a < b) ? a : b)
```

მაკროსი შეგვიძლია გამოვიყენოთ ორი ობიექტიდან უმცირესის პოვნისას, იმ პირობით რომ მათთვის განსაზღვრულია < ოპერატორი.

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

#define MIN(a,b) ((a < b) ? a : b)

int main()
{
    cout << MIN(-11, 4) << endl;
    cout << MIN(11.5, 4.3) << endl;
    cout << MIN("string1", "string2") << endl;
}
```

```
OUTPUT
-11
4.3
string1
Press any key to continue ...
```

მაკროსის გამოძახების ადგილას პრეპროცესორი „აფართოვებს“ მას, ე.ი. ახდენს მის ტექსტურ ჩანაცვლებას. ჩვენ მაგალითში გამოძახება `MIN(-11, 4)` ჩაინაცვლება გამოსახულებით `((-11 < 4) ? -11 : 4)`, რომელიც გამოითვლება ჯერ კიდევ კომპილაციის ეტაპზე.

ანალოგიურად შესრულდება მაკროსის დანარჩენი ორი გამოძახება.

ის ფაქტი, რომ მაკროსები იყენებენ ტექსტურ ჩანაცვლებას, პოტენციური შეცდომების წყაროს წარმოადგენს. მაგალითად, შემდეგი მაკროსის

```
#define CUBE(x) x * x * x
```

გამოყენება კოდის ფრაგმენტში

```
int n{ 3 };
cout << n + 1 << " ^ 3 = " << CUBE(n + 1) << endl;
```

გვაძლევს არაკორექტულ შედეგს: **4 ^ 3 = 10,** რადგან **CUBE(n + 1)** ჩაინაცვლება ტექსტით **3 + 1 * 3 + 1 * 3 + 1**

ამ მარტივ შემთხვევაში პრობლემის მოგვარებაც მარტივია:

```
#define CUBE(x) (x) * (x) * (x)
```

შედეგად **CUBE(n + 1)**-ის ტექსტურ ჩანაცვლებას ექნება სახე **(3 + 1) * (3 + 1) * (3 + 1)**,

ხოლო ფრაგმენტის შესრულებას : **4 ^ 3 = 64.**

ჩადგმადი (inline) ფუნქციები

მაკროსთან დაკავშირებული პრობლემების ასაცილებლად C++ -ში შემოღებულია ჩადგმადი (**inline**) ფუნქციები. მათ ასევე უწოდებენ ჩასმად ან ჩაშენებად ფუნქციებს.

სიტყვა **inline** ფუნქციის განაცხადში სთავაზობს (”ურჩევს”) კომპილერს ფუნქციის გამოძახების მექანიზმის ჩართვის ნაცვლად მოახდინოს ფუნქციის ტანის ასლის გენერირება (ჩაშენება, ჩასმა) მისი გამოძახების ადგილას.

```
#include <iostream>
using namespace std;
inline bool isOdd(int n)
{
     return n % 2;
}
int main()
{
     int a = 19, b = 128;
     cout << boolalpha
          << a << " is Odd? " << isOdd(a) << endl
          << b << " is Odd? " << isOdd(b) << endl;
}
```

```
OUTPUT
19 is Odd? true
128 is Odd? false
Press any key to continue ...
```

inline ფუნქციის გამოძახება კომპილაციის პროცესში გაფართოვდება მსგავს ობიექტურ კოდში

```
     cout << boolalpha
          << a << " is Odd? " << a % 2 << endl
          << b << " is Odd? " << b % 2 << endl;
```

რომელიც, ცხადია, შესრულდება გაცილებით სწრაფად ვიდრე ეს მოხდებოდა ფუნქციის გამოძახების დროს.

კომპილერს შეუძლია უგულვებელყოს **inline** სპეციფიკატორი და, ჩვეულებრივ, ასეც იქცევა, თუ ფუნქცია არ არის მარტივი და მოკლე. ამ შემთხვევაში ფუნქცია გამოიძახება ჩვეულებრივი წესით.

ამრიგად, თუ ფუნქცია შეიცავს განმეორების შეტყობინებას, ან **switch** შეტყობინებას, ან **static** ლოკალურ ცვლადებს, ან არის რეკურსიული, **inline** სპეციფიკაციას მის განაცხადში აზრი არა აქვს.

აქვე აღვნიშნოთ, რომ **inline** ფუნქციის მრავალჯერადი გამოძახება ზრდის პროგრამის მოცულობას. ამიტომ, თუ ფუნქციის ტანი მარტივია, მაგრამ დიდი, ან ჩადგმადი ფუნქცია/ფუნქციები გამოიძახება ბევრჯერ,
მაშინაც არ ღირს ასეთ ფუნქციასთან **inline** სპეციფიკატორის გამოყენება.

დასაშვებია **inline** ფუნქციის პარამეტრებს მივანიჭოთ საწყისი მნიშვნელობები.
შემდეგ მაგალითში **inline** ფუნქცია `square` ითვლის `x` გვერდიანი კვადრატის ფართობს.

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

inline double square(double =  2.5);

int main()
{
  cout << "side of square = 2.5,\narea = "
       << square() << endl;
  cout << "enter new value of side = ";
  double a;
  cin >>a;
  cout << "area of square = " << square(a) << endl;

}
inline double square(double x)
{
   return x * x;
}
```

პროგრამა დაბეჭდავს:

OUTPUT

```
side of square = 2.5,
area = 6.25
enter new value of side = 10
area of square = 100
Press any key to continue . . .
```

ინკაფსულაციის პრინციპი (Encapsulation). კლასის წევრებზე წვდომა (private, public)

ინკაპსულაცია – ობიექტზე ორიენტირებული პროგრამირების (OOP) ერთ-ერთი ძირითადი კონცეფცია – განისაზღვრება, როგორც მონაცემების და მათზე მანიპულირების მეთოდების შეკვრა ერთ მოდულში სახელწოდებით კლასი. კლასი კრძალავს მონაცემებზე პირდაპირ წვდომას კლასის გარედან: მონაცემებზე წვდომის საშუალებას იძლევა კლასის მეთოდები. ეს არის OOP–ის თვისება, რომელიც მონაცემთა დამალვას ემსახურება. ინფორმაციის დამალვა ხელს უწყობს მონაცემების დაცვას და აიოლებს პროგრამის მოდიფიცირებას.

```
class Square // მომხმარებლის მიერ განსაზღვრული აბსტრაქტული მონაცემთა ტიპის განსაზღვრა
{            // ტიპის სახელია Square
  public:
  // ღია, საჯარო წევრები – კლასის მომსახურების ღია ინტერფეისი
  // ჩვეულებრივ ესენია კლასის ფუნქციები მეთოდები), მაგრამ არა კლასის მონაცემები
     Square();
     double Area();
     double Perimeter();

  private:
  // დახურული, კერძო წევრები – წვდომადი მხოლოდ კლასის მეთოდებისთვის
  // ჩვეულებრივ ესენია კლასის მონაცემები და ზოგჯერ კლასის ფუნქციებიც
     double side;
};
```

public: (ღია, საჯარო) და **private:** (დახურული, კერძო) ჭდეებს კლასის აღწერაში ეწოდებათ *ელემენტებზე წვდომის სპეციფიკატორები.*

კლასის წევრებზე, რომელთა განაცხადები მოთავსებულია **public** სპეციფიკატორის შემდეგ ნებადართულია წვდომა პროგრამის ნებისმიერი ადგილიდან.

private სპეციფიკატორის შემდეგ განცხადებულ მონაცემებზე და ფუნქციებზე პროგრამიდან წვდომა აკრძალულია. ასეთ მონაცემებთან და ფუნქციებთან მუშაობა შეუძლიათ მხოლოდ ამავე კლასის ფუნქციებს (და ე. წ. *მეგობარ* ფუნქციებს).

კლასის განსაზღვრის სხვა სტილი

```
class Square
{
  private:
     double side;
  public:
     Square();
     double Area();
     double Perimeter();
};
```

თუ გავითვალისწინებთ, რომ კლასის მონაცემებიც და ფუნქციებიც გულისხმობის პრინციპით დახურული წევრებია, მაშინ ზემოთ მოყვანილ განაცხადში შეიძლება გამოვტოვოთ **private** სპეციფიკატორი:

```
class Square
{
     double side;
  public:
     Square();
     double Area();
     double Perimeter();
};
```

განაცხადი კლასის ობიექტზე (კლასის ეკზემპლარზე, კლასის ტიპის ცვლადზე): `Square mySquare;`

კლასას ინტერფეისის გამოყოფა იმპლემენტაციისგან

თუ კლასის მეთოდის იმპლემენტაცია განხორციელებულია კლასის შიგნით იგი არაცხადად განისაზღვრება როგორც ჩადგმადი ფუნქცია. თუმცა კომპილერი უფლებას იტოვებს არ განიხილოს იგი როგორც **inline**.

კლასის მეთოდების განსაზღვრა (არწერა) წესით უნდა მოხდეს ე.წ. კლასის იმპლემენტაციის რეალიზების (კლასის რეალიზების) ნაწილში აღწერის გარეთ. უფრო მეტიც, *იმპლემენტაციის ნაწილი* სასურველია იყოს გატანილი სხვა ფაილში, როგორც ეს ხდება მრავალფაილური პროექტის შექმნისას.

```
class Square {
    public:
    // კლასის ფუნქციების პროტოტიპები
       Square();             // default კონსტრუქტორი
       double Area();        // ფართობის გამოთვლის ფუნქცია
       double Perimeter(); // პერიმეტრის გამოთვლის ფუნქცია
    private:
       double side;
};
```

```
// კლასის იმპლემენტაცია
Square::Square(){
    side = 5;
}
```

ან

```
Square::Square():side(5){}
```

```
double Square::Area(){
     return side * side;        // სადაც :: არის ხილვადობის არის ოპერატორი
}
double Square::Perimeter(){
     return 4 * side;
}
```

კოდის შემდეგი ფრაგმენტი:

```
Square mySquare; // mySquare ობიექტის შექმნა. გვერდი უდრის 5-ს.
cout << "Area is " << mySquare.Area()
     << "\nPerimeter is " << mySquare.Perimeter() << endl;
```

დაბეჭდავს:
```
Area is 25
Perimeter is 20.
```

მაგრამ გამოსახულება `cout << mySquare.side << endl;`

მოგვცემს კომპილაციის შეცდომას

```
error C2248:'Square::side': cannot access private member declared in class 'Square'
```

კლასის დახურულ (private) წევრებზე წვდომა

იმისათვის, რომ პროგრამის შესრულების ნებისმიერ მომენტში შესაძლებელი იყოს ობიექტის **private** მონაცემების შეცვლა, კლასში უნდა გვქონდეს სპეციალური ფუნქციები (ამბობენ, Set ტიპის ფუნქციები). თუ პროგრამაში დაგვჭირდა ობიექტის დახურული მონაცემების მნიშვნელობები, მათ "ამოკითხვას" აგრეთვე ასრულებენ კლასის სპეციალური ფუნქციები (ე.წ. Get ტიპის ფუნქციები).

```
#include <iostream>
using namespace std;
class Square
{
      double side;
  public:
      Square();
      double getS();     // ფუნქცია დააბრუნებს side ველის მნიშვნელობას
      void setS(double);     // ფუნქცია მიანიჭებს side ველს ახალ მნიშვნელობას
      double Area();
      double Perimeter();
};
```

```
// კლასის იმპლემენტაციის ნაწილი
Square::Square() :side(5){}
double Square::getS(){ return side; }
void Square::setS(double s){ side = s; }
double Square::Area(){ return side * side; }
double Square::Perimeter(){ return 4 * side; }

int main()
{
    Square mySquare;
    cout << "The side of class object is " << mySquare.getS() << endl
         << "Area is " << mySquare.Area()
         << "\nPerimeter is " << mySquare.Perimeter() << endl;
    mySquare.setS(2.5);
    cout << "\nThe Area of class object with side " << mySquare.getS()
         << " is " << mySquare.Area()<< "\nPerimeter is "
         << mySquare.Perimeter() << endl;
}
```

პროგრამის შესრულების შედეგია:

```
                                OUTPUT
The side of class object is 5
Area is 25
Perimeter is 20

The Area of class object with side 2.5 is 6.25
Perimeter is 10
Press any key to continue . . .
```

კონსტრუქტორის გადატვირთვა

განხილულ მაგალითში `Square otherSquare(10.5);` ობიექტის შექმნის მცდელობა გამოიწვევს კომპილაციის შეცდომას, რადგან კლასის აღწერაში არა გვაქვს შესაბამისი ტიპის კონსტრუქტორი. `Square` კლასში რომ დავამატოთ კიდევ ერთ კონსტრუქტორი პარამეტრით კომპილაცია დასრულდება წარმატებით. ამ შემთხვევაში საქმე გვაქვს კონსტრუქტორის გადატვირთვასთან.

```
class Square {
    double side;
  public:
     // კონსტრუქტორის გადატვირთვა
      Square();
      Square(double ); // კონსტრუქტორების პროტოტიპები კლასში
      . . .
};
 // კონსტრუქტორების იმპლემენტაცია კლასის გარეთ
 Square::Square():side(3.5){}
 Square::Square(double s):side(s){}
 . . .
```

ახლა შეგვიძლია შემდეგი ორივე ობიექტის შექმნა

```
 Square mySquare;    // ობიექტის შექმნა უპარამეტრო კონსტრუქტორით
                     // კვადრატის გვერდი  3.5 –ის ტოლია
 Square otherSquare(10.5); // ობიექტის შექმნა პარამეტრიანი კონსტრუქტორით
                           // კვადრატის გვერდი  უდრის 10.5 –ს
```

კონსტრუქტორი ნაგულისხმევი არგუმენტებით

შეგვიძლია ორი გადატვირთული კონსტრუქტორი ჩავანაცვლოთ ერთით, რომლის პარამეტრს მინიჭებული აქვს საწყისი მნიშვნელობა, ე.ი. ნაგულისხმევი არგუმენტის მქონე კონსტრუქტორით:

```
Square(double = 5); // ნაგულისხმევი არგუმენტის მქონე კონსტრუქტორის პროტოტიპი კლასში

Square::Square(double s):side(s){} // ნაგულისხმევი არგუმენტის მქონე კონსტრუქტორის
                                   // იმპლემენტაცია კლასის გარეთ
```

ამ შემთხვევაში

`Square mySquare;` განაცხადის საფუძველზე შეიქმნება კვადრატი `5` -ის ტოლი გვერდით, ხოლო `Square otherSquare(7.75);` განაცხადის საფუძველზე - კვადრატს `7.75` -ის ტოლი გვერდით.

მაგალითად, შემდეგი პროგრამა

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

class Square {
      double side;
  public:
      Square(double = 5.);
      double Area();
      void showSquare();
};

// კლასის ფუნქციების იმპლემენტაცია
 Square::Square(double s):side(s){}
 double Square::Area(){
    return side * side;
 }
```

```
void Square::showSquare(){
    cout << "The Square's side is " << side << endl
         << "Area is " << Area() << endl;
 }
int main()
{
   Square S1;
   S1.showSquare();
   Square S2(10.3);
   S2.showSquare();
}
```

დაბეჭდავს

```
The Square's side is 3.5
Area is 12.25
The Square's side is 10.3
Area is 106.09
Press any key to continue . . .
```

თუ კლასში არა გვაქვს არც ერთი კონსტრუქტორი, კომპილერი თავად შექმნის ნაგულისხმევ (უპარამეტრო) კონსტრუქტორს როგორც კლასის ღია `inline` ფუნქციას და გამოიყენებს მას ობიექტის შესაქმნელად.

ამასთან, როდესაც კლასი შეიცავს მომხმარებლის მიერ შემოღებულ ერთს მაიმც კონსტრუქტორს, კომპილერი ნაგულისხმევ კონსტრუქტორს აღარ ქმნის.

C++11 სტანდარტის მიხედვით მოხმარებელს შეუძლია აიძულოს კომპილერი ნაგულისხმევი კონსტრუქტორის გენერირება მაშინაც, როდასაც კლასში არსებობს სხვა კონსტრუქტორები. ამისათვის კლასის აღწერაში უნდა ჩაიწეროს ასეთი კონსტრუქტორის პროტოტიპი მომსახურე **default** სიტყვის თანხლებით.

შემდეგ მაგალითში აიგება კლასი, რომლის დახურულ ველს **side** მინიჭებული აქვს საწყისი მნიშვნელობა. იგულისხმება, რომ **Square A;** განაცხადის შემთხვევაში შეიქმნება ობიექტი 5.-ის ტოლი ველით, თუ კომპილერი დააგენერირებს ნაგულისხმევ კონსტრუქტორს. რადგან კლასი შეიცავს რამდენიმე პარამეტრიან კონსტრუქტორს, ნაგულისხმევი კონსტრუქტორის შექმნას კომპილერს აიძულებს პროტოტიპი

```
Square() = default;
```

```
#include <iostream>
#include <fstream>
using namespace std;
class Square
{
      double side = 5.;
   public:
      Square() = default;
      Square(double);
      Square(istream&);
      Square(ifstream&);
      void showSquare();
};
// Implementation
Square::Square(double s) :side(s){}
Square::Square(istream& x){
     cout << "Enter side ";
     x >> side;
}
Square::Square(ifstream& x){
     x >> side;
}
void Square::showSquare(){
     cout << "side = " << side << endl;
}
```

```
int main()
{
    Square A;
    A.showSquare();
    Square B(10.25);
    B.showSquare();
    Square C(cin);
    C.showSquare();
    ifstream fin("squares.txt");
    // "squares.txt" contains 3.75
    Square D(fin);
    D.showSquare();
}
```

OUTPUT

```
side = 5
side = 10.25
Enter side 2.5
side = 2.5
side = 3.75
Press any key to continue . . .
```

საშინაო დავალება: გაარჩიეთ და გააანალიზეთ შემდეგი პროგრამა

```
#include <iostream>
#include <sstream>
using namespace std;
class Time
{
  public:
       Time(int = 12, int = 25, int = 40);
       Time(string);
       void showTime();
  private:
       int hour;
       int minute;
       int second;
};

Time::Time(int h, int m, int s){
     hour = h;  minute = m;  second = s;
}
Time::Time(string s)
{
     istringstream in(s);
     in >> hour; in.get();
     in >> minute; in.get();
     in >> second;
}
void Time::showTime()
{
   cout << ((hour == 0 || hour == 12) ? 12 : hour % 12)
        << ":" << (minute < 10 ? "0" : "") << minute
        << ":" << (second < 10 ? "0" : "") << second << endl;
}
```

```
int main()
{
   Time dt,  st("4:17:50"),
        t1(18, 30), t2(3, 12, 30);
   cout << "default-Time ";
   dt.showTime();
   cout << "string-Time ";
   st.showTime();
   cout << "t1 = ";   t1.showTime();
   cout << "t2 = ";   t2.showTime();
}
```

OUTPUT

```
default-Time 12:25:40
string-Time 4:17:50
t1 = 6:30:40
t2 = 3:12:30
Press any key to continue . . .
```