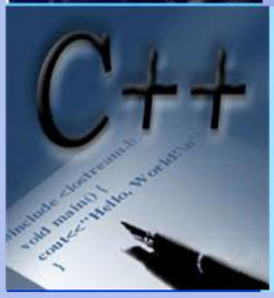
გულნარა ჯანელიძე, თამარ მეფარიშვილი

ობიექტზე ორიენტირებული დაპროგრამება



მეთოდური მითითებები ლაბორატორიული სამუშაოების შესასრულებლად

GTU

საქართველოს ტექნიკური უნივერსიტეტი

გულნარა ჯანელიძე, თამარ მეფარიშვილი

ობიექტზე ორიენტირებული დაპროგრამება

მეთოდური მითითებები ლაბორატორიული სამუშაოების შესასრულებლად დაპროგრამების C++ ენაზე

თბილისი
2014

უაკ. 681.3.06

მეთოდურ სახელმძღვანელოში წარმოდგენილია ლაბორატორიული სამუშაოების შესრულების თანამიმდევრობა საგანში „ობიექტზე ორიენტირებული დაპროგრამება c++ ენის ბაზაზე“.

მეთოდური სახელმძღვანელო განკუთვნილია ინფორმატიკის სფეროს სტუდენტებისათვის.

ISBN 978-9941-0-6426-5

შინაარსი

ლაბორატორიული სამუშაო 1.....5
ფუნქციებთან მუშაობა
ლაბორატორიული სამუშაო 2..... 12
კლასი, ობიექტი. მარტივი კლასის შექმნა
ლაბორატორიული სამუშაო 3..... 17
კლასის მეთოდების განსაზღვრა კლასის გარეთ
ლაბორატორიული სამუშაო 4.....23
კერძო და საჯარო მონაცემები
ლაბორატორიული სამუშაო 5.....30
კონსტრუქტორი და დესტრუქტორი.
კონსტრუქტორის შექმნა, გამოძახება.
კონსტრუქტორის გადატვირთვა.
დესტრუქტორი. მისი შექმნა და გამოყენება
ლაბორატორიული სამუშაო 6.....43
ოპერატორების გადატვირთვა
ლაბორატორიული სამუშაო 7.....51
სტატიკური ფუნქციები და მონაცემ-ელემენტები
ლაბორატორიული სამუშაო 8.....59
მემკვიდრეობითობა.
მარტივი მემკვიდრეობითობის მაგალითები
ლაბორატორიული სამუშაო 9..... 71
მრავლობითი მემკვიდრეობითობა
ლაბორატორიული სამუშაო 10.....80
საჯარო მემკვიდრეობითობის მაგალითები.
კერძო და დაცული მემკვიდრეობითობა

ლაბორატორიული სამუშაო 11 88
მემბრანი კლასები და მემბრანი ფუნქციები
ლაბორატორიული სამუშაო 12 105
პოლიმორფიზმი. ვირტუალური ფუნქციები
ლაბორატორიული სამუშაო 13 117
ფუნქციების და კლასების შაბლონების შექმნა
ფუნქციების და კლასების შაბლონების გამოყენება
ლაბორატორიული სამუშაო 14 134
cin და cout დამატებითი შესაძლებლობები
ლაბორატორიული სამუშაო 15 143
შეტანა-გამოტანის ფაილური ოპერაციები
ლიტერატურა 155

ლაბორატორიული სამუშაო 1
თემა: ფუნქციებთან მუშაობა

განვიხილოთ ამოცანები ფუნქციების გამოყენებით:

1. შექმენით ფუნქცია **void** print40star(**void**), რომელიც დაბეჭდავს 40 სიმბოლო '*' -ს.

```
#include<iostream.h>
#include<stdlib.h>
void print40star(void);
int main()
{
    print40star();
     system("pause");
    return 0;
}
void print40star(void){
    int i;          /* სიმბოლოების მთვლელი */
    for(i=1; i<=40; i++)
        cout<<"*";
}
```

პროგრამის შესრულების შედეგი:

```
    ****************************************

****

    Press any key to continue . . .
```

2. შექმენით ფუნქცია **void** f(**int** k); რომელიც ერთ სტრიქონად გამოიტანს ეკრანზე k ცალ შებრუნებულ სლეშს (‘\’).

```
#include <iostream.h>
#include <stdlib.h>
void f(int);
```

```
int main()
{
    int k;       /* სტრიქონში სიმბოლოთა რაოდენობა */
    cout<<"ShemoitaneT k : "<<endl;
    cin>>k;
    f(k);
    system("pause");
    return 0;
}
void f(int p){
    int i;     /* სიმბოლოების მთვლელი */
    for(i=1; i<=p; i++)
        cout<<"\");
    cout<<"\n";
}
```

პროგრამის შესრულების შედეგი:

```
ShemoitaneT k : 14

\\\\\\\\\\\\\\

Press any key to continue .
```

. .

3. შექმენით ფუნქცია, რომელიც დაბეჭდავს თქვენს სახელსა და გვარს. 3 ჯერ გამოიძახეთ ფუნქცია ძირითად პროგრამაში.

```
#include <iostream.h>
#include <stdlib.h>

void me_var(void);

int main()
{
    me_var();
    me_var();
    me_var();
    system("pause");
    return 0;
```

```
}
void  me_var(void){
cout<<"მე ვარ სანდრო ლომჯარია"<<endl;//აქ დაწერეთ თქვენი სახელი და გვარი
}
```

4. შექმენით ფუნქცია, რომელიც 12 სვეტად გამოიტანს ეკრანზე მთელ რიცხვებს 3-დან 150-მდე (ჩათვლით). გამოიძახეთ ფუნქცია ძირითად პროგრამაში.

```
#include <iostream.h>
#include <stdlib.h>
void print(void);
int main()
{
    print();
    system("pause");
    return 0;
}
void print(void){
    int i;     /* მთელი რიცხვი */
    int k;     /* რიცხვების მთვლელი */
    for(i=3, k=1; i<=150; i++, k++){
        cout<<"      "<<i;
        if(k%12 ==  0) cout<<"\n";
    }
    cout<<"\n";
}
```

5. შექმენით ფუნქცია, რომელიც დააბრუნებს მთელი რიცხვის კუბს. ეს რიცხვი ფუნქციის პარამეტრია. გამოიყენეთ ფუნქცია main-ში და დაბეჭდეთ 1-დან 20-მდე რიცხვთა კუბები.

```
#include <iostream.h>
#include <stdlib.h>
int cube(int );
```

```
int main()
{
     int a;    /* მთელი რიცხვი */
     int n;    /* ფუნქციით დაბრუნებული მნიშვნელობა */
     for(a=1; a<=20; a++){
        n =cube(a);
cout<<a<<"^3="<<n<<"\n";
     }
     system("pause");
     return 0;
}
int cube(int x){
   return x*x*x;    }
```

6. დაწერეთ ფუნქცია, რომლის პარამეტრი არის ნამდვილ რიცხვთა მასივი და რომელიც დაბეჭდავს ამ მასივის ელემენტებს შებრუნებული რიგით. მასივში N რიცხვია.

```
#include <iostream.h>
#include<stdlib.h>
const int N =7;
void bechdva(float a[]);
int main ()
{
     float mass[N];  /* განაცხადი მასივზე */
     int i;          /* მასივის ელემენტის ინდექსი */
     cout<<"SemoitaneT namdvili ricxvebi\n";
     for(i=0; i<N; i++) cin>>mass[i];
     bechdva(mass);
     system("PAUSE");
     return 0; }
void bechdva(float a[]){
     int i;
     cout<<"\nMasivi sebrunebuli rigiT\n";
     for(i=N-1; i>=0; i--)
         cout<<a[i]<<" ";
     cout<<"\n"; }
```

პროგრამის შესრულების შედეგი:

```
SemoitaneT 7 namdvili ricxvi
1.2 3.4 0 -6.32 7.25 9 -10.3

Masivi sebrunebuli rigiT
-10.30 9.00 7.25 -6.32 0.00 3.40 1.20
Press any key to continue . . .
```

7. დაწერეთ 2 ფუნქცია: პირველმა უნდა შეავსოს $K \times P$ ორგანზომილებიანი მასივი (მატრიცა) შემთხვევითი ნამდვილი რიცხვებით [0,10] დიაპაზონიდან, მეორემ – დაბეჭდოს ეს მატრიცა სტრიქონ-სტრიქონ. K და P – შესაბამისად მატრიცის სტრიქონებისა და სვეტების რაოდენობა – შემოიღეთ კონსტანტების სახით. გამოიძახეთ ორივე ფუნქცია ძირითად პროგრამაში.

```
#include <iostream.h>
#include <stdlib.h>
const int K =4, P =5;
void inputArray(float a[][P]);
void printArray(float a[][P]);
int main (){
   float matrix[K][P];
   inputArray(matrix);
   printArray(matrix);
   system("PAUSE ");
   return 0;
}
void inputArray(float a[][P]){
   int i, j;
   for(i=0; i<K; i++)
      for(j=0; j<P; j++)
         a[i][j]=rand()%10001/1000.;
}
void printArray(float a[][P]){
```

```
    int i, j;
    for(i=0; i<K; i++){
       for(j=0; j<P; j++)
          cout<<a[i][j]<<"\t";
          cout<<"\n";
    }
}
```

პროგრამის შესრულების შედეგი:

```
0.041  8.466  6.334  6.498  9.168
5.723  1.477  9.356  6.960  4.462
5.705  8.143  3.279  6.826  9.961
0.491  2.995  1.941  4.827  5.436
Press any key to continue . . .
```

დავალება:

1. დაწერეთ ფუნქცია, რომლის პარამეტრი არის მთელი რიცხვების მასივი. მასივის ელემენტთა რაოდენობა K წარმოადგენს კონსტანტას. ფუნქციამ უნდა დაადგინოს და დააბრუნოს მასივის:
   ა) 4-ის ჯერადი ელემენტების რაოდენობა. თუ ასეთი ელემენტები მასივში არ არის, დააბრუნოს –1;
   ბ) დადებითი ელემენტების რაოდენობა. თუ ასეთი ელემენტები მასივში არ არის, დააბრუნოს –1;
   გ) უდიდესი ელემენტის ინდექსი;
   დ) უმცირესი ელემენტის მნიშვნელობა.
2. დაწერეთ ფუნქცია, რომლის პარამეტრიც არის მთელ რიცხვთა მატრიცა $N \times M$. მატრიცის განზომილებები N და M წარმოადგენს კონსტანტებს. ფუნქციამ უნდა გამოითვალოს და დააბრუნოს მატრიცის 3-ის ჯერადი რიცხვების ჯამი.
3. დაწერეთ ფუნქცია, რომლის პარამეტრი არის ნამდვილ რიცხვთა მატრიცა $N \times M$. მატრიცის

განზომილებები N და M წარმოადგენს კონსტანტებს. ფუნქციამ უნდა გამოითვალოს და დააბრუნოს მატრიცის:

- არაუარყოფითი ელემენტების რაოდენობა;
- იმ ელემენტების ჯამი, რომლებიც ეკუთვნის [-10, 15] შუალედს;
- იმ ელემენტების რაოდენობა, რომლებიც არ ეკუთვნის [2, 30] შუალედს.

4. დაწერეთ ფუნქცია, რომლის პარამეტრი არის მთელი მატრიცა $M \times K$. მატრიცის განზომილებები M და K წარმოადგენს კონსტანტებს. ფუნქციამ უნდა დაადგინოს და დაბეჭდოს მასივის უდიდესი ელემენტის ინდექსები.

ლაბორატორიული სამუშაო 2
თემა: კლასი, ობიექტი. მარტივი კლასის შექმნა

კლასი წარმოადგენს მთავარ ინსტრუმენტულ საშუალებას C++ ენაში ობიექტზე ორიენტირებული დაპროგრამებისათვის. კლასი ძალიან ჰგავს სტრუქტურას, რომელშიც დაჯგუფებული მონაცემები წარმოადგენენ რაღაც ობიექტის აღწერას. ამ მონაცემებით სარგებლობენ ფუნქციები, რომელთაც მეთოდებს უწოდებენ.

ობიექტი არის არსება, როგორიც შეიძლება იყოს თანამშრომელი, წიგნი, სტუდენტი და სხვა. კლასს გამოვიყენებთ ამათუიმ ობიექტის განსაზღვრისათვის. კლასში უნდა ჩავრთოთ ობიექტის შესახებ იმდენი ინფორმაცია, რამდენსაც მოითხოვს ამოცანა. შეიძლება შეიქმნას კლასი ერთი პროგრამისთვის და გამოყენებულ იქნას სხვადასხვა პროგრამებისთვის.

განვიხილოთ კლასის ზოგადი სტრუქტურა:

```
class class_name

{
  int data_member; // მონაცემ-ელემენტი
  void show_member(int); // ფუნქცია-ელემენტი
};
```

როგორც ვხედავთ კლასს უნდა ჰქონდეს უნიკალური სახელი, შემდეგ იწერება ფიგურული ფრჩხილები, კლასის ელემენტები და იხურება ფიგურული ფრჩხილები.

კლასის გამოცხადების შემდეგ შეიძლება გამოვაცხადოთ ამ კლასის ტიპის ცვლადები, რომელთაც ჰქვია ობიექტები. ობიექტების გამოძახება ზოგადად ხდება შემდეგი სახით:

```
class_name object_one, object_two, object_three;
```

განვიხილოთ მაგალითი: შევქმნათ კლასი ***employee***, რომელიც შეიცავს მონაცემების და მეთოდების განსაზღვრას:

```
class employee

{
  public:
  char name[64] ;
  long employee_id;
  float salary;
  void show_employee(void)

  {
    cout << "saxeli: " << name << endl;
    cout << "TanamSromlis nomeri: " << employee_id << endl;
    cout << "xelfasi: " << salary << endl;
  };
};
```

მოცემულ მაგალითში კლასი შეიცავს სამ ცვლადს და ერთ ფუნქცია-ელემენტს. ყურადღება მიქცეთ ***public*** ჭდის გამოყენებას კლასის განსაზღვრაში. კლასის ელემენტები შეიძლება იყოს საჯარო და კერძო, რაზეც

დამოკიდებულია როგორ მიმართავს ჩვენი პროგრამა კლასის ელემენტებს. მოცემულ მაგალითში ყველა ელემენტი არის საჯარო, რაც ნიშნავს, რომ პროგრამა ნებისმიერ ელემენტს მიმართავს ოპერატორ წერტილის გამოყენებით.

კლასის განსაზღვრის შემდეგ პროგრამაში შეგიძლიათ გამოაცხადოთ ობიექტები შემდეგი სახით:

employee worker, boss, secretary ;

განვიხილოთ პროგრამა, რომელიც ქმნის ***employee*** კლასის ორ ობიექტს. ოპერატორ წერტილის გამოყენებით პროგრამა ანიჭებს მონაცემებს მნიშვნელობებს. შემდგომ პროგრამა გამოიყენებს ***show_employee*** ფუნქციას თანამშრომლის შესახებ ინფორმაციის გამოსატანად:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class employee

{
public:
  char name [64];
  long employee_id;
  float salary;
  void show_employee(void)

  {
    cout << "saxeli: " << name << endl;
    cout << "TanamSromlis nomeri: " << employee_id << endl;
    cout << "xelfasi: " << salary << endl;
```

```
  };
};

void main(void)

{
  employee worker, boss;
  strcpy(worker.name, "John Doe");
  worker.employee_id = 12345;
  worker.salary = 25000;
  strcpy(boss.name, "Happy Jamsa");
  boss.employee_id = 101;
  boss.salary = 101101.00;
  worker.show_employee();
  boss.show_employee();
system(“pause”);
}
```

როგორც ვხედავთ პროგრამა აცხადებს *employee* ტიპის ორ ობიექტს: *worker, boss.* შემდგომ იყენებს ოპერატორ წერტილს ელემენტებისათვის მნიშვნელობების მისანიჭებლად და *show_employee* ფუნქციის გამოსაძახებლად.

როგორც ვხედავთ ობიექტზე ორიენტირებულ დაპროგრამებაში ჩვენი პროგრამები ფოკუსირდება ობიექტებზე და ამ ობიექტებზე შესასრულებელ ოპერაციებზე.

დასკვნა:

- კლასის გამოცხადებისათვის პროგრამამ უნდა მიუთითოს კლასის სახელი, კლასის მონაცემ-ელემენტები და კლასის ფუნქციები(მეოდები);
- კლასის გამოცხადებას უზრუნველყოფს შაბლონი, რომლის დახმარებით თქვენი პროგრამა შეძლებს შექმნას ამ კლასის ტიპის ობიექტები;
- პროგრამა კლასის მონაცემ-ელემენტებს ანიჭებს მნიშვნელობებს ოპერატორ-წერტილის მეშვეობით;
- პროგრამა კლასის ფუნქცია-ელემენტს იძახებს ოპერატორ-წერტილის გამოყენებით.

დავალება: შექმენით კლასი სტუდენტი, მონაცემებით: გვარი, სახელი, ჩარიცხვის წელი, ჯგუფის ნომერი. მიანიჭეთ ელემენტებს მნიშვნელობები და გამოიტანეთ მნიშვნელობები ფუნქციის მეშვეობით.

ლაბორატორიული სამუშაო 3
თემა: კლასის მეთოდების განსაზღვრა კლასის გარეთ

წინა მასალაში შექმნილ *employee* კლასში ფუნქცია განსაზღვრული იყო თვით კლასის შიგნით (ჩაშენებული (*inline*) ფუნქცია). ჩაშენებული ფუნქციის კლასის შიგნით განსაზღვრამ კლასის აღწერაში შეიძლება უწესრიგობა შეიტანოს. ალტერნატივის სახით ფუნქციის პროტოტიპი შეიძლება განვათავსოთ კლასის შიგნით, ხოლო შემდეგ ფუნქცია განვსაზღვროთ კლასის გარეთ. ამ შემთხვევაში კლასის განსაზღვრა მიიღებს შემდეგ სახეს:

```
class employee
{
public:
  char name[64];
  long employee_id;
  float salary;
  void show_employee(void); |————————————>ფუნქციის
პროტოტიპი
};
```

რადგან სხვადასხვა კლასებში შეიძლება გამოვიყენოთ ფუნქციები ერთნაირი სახელებით, ამიტომ ფუნქციას უნდა დავურთოთ კლასის სახელი და გლობალური ნებართვის ოპერატორი (::). ამ შემთხვევაში ფუნქციის განსაზღვრა გამოიყურება შემდეგი სახით:

```
void employee:: show_employee (void) //-------------->კლასის
სახელი
```

```
{
  cout << "saxeli: " << name << endl;
cout << "TanamSromlis  nomeri: " << employee_id << endl;
  cout << "xelfasi: " << salary << endl;
};
```

შემდეგი პროგრამა *show_employee* ფუნქციის განსაზღვრებას განათავსებს კლასის გარეთ. კლასის სახელის მისათითებლად გამოიყენებს გლობალური ნებართვის ოპერატორს:

```
#include <iostream.h>
#include <string.h>
#include<stdlib>
class employee
{
public:
  char name [64];
  long employee_id;
  float salary;
  void show_employee(void);
};
void employee::show_employee(void)
{
  cout << "saxeli: " << name << endl;
  cout << "TanamSromlis nomeri: " << employee_id << endl;
  cout << "xelfasi: " << salary << endl;
};
void main(void)
        {
  employee worker, boss;
  strcpy(worker.name, "John Doe");
```

```
  worker.employee_id = 12345;
  worker.salary = 25000;
  strcpy(boss.name, "Happy Jamsa");
  boss.employee_id = 101;
  boss.salary = 101101.00;
  worker.show_employee();
  boss.show_employee();
  system(“pause”);
}
```

კლასის მეთოდები

კლასი პროგრამას საშუალებას აძლევს ობიექტის მონაცემები და ობიექტის ფუნქციები(მეთოდები), რომლებიც ოპერირებენ ამ მონაცემებით, დაჯგუფებულ იქნას ერთ ცვლადში. ჩვენ გვაქვს ობიექტის მეთოდების განსაზღვრის ორი შესაძლებლობა. პირველი – ფუნქციის მთელი კოდი შეიძლება ჩართოთ კლასის განსაზღვრის შიგნით. შეიძლება ეს მოხერხებულად მოგვეჩვენოს, მაგრამ როდესაც კლასები რთულდება და მოიცავს რამდენიმე მეთოდს, ფუნქციის ოპერატორებმა შეიძლება უწესრიგობა შეიტანოს კლასის განსაზღვრაში. ამდენად უმეტესობა პროგრამებში ფუნქცია განისაზღვრება კლასის გარეთ. კლასის განსაზღვრაში უნდა იყოს ფუნქციის პროტოტიპი, რომელიც შეიცავს ფუნქციის სახელს, დასაბრუნებელი მნიშვნელობის ტიპს და პარამეტრების ტიპებს. კლასის გარეთ ფუნქციის განსაზღვრა ზოგადად გამოიყურება შემდეგი სახით:

return_type class_name::function_name(parameters)
{ // ოპერატორები }

განვიხილოთ მაგალითი

პროგრამა ქმნის *dog* კლასს, რომელიც შეიცავს რამდენიმე მონაცემს და *show_breed* ფუნქციას. ფუნქცია განსაზღვრულია კლასის გარეთ. შემდეგ პროგრამა ქმნის *dog* ტიპის ორ ობიექტს და გამოიტანს ინფორმაციას ცალკეული ძაღლის შესახებ:

```
#include <iostream.h>
#include <string.h>
#include<stdlib>
class dogs
{
public:
  char breed[64];
  int average_weight;
  int average_height;
  void show_breed(void) ;
};
void dogs::show_breed(void)
{
  cout << "jishi: " << breed << endl;
  cout << "sashualo cona: " << average_weight << endl;
  cout << "sashualo simagle: " << average_height << endl;
}
void main(void)
{
  dogs happy, matt;
  strcpy(happy.breed, "dalmatineli") ;
  happy.average_weight = 58;
  happy.average_height = 24;
  strcpy(matt.breed, "kolli");
```

```
  matt.average_weight =22;
  matt.average_height = 15;
  happy.show_breed() ;
  matt.show_breed();
  system("pause");
}
```

ამ პროგრამაში გამოყენებულია ჭდე ***public***, რომელიც კლასის ელემენტებს მისაწვდომს გახდის მთელი პროგრამისათვის.

დასკვნა:

მოცემული მასალის შესწავლის შედეგად თქვენ უნდა იცოდეთ, რომ:

- ობიექტი წარმოადგენს არსებას, რომელთან მიმართებით თქვენი პროგრამა ასრულებს სხვადასხვა ოპერაციას;
- პროგრამა C++ ენაზე წარმოადგენს ობიექტებს, რომელიც გადმოცემულია კლასების მეშვეობით;
- კლასი სტრუქტურის მსგავსად შეიცავს ელემენტებს. კლასის ელემენტები შეიძლება იყოს მონაცემები ან ფუნქციები(მეთოდები), რომლებიც ოპერირებენ ამ მონაცემებით;
- ცალკეულ კლასს აქვს უნიკალური სახელი;
- კლასის განსაზღვრის შემდეგ თქვენ შეგიძლიათ გამოაცხადოთ ამ კლასის ობიექტები, სადაც გამოიყენებთ კლასის სახელს ტიპის სახით;
- კლასის ელემენტებზე მიმართვისათვის(როგორც მონაცემებზე, ასევე ფუნქციებზე) თქვენი პროგრამები გამოიყენებენ ოპერატორ წერტილს;

– ფუნქცია შეიძლება განისაზღვროს კლასის შიგნით ან გარეთ. თუ ფუნქციას განსაზღვრავთ კლასის განსაზღვრის გარეთ, უნდა მიუთითოთ კლასის სახელი და გლობალური ნებართვის სახელი, მაგალითად: *class:: function.*

დავალება: შექმენით კლასი ზღვა. რომელიც შეიცავს მონაცემებს: სახელი, სიღრმე, ფართობი. ფუნქცია განსაზღვრეთ კლასის გარეთ. შემდეგ შექმენით ამ კლასის ტიპის სამი ობიექტი და გამოიტანეთ ინფორმაცია ცალკეული ზღვის შესახებ.

ლაბორატორიული სამუშაო 4
თემა: კერძო და საჯარო მონაცემები

მოცემული მასალიდან თქვენ შეისწავლით თუ როგორ მართავენ public(საჯარო) და private(კერძო) ატრიბუტები პროგრამაში კლასის ელემენტებზე წვდომას. პროგრამა საჯარო ელემენტებს მიმართავს ნებისმიერი ფუნქციიდან. მეორეს მხრივ პროგრამამ შეიძლება მიმართოს კერძო ელემენტებს მხოლოდ მოცემული კლასის ფუნქციებიდან.

ინფორმაციის დაფარვა. თქვენს პროგრამას არ მოეთხოვება იცოდეს, თუ როგორ მუშაობენ მეთოდები. პროგრამამ უნდა იცოდეს მხოლოდ თუ რა ამოცანას ასრულებს მეთოდები. მაგალითად, გვაქვს კლასი file. თქვენმა პროგრამამ უნდა იცოდეს მხოლოდ ის, რომ ეს კლასი უზრუნველყოფს file.print მეთოდს, რომელიც ბეჭდავს მიმდინარე ფაილს ფორმატირებულ ასლს, ან file.delete, რომელიც წაშლის ფაილს. თქვენს პროგრამას არ მოეთხოვება როგორ მუშაობს ეს ორი მეთოდი. სხვა სიტყვებით, პროგრამა კლასს უნდა განიხილავდეს, როგორც „შავ ყუთს“. პროგრამამ იცის რომელი მეთოდები უნდა გამოიძახოს და რომელი პარამეტრები გადასცეს მას, მაგრამ პროგრამამ არაფერი იცის რეალურ სამუშაოზე, რომელიც სრულდება კლასის შიგნით. ინფორმაციის დაფარვა პროცესია, რომლის შედეგად პროგრამას წარედგინება მხოლოდ მინიმალური ინფორმაცია, რომელიც საჭიროა კლასის გამოსაყენებლად. კლასის საჯარო და კერძო ელემენტები დაგეხმარებათ მიიღოთ ინფორმაცია,

რომელიც დაფარულია პროგრამაში. წინა მეცადინეობაზე შექმნილ კლასებში გამოყენებული იყო public ჭდე, ამიტომ პროგრამა კლასის ნებისმიერ ელემენტს მიმართავდა ოპერატორ წერტილის გამოყენებით.

კლასის შექმნისას თქვენ შეიძლება გქონდეთ ელემენტები, რომელთა მნიშვნელობებიც გამოიყენება მხოლოდ კლასის შიგნით, მაგრამ მათზე მიმართვა პროგრამიდან არ არის აუცილებელი. ასეთი ელემენტები არის კერძო და ისინი დამალული უნდა იქნან პროგრამისაგან. თუ თქვენ არ გამოიყენებთ public ჭდეს, დუმილით კლასის ყველა ელემენტი ჩაითვლება კერძოდ. პროგრამა კერძო ელემენტებს ვერ მიმართავს ოპერატორ წერტილის გამოყენებით. კლასის კერძო ელემენტებზე მიმართვა შეუძლიათ მხოლოდ თვით ამ კლასის ელემენტებს.

კლასის შექმნისას ელემენტები უნდა დაყოთ საჯაროდ და კერძოდ, როგორც ქვემოთაა ნაჩვენები:

```
class some_class
{
public:
  int some_variable;
  void initialize_private(int, float);   //———> საჯარო
ელემენტები
  void show_data(void);
private:
  int key_value;              //————————————————————> კერძო
ელემენტები
```

```
  float key_number;
}
```

მოცემულ შემთხვევაში პროგრამას შეუძლია გამოიყენოს ოპერატორი წერტილი საჯარო ელემენტებზე მიმართვისათვის, როგორც ქვემოთ არის ნაჩვენები:

```
some_class object; // შეიქმნას ობიექტი
object.some_variable = 1001;
object.initialize_private(2002, 1.2345);
object.show_data()
```

თუ პროგრამიდან ვეცდებით მივმართოთ key_value ან key_number კერძო ელემენტებს წერტილის გამოყენებით, კომპილატორი შეგვატყობინებს სინტაქსურ შეცდომაზე.

ელემენტების კერძო სახით გამოცხადებით თქვენ დაიცავთ კლასის ელემენტებს პირდაპირი წვდომისაგან. ასეთ ელემენტებს პროგრამა მნიშვნელობებს ვერ მიანიჭებს ოპერატორ წერტილის გამოყენებით. იმის ნაცვლად, რომ მიანიჭოს მნიშვნელობა, პროგრამამ უნდა გამოიძახოს კლასის მეთოდი.

კლასის მეთოდები, რომლებიც მართავენ მონაცემთა ელემენტებზე წვდომას, წარმოადგენენ ინტერფეისულ ფუნქციებს. კლასების შექმნისას თქვენ გამოიყენებთ ასეთ ფუნქციებს კლასების მონაცემების დასაცავად.

შემდეგი განვიხილოთ პროგრამა, რომელიც იყენებს კლასის საჯარო და კერძო ნაწილს:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
```

```
class employee
{
public:
   int assign_values(char *, long, float);
   void show_employee(void);
   int change_salary(float);
   long get_id(void);
private:
   char name [64];
   long employee_id;
   float salary;
};
int employee::assign_values(char *emp_name, long emp_id,
float emp_salary)
{
   strcpy(name, emp_name);
   employee_id = emp_id;
   if (emp_salary < 50000.0)
    {
      salary = emp_salary;
      return(0);   // წარმატებით
    }
   else
return(-1); }  // დაუშვებელი  ხელფასი
void employee::show_employee(void)
{
   cout << "TanamSromeli " << name << endl;
   cout << "Tanamsromlis nomeri: " << employee_id << endl;
```

```
  cout << "xelfasi: " << salary << endl;
}
int employee::change_salary(float  new_salary)
{
  if (new_salary < 50000.0)
  {
    salary = new_salary;
    return(0); } //  წარმატებით
   else return(-1);}   //  დაუშვებელი  ხელფასი
void main(void)
{
  employee  worker;
  if (worker.assign_values("Happy Jamsa", 101, 10101.0) == 0)
  {
    cout << "TanamSromels daeniSna Semdegi mniSvnelobebi" <<
endl;}
    worker.show_employee();
    if (worker.change_salary(35000.00) == 0)
    {
      cout << "daniSnulia axali xelfasi" << endl;
      worker.show_employee();
    }
    else
  cout << "naCvenebia  dauSvebeli xelfasi" << endl;
system(“pause”);
}
```

როგორც ხედავთ კლასი იცავს თავის ყველა მონაცემ-ელემენტს, რადგან აცხადებს როგორც კერძოს. პროგრამა ამ ელემენტებზე წვდომას ახორციელებს ინტერფეისული ფუნქციებით.

assign_values მეთოდი ინიციალიზებას უკეთებს კლასის კერძო მონაცემებს. მეთოდი იყენებს პირობის ოპერატორს, რათა დარწმუნდეს, რომ მიენიჭება დასაშვები ხელფასი. მეთოდს show_employee მოცემულ შემთხვევაში გამოაქვს კერძო მონაცემები. change_salary ახორციელებს ხელფასის ახალი ხელფასით შეცვლას, პირობის ჭეშმარიტობის შემთხვევაში. პროგრამის მიერ კერძო მონაცემებზე წვდომა განხორციელებულია ფუნქციებით. პროგრამის კომპილაციის და გაშვების შემდეგ, შეეცადეთ კერძო ელემენტებს მიმართოთ წერტილით, კომპილატორი შეგატყობინებთ სინტაქსური შეცდომის შესახებ.

კლასის ელემენტებისათვის გამოიყენება გლობალური ნებართვის ოპერატორი იმისათვის, რომ გასაგები იყოს რომელი სახელები რომელ კლასს შეესაბამება. ქვემოთ მოყვანილ ფუნქციაში ჩანს, რომ მონაცემები შეესაბამება employee კლასს:

```
int employee::assign_values(char *name, long employee_id, float
salary)

{
  strcpy(employee::name, name) ;
  employee::employee_id = employee_id;
  if (salary < 50000.0)

{
    employee::salary = salary;
```

```
  return(0); } else          // წარმატებით
  return(-1);                //დაუშვებელი ხელფასი
}
```

ჩვენს მაგალითში კერძო ელემენტები იყო მონაცემ-ელემენტები. ზოგჯერ საჭიროა, რომ შეიქმნას ფუნქცია, რომელიც უნდა გამოიყენოს კლასის სხვა მეთოდებმა, მაგრამ პროგრამის დანარჩენი ნაწილისათვის ეს ფუნქცია იყოს დახურული. ასეთ შემთხვევაში ასეთი ფუნქციები უნდა გამოცხადდეს როგორც კერძო ელემენტები.

დასკვნა:

- კლასის ელემენტები შეიძლება იყოს საერთო და კერძო. პროგრამა საერთო ელემენტებს მიმართავს პირდაპირ, ოპერატორ წერტილის გამოყენებით. კერძო ელემენტებზე მიმართვა ხდება კლასის მეთოდების(ფუნქციების) გამოყენებით.
- დუმილით C++ ყველა ელემენტს კერძოდ ჩათვლის.
- კლასის ელემენტებში შეგიძლიათ გამოიყენოთ კლასის სახელი და გლობალური ნებართვის ოპერატორი, მაგალითად employee::name, სახელების შესაძლო კონფლიქტის თავიდან ასაცილებლად.

დავალება: მართკუთხედის სიგრძე და სიგანე აღწერეთ საჯარო ნაწილში, ხოლო ფართობი და პერიმეტრი კერძო ნაწილში. საჯაროში დაასახელეთ ორი ფუნქციის პროტოტიპი, რომელთაგან ერთი ითვლის მართკუთხედის ფართობს, ხოლო მეორე-პერიმეტრს. ფუნქციები განსაზღვრეთ კლასის გარეთ.

ლაბორატორიული სამუშაო 5

თემა: კონსტრუქტორი და დესტრუქტორი კონსტრუქტორის შექმნა, გამოძახება. კონსტრუქტორის გადატვირთვა. დესტრუქტორი. მისი შექმნა და გამოყენება

ობიექტის შექმნისას საჭიროა მათი ინიციალიზება. როგორც უკვე განვიხილეთ კერძო ელემენტებზე მიმართვა შესაძლებელია მხოლოდ კლასის ფუნქციებით. კლასის მონაცემების ინიციალიზაციის პროცესის გასამარტივებლად გამოიყენება სპეციალური ფუნქცია, რომელსაც ჰქვია კონსტრუქტორი, რომელიც გაიშვება ყოველი შექმნილი ობიექტისათვის. ხოლო დესტრუქტორი გაიშვება ობიექტის განადგურებისას.

დაიმახსოვრეთ, რომ:

- კონსტრუქტორი არის კლასის მეთოდი, რომელიც პროგრამას უადვილებს კლასის მონაცემ-ელემენტების ინიციალიზაციას;
- კონსტრუქტორს აქვს ისეთივე სახელი, როგორც კლასს;
- კონსტრუქტორს არა აქვს დასაბრუნებელი მნიშვნელობა;
- ყოველთვის, როდესაც პროგრამა ქმნის კლასის ცვლადს, C++ იძახებს კლასის კონსტრუქტორს, თუ იგი არსებობს;
- ობიექტისათვის გამოიყოფა მეხსიერება, ინფორმაციის შესანახად, როდესაც თქვენ ანადგურებთ ამ ობიექტს C++ იძახებს სპეციალურ დესტრუქტორს, რომელიც

ათავისუფლებს მეხსიერებას. ასუფთავებს მას ობიექტის შემდგომ.

- დესტრუქტორს აქვს ისეთივე სახელი, როგორც კლასს, მხოლოდ მას წინ უძღვის ~ სიმბოლო.
- დესტრუქტორს არა აქვს დასაბრუნებელი მნიშვნელობა.

კონსტრუქტორი წარმოიდგინეთ როგორც ფუნქცია, რომელიც გეხმარებათ თქვენ ობიექტის აშენებაში. ასევე დესტრუქტორი არის ფუნქცია, რომელიც გეხმარებათ ობიექტის განადგურებაში. დესტრუქტორი გამოიყენება მაშინ თუ ობიექტის განადგურებისას საჭიროა იმ მეხსიერების გათავისუფლება, რომელსაც იკავებს ობიექტი.

მარტივი კონსტრუქტორის შექმნა

შევქმნათ კლასი employee, შესაბამისად კოსტრუქტორსაც ექნება სახელი employee. კოსტრუქტორს არა აქვს დასაბრუნებელი მნიშვნელობა და იგი არც უნდა მიეთითოს.

```
class employee

{
public:
  employee(char *, long, float); //Конструктор
  void show_employee(void);
  int change_salary(float);
  long get_id(void);
private:
```

```
  char name [64];
  long employee_id;
  float salary;
};
```

მოცემულ პროგრამში თქვენ უნდა განსაზღვროთ კონსტრუქტორი ისევე, როგორც კლასის სხვა ნებისმიერი მეთოდი:

```
employee::employee(char *name, long employee_id, float salary)

{
  strcpy(employee::name, name) ;
  employee::employee_id = employee_id;
  if (salary < 50000.0)
  employee::salary = salary;
  else          //დაუშვებელი ხელფასი
  employee::salary = 0.0;
}
```

როგორც ხედავთ კონსტრუქტორი არ აბრუნებს მნიშვნელობას და არც გამოიყენება void ტიპი. მოცემულ შემთხვევაში კონსტრუქტორი გამოიყენებს გლობალური ნებართვის ოპერატორს და კლასის სახელს ცალკეული ელემენტის სახელის წინ. განვიხილოთ ჩვენი მაგალითი:

```
#include <iostream.h>
#include <string.h>
class employee
{
public:
```

```
  employee(char *, long, float);
  void show_employee(void);
  int change_salary(float) ;
  long get_id(void);
private:
  char name [64] ;
  long employee_id;
  float salary;
};

employee::employee(char *name, long employee_id, float salary)

{
  strcpy(employee::name, name) ;
  employee::employee_id = employee_id;
  if (salary < 50000.0)
    employee::salary = salary;
  else // დაუშვებელი ხელფასი
    employee::salary = 0.0;
}

void employee::show_employee(void)

{
  cout << "TanamSromeli: " << name << endl;
  cout << "TanamSromlis nomeri: " << employee_id << endl;
  cout << "xelfasi: " << salary << endl;
}

void main(void)
```

```
{
  employee worker("Happy Jamsa", 101, 10101.0);
  worker.show_employee();
}
```

მიაქციეთ ყურადღება, რომ worker ობიექტის გამოცხადების შემდეგ მოდის მრგვალი ფრჩხილები და საწყისი მნიშვნელობები, ისე როგორც ფუნქციის გამოცხადებისას. როდესაც თქვენ იყენებთ კონსტრუქტორს, მას გადასცემთ პარამეტრებს ობიექტის გამოცხადებისას:

```
employee worker("Happy Jamsa", 101, 10101.0);
```

თუ პროგრამაში მოითხოვება შეიქმნას employee კლასის რამდენიმე ობიექტი, თქვენ შეგიძლიათ ინიციალიზება გაუკეთოთ ცალკეულ მათგანს კონსტრუქტორის მეშვეობით, როგორ ქვემოთ არის ნაჩვენები:

```
employee worker("Happy Jamsa", 101, 10101.0);
employee secretary("John Doe", 57, 20000.0);
employee manager("Jane Doe", 1022, 30000.0);
```

ობიექტის ინიციალიზება კონსტრუქტორის მეშვეობით ზოგადად გამოიყურება შემდეგი სახით:

class_name object(valuel, value2, value3)

კონსტრუქტორი და პარამეტრები დუმილით

კონსტრუქტორში შეიძლება მითითებულ იქნას პარამეტრის მნიშვნელობა დუმილით. განვიხილოთ მაგალითი, სადაც employee კონსტრუქტორი იყენებს ხელფასს, რომლის დუმილით მნიშვნელობა არის 1000, თუმცა პროგრამაში უნდა მიეთითოს თანამშრომლის სახელი და ნომერი:

```
employee::employee(char *name, long employee_id, float salary =
10000.00)

{
  strcpy(employee::name, name);
  employee::employee_id = employee_id;
  if (salary < 50000.0)
    employee::salary = salary;
  else //დაუშვებელი ხელფასი
    employee::salary = 0.0;
}
```

კონსტრუქტორის გადატვირთვა

კლასი შეიძლება შეიცავდეს რამდენიმე კონსტრუქტორს. მაგალითად ქვემოთ მოყვანილი ორივე კონსტრუქტორს:

```
employee(char *, long, float);
employee(char *, long);
```

ამას ეწოდება კონსტრუქტორის გადატვირთვა.

განვიხილოთ ჩვენი პროგრამის რეალიზაცია, კონსტრუქტორის გადატვირთვის გათვალისწინებით:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class employee

{
public:
  employee(char *, long, float);
  employee(char *, long);
  void show_employee(void);
private:
  char name [64];
  long employee_id;
  float salary;
};

employee::employee(char *name, long employee_id, float salary)

{
  strcpy(employee::name, name);
  employee::employee_id = employee_id;
  if (salary < 50000.0) employee::salary = salary;
  else // დაუშვებელი ხელფასი
    employee::salary = 0.0;
}

employee::employee(char *name, long employee_id)

{
  strcpy(employee::name, name);
```

```
  employee::employee_id = employee_id;
  do

  {
   cout << "შეიტანეთ  ხელფასი " << name <<” saTvis”<<"
50000-ზე ნაკლები: ";
  cin >> employee::salary;
  }
  while (salary >= 50000.0);
}

void employee::show_employee(void)

{
  cout << "TanamSromeli: " << name << endl;
  cout << "TanamSromlis nomeri: " << employee_id << endl;
  cout << "xelfasi: " << salary << endl;
}

void main(void)

{
  employee worker("Happy Jamsa", 101, 10101.0);
  employee manager("Jane Doe", 102);
  worker.show_employee();
  manager.show_employee();
system(“pause”);

}
```

პროგრამის გაშვებისას გამოვა მოთხოვნა: შეიტანეთ ხელფასი Jane Doe - სათვის. ხელფასის შეტანის შემდეგ გამოვა ინფორმაცია ორივე თანამშრომლის შესახებ.

დესტრუქტორი

დესტრუქტორი ავტომატურად გაიშვება თითოეულ ჯერზე, როდესაც პროგრამა ანადგურებს ობიექტს. მოცემული მომენტისათვის თქვენ შეგიძლიათ შექმნათ და გაანადგუროთ ობიექტები პროგრამის შესრულების მომენტში. ასეთ შემთხვევებში აზრი აქვს დესტრუქტორის გამოყენებას.

დესტრუქტორი არის კლასის ფუნქცია-წევრი, რომელიც ავტომატურად გამოიძახება მაშინ, როდესაც ობიექტი გამოდის მოქმედების არედან: დესტრუქტორი მონიშნავს ასეთი ობიექტის მიერ დაკავებულ მეხსიერებას, როგორც თავისუფალს. დესტრუქტორის დასახელება შედგება: ~ (ტილდა) სიმბოლოსაგან, რომელსაც მოჰყვება კლასის დასახელება. დესტრუქტორი არ აბრუნებს მნიშვნელობას და მას არა აქვს პარამეტრები. კლასში უნდა იყოს აღწერილი მხოლოდ ერთი დესტრუქტორი. დესტრუქტორების გადატვირთვა დაშვებული არ არის.

<u>დესტრუქტორის ზოგადი სახეა:</u>

~class_name (void)

{დესტრუქტორის ოპერატორები}

კონსტრუქტორისგან განსხვავებით თქვენ არ შეგიძლიათ გადასცეთ პარამეტრები დესტრუქტორს. განვსაზღვროთ დესტრუქტორი **employee** კლასისათვის:

```
void employee:: ~ employee(void )

{
  cout << "ობიექტის  განადგურება " << name <<”სათვის”<<
endl;
}
```

მოცემულ შემთხვევაში დესტრუქტორს უბრალოდ გამოაქვს ეკრანზე შეტყობინება იმის შესახებ, რომ c++ ანადგურებს ობიექტს. პროგრამის დასრულებისას დესტრუქტორი ავტომატურად გამოიძახება ცალკეული ობიექტისათვის. განვიხილოთ პროგრამა:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

class employee

{
public:
  employee(char *, long, float);
  ~employee(void);
  void show_employee(void);
  int change_salary(float);
  long get_id(void);
private:
  char name [64] ;
  long employee_id;
  float salary;
};

employee::employee(char *name, long employee_id, float salary)
```

```
{
  strcpy(employee::name, name) ;
  employee::employee_id = employee_id;
  if (salary < 50000.0) employee::salary = salary;
  else // დაუშვებელი ხელფასი
  employee::salary= 0.0;
}

employee:: ~employee(void)

{
  cout << "ობიექტის განადგურება " << name <<" saTvis"<<
endl;

system("pause");
}

void employee::show_employee(void)

{
  cout << "TamamSromeli: " << name << endl;
  cout << "TanamSromlis nomeri: " << employee_id << endl;
  cout << "xelfasi: " << salary << endl;
}

void main(void)

{
  employee worker("Happy Jamsa", 101, 10101.0);
  worker.show_employee();
system("pause");
}
```

პროგრამის შესრულების შედეგად მივიღებთ:

TanamSromeli: Happy Jamsa
TanamSromlis nomeri: 101
xelfasi: 10101
obieqtis ganadgureba **Happy Jamsa** saTvis

როგორც ხედავთ პროგრამა ავტომატურად იძახებს დესტრუქტორს, დესტრუქტორის ფუნქციის ყოველგვარი ცხადი გამოძახების გარეშე.

ამდენად, როდესაც პროგრამა დაიწყებს მეხსიერების განაწილებას ობიექტების შიგნით თქვენ აღმოაჩენთ, რომ დესტრუქტორი მოხერხებული საშუალებაა ობიექტის განადგურებისას მეხსიერების განთავისუფლებისათვის.

დასკვნა:

- კონსტრუქტორი და დესტრუქტორი კლასის სპეციალური ფუნქციებია, რომელთაც პროგრამა ავტომატურად იძახებს ობიექტის შექმნის ან განადგურების დროს;
- კონსტრუქტორს პროგრამა გამოიძახებს თითოეულ ჯერზე ობიექტის შექმნისას. კონსტრუქტორს იგივე სახელი აქვს, რაც კლასს;
- კონსტრუქტორს არა აქვს დასაბრუნებელი მნიშვნელობა და საერთოდ იგი არ უნდა მიეთითოს;
- როდესაც თქვენი პროგრამა ქმნის ობიექტს, მას შეუძლია გადასცეს პარამეტრები კონსტრუქტორს ობიექტის გამოცხადების დროს;
- C++ ენა კონსტრუქტორის გადატვირთვის საშუალებას იძლევა და შესაძლებელია პარამეტრებისათვის გამოყენებულ იქნას მნიშვნელობები დუმილით;

- უმეტესობა პროგრამები კონსტრუქტორს იყენებს კლასის მონაცემ-ელემენტების ინიციალიზებისათვის;
- მარტივი პროგრამები არ მოითხოვენ დესტრუქტორის გამოყენებას;
- დესტრუქტორი არის სპეციალური ფუნქცია, რომელსაც პროგრამა იძახებს ავტომატურად თითოეულ ჯერზე ობიექტის განადგურებისას;
- დესტრუქტორს აქვს იგივე სახელი როგორც კლასს, მხოლოდ მის წინ იწერება ტილდა ნიშანი;
- დესტრუქტორი არ აბრუნებს მნიშვნელობას და მას არა აქვს პარამეტრები.

დავალება: შექმენით კლასი სტუდენტი. ღია ნაწილში იყოს ორი კონსტრუქტორი, პირველის პარამეტრებია: სტუდენტის გვარი, პირადი ნომერი და საშუალო ქულა, ხოლო მეორესი: გვარი და პირადი ნომერი. თვით ეს მონაცემები გამოაცხადეთ დახურულ ნაწილში. შექმენით ამ კლასის ორი ობიექტი. პირველის სამი პარამეტრია, ხოლო მეორესი ორი. გამოიტანეთ ინფორმაცია ეკრანზე. პროგრამაში გამოიყენეთ დესტრუქტორი.

ლაბორატორიული სამუშაო 6
თემა: ოპერატორების გადატვირთვა

ოპერატორის გადატვირთვა მდგომარეობს ოპერატორის აზრის შეცვლაში განსაზღვრულ კლასთან მისი გამოყენებისას. მოცემულ სამუშაოში თქვენ განსაზღვრავთ string კლასს და გადატვირთავთ ოპერატორ პლუს და მინუსს.

<u>დაიმახსოვრეთ შემდეგი ძირითადი საკითხები:</u>

- ოპერატორების გადატვირთვა საჭიროა თქვენი პროგრამის წაკითხვადობის გასაუმჯობესებლად, ოპერატორების გადატვირთვა საჭიროა მაშინ როცა ეს ამარტივებს პროგრამის გააზრებას;
- ოპერატორების გადასატვირთად პროგრამები იყენებენ გასაღებურ სიტყვას *operator*;
- ოპერატორის ხელახალი განსაზღვრისას თქვენ მიუთითებთ ფუნქციას, რომელსაც c++ იძახებს თითოეულ ჯერზე, როდესაც კლასი იყენებს გადატვირთულ ოპერატორს. ეს ფუნქცია თავის მხრივ ასრულებს შესაბამის ოპერაციას;
- თუ პროგრამა გადატვირთავს ოპერატორს განსაზღვრული კლასისათვის, მაშინ ამ ოპერატორის აზრი იცვლება მხოლოდ მითითებული კლასისათვის, პროგრამის დანარჩენი ნაწილი განაგრძობს ამ ოპერატორის გამოყენებას მისი სტანდარტული ოპერაციების შესასრულებლად;

– c++ უმეტესი ოპერატორების გადატვირთვის საშუალებას იძლევა, გარდა ოთხისა, რომელიც მოცემულია ცხრილში(გვ.50).

პლუს და მინუს ოპერატორების გადატვირთვა

განვიხილოთ მაგალითი, სადაც განსაზღვრულია string კლასი. ეს კლასი შეიცავს ერთ მონაცემ-ელემენტს, ეს არის სიმბოლოების სტრიქონი. გარდა ამისა შეიცავს რამდენიმე მეთოდს:

```
class string
{
public:
  string(char *); // Конструктор
  void str_append(char *);
  void chr_minus(char);
  void show_string(void);
private:
  char data[256] ;
};
```

str_append ფუნქცია უზრუნველყოფს კლასის სტრიქონისათვის მითითებული სიმბოლოების დამატებას. *chr_minus* ამოაგდებს სტრიქონიდან მითითებული სიმბოლოს ყოველ შესვლას. განვიხილოთ პროგრამა მთლიანობაში, სადაც შექმნილია ორი ობიექტი, რომელიც წარმოადგენს სიმბოლოების სტრიქონს:

```
#include <iostream.h>
#include <string.h>
class string
{
public:
```

```
  string(char *); // კონსტრუქტორი
  void str_append(char *);
  void chr_minus(char);
  void show_string(void);
private:
  char data[256] ;
};
string::string(char *str)
{
  strcpy(data, str);
}
void string::str_append(char *str)
{
  strcat(data, str);
}
void string::chr_minus(char letter)
{
  char temp[256] ;
  int i, j;
  for (i = 0, j = 0; data[i]; i++) // ეს სიმბოლო აუცილებელია
წაიშალოს?
  if (data[i] != letter) // თუ არა მივანიჭოთ მას  temp

  temp[j++] = data[i];
  temp[j] = NULL; // temp  დასასრული
//გადავაკოპიროთ  temp-ის შემადგენლობა უკან data-ში
  strcpy(data, temp);
}

void string::show_string(void)

{
  cout << data << endl;
}
```

```
void main(void)

{
  string title( "vscavlobT programirebas   C++ enaze");
  string lesson("operatorebis gdatvirTva");
  title.show_string() ;
  title.str_append(" me vscavlob!");
  title.show_string();
  lesson.show_string();
  lesson.chr_minus('p') ;
  lesson.show_string();
  }
```

როგორც ვხედავთ პროგრამა გამოიყენებს str_append ფუნქციას სტრიქონულ ცვლად title-სათვის სიმბოლოების დასამატებლად. პროგრამა გამოიყენებს chr_minus ფუნქციას ცალკეული “P” სიმბოლოს წასაშლელად lesson სიმბოლური სტრიქონიდან. მოცემულ შემთხვევაში პროგრამა იძახებს ფუნქციებს ამ ოპერაციების შესასრულებლად. თუ გამოვიყენებთ ოპერატორების გადატვირთვას, პროგრამა შეასრულებს იდენტურ ოპერაციებს (+) და (-) ოპერატორების დახმარებით.

ოპერატორის გადატვირთვისას გამოიყენება operator გასაღებური სიტყვა, თავისი პროტოტიპით და ფუნქციის განსაზღვრით. განვიხილოთ კლასი, სადაც გამოყენებულია operator გასაღებური სიტყვა, რათა დაენიშნოს პლუს და მინუს ოპერატორები str_append და chr_minus ფუნქციებს string კლასის შიგნით:

```
class string
```

```
{
public:
  string(char *); // Конструктор
  void operator +(char *);
  void operator -(char); //კლასის ოპერატორების
განსაზღვრა
  void show_string(void);
private:
  char data[256];
};
```

როგორც ხედავთ კლასი გადატვირთავს პლუს და მინუს ოპერატორებს. როგორც ვთქვით, როდესაც კლასი გადატვირთავს ოპერატორს, მან უნდა უჩვენოს ფუნქცია, რომელიც რეალიზებას უკეთებს ოპერაციას, რომელიც შეესაბამება ამ ოპერატორს. პლუს ოპერატორის შემთხვევაში ასეთი ფუნქციის განსაზღვრას ექნება შემდეგი სახე:

```
void string::operator +(char *str)

{
  strcat(data, str);
}
```

როგორც ხედავთ, ამ ფუნქციის განსაზღვრა არ შეიცავს სახელს, ეს კლასის გადატვირთული ოპერატორია. ამ ფუნქციის კოდი დარჩა იგივე. განვიხილოთ წინა პროგრამა გადატვირთული ოპერატორების გამოყენებით:

```
#include <iostream.h>
#include <string.h>
class string
```

```
{
public:
  string(char *); // კონსტრუქტორი
  void operator +(char *);
  void operator -(char);
  void show_string(void);
private:
  char data[256] ;
};

string::string(char *str)

{
  strcpy(data, str);
}

void string::operator +(char *str)

{
  strcat(data, str);
}

void string::operator -(char letter)

{
  char temp[256] ;
  int i, j;
  for (i = 0, j = 0; data[i]; i++) if (data[i]!= letter) temp[j++] =
data[i];
  temp[j] = NULL;
  strcpy(data, temp);
}

void string::show_string(void)
```

```
{
  cout << data << endl;
}

void main(void)

{
  string title( "vscavlobT programirebas  C++ enaze");
  string lesson("operatorebis gadatvirTva");
  title.show_string();
  title + " me vscavlob!";
  title.show_string() ;
  lesson.show_string();
  lesson - 'a';
  lesson.show_string();
}
```

როგორც ხედავთ პროგრამა იყენებს გადატვირთულ ოპერატორებს:

title + " me vscavlob!"; // დაემატოს ტექსტი **"me vscavlob!"**

lesson - 'a'; // წაიშალოს სიმბოლო 'a'

(შეგახსენებთ, რომ ფუნქცია strcat, აღწერილი string.h-ში, ორპარამეტრიანია. strcat(s1,s2) შესრულების შედეგად s2 სტრიქონი მიუერთდება s1-ს და გადაბმული სტრიქონი განთავსდება s1-ში, ამასთან, s2 სტრიქონი არ შეიცვლება. ცხადია, რომ s1-ის განზომილება თავიდანვე უნდა შეირჩეს ისე, რომ შერწყმით მიღებული ახალი სტრიქონი დაეტიოს s1-ის მეხსიერებაში.)

ცხრილში მოყვანილია ოპერატორები, რომლებიც არ გადაიტვირთება:

ოპერატორი	დანიშნულება	მაგალითი
.	ელემენტის ამორჩევა	object.member
.*	ელემენტზე მიმთითებელი	object.*member
: :	ხედვის არეს ნებართვა	classname::member
? :	პირობის ოპერატორი	c=(a>b)?a:b;

ოპერატორების გადატვირთვა არის შესაძლებლობა, რათა ოპერატორს მიენიჭოს ახალი არსი განსაზღვრულ კლასში მისი გამოყენებისას.

დასკვნა:

- ოპერატორის გადატვირთვისათვის უნდა განსაზღვროთ კლასი, რომელსაც დაენიშნება ეს ოპერატორი;
- ოპერატორის გადატვირთვა მოქმედებს მხოლოდ იმ კლასისათვის, რომელშიც იგი განისაზღვრება;
- კლასის ოპერატორის გადატვირთვისათვის გამოიყენება გასაღებური სიტყვა operator კლასის მეთოდის განსაზღვრისათვის, რომელსაც C++ იძახებს თითოეულ ჯერზე, როდესაც კლასის ცვლადი გამოიყენებს ამ ოპერატორს.

ლაბორატორიული სამუშაო 7
თემა: სტატიკური ფუნქციები და მონაცემ-ელემენტები

აქამდე განხილულ მასალაში ჩვენ მიერ შექმნილ ცალკეულ ობიექტს ჰქონდა თავისი მონაცემთა ნაკრები. შეიძლება იყოს სიტუაციები, როდესაც ერთიდაიმავე კლასის ობიექტებმა ერთობლივად უნდა გამოიყენონ ერთი ან რამდენიმე მონაცემ-ელემენტი. მაგალითად ვწერთ ხელფასის პროგრამას, რომელიც იკვლევს სამუშაო დროს 1000 თანამშრომლისათვის. სადაბეგვრო განაკვეთის განსაზღვრისათვის პროგრამამ უნდა იცოდეს პირობა, რომელშიც მუშაობს თითოეული თანამშრომელი. დავუშვათ ამისათვის გამოიყენება ცვლადი კლასი state_of_work. თუ ყველა თანამშრომელი მუშაობს ერთნაირ პირობებში, პროგრამას შეუძლია ერთობლივად გამოიყენოს ეს მონაცემები employee ტიპის ყველა ობიექტისათვის. ამდენად, თქვენი პროგრამა მოიშორებს რა ერთნაირი ინფორმაციის 999 ასლს, შეამცირებს მეხსიერების საჭირო რაოდენობას. კლასის ელემენტის ერთობლივი გამოყენებისათვის თქვენ ეს ელემენტი უნდა გამოაცხადოთ როგორც static(სტატიკური).

მასალაში თქვენ შეისწავლით შემდეგი ძირითად კონცეფციებს:

– C++ საშუალებას იძლევა გვქონდეს ერთნაირი ტიპის ობიექტები, რომლებიც ერთობლივად იყენებენ კლასის ერთ ან რამდენიმე ელემენტს;

- თუ თქვენი პროგრამა მიანიჭებს მნიშვნელობას ერთობლივად გამოსაყენებელ ელემენტს, მაშინ ამ კლასის ყველა ობიექტი მაშინვე მიიღებს წვდომას ამ ახალ მნიშვნელობაზე;
- კლასის ერთობლივად გამოსაყენებელი მონაცემ-ელემენტის შესაქმნელად კლასის სახელის წინ უნდა დაწეროთ გასაღებური სიტყვა static;
- მას შემდეგ რაც პროგრამა კლასის ელემენტს გამოაცხადებს როგორც static-ს, მან უნდა გამოაცხადოს გლობალური ცვლადი(კლასის განსაზღვრის გარეთ), რომელიც შეესაბამება კლასის ამ ერთობლივად გამოსაყენებელ ელემენტს;
- თქვენს პროგრამას შეუძლია გამოიყენოს გასაღებური სიტყვა static, რათა კლასის მეთოდი გახდეს გამოძახებადი მაშინაც კი, როცა პროგრამას შესაძლოა ჯერ კიდევ არა აქვს გამოცხადებული მოცემული კლასის რომელიმე ობიექტი.

მონაცემ ელემენტების ერთობლივი გამოყენება

გამოვაცხადოთ ელემენტები, როგორც საჯარო ან კერძო და შემდეგ ტიპის წინ დავუწეროთ გასაღებური სიტყვა static:

```
private:
static int shared_value;
```

კლასის გამოცხადების შემდეგ ელემენტი უნდა გამოცხადდეს როგორც ცვლადი კლასის გარეთ, როგორც ქვემოთ არის მოცემული:

int class_name::shared_value;

განვიხილოთ პროგრამა, რომელიც განსაზღვრავს book_series კლასს, ერთობლივად გამოყენებული ელემენტია – page_count, რომელიც ერთნაირია ყველა ობიექტისათვის (წიგნისათვის). თუ პროგრამა ცვლის ამ ელემენტის მნიშვნელობას, ცვლილება მაშინვე აისახება კლასის ყველა ობიექტში:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class book_series
{
public:
  book_series(char *, char *, float);
  void show_book(void);
  void set_pages(int) ;
private:
  static int page_count;
  char title[64];
  char author[ 64 ];
  float price;
};
int book_series::page_count;
void book_series::set_pages(int pages)

{
  page_count = pages;
}
```

```
book_series::book_series(char *title, char *author,
float price)

{
  strcpy(book_series::title, title);
  strcpy(book_series::author, author);
  book_series::price = price;
}

void book_series:: show_book (void)

{
  cout << "saTauri: " << title << endl;
  cout << "avtori: " << author << endl;
  cout << "fasi: " << price << endl;
  cout << "gverdebi: " << page_count << endl;
}

void main(void)

{
  book_series programming( "vscavlobT
programirebas C++ enaze", "Jamsa", 22.95);
  book_series word( "vscavlobT muSaobas Word
sistemaSi", "Wyatt", 19.95);
  word.set_pages(256);
  programming.show_book ();
  word.show_book() ;
  cout << endl << "cvlileba page_count " << endl;
  programming.set_pages(512);
  programming.show_book();
  word.show_book(); system(“pause”);
}
```

როგორც ხედავთ, კლასი *page_count* ელემენტს აცხადებს, როგორც *static int* ტიპს. კლასის განსაზღვრისთანავე პროგრამა *page_count* ელემენტს აცხადებს როგორც გლობალურ ცვლადს. როდესაც პროგრამა ცვლის page_count ელემენტს, ცვლილება მაშინვე აისახება book_series კლასის ყველა ობიექტში.

ელემენტების გამოყენება *public static* ატრიბუტებით, თუ ობიექტები არ არსებობენ

შეიძლება იყოს სიტუაციები, როდესაც პროგრამას ჯერ კიდევ არ შეუქმნია ობიექტი, მაგრამ საჭიროა ელემენტის გამოყენება. ელემენტის გამოყენებისათვის პროგრამამ უნდა გამოაცხადოს იგი როგორც *public* და *static*. მაგალითად, შემდეგი პროგრამა იყენებს *page_count* ელემენტს book_series კლასიდან, მაშინაც კი თუ ამ კლასის ობიექტები არ არსებობს:

```
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class book_series
{
public:
  static int page_count;
private:
  char title [64];
  char author[64];
  float price;
};
int book_series::page_count;
```

```
void main(void)
{
  book_series::page_count = 256;
  cout << "page_count –is mimdinare mniSvneloba
tolia " << book_series::page_count <<
endl; system(“pause”);
}
```

ამ შემთხვევაში, რადგან კლასი განსაზღვრავს *page_count* ელემენტს როგორც ღიას(საჯაროს), პროგრამას შეუძლია მიმართოს ამ ელემენტს მაშინაც კი, თუ *book_series* კლასის ობიექტები არ არსებობენ.

სტატიკური ფუნქცია-ელემენტების გამოყენება

თუ ჩვენ შევქმნით სტატიკურ მეთოდს, პროგრამა გამოიძახებს ამ მეთოდს მაშინაც კი თუ ობიექტები არ არის შექმნილი. მაგალითად, თუ კლასი შეიცავს მეთოდს, რომელიც შეიძლება გამოყენებულ იქნას კლასის გარეთ მონაცემებისათვის, ეს მეთოდი შეიძლება წარმოადგინოთ სტატიკური სახით.

განვიხილოთ პროგრამა სადაც შექმნილია კლასი menu, რომელშიც საჯარო ნაწილში გამოცხადებულია ტექსტის გამოტანის text_screen მეთოდი, კლასს არა აქვს ობიექტი, მაგრამ პროგრამა ამ მეთოდს გამოიყენებს:

```
#include <iostream.h>
#include<stdlib.h>
class menu
{
public:
  static void text_screen(void);
 // აქ შეიძლება იყოს სხვა მეთოდებიც
private:
```

```
  int number_of_menu_options;
};

void menu::text_screen(void)

{
  cout << “ekranze teqstis gamotana";
}

void main(void)

{
  menu::text_screen(); system(“pause”);
}
```

როგორც ხედავთ, თუ კლასი შეიცავს მეთოდს, რომელსაც თქვენ მოინდომებთ გამოიყენოთ კლასის ობიექტის გარეშე, მისი პროტოტიპის წინ განათავსეთ გასაღებური სიტყვა – static და ეს მეთოდი გამოაცხადეთ, როგორც საჯარო:

public:
static void text_screen(void);

ასეთი ფუნქციის გამოძახებისათვის პროგრამის შიგნით გამოიყენეთ გლობალური ნებართვის ოპერატორი, როგორც ქვემოთ არის ნაჩვენები:

menu::text_screen();

დასკვნა:

- როდესაც თქვენ გამოაცხადებთ კლასის ელემენტს static სახით, მაშინ ასეთი ელემენტი შეიძლება გამოყენებულ იქნას მოცემული კლასის ყველა ობიექტის მიერ;

- მას შემდეგ როცა თქვენს პროგრამაში კლასის ელემენტს გამოაცხადებთ static სახით, კლასის განსაზღვრის გარეთ უნდა გამოცხადდეს გლობალური ცვლადი, რომელიც შეესაბამება კლასის ერთობლივად გამოყენებად ელემენტს;
- თუ თქვენ აცხადებთ ელემენტს, როგორც public და static, თქვენ პროგრამას შეუძლია გამოიყენოს ასეთი ელემენტი მაშინაც კი, თუ მოცემული კლასის ობიექტები არ არსებობენ. ამ ელემენტზე მიმართვისათვის უნდა იქნას გამოყენებული გლობალური ნებართვის ოპერატორი, მაგალითად: class_name::member_name;
- თუ გამოაცხადებთ საერთო სტატიკურ ფუნქცია-ელემენტს თქვენ პროგრამას შეუძლია გამოიძახოს ეს ფუნქცია, მაშინაც კი, თუ ამ კლასის ობიექტები არ არსებობენ. მოცემული ფუნქციის გამოსაძახებლად პროგრამამ უნდა გამოიყენოს გლობალური ნებართვის ოპერატორი, მაგალითად:
  menu::text_screen().

დავალება: განსაზღვრეთ კლასი თანამშრომელი, მონაცემებით: გვარი, თანამდებობა, მუშაობის სტაჟი. ერთობლივად გამოყენებული ელემენტი იყოს ხელფასი, შექმენით ამ კლასის ორი ობიექტი. პროგრამამ გამოიტანოს მონაცემები ორივე ობიექტისათვის. შემდეგ პროგრამამ შეცვალოს საერთო ელემენტის მნიშვნელობა და კვლავ გამოიტანოს მონაცემები. გააანალიზეთ პროგრამის მუშაობის შედეგები. პროგრამაში დაამატეთ სტატიკური ფუნქცია და გამოიძახეთ მთავარი ფუნქციიდან.

ლაბორატორიული სამუშაო 8
თემა: მემკვიდრეობითობა. მარტივი მემკვიდრეობითობის მაგალითები

ობიექტზე ორიენტირებული დაპროგრამების მიზანი მდგომარეობს თქვენ მიერ შექმნილი კლასის განმეორებით გამოყენებაში, რაც ეკონომიას გაუკეთებს თქვენს დროს და ძალისხმევას. თუ თქვენ უკვე შექმენით რაიმე კლასი, შესაძლებელია იყოს სიტუაციები, რომ ახალ კლასს სჭირდებოდეს უკვე არსებული კლასის ბევრი ან სულაც ყველა თავისებურება და საჭიროა დაემატოს მხოლოდ ერთო ან რამდენიმე მონაცემ-ელემენტი ან ფუნქცია. ასეთ შემთხვევაში C++ საშუალებას იძლევა შეიქმნას ახალი ობიექტი უკვე არსებული ობიექტის მახასიათებლების გამოყენებით. სხვა სიტყვებით რომ ვთქვათ ახალი კლასი იქნება უკვე არსებული კლასის მემკვიდრე.

მოცემულ მასალაში თქვენ შეისწავლით შემდეგ ძირითად კონცეფციებს:

– თუ პროგრამა გამოიყენებს მემკვიდრეობითობას, მაშინ ახალი კლასის წარმოქმნისათვის საჭიროა საბაზო კლასი, ანუ ახალი კლასი ხდება საბაზო კლასის ელემენტების მემკვიდრე;

– წარმოებული კლასის ელემენტების ინიციალი-ზაციისათვის თქვენმა პროგრამამ უნდა გამოიძახოს საბაზო და წარმოებული კლასების კონსტრუქტორები;

– ოპერატორ წერტილის გამოყენებით პროგრამა მიმართავს საბაზო და წარმოებული კლასების ელემენტებს;

– დამატებით C++ წარმოადგენს protected(დაცულ) ელემენტებს, რომლებიც მისაწვდომია საბაზო და წარმოებული კლასებისათვის;

– სახელების კონფლიქტის თავიდან ასაცილებლად თქვენ პროგრამას შეუძლია გამოიყენოს გლობალური ნებართვის ოპერატორი, რომლის წინაც მითითებული იქნება საბაზო ან წარმოებული კლასის სახელი.

მემკვიდრეობითობა არის ობიექტზე ორიენტირებული დაპროგრამების ფუნდამენტური კონცეფცია.

დავუშვათ გვაქვს employee საბაზო კლასი:

```
class employee

{
public:
  employee(char *, char *, float);
  void show_employee(void);
private:
  char name[64];
  char position[64];
  float salary;
};
```

დავუშვათ პროგრამას სჭირდება manager კლასი, რომელიც ამატებს employee კლასში შემდეგ მონაცემებს:

```
float annual_bonus;
char company_car[64];
int stock_options;
```

ასეთ შემთხვევაში პროგრამას შეუძლია აირჩიოს ორი ვარიანტი: პირველი – პროგრამა შექმნის ახალ manager კლასს, სადაც დუბლირებული იქნება employee კლასის მრავალი ელემენტი, მეორე – პროგრამა ქმნის manager კლასს employee საბაზო კლასიდან, რითაც მცირდება პროგრამის მოცულობაც და არ მოხდება კოდის დუბლირება თქვენი პროგრამის შიგნით. ამ კლასის განსაზღვრისათვის უნდა დაწეროთ გასაღებური სიტყვა class, შემდეგ სახელი manager, ორი წერტილი და სახელი employee, როგორც ქვემოთ არის ნაჩვენები:

```
class manager : public employee {
// აქ განისაზღვრება ელემენტები};
```

გასაღებური სიტყვა public, რომელიც წინ უძღვის employee კლასს, უჩვენებს, რომ employee კლასის საერთო ელემენტები აგრეთვე საერთოა manager კლასშიც. ქვემოთ წარმოდგენილია manager კლასის შექმნის ოპერატორები:

```
class manager : public employee

{
public:
  manager(char *, char *, char *, float, float, int);
  void show_manager(void);
private:
  float annual_bonus;
  char company_car[64];
  int stock_options;
};
```

საბაზო კლასის კერძო ელემენტები წარმოებული კლასისათვის მისაწვდომია მხოლოდ საბაზო კლასის ფუნქციებით.

განვიხილოთ პროგრამა:

```
#include <iostream.h>
#include <string.h>
class employee

{
public:
  employee(char *, char *, float);
  void show_employee(void);
private:
  char name [ 64 ];
  char position[64];
  float salary;
};

employee::employee(char *name, char *position,float salary)

{
  strcpy(employee::name, name);
  strcpy(employee::position, position);
  employee::salary = salary;
}

void employee::show_employee(void)

{
  cout << "saxeli: " << name << endl;
  cout << "Tnamdeboba: " << position << endl;
```

```
  cout << "xelfasi: " << salary << endl;
}

class manager : public employee // manager კლასის შექმნა
საბაზო employee კლასიდან

{
public:
  manager(char *, char *, char *, float, float, int);
  void show_manager(void);
private:
  float annual_bonus;
  char company_car[64];
  int stock_options;
};

manager::manager(char *name, char *position, char *company_car,
float salary, float bonus, int stock_options) : employee(name,
position, salary)

{
  strcpy(manager::company_car, company_car) ;
  manager::annual_bonus = bonus ;
  manager::stock_options = stock_options;
}

void manager::show_manager(void)

{
  show_employee();
  cout << "firmis manqana: " << company_car << endl;
  cout << "yovelwliuri premia:" << annual_bonus << endl;
  cout << "safondo opcia: " << stock_options << endl;
}
```

```
void main(void)

{
  employee worker("BeriZe Noe", "programisti", 35000);
  manager boss("SaxvaZe Miriani", "vice-prezidenti ", "Lexus",
50000.0, 5000, 1000);
  worker.show_employee() ;
  boss.show_manager();
}
```

ყურადღება მიაქციეთ manager კონსტრუქტორს. როდესაც ქმნით ახალ კლასს საბაზო კლასიდან, წარმოებული კლასის კონსტრუქტორმა უნდა გამოიძახოს საბაზო კლასის კონსტრუქტორი, რისთვისაც წარმოებული კლასის კონსტრუქტორის შემდეგ განათავსეთ ორი წერტილი, შემდეგ მიუთითეთ საბაზო კლასის კონსტრუქტორი მოთხოვნილი პარამეტრებით:

```
 manager::manager(char *name, char *position, char
*company_car, float salary, float bonus, int stock_options)
: employee(name, position, salary) //საბაზო კლასის
კონსტრუქტორი

{
strcpy(manager::company_car, company_car);
manager::annual_bonus = bonus;
manager::stock_options = stock_options;
}
```

მიაქციეთ ყურადღება, რომ show_manager ფუნქცია იძახებს show_employee ფუნქციას, რომელიც არის employee კლასის ელემენტი. რამდენადაც manager კლასი

არის employee კლასიდან წარმოებული, manager კლასს შეუძლია მიმართვა employee კლასის საჯარო ელემენტებზე ისევე როგორც ყველა ელემენტი განსაზღვრული რომ ყოფილიყო manager კლასის შიგნით.

მაგალითი. პროგრამა ქმნის library _card კლასს book კლასიდან:

```
#include <iostream.h>
#include <string.h>
class book

{
public:
  book(char *, char *, int);
  void show_book(void);
private:
  char title [64];
  char author[64];
  int pages;
};

book::book(char •title, char *author, int pages)

{
  strcpy(book::title, title);
  strcpy(book::author, author);
  book::pages = pages;
}

void book::show_book(void)
```

```
{
  cout << "dasaxeleba: " << title << endl;
  cout << "avtori: " << author << endl;
  cout << "gverdebi: " << pages << endl;
}

class library_card : public book

{
public:
  library_card(char *, char *, int, char *, int);
  void show_card(void) ;
private:
  char catalog[64];
  int checked_out;
};

library_card::library_card(char *title, char *author, int pages, char
*catalog, int checked_out) : book(title, author, pages)

{
  strcpy(library_card::catalog, catalog) ;
  library_card::checked_out = checked_out;
}

void library_card::show_card(void)

{
  show_book() ;
  cout << "katalogi: " << catalog << endl;
  if (checked_out) cout << "statusi: Semowmebuli" << endl;
  else cout << "statusi: Tavisufali" << endl;
}

void main(void)
```

```
{
  library_card card( "vscavlobT programirebas C++ enaze",
"Jamsa", 272, "101CPP", 1);
  card.show_card();
}
```

როგორც წინა მაგალითში, მიაქციეთ ყურადღება, რომ library _card კონსტრუქტორი იძახებს book კლასის კონსტრუქტორს, book კლასის ელემენტების ინიციალიზაციისთვის. ასევე, მიაქციეთ ყურადღება book კლასის show_book ფუნქცია-ელემენტის გამოყენებას show_card ფუნქციაში. რამდენადაც library_card კლასი მემკვიდრეობს book კლასის მეთოდებს, show_card ფუნქციას შეუძლია გამოიძახოს show_book მეთოდი ოპერატორ წერტილის დახმარების გარეშე, თითქოს ეს მეთოდი ყოფილიყოს library _card კლასის მეთოდი.

რა არის დაცული ელემენტები

როგორც ვიცით, წარმოებულ კლასის შეუძლია მიმართოს საბაზო კლასის საერთო ელემენტებს, თითქოს ისინი განსაზღვრული იყოს წარმოებულ კლასში. მეორეს მხრივ წარმოებულ კლასს პირდაპირ არ შეუძლია მიმართოს საბაზო კლასის კერძო ელემენტებს. ამისათვის მან უნდა გამოიყენოს საბაზო კლასის ფუნქციები. საბაზო კლასის დაცული ელემენტებს უჭირავთ შუალედური მდგომარეობა საჯაროსა და კერძოს შორის. თუ ელემენტი არის დაცული, წარმოებული კლასის ობიექტები მას მიმართავენ როგორც საჯაროს. თქვენი პროგრამის

დანარჩენი ნაწილისათვის დაცული ელემენტები იქნება ისევე როგორც კერძო.

ქვემოთ განსაზღვრულია book კლასი, რომელიც იყენებს protected ჭდეს, რათა book კლასიდან წარმოებულ კლასს ნება დაერთოს მიმართოს title, author და pages ელემენტებს პირდაპირ, წერტილის გამოყენებით:

```
class book

{
public:
  book(char *, char *, int) ;
  void show_book(void) ;
protected:
  char title [64];
  char author[64];
  int pages;
};
```

როგორც შეამჩნიეთ მემკვიდრეობითობა ამარტივებს პროგრამირებას იმ შემთხვევაში, თუ წარმოებულ კლასებს შეუძლიათ მიმართონ საბაზო კლასის ელემენტებს ოპერატორ წერტილით. ასეთ შემთხვევაში პროგრამას შეუძლია გამოიყენოს კლასის დაცული ელემენტები. წარმოებულ კლასს შეუძლია მიმართოს საბაზო კლასის დაცულ ელემენტებს პირდაპირ, ოპერატორ წერტილის გამოყენებით. მხოლოდ პროგრამის დანარჩენ ნაწილს შეუძლია მიმართოს დაცულ ელემენტებს მხოლოდ ამ კლასის ფუნქციებით.

სახელთა კონფლიქტის პრობლემის გადაწყვეტა

ერთი კლასიდან მეორე კლასის შექმნის დროს შესაძლებელია გვქონდეს სიტუაციები, როდესაც წარმოებულ კლასში ელემენტის სახელი იგივეა, რაც საბაზო კლასში. მაგალითად კლასი book და library_card იყენებს price ელემენტს. book კლასში price ელემენტს შეესაბამება წიგნის გასაყიდი ფასი 22.95. library_card კლასში price შეიძლება შეიცავდეს ბიბლიოთეკის ფასდაკლებას, მაგალითად 18.50. დავუშვათ show_card ფუნქციამ უნდა გამოიტანოს ორივე ფასი. მაშინ მან უნდა გამოიყენოს შემდეგი ოპერატორები:

```
cout << "biblioTekis fasi" << price << endl;
cout << "gasayidi fasi:" << book::price << endl;
```

დასკვნა:

- მემკვიდრეობითობა არის არსებული საბაზო კლასიდან ახალი კლასის წარმოების უნარი;
- წარმოებული კლასი არის ახალი კლასი, ხოლო საბაზო კლასი - არსებული კლასი;
- როდესაც თქვენ ქმნით სხვა კლასს საბაზო კლასიდან, წარმოებული კლასი ხდება საბაზო კლასის ელემენტების მემკვიდრე;
- საბაზოდან ახალი კლასის წარმოქმნისთვის წარმოებული კლასის განსაზღვრა დაიწყეთ class გასაღებური სიტყვით, რომელსაც მოსდევს კლასის სახელი, ორი წერტილი და საბაზო კლასის სახელი, მაგალითად class dalmatian:dog;
- წარმოებულ კლასს შეუძლია მიმართოს საბაზო კლასის საერთო ელემენტებს ისე როგორც თავისივე კლასის ელემენტებს. საბაზო კლასის კერძო

მონაცემებზე წვდომისათვის წარმოებული კლასი იყენებს საბაზო კლასის ფუნქციებს;

- წარმოებული კლასის კონსტრუქტორში პროგრამამ უნდა გამოიძახოს საბაზო კლასის კონსტრუქტორი წარმოებული კლასის კონსტრუქტორის სათაურის შემდეგ ორი წერტილის, საბაზო კლასის კონსტრუქტორის სახელის და შესაბამისი პარამეტრების მითითებით;
- რათა წარმოებული კლასისათვის უზრუნველყოფილ იქნას საბაზო კლასის განსაზღვრულ ელემენტებზე პირდაპირი წვდომა და ამავე დროს ეს ელემენტები დაცული იყოს პროგრამის დანარჩენი ნაწილისთვის, C++ უზრუნველყოფს კლასის protected (დაცულ) ელემენტებს. წარმოებულ კლასს შეუძლია მიმართოს საბაზო კლასის დაცულ ელემენტებს როგორც საჯაროს. პროგრამის დანარჩენი ნაწილისთვის დაცული ელემენტები კერძოს ექვივალენტურია;
- თუ წარმოებულ და საბაზო კლასში არის ელემენტები ერთნაირი სახელებით, მაშინ წარმოებული კლასის ფუნქციის შიგნით C++ გამოიყენებს წარმოებული კლასის ელემენტებს. თუ წარმოებული კლასის ფუნქციებს სჭირდებათ საბაზო კლასის ელემენტებზე მიმართვა, უნდა გამოიყენოთ გლობალური ნებართვის ოპერატორი, მაგალითად base_class::member

დავალება: შექმენით კლასი ბაკალავრი მონაცემებით: გვარი, ფაკულტეტი, დაგროვებული კრედიტების რაოდენობა. შექმენით მისი მემკვიდრე კლასი: მაგისტრანტი, რომელსაც დამატებით ექნება ორი მონაცემი: ბაკალავრიატის კვალიფიკაცია და სწავლების წელი. შექმენით ამ კლასების თითო ობიექტი და გამოიტანეთ ინფორმაცია სტუდენტების შესახებ.

ლაბორატორიული სამუშაო 9
თემა: მრავლობითი მემკვიდრეობითობა

C++ ენაში შესაძლებელია შეიქმნას კლასი რამდენიმე საბაზო კლასისგან. როსდესაც კლასი მემკვიდრეობას იღებს რამდენიმე კლასისგან უნდა გამოვიყენოთ მრავლობითი მემკვიდრეობითობა.

მოცემულ მასალაში შეისწავლით შემდეგ ძირითად კონცეფციებს:

- თუ თქვენ ქმნით კლასს რამდენიმე საბაზო კლასისგან, მაშინ მიიღებთ მრავლობითი მემკვიდრეობის უპირატესობას;
- მრავლობითი მემკვიდრეობის დროს წარმოებული კლასი მიიღებს ორი ან მეტი კლასის ატრიბუტებს;
- მრავლობითი მემკვიდრეობითობის გამოყენებისას კლასის შექმნისათვის წარმოებული კლასის კონსტრუქტორმა უნდა გამოიძახოს ყველა საბაზო კლასის კონსტრუქტორი;
- წარმოებული კლასიდან ახალი კლასის შექმნისას თქვენ ქმნით მემკვიდრეობითობის იერარქიას(კლასთა იერარქიას).

მრავლობითი მემკვიდრეობა ობიქტზე ორიენტირებული დაპროგრამების მძლავრი ინსტრუმენტია.

მარტივი მაგალითი:

დავუშვათ გვაქვს კლასი computer_screen:

```
class computer_screen

{
public:
```

```
  computer_screen(char *, long, int, int);
  void show_screen(void);
private:
  char type[32] ;
  long colors;
  int x_resolution;
  int y_resolution;
};
```

დავუშვათ გვაქვს, აგრეთვე, კლასი mother_board:

```
class mother_board

{
public:
  mother_board(int, int, int);
  void show_mother_board(void);
private:
  int processor;
  int speed;
  int RAM;
};
```

ამ ორი კლასის გამოყენებით შეიძლება ავაგოთ კლასი computer, რაც ნაჩვენებია ქვემოთ:

```
class computer : public computer_screen, public mother_board

{
public:
  computer(char *, int, float, char *, long, int, int, int, int, int);
  void show_computer(void);
private:
  char name[64];
```

```
  int hard_disk;
  float floppy;
};
```

როგორც ხედავთ ეს კლასი უჩვენებს თავის საბაზო კლასებს ორი წერტილის შემდეგ, რომელიც მოსდევს computer კლასის სახელს:

```
class computer : public computer_screen, public
mother_board//საბაზო კლასები
```

ქვემოთ წარმოდგენილია პროგრამა სრული სახით:

```
#include <iostream.h>
#include <string.h>
#include<stdlib>
class computer_screen

{
public:
  computer_screen(char *, long, int, int);
  void show_screen(void);
private:
  char type[32];
  long colors;
  int x_resolution;
  int y_resolution;
};

computer_screen::computer_screen(char *type, long colors, int
x_res, int y_res)

{
  strcpy(computer_screen::type, type);
```

```
  computer_screen::colors = colors;
  computer_screen::x_resolution = x_res;
  computer_screen::y_resolution = y_res;
}

void computer_screen::show_screen(void)

{
  cout << "ekranis tipi: " << type << endl;
  cout << "ferebi: " << colors << endl;
  cout << "dashveba: " << x_resolution << " X " << y_resolution
<< endl;
}

class mother_board

{
public:
  mother_board(int, int, int);
  void show_mother_board(void);
private:
  int processor;
  int speed;
  int ram;
};

mother_board::mother_board(int processor, int speed, int ram)

{
  mother_board::processor = processor;
  mother_board::speed = speed;
  mother_board::ram = ram;
}

void mother_board::show_mother_board(void)
```

```
{
  cout << "procesori: " << processor << endl;
  cout << "sixSire: " << speed << "MH" << endl;
  cout << "operatiuli: " << ram << " MB" << endl;
}

class computer : public computer_screen, public mother_board

{
public:
  computer(char *, int, float, char *, long, int, int, int, int, int);
  void show_computer (void);
private:
  char name [64];
  int hard_disk;
  float floppy;
};

computer::computer(char *name, int hard_disk, float floppy, char
*screen, long colors, int x_res, int y_res, int processor, int speed, int
ram) : computer_screen(screen, colors, x_res, y_res),
mother_board(processor, speed, ram)

{
  strcpy(computer::name, name);
  computer::hard_disk = hard_disk;
  computer::floppy = floppy;
}

void computer::show_computer(void)

{
  cout << "tipi: " << name << endl;
  cout << "myari diski: " << hard_disk << "GB" << endl;
```

```
  cout << "moqnili diski: " << floppy << "MB" << endl;
  show_mother_board();
  show_screen();
}

void main(void)

{
  computer my_pc("Compaq", 212, 1.44, "SVGA", 16000000, 640,
480, 486, 66, 8);
  my_pc.show_computer(); system(“pause”);
}
```

თუ თქვენ გააანალიზებთ computer კლასის კონსტრუქტორს აღმოაჩენთ, რომ იგი იძახებს mother_board და computer_screen კლასების კონსტრუქტორებს, როგორც ნაჩვენებია ქვემოთ:

```
computer::computer(char *name, int hard_disk, float floppy, char
*screen, long colors, int x_res, int y_res, int processor, int speed, int
RAM) : computer_screen(screen, colors, x_res, y_res),
mother_board(processor, speed, RAM)
```

კლასთა იერარქიის აგება

ზოგჯერ საჭიროა შეიქმნას კლასი, იმ კლასისგან, რომელიც თავის მხრივ წარმოებულია რამდენიმე საბაზო კლასისგან. მაგალითად ჩვენ გვჭირდება computer კლასი გამოვიყენოთ საბაზოდ და შევქმნათ ახალი კლასი workstation:

```
class work_station : public computer
```

```
{
public:
  work_station (char *operating_system, char *name, int hard_disk,
float floppy, char *screen, long colors, int x_res, int  y_res, int
processor, int speed, int RAM);
  void show_work_station(void);
private:
  char operating_system[64];
};
```

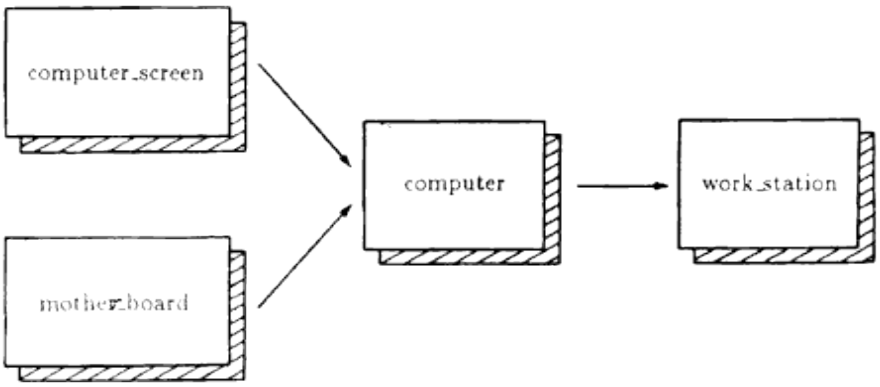
work_station კლასის კონსტრუქტორი უბრალოდ იძახებს computer კლასის კონსტრუქტორს, რომელიც თავის მხრივ იძახებს computer_screen და mother_board კონსტრუქტორებს:

```
work_station::work_station( char *operating_system, char *name,
int hard_disk, float floppy, char *screen, long colors, int  x_res, int
y_res, int processor, int speed, int RAM) : computer (name,
hard_disk, floppy, screen, colors, x_res, y_res,  processor, speed,
RAM)

{
  strcpy(work_station::operating_system, operating_system);
}
```

მოცემულ შემთხვევაში computer კლასი გამოდის საბაზო კლასის როლში. მხოლოდ, როგორც იცით იგი იყო წარმოებული computer_screen და mother_board კლასებიდან. შესაბამისად work_station კლასი მემკვიდრეობად

მიიღებს სამივე კლასის თვისებებს. ახალი კლასების „დაბადებას“ მივყავართ კლასების იერარქიამდე, როგორც ნაჩვენებია ნახაზზე:



დასკვნა:

- მრავლობითი მემკვიდრეობითობა არის უნარი წარმოებულ კლასს გადაეცეს რამდენიმე საბაზო კლასის მახასიათებლები;
- ახალი მემკვიდრე კლასის შექმნისას ახალი კლასის სახელის შემდეგ უნდა დაისვას ორი წერტილი და მიეთითოს საბაზო კლასების სახელები, რომელიც გამოყოფილია მძიმეებით, მაგალითად: class cabbit : public cat, public rabbit;
- წარმოებული კლასის კონსტრუქტორის განსაზღვრისას უნდა გამოიძახოთ ყველა საბაზო კლასის კონსტრუქტორები, საჭირო პარამეტრების გადაცემით;

– წარმოებული კლასიდან შეიძლება ახალი კლასის შექმნა. ამგვარად ჩვენი პროგრამა შექმნის კლასთა იერარქიას.

დავალება: შექმენით კლასი ბაკალავრი მონაცემებით: გვარი, ფაკულტეტი, დაგროვებული კრედიტების რაოდენობა. შექმენით მისი მემკვიდრე კლასი: მაგისტრანტი, რომელსაც დამატებით ექნება ორი მონაცემი: ბაკალავრიატის კვალიფიკაცია და სწავლების წელი. შექმენით მაგისტრანტის მემკვიდრე კლასი დოქტორანტი, რომელსაც დამატებით ექნება ორი მონაცემი: მაგისტრანტის კვალიფიკაცია და გამოქვეყნებული სტატიების რაოდენობა. შექმენით ამ კლასების თითო ობიექტი და გამოიტანეთ ინფორმაცია სტუდენტების შესახებ.

ლაბორატორიული სამუშაო 10
თემა: საჯარო მემკვიდრეობითობის მაგალითები. კერძო და დაცული მემკვიდრეობითობა

როგორც წინა მასალაში განვიხილეთ არსებული კლასის საფუძველზე შექმნილ კლასს ეწოდება წარმოებული ან მემკვიდრე კლასი, ხოლო იმ კლასს, რომლის საფუძველზე შეიქმნა ახალი, უწოდებენ – საბაზოს ან წინაპარ კლასს. ყოველი მემკვიდრე კლასი თვითონ შეიძლება გახდეს საბაზო კლასი მომავალი მემკვიდრე კლასებისთვის. მარტივი მემკვიდრეობითობის დროს წარმოებული კლასი მიიღება მხოლოდ ერთი საბაზო კლასის საფუძველზე. რთული მემკვიდრეობითობის შემთხვევაში წარმოებული კლასი იქმნება რამდენიმე კლასის საფუძველზე. მემკვიდრეობითობის მექანიზმის გამოყენებით აიგება რთული იერარქიული სტრუქტურები.

ის ფაქტი, რომ Circle კლასი წარმოიშვა Point კლასის საფუძველზე აღინიშნება შემდეგი სახით:

class Circle : public Point{ ……..};

ამას ეწოდება ღია მემკვიდრეობითობა (Public inheritance).

protected(დაცული) წვდომის რეჟიმი შემოტანილია მემკვიდრეობითობასთან დაკავშირებით. საჯარო მემკვიდრეობითობის დროს საბაზო კლასის საჯარო და დაცულ ელემენტებს მემკვიდრე კლასის ობიექტები მიმართავენ როგორც საჯაროს, ხოლო საბაზო კლასის დაცული ელემენტები პროგრამის სხვა ნაწილისათვის არის როგორც კერძო ელემენტები. საბაზო კლასის

კერძო ელემენტებზე მემკვიდრე კლასს პირდაპირი წვდომა არა აქვს, მათზე მიმართვა შეუძლია მხოლოდ საბაზო კლასის ფუნქციებს.

ზემოაღნიშნული საკითხების განმტკიცებისათვის განვიხილოთ მაგალითი:

```
#include<iostream>
class base {                    //საბაზო კლასი
  public:
        int get_i();
        int get_j();
        void set_ij(int, int);
        void show();
  private:
        int i, j;};
class derived : public base   // წარმოებული კლასი
  {  public:
        void make_k();
        void show();
  private:
        int k;
  };
  //რეალიზების ნაწილი
  int base::get_i() {return i;}
  int  base:: get_j(){return j;}
  void base::set_ij(int a, int b)  {i=a; j=b;}
  void base::show() {cout<<"i="<<i<<" j="<<j;}
  void derived ::make_k(){k=get_i()*get_j();}
  void derived::show() {base::show();
cout<<"k="<<k<<endl;}
  main()
```

```
{
base b;
  b.set_ij(2,3);
 cout<<"Object  b of  base class: "; b.show();
derived d;
  d.set_ij(5,6);  d.make_k();
  cout<<"\n\nObject d of derived class:"; d.show();
}
```

პროგრამის შედეგი:

```
  Object  b of  base class:  i=2 j=3
   Object d of derived class: i=5 j=6 k=30
```

derived კლასი არის base კლასის მემკვიდრე და მიიღება ღია(საჯარო) მემკვიდრეობითობით:

```
  class derived : public base  {…..};
```

base კლასის ყველა ელემენტი მემკვიდრეობით გადაეცემა derived კლასს. ეს ნიშნავს, რომ derived კლასის ღია ინტერფეისი შეიცავს base კლასის ღია ფუნქცია-წევრებს: get_i, get_j, set_ij და show (ანუ ფუნქციების გამოყენება შეუძლიათ derived კლასის ობიექტებს და საკუთარ ფუნქცია-წევრებს: make_k და show. გარდა ამისა derived კლასის ობიექტი ხასიათდება მონაცემებით i, j და k.

base კლასის მონაცემები i და j აღწერილია როგორც კლასის private(კერძო) წევრები. ამიტომ წარმოებულ derived კლასში მათზე პირდაპირი წვდომა აკრძალულია, ისევე როგორც პროგრამის დანარჩენ ნაწილში. derived კლასში ამ წევრებზე წვდომისათვის გამოყენებულია base კლასის ფუნქციები: get_i და get_j:

```
  void derived::make_k(){k=get_i()*get_j();}
```

თუ იგივე მაგალითში base კლასის i და j მონაცემებზე წვდომის რეჟიმს შევცვლით დაცულით (protected), მაშინ წარმოებულ კლასს i და j მონაცემებზე ექნება პირდაპირი წვდომა:

```
void derived ::make_k() {k=i*j;}
```

ამავე დროს პროგრამის დანარჩენი ფუნქციებისთვის i და j ისევ დახურული მონაცემებია ანუ მათზე პირდაპირი წვდომა არა აქვთ.

პროგრამა წარმოვადგინოთ ამ ცვლილების გათვალისწინებით:

```
#include<iostream>
class base {
  public:
        int get_i();
        int get_j();
        void set_ij(int, int);
        void show();
  protected:
        int i, j;};
class derived : public base
   {  public:
        void make_k();
        void show();
  private:
        int k;
  };
  int base::get_i() {return i;}
  int  base:: get_j(){return j;}
  void base::set_ij(int a, int b)  {i=a; j=b;}
  void base::show() {cout<<"i="<<i<<" j="<<j;}
```

```
        void derived ::make_k(){k=i * j;} //პირდაპირი მიმართვა
i და j ელემენტებზე
        void derived::show() {base::show();
cout<<"k="<<k<<endl;}
        main()
        {
        derived d;
          d.set_ij(3,7);  d.make_k();
          cout<<"Object d of derived class:"; d.show();
        }
```

პროგრამის შედეგი:

```
        Object d of derived class: i= 3  j=7 k=21
```

მივაქციოთ ყურადღება იმ ფაქტსაც, რომ base კლასის set_ij ფუნქციის გამოყენება შეუძლია როგორც ამ კლასის ობიექტს, ასევე derived კლასის d ობიექტს. საინტერესოა სახელების კონფლიქტის გადაწყვეტის საკითხი: ორივე derived და base კლასში გამოცხადებულია ერთი და იგივე სახელის მქონე show ფუნქციები. რეალიზაციის ნაწილში მათი განმარტებებია:

```
        void base::show() {cout<<"i="<<i<<" j="<<j;}
```

და

```
        void derived::show() {base::show();
    cout<<"k="<<k<<endl;}
```

derived კლასის show ფუნქცია ბეჭდავს ამ კლასის ყველა მონაცემს. ამასთან i და j მონაცემების დასაბეჭდად გამოიძახება base კლასის show ფუნქცია - base::show(); კლასებში სახელთა კონფლიქტის გადაჭრის მიზნით აუცილებელია იმის მითითება, რომ გამოძახებული show ფუნქცია არის base კლასის ხედვის არეში - base::

დასკვნა:

public მემკვიდრეობითობის შემთხვევაში საბაზო კლასის public ელემენტებზე წვდომა აქვს პროგრამის ყველა ფუნქციას. საბაზო კლასის private წევრებზე წვდომა აქვთ მხოლოდ საბაზო კლასის ფუნქცია-წევრებს და მეგობარ-ფუნქციებს, რომელსაც შემდგომ მასალაში განვიხილავთ.

protected(დაცული) წვდომის დონე წარმოადგენს წვდომის შუალედურ დონეს public და private წვდომის დონეებს შორის. საბაზო კლასის protected ელემენტებზე წვდომა აქვთ მხოლოდ: საბაზო კლასის ფუნქციებს და მეგობარ ფუნქციებს; წარმოებული კლასის ფუნქციებს და მეგობარ-ფუნქციებს.

წარმოებული კლასის ფუნქციებს შეუძლიათ საბაზო კლასის public და protected ელემენტებზე მიმართვა მათი სახელების მითითებით. ამასთან უნდა გვახსოვდეს, რომ protected წვდომის მონაცემები არღვევენ საბაზო კლასის ინკაფსულაციას: საბაზო კლასის დაცული ელემენტების შეცვლამ შეიძლება გამოიწვიოს ყველა წარმოებული კლასის მოდიფიცირების აუცილებლობა.

საბაზო და წარმოებული კლასის ობიექტებს აქვთ საერთო ნაწილი – საბაზო კლასის ატრიბუტები (მონაცემები) და ფუნქციები. ამიტომ public მემკვიდრეობითობით მიღებული წარმოებული კლასის ობიექტი შეიძლება განვიხილოთ, როგორც საბაზო კლასის ობიექტიც, მაგრამ საბაზო კლასის ობიექტი წარმოებული კლასის ობიექტს არ წარმოადგენს.

წარმოებული კლასის შექმნისას მემკვიდრეობითობის ტიპი შეიძლება იყოს როგორც public, ასევე protected და private.

protected მემკვიდრეობითობის დროს საბაზო კლასის:

- public ელემენტები წარმოებულ კლასში ხდებიან protected – მათზე პირდაპირი წვდომა აქვთ მხოლოდ წარმოებული კლასის ფუნქცია-წევრებს და მეგობარ ფუნქციებს(შემდეგში განვიხილავთ);
- protected ელემენტები მემკვიდრე კლასშიც რჩებიან protected;
- private ელემენტებზე პირდაპირი წვდომა წარმოებულ კლასში არ არის. მათზე წვდომა ხორციელდება საბაზო კლასის public ან protected ფუნქციების საშუალებით.

private მემკვიდრეობითობის დროს საბაზო კლასის:

- public ელემენტები წარმოებულ კლასში ხდებიან private – მათზე პირდაპირი წვდომა აქვთ მხოლოდ წარმოებული კლასის ფუნქცია-წევრებს და მეგობარ ფუნქციებს;

- protected ელემენტები წარმოებულ კლასში ხდებიან private;
- private ელემენტები წარმოებულ კლასში „არ ჩანან“. მათზე წვდომა ხერხდება საბაზო კლასის public ან protected ფუნქციების გამოყენებით.

დავალება: შექმენით კლასი მართკუთხედი და მისი მემკვიდრე მართკუთხა პარალელეპიპედი. შექმენით ამ კლასების თითო ობიექტი. გამოითვალეთ მართკუთხედის ფართობი და მართკუთხა პარალელეპიპედის მოცულობა. შედეგები გამოიტანეთ ეკრანზე.

ლაბორატორიული სამუშაო 11
თემა: მეგობარი კლასები და მეგობარი ფუნქციები

როგორც უკვე იცით, პროგრამას შეუძლია მიმართოს კლასის კერძო ელემენტებს მხოლოდ ამავე კლასის ფუნქცია-ელემენტების მეშვეობით. ყველა სიტუაციაში სადაც ეს შესაძლებელია საჯაროს ნაცვლად კერძო ელემენტების გამოყენებით თქვენ ამცირებთ პროგრამის შესაძლებლობას დააზიანოს კლასის ელემენტების მნიშვნელობა. მაგრამ ზოგჯერ თქვენ გჭირდებათ მწარმოებლურობის გაზრდა იმის ხარჯზე, რომ კლასს ნება დართოთ პირდაპირ მიმართოს სხვა კლასის კერძო ელემენტებს. ასეთ შემთხვევაში მცირდება დაყოვნებები. მსგავს სიტუაციებში C++ საშუალებას იძლევა კლასი განისაზღვროს მეგობრის(friend) რანგში სხვა კლასისთვის და მეგობარ-კლასს ნებას რთავს მიმართოს კერძო ელემენტებს. მოცემულ მასალაში მოცემულია თუ როგორ გაიგებს თქვენი პროგრამა, რომ ორი კლასი არის მეგობარი.

მასალაში შეისწავლით შემდეგ ძირითად კონცეფციებს:

- გასაღებური friend სიტყვის გამოყენებით კლასი იტყობინება, რომელია მისი მეგობარი ანუ იმას, რომ სხვა კლასები შეძლებენ პირდაპირ მიმართონ მის კერძო ელემენტებს;
- კლასის კერძო ელემენტები იცავენ კლასის მონაცემებს, შესაბამისად, თქვენ უნდა შეზღუდოთ კლასი-მეგობრების წრე მხოლოდ იმ კლასებით,

რომელთაც ნამდვილად სჭირდებათ პირდაპირი წვდომა საძიებო კლასის კერძო ელემენტებზე;

– C++ საშუალებას იძლევა შეიზღუდოს მეგობრული წვდომა ფუნქციების განსაზღვრული ნაკრებით.

კერძო ელემენტები კლასის დაცვის და შეცდომების შემცირების იძლევა საშუალებას. ამდენად, თქვენ უნდა შეზღუდოთ მეგობარ-კლასების გამოყენება იმდენად, რამდენადაც იგი შესაძლებელია.

კლასის მეგობრების განსაზღვრა

იმის მისათითებლად, რომ ერთი კლასი არის მეორეს მეგობარი, ამ სხვა კლასის განსაზღვრის შიგნით უნდა განათავსოთ გასაღებური სიტყვა friend და შესაბამისი მეგობარი-კლასის სახელი. ქვემოთ მოყვანილ მაგალითში book კლასი არის librarian კლასის მეგობარი. ამიტომ librarian კლასის ობიექტებს შეუძლიათ პირდაპირ მიმართონ book კლასის კერძო ელემენტებს ოპერატორ წერტილის გამოყენებით:

```
 class book
{
public:
  book (char *, char *, char *);
  void show_book(void);
  friend librarian;
private:
  char title [64] ;
  char author[64];
  char catalog[64];
};
```

როგორც ხედავთ მეგობრის მისათითებლად საჭიროა მხოლოდ ერთი ოპერატორი კლასის განსაზღვრაში. ქვემოთ წარმოდგენილ პროგრამაში librarian კლასი არის book კლასის მეგობარი. პროგრამა გამოიყენებს librarian კლასის change_catalog ფუნქციას განსაზღვრული წიგნის კატალოგის ბარათის ნომრის შესაცვლელად:

```
#include <iostream.h>
#include <string.h>
class librarian;
class book
{
public:
  book (char *, char *, char *);
  void show_book(void);
  friend librarian;
private:
  char title[64] ;
  char author[64];
  char catalog[64];
};
book::book(char *title, char *author, char *catalog)
{
  strcpy(book::title, title);
  strcpy(book::author, author) ;
  strcpy(book::catalog, catalog);
}
void book::show_book(void)

{
  cout << "dasaxeleba: " << title << endl;
  cout << "avtori: " << author << endl;
  cout << "katalogi: " << catalog << endl;
}
```

```
class librarian

{
public:
  void change_catalog(book *, char *);
  char *get_catalog(book);
};

void librarian::change_catalog(book *this_book, char
*new_catalog)

{
  strcpy(this_book->catalog, new_catalog);
}

char *librarian::get_catalog(book this_book)

{
  static char catalog[64];
  strcpy(catalog, this_book.catalog);
  return(catalog) ;
}

void main(void)

{
  book programming( "vscavlobT daprogramebas C++ enaze",
"Jamsa", "P101");
  librarian library;
  programming.show_book();
  library.change_catalog(&programming, "leqciebi C++ 101");
  programming.show_book();
}
```

როგორც ხედავთ, პროგრამა გადასცემს book ობიექტს librarian კლასის change_catalog ფუნქციაში მისამართით. რამდენადაც ეს ფუნქცია ცვლის book კლასის ელემენტს, პროგრამამ უნდა გადასცეს პარამეტრი მისამართით, ხოლო შემდეგ გამოიყენოს მიმთითებელი ამ კლასის ელემენტზე მიმართვისათვის. ჩაატარეთ ექსპერიმენტი: შეეცადეთ წაშალოთ friend ოპერატორი book კლასის განსაზღვრიდან. რამდენადაც librarian კლასს უკვე აღარ აქვს წვდომა book კლასის კერძო ელემენტებზე, კომპილატორი მოგვცემს სინტაქსური შეცდომის შესახებ შეტყობინებას book კლასის კერძო მონაცემებზე ყოველი მიმართვისას.
ქვემოთ წარმოდგენილია კლასის მეგობარად გამოცხადების ზოდაგი სახე:

class abbott

{
public:
friend costello;
// საერთო ელემენტები
private:
// კერძო ელემენტების;

რით განსხვავდება მეგობრები დაცული(protected) ელემენტებისგან

წარმოებულ კლასს შეუძლია პირდაპირ მიმართოს საბაზო კლასის დაცულ ელემენტებს, ოპერატორ წერტილის გამოყენებით. გახსოვდეთ, რომ კლასის დაცულ ელემენტებს შეუძლია მიმართოს მხოლოდ იმ კლასებმა, რომლებიც არიან წარმოებული ამ საბაზო

კლასიდან ანუ რომლებიც არიან საბაზო კლასის მემკვიდრეები. კლასის დაცული ელემენტები პროგრამის სხვა დანარჩენი ნაწილისთვის არიან როგორც კერძოები. მეგობარი-კლასები ერთმანეთთან არ არიან დაკავშირებულნი მემკვიდრეობითობით. მემკვიდრული კავშირის არქონის შემთხვევაში სხვა კლასის კერძო ელემენტებზე წვდომის მიღების ერთადერთი ხერხი არის კომპილატორის ინფორმირება, რომ მოცემულ კლასი არის მეგობარი.

მეგობრების რაოდენობის შეზღუდვა

დავუშვათ, წინა პროგრამაში წარმოდგენილი librarian კლასი შეიცავს მრავალ განსხვავებულ ფუნქციას. დავუშვათ, მხოლოდ change_catalog და get_catalog ფუნქციებს სჭირდებათ წვდომა book კლასის კერძო ელემენტებზე. book კლასის განსაზღვრაში ჩვენ შეგვიძლია შევზღუდოთ წვდომა კერძო ელემენტებზე მხოლოდ ამ ორი ფუნქციით, როგორც ქვემოთ არის ნაჩვენები:

```
class book
{
public:
  book(char *, char *, char *);
  void show_book(void);
  friend char *librarian::get_catalog(book);
  friend void librarian: :change_catalog( book *, char *);
private:
  char title[64];
  char author[ 64 ];
  char catalog[64];
};
```

როგორც ხედავთ, friend ოპერატორები შეიცავენ იმ მეგობარი ფუნქციის პროტოტიპებს, რომელთაც შეუძლიათ პირდაპირ მიმართონ კერძო ელემენტებს.

მეგობარი-ფუნქციები

თუ პროგრამა იყენებს მეგობრებს კლასის კერძო მონაცემებთან წვდომისთვის, თქვენ შეგიძლიათ შეზღუდოთ მეგობარი კლასის ფუნქცია-ელემენტების რაოდენობა, რომელთაც შეუძლიათ პირდაპირ მიმართონ კერძო ელემენტებს. მეგობარი-ფუნქციის გამოცხადებისათვის მიუთითეთ friend გასაღებური სიტყვა, რომელსაც მოჰყვება სრული პროტოტიპი, როგორც ქვემოთ არის ნაჩვენები:

public:
friend class_name::function_name(parameter types);

მხოლოდ მეგობარ ფუნქცია-ელემენტებს შეუძლიათ პირდაპირ მიმართონ კლასის კერძო ელემენტებს, ოპერატორ წერტილის გამოყენებით.

ერთი კლასიდან მეორეზე მიმართვისას კლასების განსაზღვრის მიმდევრობა უნდა იყოს დაცული, წინააღმდეგ შემთხვევაში პროგრამა მოგვცემს შეტყობინებას სინტაქსური შეცდომის შესახებ. მოცემულ შემთხვევაში book კლასის განსაზღვრა გამოიყენებს ფუნქციების პროტოტიპებს, რომლებიც განსაზღვრულია librarian კლასში. შესაბამისად, librarian კლასის განსაზღვრა წინ უნდა უძღოდეს book კლასის განსაზღვრას. თუ გააანალიზებთ librarian კლასს, აღმოაჩენთ, რომ იგი მიმართავს book კლასს:

```
class librarian
{
public:
  void change_catalog(book *, char *);
  char *get_catalog(book);
};
```

რამდენადაც თქვენ არ შეგიძლიათ book კლასის განსაზღვრა განათავსოთ librarian კლასის განსაზღვრის წინ, C++ საშუალებას იძლევა გამოცხადდეს book კლასი ანუ შეატყობინოს კომპილატორს, რომ ასეთი კლასი არის, ხოლო მოგვიანებით განისაზღვროს იგი. კლასი გამოვაცხადოთ შემდეგი სახით:

class class_name;

მაგალითად:

```
class book; // კლასის გამოცხადება
```

შემდეგი პროგრამა გამოიყენებს მეგობარ ფუნქციებს librarian კლასის წვდომის შეზღუდვისათვის book კლასის კერძო მონაცემებზე. ყურადღება მიაქციეთ კლასების განსაზღვრის მიმდევრობას:

```
#include <iostream.h>
#include <string.h>
class book;
class librarian
{
public:
  void change_catalog(book *, char *);
  char *get_catalog(book);
};
```

```
class book

{
public:
  book(char *, char *, char *) ;
  void show_book (void);
  friend char *librarian::get_catalog(book);
  friend void librarian::change_catalog( book *, char *);
private:
  char title[64];
  char author[64];
  char catalog[64];
};

book::book(char *title, char *author, char *catalog)

{
  strcpy(book::title, title);
  strcpy(book::author, author);
  strcpy(book::catalog, catalog);
}

void book::show_book(void)

{
  cout << "dasaxeleba: " << title << endl;
  cout << "avtori: " << author << endl;
  cout << "katalogi: " << catalog << endl;
}

void librarian::change_catalog(book *this_book, char
*new_catalog)
```

```
{
  strcpy(this_book->catalog, new_catalog) ;
}

char *librarian::get_catalog(book this_book)

{
  static char catalog[64];
  strcpy(catalog, this_book.catalog);
  return(catalog) ;
}

void main(void)

{
  book programming( "vswavlobT programirebas C++ enaze",
"Jamsa", "P101");
  librarian library;
  programming.show_book();
  library.change_catalog(&programming, "leqcia C++ 101");
  programming.show_book();
}
```

როგორც ხედავთ, პროგრამა თავიდან იყენებს განცხადებას, რათა შეატყობინოს კომპილატორს, რომ book კლასი განსაზღვრული იქნება შემდგომ. რადგან book კლასზე განაცხადი გაკეთებულია, librarian კლასის განსაზღვრა შეიძლება გადაიგზავნოს book კლასზე, რომელიც პროგრამაში ჯერ კიდევ არ არის განსაზღვრული.

დასკვნა:

– C++ -ში მეგობრების გამოყენება სხვა კლასის კერძო ელემენტებზე პირდაპირი მიმართვის საშუალებას იძლევა, ოპერატორ წერტილის გამოყენებით;

– ერთი კლასის მეორე კლასის მეგობრად გამოცხადებისათვის, მეორე კლასის გამოცხადებაში უნდა უჩვენოთ friend გასაღებური სიტყვა, რომლის შემდეგაც იწერება პირველი კლასის სახელი;

– კლასის სხვა კლასთან მიმართებით მეგობრად გამოცხადების შემდეგ, მეგობარი-კლასის ყველა ფუნქცია-ელემენტს შეუძლია მიმართოს ამ სხვა კლასის კერძო ელემენტებს;

– მეგობარი მეთოდების შეზღუდვის მიზნით, C++ საშუალებას იძლევა მითითებულ იქნა მეგობარი ფუნქციები. მეგობარი ფუნქციის გამოცხადებისათვის უნდა მიეთითოს friend გასაღებური სიტყვა, რომლის შემდეგაც იწერება ფუნქციის პროტოტიპი;

– მეგობარი ფუნქციების გამოცხადებისას უნდა დაიცვათ კლასების განსაზღვრის მიმდევრობა. თუ საჭიროა კომპილატორზე შეტყობინება, რომ რომ იდენტიფიკატორი წარმოადგენს კლასის სახელს, რომელსაც პროგრამა განსაზღვრავს მოგვიანებით, თქვენ შეგიძლიათ გამოიყენოთ შემდეგი სახის ოპერატორი: class class_name.

ქვემოთ მოყვანილ მაგალითში frd() ფუნქცია გამოცხადებულია როგორც cl კლასის მეგობარი:

```
class cl {
...
public:
friend void frd();
...
};
```

ერთ-ერთი მიზეზი, რომლის გამოც C++ ენა უშვებს ფუნქცია-მეგობრების არსებობას დაკავშირებულია ისეთ სიტუაციასთან, როდესაც ორმა კლასმა უნდა გამოიყენოს ერთიდაიგივე ფუნქცია. განვიხილოთ მაგალითი, სადაც განსაზღვრულია ორი კლასი – line და box. line კლასი შეიცავს ყველა საჭირო მონაცემს და ნებისმიერი მოცემული სიგრძის ჰორიზონტალური პუნქტირი მონაკვეთის დასახაზ კოდს. box კლასი შეიცავს საჭირო კოდს და მონაცემებს იმისათვის, რომ გამოისახოს მართკუთხედი ზედა მარცხენა და ქვედა მარჯვენა წერტილებით, მოცემული ფერით. ორივე კლასი გამოიყენებს same_color() ფუნქციას იმისათვის, რომ განისაზღვროს დახატულია თუ არა მონაკვეთი და მართკუთხედი ერთიდაიგივე ფერით. ამ კლასების გამოცხადება მოცემულია პროგრამის შემდეგ ფრაგმენტში:

```
class line;
class box {
int color; // მართკუთხედის ფერი
int upx, upy; //ზედა მარცხენა კუთხე
int lowx, lowy; //ქვედა მარჯვენა კუთხე
public:
friend int same_color (line l, box b);
void set_color(int c);
```

```
void define_box (int x1, int y1, int x2, int y2);
void show_box();
};
class line {
int color;
int startx, starty;
int len;
public:
friend int same_color (line I, box b);
void set_color(int c);
void define_line (int x, int y, int I);
void show_line();
};
```

same_color() ფუნქცია არც ერთი კლასის წევრი არ არის, მაგრამ არის ორივეს მეგობარი. იგი აბრუნებს შეტყობინებას: “Are the same color” როდესაც line ტიპის ობიექტი და box ტიპის ობიექტი დახაზულია ერთიდაიმავე ფერით, ხოლო შეტყობინებას: – “Not the same color” წინააღმდეგ შემთხვევაში. ქვემოთ წარმოდგენილია same_color() ფუნქციის განსაზღვრის კოდი:

```
int same_color (line I, box b)
{
if (I.color == b.color) cout << " Are the same color”;
else cout<<”Not the same color";
}
```

რადგან same_color() ფუნქცია არის ორივე კლასის მეგობარი მას ექნება ორივე კლასის კერძო მონაცემებზე წვდომა. უფრო მეტიც, მივაქციოთ ყურადღება, რომ

რამდენადაც same_color() ფუნქცია არ არის წევრი, გამოყენებისას არ არის საჭირო კლასის სახელის გამოყენება. იმავე მიზნით შეიძლებოდა შეგვექმნა ფუნქცია-წევრი public წვდომის სპეციფიკატორით, რომელიც დააბრუნებდა line და box ტიპის ობიექტების ფერებს და კიდევ ერთი ფუნქცია ამ ფერების შესადარებლად. მაგრამ ასეთი მიდგომა მოითხოვს ფუნქციის დამატებით გამოძახებას, რაც მოცემულ შემთხვევაში არაეფექტურია.

ქვემოთ მოყვანილი პროგრამა დემონსტრირებას უკეთებს line და box კლასებს და უჩვენებს თუ როგორ მიიღებს მეგობარი ფუნქცია წვდომას კლასის კერძო წევრებთან:

```
#include <iostream.h>
#include <conio.h>
class line;
class box {
int color;
int upx, upy;
int lowx, lowy;
public:
friend int same_color (line l, box b);
void set_color(int c);
void define_box (int x1, int y1, int x2, int y2);
void show_box();
};
class line {
int color;
int startx, starty;
```

```
int len;
public:
friend int same_color (line I, box b);
void set_color(int c);
void define_line (int x, int y, int I);
void show_line();
};
int same_color (line I, box b)
{
if (I.color == b.color) cout << " Are the same color”;
        else cout<<”Not the same color";
        }
void box::set_color(int c)
{
color = c;
}
void line::set_color(int c)
{
color = c;
}
void box::define_box (int x1, int y1, int x2, int y2)
{
upx = x1;
upy = y1;
lowx = x2;
lowy = y2;
}
void box::show_box()
{
int i;
```

```
textcolor(color);
gotoxy(upx, upy);
for (i=upx; i<=lowx; i++) cout<<"-";
gotoxy(upx, lowy-1);
for (i=upx; i<=lowx; i++) cout<<"-";
gotoxy(upx, upy);
for (i=upy; i<=lowy; i++) {
cout<<" | ";
gotoxy(upx , i);
}
gotoxy(lowx, upy);
for (i=upy; i<=lowy; i++) {
cout<<"I";
gotoxy(lowx, i);
}
}
void line::define_line (int x, int y, int I)
{
startx = x;
starty = y;
len = I;
}
void line::show_line()
{
int i;
textcolor(color);
gotoxy(startx, starty);
for (i=0; i<len; i++) cout<<"-";
}
int main()
```

```
{
box  b;
line  I;
b.define_box (10, 10, 15, 15);
b.set_color(3);
b.show_box();
I.define_line (2, 2, 10);
I.set_color(2);
I.show_line();
same_color (I, b);
I.define_line (22, 22, 10);
I.set_color(3);
I.show_line();
same_color (I, b);
        cout << "\nPress a key.";
getch();
}
```

არსებობს მეგობარ ფუნქციებთან მიმართებით ორი მნიშვნელოვანი შეზღუდვა:

- წარმოებული კლასები არ მემკვიდრეობენ მეგობარ ფუნქციებს;
- მეგობარი ფუნქციები არ შეიძლება გამოცხადებულ იქნან static და extern გასაღებური სიტყვებით.

დავალება: შექმენით ორი კლასი: თსუ და სტუ, რომელთაც ექნებათ კერძო მონაცემი სტუდენტების რაოდენობა. მეგობარმა ფუნქციამ შეადაროს ერთმანეთს თსუ-ს და სტუ-ს სტუდენტების რაოდენობები და გამოიტანოს შესაბამისი შეტყობინება: ტოლია თუ განსხვავებულია.

ლაბორატორიული სამუშაო 12
თემა: პოლიმორფიზმი. ვირტუალური ფუნქციები

ზოგადად პოლიმორფიზმი არის ობიექტის ფორმის შეცვლის უნარი. თუ ამ ტერმინს დავყოფთ ნაწილებად, აღმოვაჩენთ, რომ, პოლი არის ბევრი, ხოლო მორფიზმი ფორმის ცვლილება. შესაბამისად, პოლიმორფული ობიექტი არის ისეთი ობიექტი, რომელსაც შეუძლია მიიღოს სხვადასხვა ფორმა. მოცემული სამუშაო გაგაცნობთ პოლიმორფიზმის არსს და შეგასწავლით როგორ გამოვიყენოთ პროგრამაში პოლიმორფული ობიექტები კოდის შემცირებისა და გამარტივებისათვის.

მასალაში შეისწავლით შემდეგ ძირითად კონცეფციებს:

- პოლიმორფიზმი წარმოადგენს ობიექტის უნარს შეიცვალოს ფორმა პროგრამის შესრულების დროს;
- C++ ამარტივებს პოლიმორფული ობიექტების შექმნას;
- პოლიმორფული ობიექტების შექმნისათვის თქვენმა პროგრამამ უნდა გამოიყენოს ვირტუალური(*virtual)* ფუნქციები;
- ვირტუალური(*virtual)* ფუნქცია – ეს არის საბაზო კლასის ფუნქცია, რომლის სახელის წინ განთავსებულია გასაღებური სიტყვა *virtual;*
- საბაზო კლასიდან ნებისმიერ წარმოებულს შეუძლია გამოიყენოს ან გადატვირთოს ვირტუალური ფუნქცია;
- პოლიმორფული ობიექტების შექმნისათვის უნდა გამოიყენოთ საბაზო კლასის ობიექტზე მიმთითებელი.

დავუშვათ, პროგრამისტს, რომელიც მუშაობს სატელეფონო კომპანიაში სჭირდება დაწეროს პროგრამა,

რომელიც სახეს უცვლის სატელეფონო ოპერაციებს. როგორც ვიცით, ტელეფონის გამოყენებისას სრულდება ოპერაციები: ნომრის აკრეფა, ზარი, გაწყვეტა, დაკავების ინდიკაცია. ამ ოპერაციების დახმარებით, თქვენ განსაზღვრავთ შემდეგ კლასს:

```
class phone

{
public:
  void dial(char "number) { cout << "nomris akrefa " << number << endl; }
  void answer(void) { cout << "pasuxis molidini" << endl; }
  void hangup(void) { cout << "zari Sesrulebulia – kurmilis dakideba" << endl; }
  void ring(void) { cout << "zari,zari,zari" << endl;)
  phone(char *number) { strcpy(phone::number, number); };
private:
  char number[13];
);
```

ქვემოთ წარმოდგენილია პროგრამა, რომელიც გამოიყენებს *phone* კლასს ობიექტ – ტელეფონის შესაქმნელად:

```
#include <iostream.h>
#include <string.h>
class phone
{
public:
  void dial(char *number) { cout << "nomris akrefa " << number << endl; }
  void answer(void) { cout << "pasuxis molodini" << endl; }
  void hangup(void) { cout << "zari Sesrulebulia – kurmilis
```

```
dakideba" << endl; }
  void ring(void) { cout << "zari, zari, zari" << endl; }
  phone(char *number) { strcpy(phone::number, number); };
private:
  char number[13];
};

void main(void)

{
  phone telephone("555-1212");
  telephone.dial("212-555-1212");
}
```

მოცემული პროგრამა არ აკეთებს განსხვავებას დისკურ და ღილაკებიან ტელეფონებს შორის. ამავე დროს მას არა აქვს ფასიანი ტელეფონის მხარდაჭერა. აქედან გამომდინარე ლოგიკურია მივიღებთ გადაწყვეტილებას შევქმნათ *touch_tone* და pay_phone კლასები, *phone* კლასიდან, როგორც ქვემოთ არის ნაჩვენები:

```
class touch_tone : phone

{
public:
  void dial(char * number) { cout << "Rilakebianze nomris
akrefa " << number << endl; }
  touch_tone(char *number) : phone(number) { }
};

class pay_phone : phone
```

```
{
public:
  void dial(char * number)

  {
    cout << "gTxovT gadaixadoT " << amount << " TeTri" <<
endl;
    cout << "nomris akrefa " << number << endl;
  }

  pay_phone(char *number, int amount) : phone(number)
{pay_phone::amount = amount; }
private:
  int amount;
};
```

როგორც ხედავთ *touch_tone* და pay_phone კლასები განსაზღვრავენ თავის საკუთარ *dial* მეთოდს. phone კლასის *dial* მეთოდი გათვალიწინებულია დისკური ტელეფონისთვის. შემდეგი პროგრამა გამოიყენებს ამ კლასებს rotary ობიექტის შექმნისათვის:

```
#include <iostream.h>
#include <string.h>
class phone
{
public:
  void dial(char *number) { cout << "nomris akrefa " << number
<< endl; }
  void answer(void) { cout << "pasuxis molodini" << endl; }
  void hangup(void) { cout << "zari Sesrulebulia – kurmilis
dakideba" << endl; }
  void ring(void) { cout << "zari, zari, zari" << endl; }
  phone(char *number) { strcpy(phone::number, number); };
```

```
protected:
  char number[13];
);

class touch_tone : phone

{
public:
  void dial(char *number) { cout << "Rilakebianze  nomris akrefa "
<< number << endl; }
  touch_tone(char *number) : phone(number) { }
};

class pay_phone : phone

{
public:
  void dial(char * number) { cout << "gTxovT gadaixadoT " <<
amount << " TeTri" << endl; cout << "nomris akrefa       "
<<number << endl; }
  pay_phone(char * number, int amount) : phone(number)
{pay_phone::amount = amount; }
private:
  int amount ;
};
void main (void)
{
  phone rotary("303-555-1212");
  rotary.dial("602-555-1212");
  touch_tone telephone("555-1212");
  telephone.dial("212-555-1212");
```

```
  pay_phone city_phone("555-1111", 25);
  city_phone.dial("212-555-1212");
}
```

გაუშვით პროგრამა კომპილაციაზე და შემდეგ შესრულებაზე, ეკრანზე გამოვა შემდეგი შედეგები:

nomris akrefa 602-555-1212
Rilakebianze nomris akrefa 212-555-1212
gTxovT gadaixadoT 25 TeTri
nomris akrefa 212-555-1212

განხილული პროგრამა არ გამოიყენებს პოლიმორფულ ობიექტებს. ანუ მასში არ არის ობიექტები, რომლებიც შეიცვლიან ფორმას. მიზანშეწონილია სხვადასხვა ზარის დროს ჩვენი ობიექტი-ტელეფონი იცვლიდეს ფორმას.

პოლიმორფული ობიექტი-ტელეფონის შექმნა

ტელეფონის სხვადასხვა კლასებს შორის არსებობს ერთადერთი განმასხვავებელი ფუნქცია – ეს არის *dial* მეთოდი. პოლიმორფული ობიექტის შესაქმნელად თავდაპირველად უნდა განისაზღვროს საბაზო კლასის ფუნქციები, რომლებიც განსხვავდებიან წარმოებული კლასების ფუნქციებიდან იმით, რომ ისინი ვირტუალურია. მათ პროტოტიპებს წინ უძღვის *virtual* გასაღებური სიტყვა, როგორც ნაჩვენებია ქვემოთ:

```
class phone
{
public:
  virtual void dial(char •number) { cout << "nomris akrefa " <<
number << endl; }
```

```
  void answer(void) { cout << "pasuxis molodini" << endl; }
  void hangup(void) { cout << "zari Sesrulebulia  - kurmilis
dakideba" << endl; }
  void ring(void) { cout << "zari, zari, zari" << endl; }
  phone(char *number) { strcpy(phone::number, number); };
protected:
  char number[13];
};
```

შემდეგ, პროგრამაში შევქნათ საბაზო კლასის ობიექტზე მიმთითებელი. ჩვენს შემთხვევაში მიმთითებელი *phone* საბაზო კლასზე:

```
phone *poly_phone;
```

ობიექტის ფორმის შესაცვლელად უბრალოდ მივანიჭოთ ამ მიმთითებელს წარმოებული კლასის ობიექტის მისამართი, როგორც ქვემოთ არის ნაჩვენები:

```
poly_phone = (phone *) &home_phone;
```

სიმბოლოები *(phone *),* რომელიც იწერება მინიჭების ოპერატორის შემდგომ, წარმოადგენს ტიპებში მოყვანის ოპერატორს, რომელიც ატყობინებს კომპილატორს, რომ ყველაფერი წესრიგშია, საჭიროა ერთი ტიპის (*touch_tone)* ცვლადის მისამართს მიენიჭოს სხვა ტიპის *(phone)* ცვლადზე მიმთითებელი. რამდენადაც ქვემოთ წარმოდგენილ პროგრამას შეუძლია poly_phone ობიექტზე მიმთითებელს მიანიჭოს სხვადასხვა ობიექტის მისამართი, ამდენად ობიექტს შეუძლია ფორმის შეცვლა, რის გამოც იგი პოლიმორფულია.

```
#include <iostream.h>
#include <string.h>
class  phone

{
public:
  virtual void dial(char *number) { cout << "nomris akrefa " <<
number << endl; }
  void answer(void) { cout << "pasuxis molodini" << endl; }
  void hangup(void) { cout << "zari Sesrulebulia – kurmilis
dakideba" << endl; }
  void ring(void) { cout << "zari, zari, zari" << endl; }
  phone(char *number) { strcpy(phone::number, number); };
protected:
  char number[13] ;
};

class touch_tone : phone

{
public:
  void dial(char * number) { cout << "RilakebiT nomris akrefa " <<
number << endl; }
  touch_tone(char *number) : phone(number) { }
};

class pay_phone: phone

{
public:
  void dial(char *number) { cout << "gTxovT gadaixadoT " <<
amount << " TeTri" << endl; cout << "nomris akrefa "       <<
```

```
number << endl; }
  pay_phone(char *number, int amount) : phone(number)
{pay_phone::amount = amount; }
private:
  int amount;
};

void main(void)

{
  pay_phone city_phone("702-555-1212", 25);
  touch_tone home_phone("555-1212");
  phone rotary("201-555-1212") ;
    //obieqtis Seqmna  diskur telefonad
  phone *poly_phone = &rotary;
  poly_phone->dial("818-555-1212");
    // obieqtis formis Secvla Rilakebian telefonad
  poly_phone = (phone *) &home_phone;
  poly_phone->dial("303-555-1212");
    // obieqtis formis Secvla fasian telefonad
  poly_phone = (phone *) &city_phone;
  poly_phone->dial("212-555-1212");
}
```

პროგრამას გავუკეთოთ კომპილაცია და გავუშვათ შესრულებაზე, ეკრანზე გამოვა შემდეგი შედეგი:

nomris akrefa 818-555-1212
RilakebiT nomris akrefa 303-555-1212

gTxovT gadaixadoT 25 TeTri
nomris akrefa 212-555-1212

რამდენადაც *poly_phone* ობიექტი იცვლის ფორმას პროგრამის შესრულებისას, იგი არის პოლიმორფული.

როგორც ვხედავთ, პოლიმორფული ობიექტის შექმნისას პროგრამამ უნდა გამოიყენოს საბაზო კლასის ობიექტზე მიმთითებელი. შემდგომ პროგრამამ უნდა მიანიჭოს ამ მიმთითებელს წარმოებული კლასის ობიექტის სახელი. ცალკეულ შემთხვევაში, როდესაც პროგრამა მიმთითებელს ანიჭებს სხვა კლასის ობიექტის მისამართს, ამ მიმთითებლის ობიექტი(რომელიც არის პოლიმორფული) იცვლის ფორმას. პროგრამები აგებენ პოლიმორფულ ობიექტებს, საბაზო კლასის ვირტუალურ ფუნქციებზე დაფუძნებით.

რა არის წმინდად ვირტუალური ფუნქციები

როგორც ვიცით, პოლიმორფული ობიექტის შექმნისათვის პროგრამები განსაზღვრავენ საბაზო კლასის ერთ ან რამდენიმე მეთოდს, როგორც ვირტუალურ ფუნქციას. წარმოებულ კლასს შეუძლია განსაზღვროს თავისი საკუთარი ფუნქცია, რომელიც სრულდება საბაზო კლასის ვირტუალური ფუნქციის ნაცვლად, ან გამოიყენოს საბაზო ფუნქცია. სხვა სიტყვებით რომ ვთქვათ წარმოებულ კლასს შეუძლია არც განსაზღვროს თავისი საკუთარი მეთოდი. პროგრამის მიხედვით ზოგჯერ აზრი არა აქვს საბაზო კლასში ვირტუალური ფუნქციის განსაზღვრას. მაგალითად, წარმოებული ტიპის ობიექტები შეიძლება

იმდენად იყოს განსხვავებული, მათ არც კი ესაჭიროებოდეთ საბაზო კლასის მეთოდის გამოყენება. ასეთ შემთხვევაში საბაზო კლასის ვირტუალური ფუნქციისათვის ოპერატორების განსაზღვრის ნაცვლად, თქვენ პროგრამებს შეუძლიათ შექმნან წმინდად ვირტუალური ფუნქცია, რომელიც არ შეიცავს ოპერატორებს.

წმინდად ვირტუალური ფუნქციის შესაქმნელად პროგრამა მიუთითებს ფუნქციის პროტოტიპს, მაგრამ არ მიუთითებს მის ოპერატორებს. მათ ნაცვლად პროგრამა ფუნქციას მიანიჭებს ნულ მნიშვნელობას, როგორც ქვემოთ არის ნაჩვენები:

```
class phone
{
public:
  virtual void dial (char *number) =0; // წმინდად ვირტუალური
ფუნქცია
  void answer(void) { cout << "pasuxis molodini" << endl; }
  void hangup(void) { cout << "zari Sesrulebulia – kurmilis
dakideba" << endl; }
  void ring(void) { cout << "zari, zari, zari " << endl; }
  phone(char *number) { strcpy(phone::number, number); };
protected:
  char number[13];
};
```

დასკვნა:

- პოლიმორფულ ობიექტს შეუძლია შეიცვალოს ფორმა პროგრამის შესრულების დროს;
- თქვენ ქმნით პოლიმორფულ ობიექტებს კლასების გამოყენებით, რომლებიც შექმნილია არსებული საბაზო კლასებიდან;
- საბაზო კლასში პოლიმორფული ობიექტისათვის უნდა განსაზღვროთ ერთი ან რამდენიმე ფუნქცია, როგორც ვირტუალური(*virtual)*;
- ზოგადად პოლიმორფული ობიექტები განსხვავდებიან საბაზო კლასის ვირტუალური ფუნქციების გამოყენებით;
- პოლიმორფული ობიექტის შესაქმნელად საჭიროა შექმნათ საბაზო კლასის ობიექტზე მიმთითებელი;
- პოლიმორფული ობიექტის ფორმის შესაცვლელად მიმთითებელი უნდა მიმართოთ სხვადასხვა ობიექტზე;
- წმინდად ვირტუალური ფუნქცია არის საბაზო კლასის ვირტუალური ფუნქცია, რომლისთვისაც საბაზო კლასში არ არის განსაზღვრული ოპერატორები. მათ ნაცვლად საბაზო კლასი ასეთ ფუნქციას ანიჭებს 0 მნიშვნელობას;
- წარმოებული კლასები უნდა უზრუნველყოფდნენ ფუნქციის განსაზღვრას საბაზო კლასის ცალკეული წმინდად ვირტუალური ფუნქციისათვის.

ლაბორატორიული სამუშაო 13
თემა: ფუნქციების და კლასების შაბლონების შექმნა
ფუნქციების შაბლონების გამოყენება

ფუნქციების შექმნისას წარმოიშვება სიტუაციები, როდესაც ორი ფუნქცია ასრულებს ერთიდაიგივე მოქმედებას, მაგრამ მუშაობენ სხვადასხვა მონაცემთა ტიპებთან(მაგალითად, ერთი გამოიყენებს int ტიპის პარამეტრებს, ხოლო მეორე – float ტიპის). დავუშვათ, გვაქვს ფუნქცია max სახელით, რომელიც აბრუნებს ორი მთელი რიცხვიდან უდიდესს. თუ მოგვიანებით ჩვენ დაგვჭირდება მსგავსი ფუნქცია, რომელიც აბრუნებს უდიდესს ორი მცოცავწერტილიანი მნიშვნელობიდან, საჭიროა განვსაზღვროთ სხვა ფუნქცია, მაგალითად fmax. მოცემულ მასალაში თქვენ შეისწავლით თუ როგორ გამოვიყენოთ C++ ენის შაბლონები ისეთი ფუნქციების სწრაფად შექმნისათვის, რომლებიც აბრუნებენ სხვადასხვა ტიპის მნიშვნელობებს.

მოცემულ მასალაში თქვენ შეისწავლით შემდეგ კონცეფციებს:

– შაბლონი განსაზღვრავს ოპერატორების ნაკრებს, რომელთა დახმარებით თქვენ პროგრამას მოგვიანებით შეუძლია რამდენიმე ფუნქციის შექმნა;

– ფუნქციების შაბლონი ხშირად გამოიყენება რამდენიმე ფუნქციის სწრაფად განსაზღვრისათვის, რომლებიც ერთნაირი ოპერატორების დახმარებით მუშაობენ სხვადასხვა ტიპის პარამეტრებთან ანუ

აქვთ დასაბრუნებელი მნიშვნელობის სხვადასხვა ტიპი;

– ფუნქციის შაბლონებს აქვთ სპეციფიური სახელები, რომლებიც შეესაბამებიან პროგრამაში თქვენ მიერ გამოყენებულ ფუნქციის სახელებს;

– პროგრამაში ფუნქციის შაბლონის განსაზღვრის შემდგომ, შეიძლება შეიქმნას კონკრეტული ფუნქცია, სადაც გამოიყენება ეს შაბლონი პროტოტიპის დასაწერად, რომელიც შეიცავს მოცემული შაბლონის სახელს, დასაბრუნებელ მნიშვნელობას და პარამეტრების ტიპს;

– კომპილაციის პროცესში C++ ენის კომპილატორი თქვენს პროგრამაში შექმნის ფუნქციებს პროტოტიპებში მითითებული ტიპების გამოყენებით, რომლებიც იგზავნებიან შაბლონის სახელზე.

ფუნქციის საბლონებს აქვთ უნიკალური სინტაქსი, რომელიც ერთი შეხედვით შეიძლება გაუგებარი იყოს. რამდენიმე შაბლონის შექმნის შემდეგ თქვენ მიხვდებით, რომ რეალურად მათი გამოყენება იოლია.

ფუნქციის მარტივი შაბლონის შექმნა

ფუნქციის შაბლონი განსაზღვრავს ტიპობრივად დამოუკიდებელ ფუნქციას. ასეთი შაბლონის დახმარებით პროგრამაში შეიძლება განისაზღვროს კონკრეტული ფუნქციები საჭირო ტიპებით. მაგალითად, ქვემოთ განსაზღვრულია შაბლონი max ფუნქციისათვის, რომელიც აბრუნებს უდიდესს ორი მნიშვნელობიდან:

```
template<class T> T max(T a, T b)
```

```
{
  if (a > b) return(a);
  else return(b);
}
```

T სიმბოლო მოცემულ შემთხვევაში წარმოადგენს შაბლონის ზოგად ტიპს. შაბლონის განსაზღვრის შემდეგ თქვენ პროგრამაში გამოაცხადეთ პროტოტიპი ცალკეული მოთხოვნილი ტიპისათვის. ჩვენს შემთხვევაში:

```
float max(float, float);
int max(int, int);
```

როდესაც C++ კომპილატორს შეხვდება ეს პროტოტიპები, ფუნქციების აგებისას T შაბლონის ტიპს იგი შეცვლის თქვენ მიერ მითითებული ტიპით. float ტიპის შემთხვევაში max ფუნქცია შეცვლის შემდეგ მიიღებს სახეს:

```
float max(float a, float b)

{
  if (a > b) return(a) ;
  else return(b);
}
```

ქვემოთ წარმოდგენილი პროგრამა გამოიყენებს max შაბლონს int და float ტიპის ფუნქციების შესაქმნელად:

```
#include <iostream.h>
template<class T> T max(T a, T b)
{
  if (a > b) return(a);
```

```
  else return(b);
}
float max(float, float);
int max(int, int);
void main(void)
{
  cout << "udidesi 100 da 200 Soris tolia " << max(100, 200) <<
endl;
  cout << "udidesi 5.4321 da 1.2345 Soris tolia " << max(5.4321,
1.2345) << endl;
}
```

რადგან ეს ფუნქცია შექმნილია შაბლონის დახმარებით, იგი ნებას რთავს გამოყენებულ იქნას ერთნაირი სახელი ფუნქციებისათვის, რომლებიც აბრუნებენ განსხვავებული ტიპების მნიშვნელობებს. ამის გაკეთება შეუძლებელი იქნებოდა მხოლოდ ფუნქციების გადატვირთვის გამოყენებით.

შაბლონები, რომლებიც გამოიყენებენ რამდენიმე ტიპს

შემდეგი ოპერატორები ქმნიან შაბლონს show_array ფუნქციისათვის, რომელიც გამოიტანს მასივის ელემენტებს. შაბლონი გამოიყენებს T ტიპს მასივის ტიპის განსაზღვრისათვის და T1 ტიპს count პარამეტრის ტიპის მისათითებლად:

```
template<class T,class T1> void show_array(T *array,T1 count)

{
  T1 index;
 for (index =0; index < count; index++) cout << array[index] << ' ';
  cout << endl;
}
```

როგორც ზემოთ, პროგრამამ უნდა მიუთითოს ფუნქციების პროტოტიპები მოთხოვნილი ტიპებისათვის:

```
void show_array(int *, int);
void show_array(float *, unsigned);
```

ქვემოთ წარმოდგენილი პროგრამა გამოიყენებს შაბლონს ფუნქციის შექმნისათვის, რომლებიც გამოიტანენ int და float ტიპის მასივებს:

```
#include <iostream.h>
template<class T,class T1> void show_array( T *array,T1 count)
{
  T1 index;
  for (index =0; index < count; index++) cout << array[index] “ ' ';
  cout << endl;
}
void show_array(int *, int);
void show_array(float *, unsigned);
void main(void)
{
  int pages[] = { 100, 200, 300, 400, 500 };
  float pricesH = { 10.05, 20.10, 30.15 };
  show_array(pages, 5);
  show_array(prices, 3);
}
```

მოცემულ შემთხვევაში ფუნქცია იყენებს ორ პარამეტრს. ერთი შეესაბამება მასივს, ხოლო მეორე - მასივში ელემენტების რაოდენობას. უფრო უნივერსალური შაბლონი პროგრამას ამ პარამეტრის საკუთარი ტიპის მითითების შესაძლებლობას მისცემდა, როგორც ქვემოთ არის ნაჩვენები:

template<class T, class T1> void array_sort(T array[], T1 elements)
{
// операторы
}

array_sort შაბლონის დახმარებით პროგრამას შეუძლია შექმნას ფუნქცია, რომელიც სორტირებას უკეთებს ***float*** ტიპის პატარა მასივებს (128 ელემენტზე ნაკლები) და ***int*** ტიპის ძალიან დიდი ზომის მასივებს შემდეგი პროტოტიპების გამოყენებით:

void array_sort(float, char);
void array_sort(int, long);

დასკვნა:

- ფუნქციების შაბლონები საშუალებას იძლევა გამოცხადდეს ტიპობრივად დამოუკიდებელი ანუ ზოგადი ფუნქციები;
- როდესაც პროგრამაში მოითხოვება ფუნქციის გამოყენება განსაზღვრული მონაცემთა ტიპებით, მან უნდა მიუთითოს ფუნქციის პროტოტიპი, რომელიც განსაზღვრავს მოთხოვნილ ტიპებს;
- როდესაც C++ კომპილატორს შეხვდება ასეთი ფუნქციის პროტოტიპი, იგი ქმნის ამ ფუნქციის შესაბამის ოპერატორებს, მოთხოვნილი ტიპების გამოყენებით;
- პროგრამამ უნდა შექმნას შაბლონები საერთო ფუნქციებისათვის, რომლებიც მუშაობენ განსხვავებული ტიპებისთვის. სხვა სიტყვებით, თუ თქვენ გამოიყენებთ რომელიმე ფუნქციაში მხოლოდ ერთ

ტიპს, არ არის შაბლონის გამოყენების აუცილებლობა;

– თუ ფუნქცია მოითხოვს რამდენიმე ტიპს, შაბლონი უბრალოდ უნიშნავს ცალკეულ ტიპს უნიკალურ იდენტიფიკატორს, მაგალითად T, T1 და T2. მოგვიანებით კომპილაციის პროცესში კომპილატორი კორექტულად დანიშნავს თქვენ მიერ ფუნქციის პროტოტიპში მითითებულ ტიპებს.

კლასების შაბლონების გამოყენება

მოცემულ მასალაში თქვენ შეისწავლით შემდეგ ძირითად კონცეფციებს:

– template გასაღებური სიტყვის და ტიპების სიმბოლოების (მაგალითად, T, T1 და T2) გამოყენებით პროგრამას შეუძლია შექმნას კლასის შაბლონი – კლასის შაბლონის განსაზღვრას შეუძლია გამოიყენოს ეს სიმბოლოები მონაცემთა ელემენტების გამოსაცხადებლად, პარამეტრების ტიპების და ფუნქციის დასაბრუნებელი მნიშვნელობის საჩვენებლად და ა.შ.

– შაბლონის გამოყენებით კლასის ობიექტების შესაქმნელად პროგრამა აკეთებს მიმართვას კლასის სახელზე, რომლის შემდეგაც კუთხოვან ფრჩხილში იწერება ტიპი (მაგალითად, <int, float>), რომელთაგან თითოეულს კომპილატორი დაუნიშნავს ტიპების სიმბოლოებს და ცვლადის სახელს;

– თუ კლასს აქვს კონსტრუქტორი, რომლის დახმარებით ინიციალიზებას უკეთებთ მონაცემ-

ელემენტებს, შეგიძლიათ გამოიძახოთ ეს კონსტრუქტორი ობიექტის შექმნისას შაბლონის გამოყენებით, მაგალითად:

class_name<int,float>values(200);

– თუ კომპილატორს შეხვდება ობიექტის გამოცხადება, იგი ქმნის კლასს შაბლონიდან შესაბამისი ტიპების გამოყენებით.

კლასის შაბლონის შექმნა

ზოგიერთი პროგრამისათვის შექმნილი კლასი შეიძლება გამოდგეს სხვა პროგრამისთვისაც. ხშირად კლასები შეიძლება განსხვავდებოდნენ მხოლოდ ტიპებით. სხვა სიტყვებით, ერთი კლასი მუშაობდეს მთელრიცხვა მნიშვნელობებით, ხოლო მოცემულ მომენტში მოთხოვნილმა კლასმა იმუშაოს ***float*** ტიპის მნიშვნელობებით. არსებული კოდის ხელმეორედ გამოყენების მიზნით ***C++*** საშუალებას იძლევა განისაზღვროს კლასების შაბლონი. მოკლედ რომ ვთქვათ, კლასის შაბლონი განვსაზღვროთ როგორც ტიპზე დამოუკიდებელი კლასი, რომელიც მომავალში ემსახურება მოთხოვნილი ტიპების ობიექტების შექმნას. თუ კომპილატორს შეხვდება კლასის შაბლონზე დაფუძნებული ობიექტის გამოცხადება, მაშინ მოთხოვნილი ტიპის კლასის ასაგებად იგი გამოიყენებს გამოცხადებისას მითითებულ ტიპებს. მხოლოდ ტიპებით განსხვავებული კლასების სწრაფი შექმნის თვალსაზრისით კლასების შაბლონები ამცირებენ დაპროგრამების მოცულობას.

დავუშვათ ვქმნით მთელი ტიპის მასივის კლასს, რომელშიც არის მეთოდები წევრთა ჯამის და საშუალო

არითმეტიკულის გამოსათვლელად. კლასი გამოიყურება შემდეგი სახით:

```
class array

{
public:
  array(int size);
  long sum(void);
  int average_value(void);
  void show_array(void);
  int add_value(int);
private:
  int *data;
  int size;
  int index;
};
```

ქვემოთ წარმოდგენილია პროგრამა, რომელიც იყენებს array კლასს int ტიპის მნიშვნელობებთან სამუშაოდ:

```
#include <iostream.h>
#include <stdlib.h>
class array
{
public:
  array(int size);
  long sum(void);
  int average_value(void);
  void show_array(void);
  int add_value(int) ;
private:
  int *data;
```

```
  int size;
  int index;
};

array::array(int size)

{
  data = new int [size];
  if (data == NULL)

  {
    cerr << "arasakmarisia mexsiereba –programa dasruldeba " <<
endl;
    exit(l);
  }

  array:: size = size;
  array::index = 0;
}

long array::sum(void)

{
  long sum = 0;
  for (int i = 0; i < index; i++) sum += data[i];
  return(sum);
}

int array::average_value(void)

{
  long sum = 0;
  for (int i = 0; i < index; i++) sum += data[i];
```

```
  return (sum / index);
}
void array::show_array(void)
{
  for (int i = 0; i < index; i++) cout << data[i] << ' ';
  cout << endl;
}
int array::add_value(int value)
{
  if (index == size) return(-1); // masivi savsea
  else
  {
    data[index] = value;
    index++;
    return(0); //warmatebiT
  }
}
void main(void)

{
  array numbers (100); // 100 elementiani masivi
  int i;
  for (i = 0; i < 50; i++) numbers.add_value(i);
  numbers.show_array();
  cout << "ricxvebis jami tolia " << numbers.sum () << endl;
  cout << "saSualo mniSvneloba tolia " <<
numbers.average_value() << endl;
}
```

როგორც ვხედავთ, პროგრამა ანაწილებს მასივის 100 ელემენტს, შემდეგ შეაქვს მასივში 50 მნიშვნელობა add_value მეთოდის დახმარებით. array კლასში index ცვლადი მიედევნება ელემენტების რაოდენობას, რომელიც შენახულია მოცემულ მომენტში მასივში. თუ

მომხმარებელი შეეცდება დაამატოს იმაზე მეტი ელემენტი რასაც მასივი იტევს, add_value ფუნქცია აბრუნებს შეცდომას. როგორც ხედავთ average_value ფუნქცია იყენებს index ცვლადს მასივის საშუალო მნიშვნელობის განსაზღვრისათვის. პროგრამა ითხოვს მეხსიერებას მასივისათვის new ოპერატორის გამოყენებით.

დავუშვათ პროგრამას, რომელიც მუშაობს მთელ რიცხვებთან, სჭირდება მუშაობა მცოცავწერტილიანი მნიშვნელობების მასივებთან. ერთ-ერთი ხერხი ამ შემთხვევაში მდგომარეობს სხვადასხვა კლასების შექმნაში. მეორეს მხრივ კლასების შაბლონის გამოყენებით თქვენ გვერდს აუვლით კლასების დუბლირებას. ქვემოთ წარმოდგენილია კლასის შაბლონი, რომელიც ქმნის ზოგად array კლასს:

```
template<class T, class T1> class array

{
public:
  array(int size);
  T1 sum (void);
  T average_value(void);
  void show_array(void);
  int add_value(T);
private:
  T *data;
  int size;
  int index;
};
```

ეს შაბლონი განსაზღვრავს T და T1 ტიპების სიმბოლოებს. მთელრიცხვა მასივის შემთხვევაში T შეესა-

ბამება int ტიპს, ხოლო T1 – long ტიპს. ანალოგიურად მცოცავწერტილიანი მნიშვნელობების შემთხვევაში T და T1 უდრის float-ს.

შემდგომში კლასის ცალკეული ფუნქციის წინ უნდა მიუთითოთ ისეთივე ჩანაწერი template სიტყვით. კლასის სახელის შემდეგ უნდა მიუთითოთ კლასის ტიპები, მაგალითად array <T, T1>::average_value. შემდეგი ოპერატორი გვიჩვენებს ამ კლასისათვის average_value ფუნქციის განსაზღვრას:

```
template<class T, class T1> T array<T, T1>::average_value(void)

{
  T1 sum = 0;
  int i;
  for (i = 0; i < index; i++) sum += data[i] ;
  return (sum / index);
}
```

შაბლონის შექმნის შემდეგ შეგიძლიათ შექმნათ მოთხოვნილი ტიპის კლასი, კლასის სახელის მითითებით, რომელსაც თან ახლავს კუთხურ ფრჩხილებში საჭირო ტიპები, როგორც ქვემოთ არის ნაჩვენები:

შაბლონის სახელი//----> array <int, long> numbers (100); <----// *შაბლონის ტიპი*

array <float, float> values(200);

ქვემოთ წარმოდგენილი პროგრამა გამოიყენებს array კლასის შაბლონს ორი კლასის შესაქმნელად,

რომელთაგან ერთი მუშაობს int ტიპის, ხოლო მეორე – float ტიპის მნიშვნელობებთან:

```
#include <iostream.h>
#include <stdlib.h>
template<class T, class T1> class array
{
public:
  array(int size);
  T1 sum(void);
  T average_value(void);
  void show_array(void);
  int add_value(T);
private:
  T *data;
  int size;
  int index;
};
template<class T, class T1> array<T, t1>::array(int size)
{
  data = new T[size];
  if (data == NULL)
  {
    cerr << "მეხსიერება არასაკმარისია – პროგრამა
დასრულდება" << endl;
    exit(l);
  }
  array::size = size;
  array::index = 0;
}
template<class T, class T1> Tl array<T, Tl>::sum(void)
```

```
{
  T1 sum = 0;
  for (int i = 0; i < index; i++) sum += data[i];
  return(sum);
}
template<class T, class T1> T array<T, T1>::average_value(void)
{
  Tl sum =0;
  for (int i = 0; i < index; i++) sum += data[i];
  return (sum / index);
}
template<class T, class T1> void array<T, T1>::show_array(void)
{
  for (int i = 0; i < index; i++) cout << data[i] << ' ';
  cout << endl;
}
template<class T, class T1> int array<T, T1>::add_value(T value)
{
  if (index == size)
  return(-1); // მასივი  სავსეა
  else
  {
    data[index] = value;
    index++;
    return(0); // წარმატებით
  }
}
void main(void)
{
  // 100 ელემენტიანი  მასივი
  array<int, long> numbers(100)7
  // 200 ელემენტიანი  მასივი
  array<float, float> values(200);
  int i;
```

```
  for (i = 0; i < 50; i++) numbers.add_value(i);
  numbers.show_array();
  cout << "რიცხვების ჯამი ტოლია " << numbers.sum () <<
endl;
  cout << "საშუალო მნიშვნელობა ტოლია " <<
numbers.average_value() << endl;
  for (i = 0; i < 100; i++) values.add_value(i * 100);
  values.show_array();
  cout << "რიცხვების ჯამი ტოლია " << values.sum() <<
endl;
  cout << "საშუალო მნიშვნელობა ტოლია " <<
values.average_value() << endl;
}
```

ობიექტების გამოცხადება კლასის შაბლონის საფუძველზე

კლასის შაბლონის გამოყენებით ობიექტების შექმნისათვის უნდა მიუთითოთ კლასის შაბლონის სახელი, რომლის შემდეგაც კუთხოვან ფრჩხილებში მიუთითეთ ტიპები, რომლითაც კომპილატორი შეცვლის ***T, T1, T2*** და ა.შ. სიმბოლოებს. ამის შემდეგ პროგრამამ უნდა მიუთითოს ობიექტის (ცვლადის) სახელი იმ პარამეტრების მნიშვნელობებით, რომელიც უნდა გადასცეთ კლასის კონსტრუქტორს, როგორც ქვემოთ არის ნაჩვენები:

template_class_name<typel, type2> object_name(parameter1, parameter2);

როდესაც კომპილატორს ხვდება ასეთი გამოცხადება, იგი ქმნის მითითებულ ტიპებზე დაფუძნებულ კლასს. მაგალითად, შემდეგი ოპერატორი იყენებს ***array*** კლასის

შაბლონს ***char*** ტიპის მასივის შესაქმნელად, რომელშიც ინახება 100 ელემენტი:

array<char, int> small_numbers(100) ;

დასკვნა:

- კლასების შაბლონები საშუალებას იძლევა თავი ავარიდოთ კოდის დუბლირებას ისეთი კლასებისათვის, რომლის ობიექტები განსხვავდებიან მათი ელემენტების მხოლოდ ტიპებით;
- კლასის შაბლონის შესაქმნელად კლასის გამოცხადებას წაუმძღვარეთ template გასაღებური სიტყვა და ტიპების სიმბოლოები, მაგალითად T და T1;
- შემდგომ კლასის ცალკეული ფუნქციის განსაზღვრას წაუმძღვარეთ template გასაღებური სიტყვა. გარდა ამისა მიუთითეთ შაბლონის ტიპი კუთხოვან ფრჩხილებში, ხოლო გამოსახულება კუთხოვან ფრჩხილებში განათავსეთ კლასის სახელსა და ხედვის არეს დაშვების ოპერატორს შორის, მაგალითად:
  class_name<T,T1>::function_name.
- შაბლონის გამოყენებით კლასის შესაქმნელად მიუთითეთ კლასის სახელი და ტიპებისათვის ჩამნაცვლებელი მნიშვნელობები, კუთხოვან ფრჩხილებში. მაგალითად, class_name<int, long> object.

ლაბორატორიული სამუშაო 14
თემა: **cin** და **cout** დამატებითი შესაძლებლობები

როგორც განვლილი მასალებიდან ჩანს, **c++** ენაზე დაწერილი ყოველი პროგრამა შეიცავს *iostream.h* სათაო ფაილს. მაგრამ აქამდე ჩვენ არ ვიცოდით რა არის ამ ფაილში. ეს ფაილი განსაზღვრას კლასებს *istream* და *ostream* (შემავალი ნაკადი და გამომავალი ნაკადი), ხოლო *cin* და *cout* არიან ამ კლასების ცვლადები (ობიექტები).

მოცემულ მასალაში თქვენ გაეცნობით როგორ ფართოვდება შეტანა-გამოტანის შესაძლებლობები ფუნქციების გამოყენებით, რომლებიც ჩაშენებულია **cin** და **cout** კლასებში.

თქვენ შეისწავლით შემდეგ ძირითად კონცეფციებს:

- **iostream.h** სათაო ფაილი შეიცავს კლასის განსაზღვრას, რომელიც თქვენ შეგიძლიათ გააანალიზოთ, რათა უკეთ გაიგოთ ნაკადური შეტანა/გამოტანა;
- **cout.width** მეთოდის გამოყენებით თქვენ პროგრამებს შეუძლიათ გამოტანის სიგანის მართვა;
- **cout.fill** მეთოდის გამოყენებით თქვენ პროგრამებს შეუძლიათ ცარიელი გამომავალი სიმბოლოების (ტაბულაცია და ხარვეზი) შეცვლა ზოგიერთი განსაზღვრული სიმბოლოთი;
- მცოცავწერტლიანი მნიშვნელობების გამომავალ ნაკადში ციფრების რაოდენობის მართვისათვის შეიძლება გამოყენებულ იქნას **cout.setprecision** მეთოდი;

– სიმბოლოების სათითაოდ შეტანა-გამოტანისათვის პროგრამას შეუძლია გამოიყენოს **cout.put** და **cin.get** ნაკადური მეთოდები;
– **cin.getline** მეთოდის გამოყენებით შეიძლება მთელი სტრიქონის გამოტანა ერთდროულად.

cout გამოყენება

როგორც უკვე იცით **cout** წარმოადგენს კლასს, რომელიც შეიცავს სხვადასხვა მეთოდს. ქვემოთ წარმოდგენილ პროგრამაში ნაჩვენებია ზოგიერთი მეთოდის გამოყენება გამოტანის ფორმატირებისათვის. *setw* მანიპულატორი პროგრამას საშუალებას აძლევს მიუთითოს სიმბოლოების მინიმალური რაოდენობა, რომელიც შეუძლია დაიკავოს გამომავალმა მნიშვნელობამ:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
  cout << "გამოიტანს რიცხვს" << setw(3) << 1001 << endl;
  cout << "გამოიტანს რიცხვს" << setw (4) << 1001 << endl;
  cout << "გამოიტანს რიცხვს" << setw (5) << 1001 << endl;
  cout << "გამოიტანს რიცხვს" << setw(6) << 1001 << endl;
}
```

ანალოგიურად მოქმედებს *cout.width* მეთოდი, რომელიც წარმოდგენილია ქვემოთ:

```
#include <iostream.h>
#include <iomanip.h>
```

```
void main (void)
{
  int i;
  for (i = 3; i < 7; i++)
  {
    cout << "გამოიტანს რიცხვს";
    cout. width (i);
    cout << 1001 << endl;
  }
}
```

პროგრამის შესრულებისას ეკრანზე გამოვა:

გამოიტანს რიცხვს**1001**
გამოიტანს რიცხვს**1001**
გამოიტანს რიცხვს **1001**
გამოიტანს რიცხვს **1001**

შემავსებელი სიმბოლოს გამოყენება

ზოგჯერ საჭიროა შემავსებელი სიმბოლოს გამოყენება. დავუშვათ ვაკეთებთ სარჩევს, სადაც ჩამონათვალსა და გვერდის ნომერს შორის შემავსებლად გამოყენებულია სიმბოლო წერტილი.

ინფორმაციის ცხრილი

კომპანიის პროფილი.....10
კომპანიის შემოსავლები და დანაკარგები.....13
კომპანიის წევრები.....15

განვიხილოთ პროგრამა, სადაც *cout.fill* ფუნქცია საშუალებას იძლევა მივუთითოთ სიმბოლო, რომელიც შეავსებს ცარიელ სივრცეს:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
```

```
{
  cout << "ინფორმაციის ცხრილი" << endl;
  cout.fill ('.');
  cout << "კომპანიის პროფილი" << setw(20) << 10 << endl;
  cout << "კომპანიის შემოსავლები და დანაკარგები" <<
setw(12) << 13 << endl;
  cout << "კომპანიის წევრები" << setw(14) << 15 << endl;
}
```

მცოცავწერტილიანი მნიშვნელობების თანრიგების მართვა

თუ პროგრამაში *cout-ს* ვიყენებთ მცოცავწერტილიანი მნიშვნელობების გამოსატანად, წინასწარ ვერ ვივარაუდებთ წერტილის შემდეგ თუ რამდენი ციფრი იქნება დუმილით გამოტანილი. **setprecision** მანიპულატორის გამოყენებით შეგვიძლია მივუთითოთ საჭირო ციფრების რაოდენობა. ქვემოთ წარმოდგენილ პროგრამაში გამოყენებულია **setprecision** მანიპულატორი წერტილიდან მარჯვნივ ციფრთა რაოდენობის მართვისათვის:

```
#include <iostream.h>
#include <iomanip.h>
void main(void)
{
  float value = 1.23456;
  int i;
  for (i = 1; i < 6; i++) cout << setprecision(i) << value << endl;
}
```

პროგრამის შესრულებისას გამოვა შემდეგი შედეგი:

1.2
1.23

1.235
1.2346
1.23456

setprecision მანიპულატორის აწყობები მოქმედებს მანამ, სანამ პროგრამა ხელახლა არ გამოიყენებს მოცემულ მანიპულატორს.

ცალკეულ ჯერზე თითო სიმბოლოს გამოტანა

ქვემოთ წარმოდგენილ პროგრამაში გამოყენებულია *cout.put* ფუნქცია შეტყობინების: ”ვსწავლობთ დაპროგრამებას c++ ენაზე” სიმბოლო-სიმბოლოდ გამოსატანად:

```
#include <iostream.h>
void main(void)
{
  char string[] = "ვსწავლობთ დაპროგრამებას C++ ენაზე!";
  int i;
  for (i = 0; string[i]; i++) cout.put(string[i]) ;
}
```

შემდეგ პროგრამაში გამოყენებულია *toupper* ფუნქცია სიმბოლოს ზედა რეგისტრში გადასაყვანად, ხოლო შემდეგ ეს სიმბოლო გამოდის *cout.put* ფუნქციის დახმარებით:

```
#include <iostream.h>
#include <ctype.h> // toupper პროტოტიპი
void main(void)
{
  char string[] = "C++ language";
  int i;
  for (i = 0; string[i]; i++) cout.put(toupper(string[i]));
  cout << endl << "შედეგი  სტრიქონი: " << string << endl;
}
```

პროგრამის გაშვების შემდეგ ეკრანზე გამოვა შედეგი:
C++ LANGUAGE
შედეგი სტრიქონი**: C++ language**

კლავიატურიდან სტრიქონის ცალ-ცალკე სიმბოლოებად წაკითხვა

cin წარმოგვიდგენს *cin.get* ფუნქციას, რომელიც ცალკეული სიმბოლოს წაკითხვის საშუალებას იძლევა. მოცემული ფუნქციით სარგებლობისათვის სიმბოლოს ანიჭებთ ამ ფუნქციის დასაბრუნებელ ცვლადს:

```
letter = cin.get();
```

ქვემოთ წარმოდგენილია პროგრამა, რომელსაც გამოაქვს შეტყობინება, რომლის საპასუხოდ უნდა შეიტანოთ Y ან N. შემდეგ იგი იმეორებს ციკლში cin.get ფუნქციის გამოძახებას სიმბოლოს წასაკითხად, სანამ არ მიიღებს Y ან N პასუხს:

```
#include <iostream.h>
#include <ctype.h>
void main(void)
{
  char letter;
  cout << "გავაგრძელო? (Y/N): ";
  do
  {
    letter = cin.get();
      // ზედა რეგისტრში გარდაქმნა
    letter = toupper(letter);
  } while ((letter != 'Y') && (letter != 'N'));
  cout << endl << "თქვენ  შეიტანეთ " << letter << endl;
}
```

კლავიატურიდან მთელი სტრიქონის წაკითხვა

ხშირ შემთხვევაში საჭიროა, რომ პროგრამამ წაიკითხოს მთელი სიმბოლური სტრიქონი, რისთვისაც გამოიყენება *cin.getline* ფუნქცია. აღნიშნული ფუნქციის გამოყენებისას საჭიროა მიუთითოთ სიმბოლური სტრიქონი, რომელშიც განთავსდება სიმბოლოები და ასევე სტრიქონის სიგრძე, როგორც ქვემოთ არის ნაჩვენები:

```
cin.getline(string, 64);
```

როდესაც *cin.getline* კითხულობს სიმბოლოებს კლავიატურიდან, იგი არ წაიკითხავს იმაზე მეტს რასაც დაიტევს სტრიქონი. მასივის ზომის განსაზღვრისათვის მოსახერხებელია *sizeof* ოპერატორის გამოყენება, როგორც ქვემოთ არის ნაჩვენები:

```
cin.getline(string, sizeof(string));
```

თუ თქვენ მოგვიანებით შეცვლით სტრიქონის ზომას, არ მოგიწევთ თქვენს პროგრამაში არსებულ *cin.get*-თან დაკავშირებული ცალკეული ოპერატორის ძებნა და შეცვლა. ამის ნაცვლად *sizeof* ოპერატორი გამოიყენებს სტრიქონის კორექტირებულ ზომას. ქვემოთ მოყვანილი პროგრამა გამოიყენებს cin.getline ფუნქციას ტექსტური სტრიქონის კლავიატურიდან წასაკითხად:

```
#include <iostream.h>
void main(void)
{
  char string[128];
  cout << "შეიტანეთ ტექსტური სტრიქონი და დააჭირეთ Enter"
<< endl;
```

```
  cin.getline(string, sizeof(string));
  cout << "თქვენ შეიტანეთ: " << string << endl;
}
```

ზოგჯერ საჭიროა არა მთელი სტრიქონის, არამედ რომელიმე სიმბოლომდე წაკითხვა, მსგავსი ოპერაციის შესასრულებლად საძიებო სიმბოლო უნდა იქნას გადაცემული *cin.getline* ფუნქციაში. ქვემოთ მოყვანილი ფუნქცია სტრიქონს წაიკითხავს მანამ, სანამ არ შეხვდება სტრიქონზე გადასვლა ან სანამ არ წაიკითხავს 64 სიმბოლოს ან სანამ არ შეხვდება სიმბოლო “z”:

```
cin.getline(string, 64, 'z');
```

განვიხილოთ პროგრამა, რომელიც გამოიყენებს *cin.getline* ფუნქციას სტრიქონის წასაკითხად “z” სიმბოლოს ჩათვლით:

```
#include <iostream.h>
void main(void)
{
  char string[128];
  cout << "შეიტანეთ სტრიქონი და დააჭირეთ Enter" << endl;
  cin.getline(string, sizeof(string), 'z');
  cout << "თქვენ შეიტანეთ: " << string << endl;
}
```

გააკეთეთ ტესტირება სხვადახვა სტრიქონისათვის. ზოგიერთი მათგანი დაიწყეთ z სიმბოლოთი, ზოგიერთი დაასრულეთ z სიმბოლოთი, ხოლო ზოგიერთი საერთოდ არ შეიცავდეს z სიმბოლოს.

დასკვნა:

- *cin* და *cout* არიან *istream* და *ostream* კლასების ობიექტები (ცვლადები), რომლებიც განსაზღვრულია *iostream.h* სათაო ფაილში. ამდენად ისინი წარმოადგენენ ფუნქციებს, რომელთაც თქვენი პროგრამები გამოიძახებენ განსაზღვრული ამოცანების ამოსახსნელად;
- cout.width საშუალებას იძლევა გამოტანის დროს მითითებულ იქნას სიმბოლოების მინიმალური რაოდენობა;
- cout.fill ფუნქცია საშუალებას იძლევა მითითებულ იქნას cout.width ან setw გამოყენებით წარმოქმნილი ცარიელი სივრცის შემავსებელი სიმბოლო;
- setprecision მანიპულატორი საშუალებას იძლევა მცოცავწერტილიანი რიცხვების გამოტანის დროს მითითებულ იქნას წერტილის შემდეგ ციფრთა რაოდენობა;
- cin.get და cout.put ფუნქციები საშუალებას იძლევა შეტანილ ან გამოტანილ იქნას ერთი სიმბოლო;
- cin.getline ფუნქცია სტრიქონის კლავიატურიდან წაკითხვის საშუალებას იძლევა.

ლაბორატორიული სამუშაო 15
თემა: შეტანა-გამოტანის ფაილური ოპერაციები

პროგრამების გართულებასთან ერთად პროგრამებმა უნდა შეინახონ და მიიღონ ინფორმაცია ფაილების გამოყენებით.

მოცემულ მასალაში განხილულია შემდეგი ძირითადი კონცეფციები:

- გამომავალი ფაილური ნაკადის გამოყენებით თქვენ შეგიძლიათ ჩაწეროთ ინფორმაცია ფაილში ჩასმის ოპერატორის (<<) დახმარებით;
- შემავალი ფაილური ნაკადის გამოყენებით თქვენ შეგიძლიათ ფაილში შენახული ინფორმაციის წაკითხვა ამოღების ოპერატორის (>>) დახმარებით;
- ფაილის გახსნისა და დახურვისათვის გამოიყენებთ ფაილური კლასების მეთოდებს;
- ფაილური მონაცემების წაკითხვისა და ჩაწერისათვის შეგიძლიათ გამოიყენოთ ჩასმის და ამოღების ოპერატორები, ასევე ფაილური კლასების ზოგიერთი მეთოდები.

შეტანა ფაილურ ნაკადში

სათაო ფაილი *iostream.h* განსაზღვრავს *cout* გამომავალ ნაკადს. ანალოგიურად, სათაო ფაილი *fstream.h* განსაზღვრავს გამომავალი ფაილური ნაკადის კლასს, რომლის სახელია *ofstream.* ამ კლასის ობიექტების გამოყენებით პროგრამებს შეუძლია შეასრულოს გამოტანა ფაილში. დასაწყისისათვის თქვენ უნდა გამოაცხადოთ *ofstream* ტიპის ობიექტი, სადაც მოთხოვნილი გამომავალი ფაილის სახელი

მიეთითება სიმბოლური სტრიქონის სახით, როგორც ქვემოთ არის ნაჩვენები:

```
ofstream  file_object("FILENAME.EXT");
```

ofstream ტიპის ობიექტის გამოცხადებისას თუ თქვენ მიუთითებთ ფაილის სახელს, დისკზე შეიქმნება ახალი ფაილი მითითებული სახელის გამოყენებით ან გადაეწერება ამავე სახელის ფაილს თუ იგი უკვე არსებობს დისკზე. ქვემოთ წარმოდგენილი პროგრამა ქმნის *ofstream* ტიპის ობიექტს და შემდეგ გამოიყენებს ჩასმის ოპერატორს ტექსტის რამდენიმე სტრიქონის შესატანად BOOKINFO.TXT ფაილში:

```
#include <fstream.h>
void main(void)
{
  ofstream  book_file("d:\BOOKINFO.TXT");
  book_file << "ვსწავლობთ დაპროგრამებას C++ ენაზე, " <<
"მეორე რედაქცია" << endl;
  book_file << "Jamsa Press" << endl;
  book_file << "22.95" << endl;
}
```

მოცემულ მაგალითში პროგრამა ხსნის BOOKINFO.TXT ფაილს და შემდეგ ჩაწერს სამ სტრიქონს ფაილში:

```
ვსწავლობთ დაპროგრამებას ჩ++ ენაზე, მეორე რედაქცია
Jamsa Press
22.95
```

წაკითხვა შემავალი ფაილური ნაკადიდან

წაკითხვა ფაილიდან ხდება *ifstream* ტიპის ობიექტების გამოყენებით. ანალოგიურად თქვენ ქმნით ობიექტს, რომელსაც პარამეტრად გადასცემთ მოთხოვნილი ფაილის სახელს:

```
ifstream input_file("filename.EXT");
```

ქვემოთ წარმოდგენილი პროგრამა ხსნის BOOKINFO.TXT ფაილს, რომელიც თქვენ შექმენით წინა პროგრამაში და კითხულობს, შემდეგ კი ასახავს ფაილის პირველ-სამ ელემენტს:

```
#include <iostream.h>
#include <fstream.h>
    void main(void)
        {
  ifstream input_file("d:\BOOKINFO.TXT") ;
  char one[64], two[64], three[64];
  input_file >> one;
  input_file >> two;
  input_file >> three;
  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

მოცემული პროგრამის გაშვების შემდეგ იგი ასახავს ფაილის პირველ-სამ სიტყვას. *cin* მსგავსად შემავალი ფაილური ნაკადები გამოიყენებენ ცარიელ სიმბოლოებს რათა განისაზღვროს სად მთავრდება ერთი

მნიშვნელობა და სად იწყება ახალი. ეკრანზე გამოჩნდება შედეგი:

ვსწავლობთ
დაპროგრამებას
c++

მთელი სტრიქონის წაკითხვა

როგორც წინა მასალაში განვიხილეთ კლავიატურიდან მთელი სტრიქონის წასაკითხად გამოიყენება *cin.getline*. მსგავსად *ifstream* ტიპის ობიექტებს შეუძლიათ *getline* გამოიყენონ ფაილური შეტანის სტრიქონების წასაკითხად. ქვემოთ წარმოდგენილი პროგრამა გამოიყენებს *getline*-ს BOOKINFO.TXT ფაილიდან სამივე სტრიქონის წასაკითხად:

```
#include <iostream.h>
#include <fstream.h>
      void main(void)
      {
  ifstream input_file("d:\BOOKINFO.TXT");
  char one[64], two[64], three [64] ;
  input_file.getline(one, sizeof(one)) ;
  input_file.getline(two, sizeof(two));
  input_file.getline(three, sizeof(three)) ;
  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

მოცემულ შემთხვევაში პროგრამა წარმატებით კითხულობს ფაილის შინაარსს, რადგან მან იცის, რომ ფაილი შეიცავს სამ სტრიქონს. ხშირ შემთხვევაში

პროგრამამ არ იცის რამდენი სტრიქონია ფაილში. ასეთ შემთხვევებში წაკითხვა გაგრძელდება მანამ, სანამ არ შეხვდება ფაილის დასასრული.

ფაილის დასასრულის განსაზღვრა

პროგრამაში ჩვეულებრივი ფაილური ოპერაცია არის ფაილის შინაარსის წაკითხვა, სანამ არ შეხვდება ფაილის დასასრული. ფაილის დასასრულის განსაზღვრისათვის პროგრამაში გამოიყენება eof ფუნქცია. ეს ფუნქცია აბრუნებს 0 მნიშვნელობას, თუ ფაილის დასასრული ჯერ კიდევ არ შეხვდა, ხოლო 1-ს თუ ფაილის დასასრული შეხვდა. *while* ციკლის გამოყენებით პროგრამას შეუძლია განუწყვეტლივ წაიკითხოს ფაილის შინაარსი, სანამ არ იპოვის ფაილის დასასრულს, როგორც ქვემოთ არის ნაჩვენები:

```
while (! input_file.eof())
{
  // ოპერატორები
}
```

მოცემულ შემთხვევაში პროგრამა აგრძელებს ციკლის შესრულებას, სანამ *eof* ფუნქცია აბრუნებს მცდარს. ქვემოთ წარმოდგენილი პროგრამა ფაილს კითხულობს დასასრულის მიღწევამდე:

```
#include <iostream.h>
#include <fstream.h>
void main (void)
{
  ifstream  input_file("d:\BOOKINFO.TXT");
```

```
  char line[64];
  while (! input_file.eof())
        {
    input_file.getline(line, sizeof(line));
    cout << line << endl;
  }
}
```

ანალოგიურად, შემდეგი პროგრამა კითხულობს ფაილის შინაარსს თითო ჯერზე თითო სიტყვას, სანამ არ შეხვდება ფაილის დასასრული:

```
       #include <iostream.h>
       #include <fstream.h>
       void main(void)
       {
  ifstream input_file("d:\BOOKINFO.TXT");
  char word[64] ;
  while (! input_file.eof())
        {
    input_file >> word;
    cout << word << endl;
  }
}
```

ხოლო შემდეგი პროგრამა *get* ფუნქციის გამოყენებით კითხულობს ფაილის შინაარსს თითო ჯერზე თითო სიმბოლოს, სანამ არ შეხვდება ფაილის დასასრული:

```
       #include <iostream.h>
       #include <fstream.h>
       void main(void)
       {
  ifstream input_file("d:\BOOKINFO.TXT");
```

```
  char letter;
  while (! input_file.eof())
        {
    letter = input_file.get();
    cout << letter;
  }
}
```

ფაილური ოპერაციების შესრულებისას შეცდომების შემოწმება

აქამდე წარმოდგენილი პროგრამები გვთავაზობდნენ, რომ შეტანა-გამოტანის ფაილური ოპერაციების დროს არ ხდება შეცდომები. მაგრამ ეს ყოველთვის ასე არ არის. მაგალითად, თუ ხსნით ფაილს შესატანად, პროგრამამ უნდა შეამოწმოს, რომ ფაილი არსებობს. ანალოგიურად, თუ პროგრამას შეაქვს მონაცემები ფაილში, აუცილებელია დარწმუნდეთ, რომ ოპერაციამ წარმატებით ჩაიარა. შეცდომებზე თვალყურის მიდევნების მიზნით შესაძლებელია ფაილური ობიექტის fail ფუნქციის გამოყენება. თუ ფაილური ოპერაციის პროცესში შეცდომები არ იყო, ფუნქცია დააბრუნებს მცდარს(0). მხოლოდ თუ შეხვდა შეცდომა fail ფუნქცია დააბრუნებს ჭეშმარიტს. მაგალითად, თუ პროგრამა ხსნის ფაილს, მან უნდა გამოიყენოს fail ფუნქცია, რათა განისაზღვროს მოხდა თუ არა შეცდომა, როგორც ქვემოთ არის ნაჩვენები:

```
    ifstream input_file("FILENAME.DAT");
if (input_file.fail())
```

```
        {
  cerr << " FILENAME.EXT გახსნის შეცდომა" << endl;
  exit(1);
}
```

ამდენად, პროგრამები უნდა დარწმუნდნენ, რომ წაკითხვის და ჩაწერის ოპერაციებმა ჩაიარა წარმატებით. შემდეგი პროგრამა გამოიყენებს fail ფუნქციას სხვადასხვა შეცდომითი სიტუაციების შესამოწმებლად:

```
        #include <iostream.h>
        #include <fstream.h>
        void main(void)
        {
  char line[256] ;
  ifstream input_file("BOOKINFO.TXT") ;
  if (input_file.fail()) cerr << "BOOKINFO.TXT ფაილის გახსნის
შეცდომა"<< endl;
  else
          {
    while ((! input_file.eof()) && (! input_file.fail()))
            {
      input_file.getline(line, sizeof(line)) ;
      if (! input_file.fail()) cout << line << endl;
    }
  }
}
```

ფაილის დახურვა

ფაილითან მუშობის დასრულების შემდეგ იგი უნდა დაიხუროს, რისთვისაც გამოიყენება *close* ფუნქცია, როგორც ქვემოთ არის ნაჩვენები:

```
input_file.close ();
```

პროგრამის მიერ შეტანილი ყველა ინფორმაცია ინახება ფაილში.

ფაილის გახსნის მართვა

არსებული ფაილის ბოლოში ინფორმაციის დასამატებლად საჭიროა ფაილი გაიხსნას დამატების რეჟიმში, რისთვისაც ფაილის გახსნისას უნდა მიეთითოს მეორე პარამეტრი, როგორც ქვემოთ არის ნაჩვენები:

```
ifstream output_file("FILENAME.EXT", ios::app);
```

მოცემულ შემთხვევაში *ios::app* პარამეტრი უჩვენებს ფაილის გახსნის რეჟიმს. პროგრამების სირთულის მიხედვით ფაილის გახსნის რეჟიმისათვის გამოიყენება მნიშვნელობათა შერწყმა, რომელიც წარმოდგენილია ცხრილში:

გახსნის რეჟიმი	დანიშნულება
ios::app	ხსნის ფაილს დამატების რეჟიმში, ფაილზე მიმთითებელს განათავსებს ფაილის ბოლოში
ios::ate	ფაილზე მიმთითებელს განათავსებს ფაილის ბოლოში
ios::in	მიუთითებს ფაილის გახსნას შესატანად
ios::nocreate	თუ მითითებული ფაილი არ არსებობს, არ შეიქმნას ფაილი და დააბრუნოს შეცდომა
ios::noreplace	თუ ფაილი არსებობს, გახსნის ოპერაცია უნდა შეწყდეს და დააბრუნოს შეცდომა
ios::out	მიუთითებს ფაილის გახსნას გამოსატანად
ios::trunc	ჩამოყრის(გადაწერს) შინაარსს არსებული ფაილიდან

ფაილის გახსნის შემდეგი ოპერაცია ხსნის ფაილს გამოსატანად, *ios::noreplace* რეჟიმის გამოყენებით, რათა თავიდან აიცილოს არსებულ ფაილზე გადაწერა:

```
ifstream output_file("FIlename.EXT", ios::out | ios::noreplace);
```

წაკითხვის და ჩაწერის ოპერაციების შესრულება

მოცემულ მასალაში წარმოდგენილი ყველა პროგრამა ასრულებდა ფაილურ ოპერაციებს სიმბოლურ სტრიქონებზე. არანაკლებ საჭიროა მასივების და სტრიქონების წაკითვა და ჩაწერაც. ამისათვის უნდა იქნას გამოყენებული *read* და *write* ფუნქციები. ამ ფუნქციების გამოყენებისას უნდა მიუთითოთ მონაცემთა ბუფერი, რომელშიც მონაცემები წაიკითხება ან საიდანაც ისინი ჩაიწერება, ასევე ბუფერის სიგრძე ბაიტებში, როგორც ქვემოთ არის ნაჩვენები:

```
input_file.read(buffer, sizeof(buffer)) ;
output_file.write(buffer, sizeof(buffer));
```

შემდეგი პროგრამა გამოიყენებს *write* ფუნქციას სტრუქტურის შემადგენლობის EMPLOYEE.DAT ფაილში ჩასაწერად:

```
#include <iostream.h>
#include <fstream.h>
void main(void)
{
  struct employee
  {
    char name[64];
    int age;
```

```
    float salary;
  } worker = { "Happy Jamsa", 33, 25000.0 };
  ofstream emp_file("d:\EMPLOYEE.DAT") ;
  emp_file.write((char *) &worker, sizeof(employee));
}
```

write ფუნქცია ჩვეულებრივ იღებს მიმთითებელს სიმბოლურ სტრიქონზე. სიმბოლოები *(char *)* წარმოადგენს ტიპებში მოყვანის ოპერატორს, რომელიც კომპილატორს უკეთებს ინფორმირებას, რომ თქვენ გადასცემთ სხვა ტიპზე მიმთითებელს. მსგავსად, მომდევნო პროგრამა გამოიყენებს *read* მეთოდს ფაილიდან თანამშრომლის შესახებ ინფორმაციის წასაკითხად:

```
#include <iostream.h>
#include <fstream.h>
    void main(void)
      {
  struct employee
        {
    char name [6 4] ;
    int age;
    float salary;
  } worker = { "Happy Jamsa", 33, 25000.0 };
       ifstream emp_file("d:\EMPLOYEE.DAT");
  emp_file.read((char *) &worker, sizeof(employee));
  cout << worker.name << endl;
  cout << worker.age << endl;
  cout << worker.salary << endl;
}
```

დასკვნა:

- *fstream.h* სათაო ფაილი განსაზღვრავს *ifstream* და *ofstream* კლასებს, რომელთა დახმარებით პროგრამას შეუძლია შეასრულოს ფაილური შეტანა-გამოტანის ოპერაციები;
- ფაილის შესატანად ან გამოსატანად გახსნისათვის ობიექტი უნდა გამოაცხადოთ *ifstream* ან *ofstream* ტიპად, ამ ობიექტის კონსტრუქტორზე მოთხოვნილი ფაილის სახელის გადაცემით;
- მას შემდეგ რაც პროგრამაში გაიხსნება ფაილი შეტანის ან გამოტანისათვის, შესაძლებელია მონაცემების წაკითხვა ან ჩაწერა ამოღების (<<) ან ჩასმის (>>) ოპერატორების გამოყენებით;
- პროგრამას შეუძლია შეასრულოს სიმბოლოების შეტანა ან გამოტანა ფაილში ან ფაილიდან, *get* და *put* ფუნქციების გამოყენებით;
- პროგრამას შეუძლია ფაილიდან წაიკითხოს მთელი სტრიქონი *getline* ფუნქციის გამოყენებით;
- უმეტესი პროგრამები კითხულობენ ფაილის შინაარსს, ვიდრე არ შეხვდება ფაილის დასასრული. ფაილის დასასრული შეიძლება განისაზღვროს *eof* ფუნქციის დახმარებით;
- როდესაც პროგრამა ასრულებს ფაილურ ოპერაციებს, მან უნდა შეამოწმოს ყველა ოპერაციის მდგომარეობა, რათა დარწმუნდეს, რომ ოპერაციები შესრულებულია წარმატებულად. შეცდომების შესამოწმებლად შეიძლება გამოყენებულ იქნას *fail* ფუნქცია;
- როდესაც ფაილში საჭიროა მასივების ან სტრუქტურების შეტანა ან გამოტანა, შეიძლება გამოყენებულ იქნას *read* და *write* მეთოდები;
- ფაილთან მუშაობის დასრულების შემდეგ საჭიროა მისი დახურვა *close* ფუნქციის გამოყენებით.

ლიტერატურა

1. Роберт Лафоре `Object-Oriented Programming in c++", изд. Питер, Классика Computer Science, ISBN 978-5-4237-0038-6, 0-672-32308-7; 2011г.
2. გ. სურგულაძე, ობიექტ-ორიენტირებული დაპროგრამების მეთოდი, ტექნიკური უნივერსიტეტი,2008.ISBN 99940-56-18-2.
3. Харви Дейтел, Пол Дейтел `Как программировать на c++~, стр.1037.
4. Грэхем И. Объектно-ориентированные методы: Принципы и практика: пер. с англ. Изд. 3-е. М: Вильямс, 2004. 880 с.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2003. 368 с.
6. Элиенс А. Принципы объектно-ориентированной разработки программ. М.: Вильямс, 2002. 496 с.
7. Кендалл Скотт. UML. Основные концепции. Пер. с англ. М.: Издательский дом «Вильямс», 2002.144 с.: ил.
8. Учебный курс «Основы объектно-ориентированного программирования». http://www.intuit.ru/department/se/oopbases/