ლევან შოშიაშვილი

მოდელირება და ვიზუალიზაცია

(ლექცია/პრაქტიკული/შენიშვნები)

2025

სარჩევი

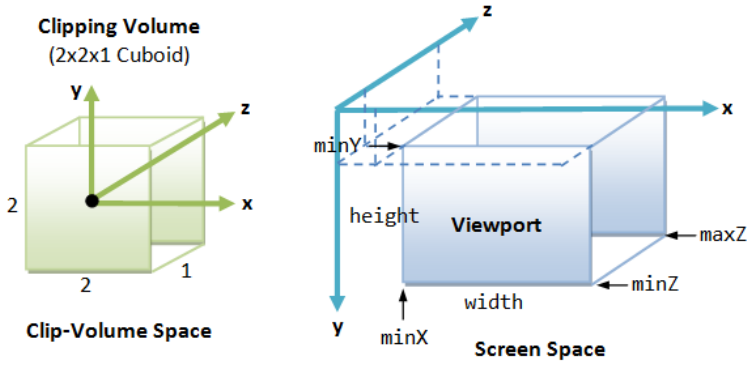
საკოორდინატო სისტემები

{refsection:1}

1.1 საკოორდინატო სისტემები

გამოვიყენებთ ბიბლიოთეკას **OpenGL**.

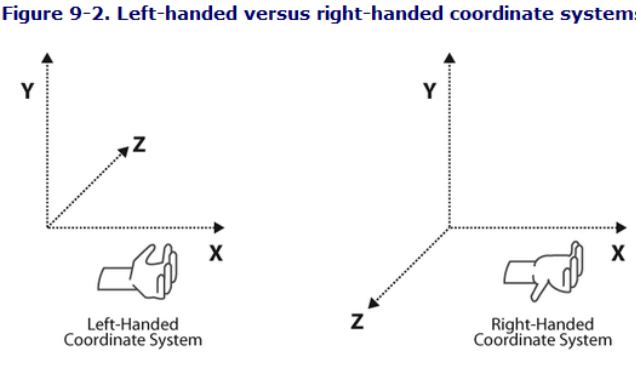
მონიტორზე კოორდინატები აითვლება მარცხენა ზედა კუთხიდან.



{fig:monitor}

სურ 1.1: ეკრანის საკოორდინატო სისტემა

OpenGL კოორდინატები აითვლება მარცხენა ქვედა კუთხიდან. საკოორდინატო სისტემა არის X, Y, Z ოღონდ Z მიმართულება არის ჩვნსკენ ეკრანის სიბრტყიდან(მარჯვენა საკოორდინატო სისტემა). ანუ გეომეტრია რომელიც იხატება **OpenGL**-ში იხატება ამრჯვენა საკოორდინატო სისტემაში (იხ. სურ. 1.2). კამერა (სხვაგვარად თუ როგორ ვუყურებთ ჩვენ 3D სივრცეს სადაც არის ობიექტები) იყენებს მარცხენა საკოორდინატო სისტემას.



g:coordinates}

სურ 1.2: მარცხენა და მარჯვენა საკოორდინატო სისტემები

OpenGL 3D/2D გამოსახულება დამოუკიდებელია მომხმარებლის მიერ გამოყენებული ერთეულებისაგან. ანუ თქვენ შეგიძლიათ იმუშაოთ სმ, მ, მმ, ინჩ ერთეულებში. როგორ ფორმირდება **OpenGL** გამოსახულება მაგ. ე.წ. „კამერა ობსკურაში“ მოცემულია ენჯელის წიგნი გვ. 40[2]. x, y, z არის წერტილის კოორდინატები, ხოლო x^*, y^* პროექციის კოორდინატები. z კოორდინატი პროეცირდება ცხადია $-d$ წერტილში.

მნიშვნელოვანი პარამეტრია კამერის ხედვის კუთხე. ეს ახასიათებს ობიექტის მაქსიმალურ სიმაღლეს რომელიც პროეცირდება კამერის ფირფიტაზე და იკავებს ფირფიტის მაქსიმალურ სი-

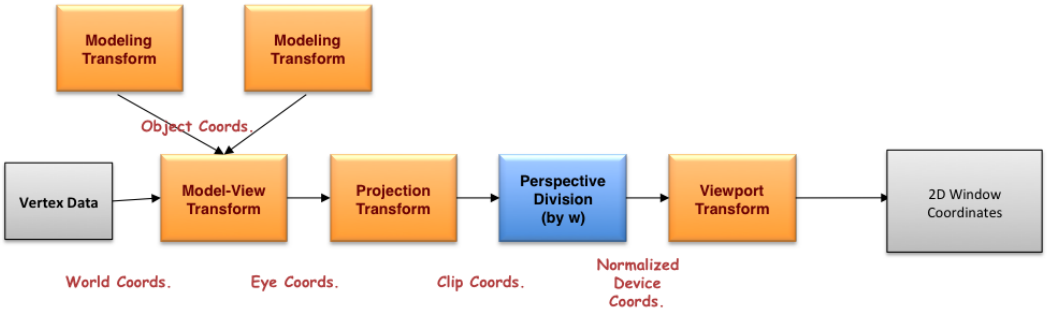
[2] შემდგომში მივუთითებთ როგორც „ენჯ.“

მაღლეს. იხ.(ენჯ. სურ. 1.16. გვ 41)

$$tg(\theta/2) = h/(2d)$$

ამ კამერას აქვს რამდენიმე ნაკლი. ვინაიდან ნახვრეტი საიდანაც შედის სხივი უნდა იყოს რაც შეიძლება მცირე (იმისათვის რომ შევიდეს მხოლოდ ერთი სხივი და წერტილი აისახოს წერტილში) განათებულობა მცირეა. ამავე დროს შეუძლებელია ხედვის კუთხის შეცვლა. ეს ორი პრობლემა მოხსნილია თანამედროვე ფოტო აპარატებში, ვინაიდან გამოყენებულია ლინზები, რაც საშუალებას იძლევა მიიღოს მეტი ენერგია(ობიექტივის ზომის შეცვლით) ხოლო ფოკუსური მანძილის ცვლილებით შეიძლება შეიცვალოს ხედვის კუთხე.

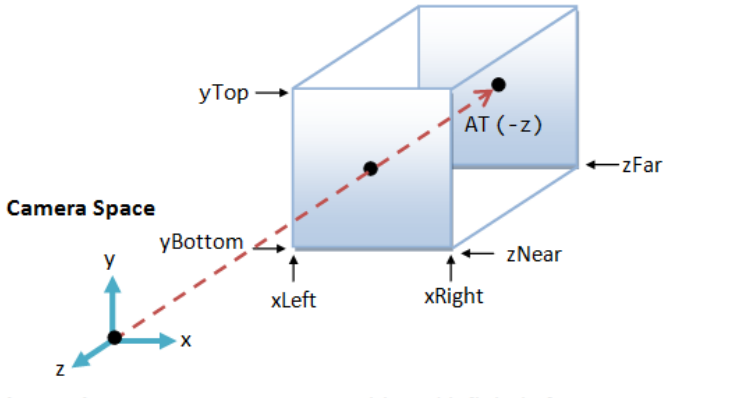
3D გამოსახულება პროეცირდება ეკრანზე შემდეგი სქემის გავლით:



{fig:pipeline}

სურ 1.3: 3D გამოსახულების ეკრანზე გამოტანის "გზა"

პროექციული გარდაქმნა შეიძლება იყოს ორი სახის: ორთოგრაფიული და პერსპექტივაში:



hics3DOrthographic}

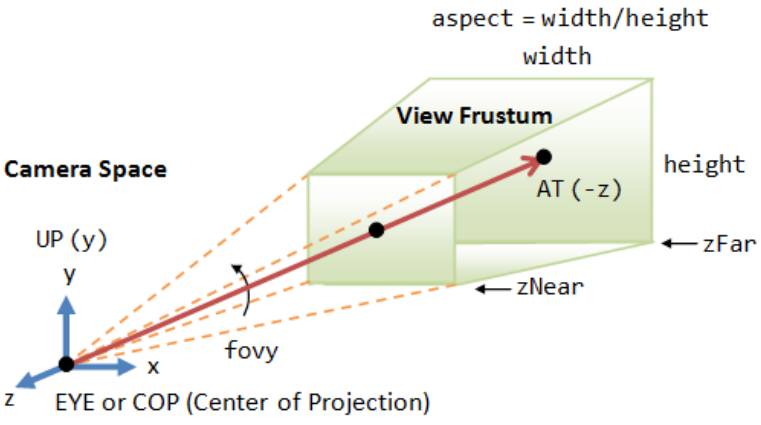
სურ 1.4: ორთოგრაფიული პროექცია

ის ობიექტები რომლებიც ხვდება $xLeft$, $xRight$, $zNear$, $zFar$, $yBotton$, $yTop$, კუბში ორთოგრაფიული პროექციის შემტხვევაში აისახება ეკრანზე.

ის ობიექტები რომლებიც ხვდება $zNear$, $zFar$, წაკვეთილ პირამიდაში პერსპექტიული პროქციის შემტხვევაში აისახება ეკრანზე.

„eye“-„თვალი“ ან რაც იგივეა პროექციის ცენტრი არის ის წერტილები მსოფლიო საკოორდინატო სისტემაში საიდანაც ვუყურებთ 3D "სცენას".

„fovy“– ესაა ე.წ. ხედვის კუთხე. მნიშვნელოვანია შეფარდება $width/height$ ეს პროპორცია დაცული უნდა იყოს იმისათვის რომ ეკრანულ კოორდინატებზე ასახული სურათში დაცული იყოს სცენის/ობიექტის პროპორციები.



სურ 1.5: პერსპექტიული პროექცია

3dperspective}

კამერის სურათიდან ჩანს რომ $\vec{z}$ მიმართულია ხედვის მიმართულების საპირისპიროდ. ანუ ხედვა ხდება კამერიდან უარყოფითი მიმართულებით. კამერის საკოორდინატო სისტემა „მარცხენაა“.

თუ როგორ მუშაობს ხედვა და პროექცია. გაუშვით პროგრამები matrixModelView, matrixProjection და/ან ნეიტ რობინსის სასწავლო პროგრამები : transformation, projection.

C++ ელემენტები

refsection:2}

2.1 მონაცემების გადაცემა მისამართით

გავიხსენოთ Cკოდი სადაც ხდებოდა მონაცემების გადაცემა ფუნქციაში მიმთითებლის საშუალებით(იხ. არქივი *sqequation*)

c_declaration}

კოდი 2.1 ფუნქციის დეკლარირება

```
#ifndef FUNCTIONS___H
#define FUNCTIONS___H
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
    #ifdef __cplusplus
    extern "C" {
    #endif
    int solve_sqeq(const double* _a, const  double* _b,
    const double* _c, double* _x1, double* _x2);
    #ifdef __cplusplus
    };
    #endif
#endif // MYHEADER___H
```

mplementation}

კოდი 2.2 ფუნქციის იმპლემენტაცია

```
#include <stdio.h>
#include <math.h>
#include "functions.h"
#define EPS 0.000001
int solve_sqeq(const double*_a, const double* _b,
const  double* _c, double* _x1, double* _x2)
{
  double d=(*_b)*(*_b)-4*(*_a)*(*_c);

  if(d<0){
  #ifdef _DEBUG
     printf("D<0 araa amoxsna\n");
  #endif
    return 0;
  }
  else if(d<EPS)
  {
    #ifdef _DEBUG
    printf("D=0 aqvs 1 amonaxseni\n");
    #endif
    (*_x1)=-(*_b)/(2*(*_a));
   _x2=_x1;
   #ifdef _DEBUG
   printf("x1=x2=%g\n",(*_x1));
   #endif // _DEBUG
   return 1;
  }
  #ifdef _DEBUG
  printf("aris 2 fesvi:\n");
  #endif // _DEBUG
  (*_x1)=(-(*_b)-sqrt(d))/(2*(*_a));
```

```
32    (*_x2)=(-(*_b)+sqrt(d))/(2*(*_a));
33    #ifdef _DEBUG
34    printf("x1=%g:\n x2=%g\n",(*_x1),(*_x2));
35    #endif
36    return 2;
37  }
```

{kodi:quadratic}

კოდი 2.3 quadratic.c პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას

```
1   #include "functions.h"
2   #define EPS 0.0000001
3
4   int main(int argc, char* argv[])
5   {
6       double a=1;
7     double b=2;
8     double c=4;
9     double x1=0;
10    double x2=0;
11    int result=-1;
12    puts("shemoitanet kv. gant. koeficientebi: a, b, c");
13    scanf("%lf,%lf,%lf",&a,&b,&c);
14    result=solve_sqeq(&a,&b,&c,&x1,&x2);
15    switch(result)
16    {
17      case 0:
18      printf("D<0 ar aqvs amoxsna\n");
19      break;
20      case 1:
21      printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
22      printf("x1=%g:\n x2=%g\n", x1, x2);
23      break;
24      case 2:
25      printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
26      printf("x1=%g:\n x2=%g\n", x1, x2);
27      break;
28      default:
29      break;
30    }
31    return 0;
32  }
```

კოდში 2.2 ჩანს როგორ უნდა ვიმუშაოთ მიმთითებელთან. ფუნქციის გამოძახებისას კი ფუნქციას უნდა გადავცეთ ცვლადის მისამართი(მიმთითებლის მნიშვნელობა არის მისამართი.). C-ში მონაცემების გადაცემა ხდება მნიშვნელობით. ეს შეიძლება იყოს რიცხვი –ცვლადის მნიშვნელობა ან მისამართი ასევე ცვლადის(მისამართის შემთხვევაში მიმთითებლის მნიშვნელობა.

C++-ში ცხადია გვაქვს, როგორც მონაცემების გადაცემა მნიშვნელობით, ასევე მიმთითებლით. . ამას გარდა C++-ში გვაქვს მონაცემების გადაცემა მისამართით(Reference). ამისათვის გამოიყენება ნიშანი ამპერსანდი „&“.

განვიხილოთ კოდები სადაც გამოყენებულია მისამართით გადაცემა (კოდების არქივი *sqequationcpp*)

di:cpp_declaration}

კოდი 2.4 ფუნქციის დეკლარირება *functionscpp.h*

```
1   #ifndef FUNCTIONSCPP___H
2   #define FUNCTIONSCPP___H
3   #include <math.h>
4   #include <stdio.h>
```

```
5  #include <stdlib.h>
6
7  int solveSQeq(const double& _a, const  double& _b,
8      const double& _c, double& _x1, double& _x2);
9
10 #endif // MYHEADER___H
```

mplementation}

კოდი 2.5 ფუნქციის იმპლემენტაცია *functionscpp.cpp*

```
1  #include <stdio.h>
2  #include <math.h>
3  #include "functionscpp.h"
4  #define EPS 0.000001
5  int solveSQeq(const double& _a, const double& _b,
6  const  double& _c, double& _x1, double& _x2)
7  {
8    double d=_b*_b-4*_a*_c;
9
10   if(d<0){
11   #ifdef _DEBUG
12      printf("D<0 araa amoxsna\n");
13   #endif
14     return 0;
15   }
16   else if(d<EPS)
17   {
18     #ifdef _DEBUG
19     printf("D=0 aqvs 1 amonaxseni\n");
20     #endif
21     _x1=-_b/(2*_a);
22    _x2=_x1;
23    #ifdef _DEBUG
24    printf("x1=x2=%g\n",(*_x1));
25    #endif // _DEBUG
26    return 1;
27   }
28   #ifdef _DEBUG
29   printf("aris 2 fesvi:\n");
30   #endif // _DEBUG
31   _x1=(-_b-sqrt(d))/(2*_a);
32   _x2=(-_b+sqrt(d))/(2*_a);
33   #ifdef _DEBUG
34   printf("x1=%g:\n x2=%g\n",(*_x1),(*_x2));
35   #endif
36   return 2;
37 }
```

quadraticcpp}

კოდი 2.6 პროგრამის ფაილი სადაც ვიყენებთ ფუნქციას *quadraticcpp.cpp*

```
1   #include "functionscpp.h"
2   #define EPS 0.0000001
3   int main(int argc, char* argv[])
4   {
5     double a=1;
6     double b=2;
7     double c=4;
8     double x1=0;
9     double x2=0;
10    int result=-1;
11    puts("shemoitanet kv. gant. koeficientebi: a, b, c");
```

```
12      scanf("%lf,%lf,%lf",&a,&b,&c);
13      result=solveSQeq(a,b,c,x1,x2);
14      switch(result)
15      {
16        case 0:
17        printf("D<0 ar aqvs amoxsna\n");
18        break;
19        case 1:
20        printf("D<EPS=%g aqvs 1 fesvi\n",EPS);
21        printf("x1=%g:\n x2=%g\n", x1, x2);
22        break;
23        case 2:
24        printf("D>EPS=%g aqvs 2 fesvi\n",EPS);
25        printf("x1=%g:\n x2=%g\n", x1, x2);
26        break;
27        default:
28        break;
29      }
30      return 0;
31    }
```

თუ შევადარებთ კოდებს 2.2 და 2.5 , ასევე 2.3 და 2.6 ვნახავთ რომ C++ ვარიანტში(უფრო სწორად მონაცემების მისამართით გადაცემისას) აღარაა საჭირო ფრჩხილები და „*“ გამოყენება. ასევე ფუნქციის გამოძახება (ხაზი 13 კოდში 2.3) არაფრით განსხვავდება ფუნქციის გამოძახებისგან, მისამართების ნაცვლად ცვლადის მნიშვნელობით გადაცემა რომ გამოგვეყენებინა(მაგრამ ამ შემთხვევაში როგორც ვნახეთ C-ს კურსში ცვლადის მნიშვნელობას „ვერ გამოვიტანდით“ ფუნქციის გარეთ).

როგორც ვხედავთ, მისამართის გადაცემის შემთხვევაშიც შენარჩუნებული გვაქვს const სპეციფიკატორი (ხაზი 5 კოდში 2.5), რაც ნიშნავს რომ აკრძალულია ამ ცვლადების მნიშვნელობების შეცვლა ფუნქციის შიგნით.

ხშირად const პარამეტრებს ფუნქციაში გადასცემენ მისამართის საშუალებით როგორც ზემოთ მაგალითში გვაქვს, და არა const ცვლადებს მიმთითებლის საშუალებით(როგორც ეს გვაქვს Cვარიანტში).

C++-ში არის შემთხვევები როცა ფუნქციაში ცვლადის გადაცემა მიმთითებლით აუცილებელია. მაგალითად როცა არ ვიცით რა ტიპის ცვლადია და საჭიროა ცვლადის დაყვანა(კასტირება, რეინტერპრეტაცია) ფუნქციის შიგნით.

სადაც შესაძლებელია, ჩვენ გამოვიყენებთ ცვლადის გადაცემას მისამართით, რაც კოდის ადვილად წაკითხვას და შეცდომებისგან ცოტა მეტ დაზღვევას განაპირობებს.

2.2 კლასები

C++ ენას თავიდან ერქვა „Cკლასებით“, რაც აღნიშნავდა რომ C++ წარმოადგენდა Cენის შემდგომ გაფართოებას. კლასის იდეა C++ ენაში შევიდა იმ დროს არსებული სხვა ობიექტზე ორიენტირებული ენებიდან *SmallTalk* და *Simula*.

კლასი class მსგავსია struct, მაგრამ თუ სტრუქტურაში ყველა წევრი ხელმისაწვდომია, კლასში შეიძლება კლასის წევრ ობიექტებზე პირდაპირი წვდომა აკრძალული იყოს. ასევე კლასში შეიძლება გვქონდეს ფუნქციები. მათ შორის სპეციალური ფუნქციები, რომლებიც უზრუნველყოფენ კლასის ობიექტის შექმნას და წაშლას მეხსიერებაში. ამ ფუნქციების ეწოდებათ კლასის კონსტრუქტორი და დესტრუქტორი,

კლასში ღია წვდომის წევრები აღწერილი უნდა იყოს, როგორც public. წვდომა შეზღუდული წევრებისთვის გამოიყენება სიტყვა private.

C++ საშუალებას იძლევა წევრები და ფუნქციები გვქონდეს სტრუქტურაშიც. განსხვავება კლასისაგან ისაა, რომ თუ სპეციალურად აღნიშნული არაა სტრუქტურისთვის ყველა წევრი იგულისხმება თავისუფალი წვდომის წევრად (public), ხოლო კლასისათვის აკრძალული წვდომის

წევრად (private).

ქვემოთ მოყვანილი კოდი კომპილირდება და მუშაობს როგორც **C++**, მაგრამ არა როგორც **C**

```
typedef struct type_A
{
  private:
  double z;
  public:
  double x;
  double y;
  double getX()
  { return x; }
  void setX(double x_)
  { x = x_; }
  setZ(double _z)
  { z=_z;}

} A;
int main(int argc, char* argv[])
{
  A x;
  x.x=1;
  x.y=3;
  x.z=1;// shecdomaa. z-ze pirdapiri cvdoma akrdzalulis
  double t=x.getX();
  x.setX(6);
  x.setZ(6);//amis ufleba gvaqvs
  return 0;
}
```

სწორი პრაქტიკაა შემდეგი: თუ ობიექტის წევრები არის თავისუფალი წვდომის და ობიექტისთვის რამდენიმე ფუნქცია გჭირდებათ აღწერეთ ეს ობიექტი როგორც სტრუქტურა. ცალკე აღწერეთ ამ ობიექტზე სამუშაო ფუნქციები(ანუ მოექეცით ობიექტს როგორც **C**ენის სტრუქტურას). წინააღმდეგ შემთხვევაში ობიექტისათვის შექმენით კლასი.

კლასის შექმნა და კლასთან მუშაობა განვიხილოთ `Point2D` კლასის მაგალითზე.

2.3 კლასი 2D წერტილი — `Point2D`

როცა რაღაცას განვიხილავთ როგორც ობიექტს და გვსურს შევქმნათ კლასი უნდა გავიაზროთ შემდეგი:

- ☑ რა თვისებები/წევრები აქვს ობიექტს
- ☑ რა თვისებები/წევრები გვსურს იყოს თავისუფლად ხელმისაწვდომი და რა შეზღუდული.
  ხშირად ყველა წევრს აკეთებენ private *და ამატებენ ფუნქციებს(ფუნქციებს უწოდებენ „მეთოდებს“) წევრებთან სამუშაოდ.*
- ☑ რა მეთოდები(ფუნქციები) გვჭირდება ობიექტთან სამუშაოდ. რა მეთოდები უნდა იყოს თავისუფლად ხელმისაწვდომი და რა შეზღუდული.
  არის შემთხვევები როცა გვჭირდება ფუნქცია კლასის შეიგნით სამუშაოდ, მაგრამ არაა საჭირო, რომ ეს ფუნქცია ხელმისაწვდომი იყოს კლასის გარეთ.
- ☑ რა ოპერაციები შეიძლება გაკეთდეს ობიექტებზე (რიცხვზე გამრავლება გაყოფა. ორი ობიექტის შეკრება გამოკლება.)

ორ განზომილებაში წერტილის კლასის (იხ. კოდების არქივი `Point2D`) (ჯერჯერობით განსხვავებას არ ვაკეთებთ წერტილს და ვექტორს შორის) მაგალითზე განვიხილავთ **C++** კლასის ძირითად კონსტრუქციებს.

კლასის კონსტრუქცია არის შემდეგი: როგორც წესი გვაქვს ორი ფაილი `'კლასის სახელი'.h` და `'კლასის სახელი'.cpp`

```
1   //ფაილი 'კლასის სახელი'.h
2   class ' კლასისსახელი '{
3   public:
4   'კლასის კონსტრუქტორი1 'კლასის სახელი'(){}'
5   'კლასის კონსტრუქტორი2 'კლასის სახელი'(parametri)'
6   'კლასის კონსტრუქტორი3 'კლასის სახელი'(parametrebi)'
7   'კლასის წევრები გარედან თავისუფალიწვდომით'
8   'ფუნქციების დეკლარაცია'
9   'კლასის დესტრუქტორი '~ კლასის სახელი(){}'
10   private:
11  ' კლასის წევრები გარედან წვდომის შეზღუდვით'
12  ' კლასის ფუნქციების დეკლარაცია გარედან წვდომის შეზღუდვით'
13   public:
14  ' რაღაც წევრები ფუნქციები'
15   private:
16  ' რაღაც წევრები ფუნქციები'
17 };//„;“— წერტილმძიმე აუცილებელია
```

განვიხილოთ Point2df.h. სწორი პრაქტიკაა, კლასის სახელი იყოს მარტივი და ასახავდეს კლასის არსს. ხშირად კლასის სახელს იწყებენ 'C' ასო ნიშნით, თუმცა ეს აუცილებელი მოთხოვნა არაა. როცა ბევრი კლასები გვაქვს ამას არავითარი უპირატესობა არ აქვს.

ჩვენს შემთხვევაში **Point** აღნიშნავს, რომ გვაქვს წერტილის კლასი; **2d**— გვაქვს 2 განზომილება და **f**— float ტიპი.

2.4 კონსტრუქტორი და დესტრუქტორი

კოდში 2.7 გამოცხადებული გვაქვს კლასი **Point2df**, რომელიც შედგება ორი **float** ტიპის ცვლადი **x** და **y** public წვდომის (ხაზი 5-8)..

ხაზზე 10-22 გამოცხადებული გვაქვს კლასის სამი კონსტრუქტორი. ანუ როგორ შეგვიძლია შევქმნათ კლასის ობიექტი. როგორც ვხედავთ ოთხი სხვადასხვა ვარიანტია:

1. ხაზი 13. ესაა ე.წ. ნაგულისხმევი (default) კონსტრუქტორი. იქმნება ობიექტი და მისი წევრები ინიციალიზდება ნულით
2. ხაზი 16. ესაა ე.წ. კოპირების კონსტრუქტორი. ობიექტი იქმნება სხვა ობიექტის კოპირების გზით. იქმნება ახალი ობიექტი და **v** ობიექტის **x** და **y** მნიშვნელობები ენიჭება ახალი ობიექტის **x** და **y** მნიშვნელობებს.
3. ხაზი 19. ობიექტი იქმნება float ტიპის მასივის საშუალებით.
4. ხაზი 22. ობიექტი იქმნება ორი float ტიპის ცვლადის საშუალებით.
5. ხაზი 65. ესაა დესტრუქტორი. როგორც ვხედავთ დესტრუქტორი ცარიელია, რაც ნიშნავს, რომ ობიექტი წაიშლება მეხსიერებიდან ისე როოგრც ეს განმარტებულია ოპერატიულ სისტემაში (ანუ უბრალოდ წაიშლება **x** და **y** ცვლადები ოპ. სისტემის მიერ). უფრო რთული კლასის შემთხვევაში, როცა მაგალითად დინამიურად ხდება ცვლადების ალოკაცია კონსტრუქტორში, დესტრუქტორში შეგვიძლია დავამატოთ დინამიურად ალოკირებული ცვლადების წაშლა.

int2df_declaration}

კოდი 2.7 კლასის დეკლარირება Point2df.h

```
1  #ifndef POINT2DF_H
2  #define POINT2DF_H
3  #include <math.h>
4
5  class  Point2df {
6  public:
7    float x;
8    float y;
9
```

```
10 public:
11
12   /// Default constructor; value is not initialized
13   Point2df() {x=0;y=0;}
14
15   /// Initialize from another vector
16   Point2df(const Point2df& v) :x(v.x), y(v.y) {}
17
18   /// Initialize from array of floats
19   Point2df(const float v[]) :x(v[0]), y(v[1]) {}
20
21   /// Initialize from components
22   Point2df(float xx, float yy) :x(xx), y(yy) {}
23
24   /// Return a non-const reference to the ith element
25   float& operator[](int i) { return (&x)[i]; }
26
27   /// Return a const reference to the ith element
28   const float& operator[](int i) const { return (&x)[i]; }
29
30   /// Assignment
31   Point2df& operator=(const Point2df& v) { x = v.x; y = v.y; return *this; }
32
33   /// Assignment from array of floats
34   Point2df& operator=(const float v[]) { x = v[0]; y = v[1]; return *this; }
35
36   /// Set value from another vector
37   Point2df& set(const Point2df& v) { x = v.x; y = v.y; return *this; }
38
39   /// Set value from array of floats
40   Point2df& set(const float v[]) { x = v[0]; y = v[1]; return *this; }
41
42   /// Set value from components
43   Point2df& set(float xx, float yy) { x = xx; y = yy; return *this; }
44
45   /// Assigning operators
46   Point2df& operator*=(float n) { return set(x*n, y*n); }
47   Point2df& operator/=(float n) { return set(x / n, y / n); }
48   Point2df& operator+=(const Point2df& v) { return set(x + v.x, y + v.y); }
49   Point2df& operator-=(const Point2df& v) { return set(x - v.x, y - v.y); }
50
51   /// Test if zero
52   bool operator!() const { return x == 0.0f && y == 0.0f; }
53
54   /// Unary
55   Point2df operator+() const { return *this; }
56   Point2df operator-() const { return Point2df(-x, -y); }
57
58   /// Length and square of length
59   float length2() const { return x*x + y*y; }
60   float length() const { return sqrt(length2()); }
61   float getX() const;
62   float getY() const;
63
64   /// Destructor
65   ~Point2df() {}
66 };
67
68
69 /// Dot product
70 inline float operator*(const Point2df& a, const Point2df& b)
71 { return a.x*b.x + a.y*b.y; }
```

```
72
73  /// Scaling
74  inline Point2df operator*(const Point2df& a, float n)
75  { return Point2df(a.x*n, a.y*n); }
76  inline Point2df operator*(float n, const Point2df& a)
77  { return Point2df(n*a.x, n*a.y); }
78  inline Point2df operator/(const Point2df& a, float n)
79  { return Point2df(a.x / n, a.y / n); }
80
81
82  /// Vector and vector addition
83  inline Point2df operator+(const Point2df& a, const Point2df& b)
84  { return Point2df(a.x + b.x, a.y + b.y); }
85  inline Point2df operator-(const Point2df& a, const Point2df& b)
86  { return Point2df(a.x - b.x, a.y - b.y); }
87
88
89
90  /// Equality tests
91  inline bool operator==(const Point2df& a, const Point2df& b)
92  { return a.x == b.x && a.y == b.y; }
93  inline bool operator!=(const Point2df& a, const Point2df& b)
94  { return a.x != b.x || a.y != b.y; }
95
96
97  /// Lowest or highest components
98  inline Point2df lo(const Point2df& a, const Point2df& b)
99  { return Point2df(fmin(a.x, b.x), fmin(a.y, b.y)); }
100 inline Point2df hi(const Point2df& a, const Point2df& b)
101 { return Point2df(fmax(a.x, b.x), fmax(a.y, b.y)); }
102
103 /// Normalize vector
104 extern  Point2df normalize(const Point2df& v);
105
106
107 /// Linearly interpolate
108 extern  Point2df lerp(const Point2df& u, const Point2df& v, float f);
109 extern Point2df perp(const Point2df &v);
110
111 #endif
```

2.5 ფუნქციის გადატვირთვა

C-ში აკრძალულია ორი ფუნქცია ერთი და იგივე სახელით. C++ შესაძლებელია რამდენიმე ფუნქცია ერთი და იგივე სახელით და ერთი და იგივე დაბრუნების ტიპით(ეს აუცილებელია), მაგრამ სხვადასხვა არგუმენტებით. ამას ეწოდება ფუნქციის გადატვირთვა.

ამის მაგალითია კოდი ხაზზე 37, 40 და 43. როგორც სხვადასხვა კონსტრუქტორის შემთხვევაში შეგვეძლო ობიექტის შექმნა სხვადასხვა გზით, ასევე შეგვიძლია სხვადასხვა გზით შევცვალოთ კლასის წევრები. ამას აკეთებს ფუნქცია `set`. კლასის წევრების მნიშვნელობების შეცვლა შეგვიძლია სხვა ობიექტის, რიცხვები მასივის ან ორი რიცხვის საშუალებით.

2.6 მიმთითებელი `this`

`this` არის მიმთითებელი რომელიც უთითებს კლასის ობიექტზე რომელშიც ვმუშაობთ, მაგალითად კლასის ფუნქციაში. ამის მაგალითია ზემოთ `set`ფუნქციის მაგალითი — შეიცვალა რა კლასის წევრების მნიშვნელობა `set` ფუნქცია აბრუნებს კლასის ობიექტის მისამართს, რომელიც შეიცვალა(რომელსაც ეკუთვნის ფუნქცია `set`).

2.7 ოპერატორები

C++ შესაძლებელია ოპერატორების განმარტება მომხმარებლის მიერ. ამას ეწოდება ოპერატორების გადატვირთვა. კოდის ხაზი [24-56] ნაწილში განმარტებულია ოპერატორები კლასის წევრებთან სამუშაოდ.

25 და 28 კვადრატული ფრჩხილები. ინდექსის(0 ან 1) მითითებით აბრუნებს x, y მნიშვნელობებს[1]. ყურადღება მიაქციეთ, რომ ინდექსის მნიშვნელობა არ მოწმდება. ანუ პროგრამისტი ვალდებულია სწორად მიუთითოს მნიშვნელობები. პირველი განმარტება საჭიროა არა მუდმივ ობიექტზე სამუშაოდ მეორე კი `const` ტიპის ობიექტზე სამუშაოდ.

31 და 34 ხაზებზე განმარტებულია მინიჭების ოპერატორი. პირველ შემთხვევაში ობიექტს ენიჭება მეორე ობიექტი. მეორე შემთხვევაში კი მასივი.

37,40,43 ხაზებზე განმარტებულია `set` რის შესახებაც ზემოთ ვისაუბრეთ

46-49 ხაზებზე განმარტებულია ობიექტზე „ოპერაცია=“ ოპერატორები ზემოთ აღწერილი `set` ფუნქციის გამოყენებით.

52 ხაზი ნულზე ტოლობის შემოწმების ოპერატორი (!)

55 და 56 ხაზი + და - ოპერატორები

59-62 განმარტებულია ფუნქციები: სიგრძის კვადრატი, სიგრძე და x და y მნიშვნელობების დაბრუნების ფუნქციები. როგორც ვხედავთ ეს ორი ფუნქცია მხოლოდ დეკლარირებულია. ცხადი სახე, ანუ იმპლემენტაცია მოცემულია კლასის შესაბამის `cpp` ფაილში.

შეგახსენებთ რომ `const` ფუნქციის სახელის შემდეგ აღნიშნავს, რომ ეს ფუნქცია არ ცვლის კლასის წევრების მნიშვნელობებს. `const` საშუალებას აძლევს კომპილატორს ოპტიმიზაცია გაუკეთოს კოდს თუ ეს შესაძლებელია.

ჩვენ ასევე შეგვეძლო ზემოთ მოყვანილი ფუნქციებისთვის და ოპერატორებისთვის კლასის `.h` ფაილში გვქონოდა მხოლოდ დეკლარაციები და იმპლემენტაციები გვქონოდა შესაბამის `cpp` ფაილში. ორივე მიდგომა თანაბარუფლებიანია და მუშაობს. კლასის შესაბამისი `cpp` ფაილის სახეს და სტრუქტურას მოგვიანებით განვიხილავთ.

65 ხაზზე გვაქვს დესტრუქტორი, რის შესახებაც ზემოთ გვქონდა საუბარი.

66 ხაზი ამთავრებს კლასის დეკლარაციას. წერტილმძიმე აუცილებელია

2.8 კლასის ობიექტებზე სამუშაო ფუნქციები

70-94 აღწერილი ფუნქციები არაა კლასის წევრები. ისინი უზრუნველყოფენ კლასის ობიექტებზე მუშაობას.

ხაზი 70: გამრავლების ოპერატორი განმარტავს ორი ვექტორის სკალარულ ნამრავლს.

74 და 76 ხაზზე აღწერილია წერტილის მარცხნიდან და მარჯვნიდან რიცხვზე გამრავლების ოპერატორი $*$.

78-ე ხაზზე წერტილის რიცხვზე გაყოფის ოპერატორი /

83 და 85 ხაზზე აღწერილია ორი ვექტორის შეკრება და გამოკლების ოპერატორები + და —

91 და 93 ხაზზე გვაქვს ოპერატორი ==, რომელიც ამოწმებს ორი ვექტორის ტოლობას და ოპერატორი !=, რომელიც ამოწმებს ორი ვექტორის არა ტოლობას.

98 და 100 ხაზზე განმარტებულია ორი ფუნქცია, რომელიც იღებს არგუმენტებად ორ ვექტორს და ამ ორი ვექტორიდან პირველ შემთხვევაში აგებს ვექტორის მინიმალური `x y` კოორდინატებით. მეორე შემთხვევაში კი მინიმალური `x y` კოორდინატებით. ეს ფუნქციები დაგვჭირდება ასეთ ამოცანაში: ვთქვათ მოცემულია რაღაც წერტილები. ვიპოვნოთ მინიმალური მართკუთხედი რომელიც მოიცავს ყველა განსახილველ წერტილს.

104 ხაზზე დეკლარირებულია ნორმალიზაციის ფუნქცია— იღებს ვექტორის მისამართს და

[1] ყურადღება მიაქციეთ რომ კლასი იწყება წევრებით x და y. ეს ნიშნავს, რომ კლასის მისამართი იგივეა და x-ის მისამართი(არ გვაქვს "ღრეჩო"კლასის ობიექტს და კლასის პირველ წევრს შორის(მსგავსი მდგომარეობა გვქონდა მასივის შემთხვევაშიც)). ეს გვაძლევს საშუალებას გვქონდეს კვადრატული ფრჩხილების ოპერატორი(ხაზი 25 და 28). კლასის წევრებზე სწრაფად მიმართვისათვის უმჯობესია კლასის წევრები განვმარტოთ კლასის დასაწყისში. რა თანმიმდევრობით გვაქვს შემოტანილი კლასის წევრები იგივე თანმიმდევრობით იქმნებიან ისინი კლასის ობიექტში მეხსიერებაში ერთმანეთის გვერდით. სადაც მთავრდება ერთი იწყება მეორე(მასივის მსგავსად).

აბრუნებს იგივე მიმართულების ერთეულოვან ვექტორს. ამ ფუნქციის იმპლემენტაცია გვაქვს მინიმალური cpp ფაილში და ქვემოთ განვიხილავთ. extern შეგვეძლო არც დაგვეწერა ვინაიდან ამ ფუნქციას ამ ფაილში არ ვიყენებთ, მაგრამ თავიდან რომ ავიცილოთ კომპილაციის თანმიმდევრობაზე და ერთი ფაილის მეორეში ჩართვაზე დამოკიდებულება გვაქვს extern, რაც კომპილატორს ეუბნება:„არ იდარდო ფუნქციის განმარტებაზე სადღაც გვაქვს განმარტებული“. თუ ფუნქცია განმარტებული არ იქნება შეცდომა გვექნება მიკავშირების (ლინკირების) ეტაპზე.

108 ხაზზე დეკლარირებულია წრფივი ინტერპოლაციის ფუნქცია.

109 ხაზზე დეკლარირებულია ფუნქცია რომელიც ითვლის და აბრუნებს მოცემული ვექტორის მართობულ ვექტორს.

extern იგივე ფუნქციას ასრულებს რასაც ზემოთ განხილულ ფუნქციებისთვის.

2.9 inline

inline ორი დატვირთვა აქვს. inline დეკლარირებული ფუნქცია ეუბნება კომპილატორს ფუნქციის ტანი ჩაეშენოს ფუნქციის კოდში გამოძახების ადგილას, როგორც ეს გვქონდა #define საშუალებით განმარტებული ფუნქციისთვის, მაგრამ #define-ისაგან განსხვავებით inline შემთხვევაში ეს კომპილატორისთვის არის რეკომენდაცია და კომპილატორი თავად წყვეტს ჩააშენოს ფუნქცია ორობით კოდში თუ არა. არის კიდევ ერთი სიტყვა __forceinline(მიკროსოფტ კომპილატორისთვის), რაც უკვე რჩევა კი არა ბრძანებაა კომპილატორისათვის ჩააშენოს ფუნქცია კოდში. შეძლებს თუ არა ამას კომპილატორი სხვა საკითხია.

todi:inlinevsdefine}

კოდი 2.8 "inline და # define "

```
1  #include <iostream>
2
3  #define mult1(a, b) a * b
4  #define mult2(a, b) (a) * (b)
5  #define mult3(a, b) ((a) * (b))
6
7  inline int multiply(int a, int b)
8  {
9    return a * b;
10 }
11
12 int main()
13 {
14   std::cout << (48 / mult1(2 + 2, 3 + 3)) << std::endl; // outputs 33
15   std::cout << (48 / mult2(2 + 2, 3 + 3)) << std::endl; // outputs 72
16   std::cout << (48 / mult3(2 + 2, 3 + 3)) << std::endl; // outputs 2
17   std::cout << (48 / multiply(2 + 2, 3 + 3)) << std::endl; // outputs 2
18
19   std::cout << mult3(2, 2.2) << std::endl; // no warning
20   std::cout << multiply(2, 2.2);
21 // Warning C4244 'argument': conversion from 'double' to 'int', possible loss of data
22  }
```

თითქოს 3,4 და 5 ხაზზე განმარტებული მაკროსები ერთი და იგივეა, მაგრამ სწორი და უსაფრთხო არის მე-5 ხაზზე მოცემული განმარტება. ამიტომ უნდა ვერიდოთ #define და გამოვიყენოთ inline ან __forceinline.

შენიშვნა
gcc კომპილატორისთვის __forceinline ეკვივალენტი არის #define __forceinline __attribute__((always_inline))

inline მეორე დანიშნულება:ზემოთ მოყვანილ კლასის ფაილში inline საჭიროა იმიტომ, რომ ფუნქციების(რომლებიც არ არიან კლასის წევრები) დეკლარაცია და იმპლემენტაცია არის ერთ .h ფაილში. როცა ამ ფაილის რამდენჯერმე ჩართვა მოხდება პროექტის სხვადასხვა ფაილებ-

ში, ბინარული პროგრამული კოდის მიკავშირების(ლინკირების) ეტაპზე გვექნება შეცდომა ერთი და იგივე სიმბოლოების(ჩვენს შემთხვევაში ფუნქციების) მრავალჯერადად განმარტების შესახებ(მსგავსი პრობლემა გვქონდა როცა არ გვქონდა #define-ით განმარტებული მაკროსები .h ფაილში). inline უზრუნველყოფს, რომ ფუნქცია საბოლოო კოდში ჩართული იქნება მხოლოდ ერთხელ.

inline არ იქნებოდა საჭირო .h ფაილში რომ გვქონდეს მხოლოდ დეკლარაციები და იმპლემენტაციები გვქონდეს .cpp ფაილში. უფრო მეტიც კლასის ეს დამხმარე ფუნქციები დეკლარაციით და იმპლემენტაციით შეგვეძლო გაგვეკეთებინა არა კლასის ფაილებში არამედ სხვა .h და .cpp ფაილებში. ამ შემთხვევაში კლასის ფაილის ჩართვასთან ერთად ჩართვა უნდა გავუკეთოთ ამ დამხმარე ფუნქციების .h ფაილსაც.

2.10 კლასის იმპლემენტაცია Point2df.cpp

კლასის იმპლემენტაციის ფაილს აქვს ასეთი სახე

```
1   //ფაილი 'კლასის სახელი'.cpp
2   #include "კლასისსახელი.h"
3   'ტიპი' ' კლასისსახელი '::  ფუნქცია1(არგუმენტები){
4     ფუნქციის აღწერა
5     return 'ტიპი';
6   }
7   'ტიპი' ' კლასისსახელი '::  ფუნქცია2(არგუმენტები){
8   ფუნქციის აღწერა
9   return 'ტიპი';
10  }
11  ......
```

ცხადია ფუნქცია შეიძლება არაფერს არ აბრუნებდეს. ფუნქციის ქვეშ შეგვიძლია ვიგულისხმოთ კლასის კონსტრუქტორი და დესტრუქტორიც.

Point2df კლასის შემთხვევაში იმპლემენტაციის ფაილი მოყვანილია ქვემოთ 2.9.

{Point2df_cpp}

კოდი 2.9 კლასის იმპლემენტაცია Point2df.cpp

```
1   #include "Point2df.h"
2   float Point2df::getX() const{
3       return x;
4   }
5   float Point2df::getY() const{
6       return y;
7   }
8   //utility functions
9   // Normalize vector
10  Point2df normalize(const Point2df& v) {
11    const double m = v.length2();
12    Point2df result(v);
13    if (0.0 < m) { result /=sqrt(m); }
14    return result;
15  }
16  Point2df perp(const Point2df &v){
17    Point2df result(-v.y, v.x);
18    return result;
19  }
20  // Linearly interpolate
21  Point2df lerp(const Point2df& u, const Point2df& v, float f) {
22    return Point2df(u.x + (v.x - u.x)*f, u.y + (v.y - u.y)*f);
23  }
```

როგორც ვხედავთ აქ განმარტებულია როგორც კლასის შიგნით დეკლარირებული ფუნქციები ასევე დამხმარე ფუნქციები რომლებიც დეკლარირებული იყო კლასის ფაილში.

2.11 Point2df კლასის გამოყენების მაგალითი

di:testPoint2D_cpp}

კლასის გამოყენების მაგალითი იხილეთ *Point2df* არქივში

კოდი 2.10 Point2df კლასის გამოყენების მაგალითი

```
1  #include "Point2df.h"
2  #include "Triangle.h"
3  #include <stdio.h>
4
5  int main()
6  {
7    Point2df A,B,C;
8    Point2df D(10, 10);
9    B.set(3,4);
10   A.set(1, 1);
11   Point2df E(A);
12   E =A+B+C;
13   printf("\n B.x=%f, B.y=%f\n", B.x, B.y);
14   Point2df F(A);
15   float lsq=B.length2();
16   printf("B.length2()=%f\n",lsq);
17   F += D;
18   F = F + D;
19   Triangle tria(A,B,C);
20 return 0;
21 }
```

არქივში ასევე არის სამკუთხედის Triangle კლასი.

{hw:Triangle}

დავალება №2.1 სამკუთხედის კლასი

სამუთხედის კლასისთვის დაამატეთ:

1. მინიჭების ოპერატორი
2. ტოლობაზე შედარების ოპერატორი
3. არტოლობაზე შედარების ოპერატორი
4. კლასში დაამატეთ private წევრები პერიმეტრი და ფართობი.
5. პერიმეტრის გამოთვლის ფუნქცია.
6. ფართობის გამოთვლის ფუნქცია.
7. getPerimeter და getArea ფუნქციები რომელიც დააბრუნებს პერიმეტრს და ფართობს თუ ისინი გამოთვლილია და თუ გამოთვლილი არაა გამოთვლის და შემდეგ დააბრუნებს მიღებულ მნიშვნელობას.
8. set და get ფუნქციები(set ღებულობს სამ წერტილს და ანიჭებს სამკუთხედის წევრ წერტილებს. get მიეწოდება სამი წერტილის მიმთითებლები და ამ წერტილებში კოპირდება სამკუთხედის შემადგენელი წერტილების მონაცემები)
9. ფუნქცია, რომელიც ეკრანზე ბეჭდავს სამკუთხედის წერტილების კოორდინატებს.
10. ფუნქცია, რომელიც ეკრანზე ბეჭდავს სამკუთხედის ფართობს და პერიმეტრს.

2.12 friend ფუნქციები

განხილულ კოდში x y ცვლადები დეკლარირებულია როგორც public. შეცვალეთ კოდში დეკლარაცია private-ით და ნახავთ რომ კოდი კომპილაციისას მოგვცემს შეცდომას. ეს იმიტომ, რომ ობიექტებთან სამუშაო ფუნქციებს, რომლებიც არ არიან კლასის წევრები, აკრძალული აქვთ x და y წევრებზე წვდომა. იმისათვის, რომ ასეთ შემთხვევებში კოდმა იმუშაოს საჭიროა ეს ოპერატორები და ფუნქციები გამოვაცხადოთ კლასის „მეგობრებად“. იხილეთ არქივი Point2D_private. ამ არქივიდან კოდის ნაწილი Point2D.h ფაილიდან მოყვანილია ქვემოთ პროგრამის კოდში 2.11.

კლასის ობიექტებთან სამუშაო ოპერატორები და ფუნქციები გადატანილია კლასის შიგნით და დეკლარაციაში დამატებულია სიტყვა friend. არ არის აუცილებელი ოპერატორის/ფუნქციის

იმპლემენტაცია იყოს კლასის შიგნით. მხოლოდ დეკლარაცია საკმარისია, ხოლო იმპლემენტაცია შიძლება იყოს სხვა ფაილში, მაგალითად კლასის შესაბამის `.cpp` ფაილში, მაგრამ ცხადია ეს ოპერატორი ფუნქციები არ არიან კლასის წევრები.

{:2D_private_h}

კოდი 2.11 კლასი `private` წევრებით

```
65  /// Dot product
66  inline friend float operator*(const Point2df& a, const Point2df& b) {
67    return a.x*b.x + a.y*b.y; }
68
69  /// Scaling
70  inline friend Point2df operator*(const Point2df& a, float n) {
71    return Point2df(a.x*n, a.y*n); }
72  inline friend Point2df operator*(float n, const Point2df& a) {
73    return Point2df(n*a.x, n*a.y); }
74  inline friend Point2df operator/(const Point2df& a, float n) {
75    return Point2df(a.x / n, a.y / n); }
76
77  /// Vector and vector addition
78  inline friend Point2df operator+(const Point2df& a, const Point2df& b) {
79    return Point2df(a.x + b.x, a.y + b.y); }
80  inline friend Point2df operator-(const Point2df& a, const Point2df& b) {
81    return Point2df(a.x - b.x, a.y - b.y); }
82
83  // Equality tests
84  inline friend bool operator==(const Point2df& a, const Point2df& b) {
85    return a.x == b.x && a.y == b.y; }
86  inline friend bool operator!=(const Point2df& a, const Point2df& b) {
87    return a.x != b.x || a.y != b.y; }
88
89  // Lowest or highest components
90  inline friend Point2df lo(const Point2df& a, const Point2df& b) {
91    return Point2df(fmin(a.x, b.x), fmin(a.y, b.y)); }
92  inline friend Point2df hi(const Point2df& a, const Point2df& b) {
93    return Point2df(fmax(a.x, b.x), fmax(a.y, b.y)); }
94
95  /// Normalize vector
96  friend  Point2df normalize(const Point2df& v);
97
98  /// Linearly interpolate
99  friend  Point2df lerp(const Point2df& u, const Point2df& v, float f);
100 friend Point2df perp(const Point2df &v);
```

2.13 წინასწარი დეკლარირება და ორი კლასის მეგობარი ფუნქცია

შესაძლებელია რომ ერთ კლასს სამუშაოდ სჭირდებოდეს მეორე კლასი ან პირიქით მეორეს პირველი. ამ შემთხვევაში თუ საჭირო კლასის ობიექტი უკვე დაკომპილირებული არ იქნა მივიღებთ შეცდომას. წინასწარი დეკლარირება მსგავსია `extern` გამპცხადების, როცა კომპილატორს ვეუბნებით რომ არ იფიქროს ფუნქციის იმპლემენტაციაზე.

განვიხილოთ მაგალითი

```
1   #include <iostream>
2   using namespace std;
3
4   // Add members of two different classes using friend functions
5   // forward declaration
6   class ClassB;
7   class ClassA {
8     public:
9     // constructor to initialize numA to 12
10    ClassA() : numA(12) {}
```

```
11
12      private:
13      int numA;
14
15      // friend function declaration
16      friend int add(ClassA, ClassB);
17    };
18
19    class ClassB {
20
21      public:
22      // constructor to initialize numB to 1
23      ClassB() : numB(1) {}
24
25      private:
26      int numB;
27
28      // friend function declaration
29      friend int add(ClassA, ClassB);
30    };
31    // access members of both classes
32    int add(ClassA objectA, ClassB objectB) {
33      return (objectA.numA + objectB.numB);
34    }
35
36    int main() {
37      ClassA objectA;
38      ClassB objectB;
39     "Sum: " << add(objectA, objectB);
40      return 0;
41    }
```

2.14 friend კლასი

კლასის „მეგობარი“ შეიძლება იყოს არა მხოლოდ გლობალური ფუნქცია არამედ სხვა კლასიც

```
1   #include <iostream>
2   using namespace std;
3   class A {
4     private:
5     int private_variable;
6     public:
7     A()
8     {
9       private_variable = 10;
10    }
11    // friend class declaration
12    friend class C;
13  };
14  // Here, class C is declared as a
15  // friend inside class A. Therefore,
16  // C is a friend of class A. Class C
17  // can access the private members of
18  // class A.
19  class C {
20    public:
21    void display(A& objectA)
22    {
23      cout << "The value of Private Variable = "
24      << objectA.private_variable << endl;
25    }
```

```
26  };
27  // Driver code
28  int main()
29  {
30    A g;
31    C fri;
32    fri.display(g);
33    return 0;
34  }
```

2.15 friend კლასის ფუნქცია

მეგობრად ასევე შეიძლება გამოცხადდეს არა კლასი არამედ კლასის ფუნქციაც

```
1   #include <iostream>
2   using namespace std;
3   class A; // forward definition needed
4   // another class in which function is declared
5   class anotherClass {
6     public:
7     void memberFunction(A& obj);
8   };
9
10  // base class for which friend is declared
11  class A {
12    private:
13    int private_variable;
14
15    public:
16    A()
17    {
18      private_variable = 10;
19    }
20
21    // friend function declaration
22    friend void anotherClass::memberFunction(A&);
23  };
24
25  // friend function definition
26  void anotherClass::memberFunction(A& obj)
27  {
28    cout << "Private Variable: " << obj.private_variable
29    << endl;
30  }
31
32  // driver code
33  int main()
34  {
35    A object1;
36    anotherClass object2;
37    object2.memberFunction(object1);
38    return 0;
39  }
```

2.16 namespace

ზემოთ კოდებში რამდენჯერმე შეგვხვდა `using namespace std;` C++ შემოტანილია ე.წ. namespace ცნება. ესაა არე რომლის შიგნითაც შეიძლება განმარტებულ იქნას კლასები, ცვლადები, ფუნქციები.

namespace უზრუნველყოფს რომ როცა ვიყენებთ რამდენიმე ბიბლიოთეკას გამოვიყენოთ მოცემული ფუნქცია/კლასი მოცემული ბიბლიოთეკიდან. ანუ შეიძლება გვქონდეს სხვადასხვა კლასე-

ბი ფუნქციები ერთი და იგივე სახელით, მაგრამ თუ ისინი განმარტებულია სხვადასხვა namespace-ში გამოყენება არ იქნება პრობლემა.

განვიხილოთ მაგალითი:

```
namespace NameSpaceA
{
  class A
  {
    public:
    void DoSomething() {}
  };
  void Func(A &objectA) {}
}
```

განმარტებული გვაქვს namespace NameSpaceA და მის შიგნით რაღაც კლასი და ფუნქცია. ამ კლასზე და ფუნქციაზე მუშაობა შეიძლება რამდენიმე გზით, მაგალითად ვუთითებთ namespace და კლასს/ფუნქციას

```
NameSpaceA::A aklasisobieqti;
aklasisobieqti.DoSomething();
NameSpaceA::Func(aklasisobieqti);
```

ან სიტყვა using გამოყენებით

```
using NameSpaceA::A;
A mgr;
mgr.DoSomething();
```

ან ვაცხადებთ namespace NameSpaceA გლობალურად

```
using namespace NameSpaceA;
A mgr;
mgr.DoSomething();
Func(mgr);
```

ეს არაა სასურველი როცა რამდენიმე namespace გვაქვს. ერთი შეიძლება გამოვაცხადოთ გლობალურად და სხვა namespace-ების ელემენტებს, ფუნქციებს კლასებს მივმართოთ „ :: “ საშუალებით.

2.17 std::vector

C++-ში გვაქვს ე.წ. სტანდარტული ბიბლიოთეკა. namespace std სწორედ სტანდარტული ბიბლიოთეკის „ნეიმსპეისია“. განვიხილოთ სტანდარტული ბიბლიოთეკის (***Standard Template Library (STL)***) კონტეინერ კლასი ვექტორი (vector.)

vector არის დინამიური მასივი რომელშიც შეიძლება გვქონდეს ერთი ტიპის ელემენტები. მას შეუძლია ავტომატურად შეიცვალოს ზომა როცა ხდება ელემენტის დამატება ან ელემენტის წაშლა.

vector არის შაბლონური კლასი, რაც ნიშნავს რომ ტიპი არის პარამეტრი და რა ტიპზეა საუბარი ეს მომხმარებელმა უნდა დააკონკრეტოს. vector<T> v; T არის პარამეტრი. შეიძლება იყოს ნებისმიერი ტიპი C++ ენაში არსებული ტიპი(მათ შორის მიმთითებელიც) ან მომხმარებლის მიერ განმარტებული კლასი(მიმთითებელი კლასზე).

განვიხილოთ მაგალითი:

```
#include <vector>
#include <iostream>

int main() {
  // ინიციალიზაციის მაგალითები
```

```
  std::vector<int> numbers = {1, 2, 3, 4, 5}; // ცხადი სახით ინიციალიზაცია
  int firstNumber = numbers[0]; // firstNumber will be 1
  std::cout << "The first number is: " << firstNumber << std::endl;

   int arr[] = {11, 23, 45, 89};
  int n = sizeof(arr) / sizeof(arr[0]);

  // Initialize the std::vector v by arr
  vector<int> v = {arr, arr + n};// მასივით ინიციალიზაცია

   vector<int> v1 = {11, 23, 45, 89};

  // Initialize the vector v2 from vector v1
  // სხვა ვექტორით ინიციალიზაცია
  vector<int> v2(v1.begin(), v1.end());

   // Initializing all the elements of a
  // vector using a single value
  //  ინიციალიზაცია კონსტანტით.
   // იქმნება 5 ელემენტიანი მასივი
   // ყველა ელემენტი არის 11
  vector<int> v3(5, 11);
   // იქმნება მასივი ზომით 0
    vector<int> v4;
  return 0;
}
```

ვექტორზე მუშაობა:

```
  #include <vector>
  #include <iostream>

  int main() {
    std::vector<int> numbers = {1, 2, 3, 4, 5};
    numbers[1] = 20; // The second element of the vector will now be 20
    std::cout << "The modified second number is: " << numbers[1] << std::endl;
    numbers.push_back(4); // Adds the number 4 to the end of the vector
    numbers.pop_back(); // Removes the last element (4)
    return 0;
  }
```

2.17.1 იტერატორი

იტერატორი C++-ში არის მიმთითებლის მსგავსი ობიექტი, რომელიც მიუთითებს STL კონტეინერის ელემენტზე. ისინი ძირითადად გამოიყენება C++-ში STL კონტეინერის ელემენტებთან სამუშაოდ. STL იტერატორების მთავარი უპირატესობა ის არის, იტერატორების გამოყენებით STL ალგორითმები დამოუკიდებლი ხდება გამოყენებული კონტეინერის ტიპისგან. ჩვენ შეგვიძლია უბრალოდ მივაწოდოთ კონტეინერის იტერატორი ალგორითმს და არა თავად კონტეინერი ან მისი ელემენტი.

იტერატორის ფუმქცია	დაბრუნებული მნიშვნელობა
begin()	აბრუნებს იტერატორს რომელიც უთითებს კონტეინერის დასაწყისზე.
end()	აბრუნებს იტერატორს კონტეინერის ბოლო ელემენტის შემდგომ ელემენტზე. ესაა სპეც. ელემენტი კონტეინერებში რომელიც უთითებს კონტეინერის დასასრულს.

cbegin()	იგვეა რაც begin() ოღნოდ აბრუნებს მუდმივ იტერატორს(ანუ შეუძლებელია ელემენტის მნიშვნელობის შეცვლა, რომელზეც უთითებს იტერატორი).
cend()	იგივეა რაც end() ოღონდ აბრუნებს მუდმივ იტერატორს.
rbegin()	ესაა შებრუნებული იტერატორი რომელიც უთითებს დასაწყისს(ანუ ბოლოს).
rend()	ესაა შებრუნებული იტერატორი რომელიც უთითებს კონტეინერის დასასრულს.
crbegin()	ესაა შებრუნებული მუდმივი იტერატორი რომელიც უთითებს დასაწყისს.
crend()	ესაა შებრუნებული მუდმივი იტერატორი რომელიც უთითებს კონტეინერის დასასრულს.

იტერატორის მაგალითი:

```
  // Create a vector called cars that will store strings
  vector<string> cars = {"Volvo", "BMW", "Ford", "Mazda"};

  // Create a vector iterator called it
  vector<string>::iterator it;

  // Loop through the vector with the iterator
  for (it = cars.begin(); it != cars.end(); ++it) {
    cout << *it << "\n";
  }
```

2.18 std::vector და Point2df მაგალითი

იხ. არქივი *randomPoints*.

utility.h ფაილში დეკლარირებულია int random_number(int low_num, int hi_num) ფუნქცია, რომელიც აგენერირებს შემთხვევით რიცხვს მოცემულ შუალედში. იმპლემენტაცია მოყვანილია ქვემოთ.

Points_utility_cpp}

კოდი 2.12 random_number ფუნქციის იმპლემენტაცია ფაილი utility.cpp

```
#include "utility.h"
#include <stdlib.h>
#include<time.h>
#include<stdio.h>
int random_number(int low_num, int hi_num)
{
  int result = 0;
  result = (rand() % (hi_num+1 - low_num)) + low_num;
  return result;
}
```

randomPoints.cpp კოდი მოყვანილია ქვემოთ. აქ დემონსტრირებულია წერილის კლასის ფუნქციები lo და hi.

i:randomPoints_cpp}

კოდი 2.13 ფაილი randomPoints.cpp

```
#include "Point2df.h"
#include "utility.h"
#include <stdlib.h>
#include<time.h>
```

```
5  #include <stdio.h>
6  #include <vector>
7  using namespace std;
8  vector<Point2df> m_points;
9  int main()
10 {
11   srand(time(NULL));
12   int low = -50;
13   int high = 50;
14   for (int i=low;i<high;i++)
15   {
16     int ival_x = random_number(low, high);
17     int ival_y = random_number(low, high);
18     Point2df rpoint((float)ival_x, (float)ival_y);
19     m_points.push_back(rpoint);
20     printf("p.x=%f,p.y=%f\n", rpoint.x, rpoint.y);
21   }
22   Point2df left(m_points[0]);
23   Point2df rigth(m_points[0]);
24   for (size_t i = 1; i < m_points.size(); i++)
25   {
26     left = lo(left, m_points[i]);
27     rigth = hi(rigth, m_points[i]);
28   }
29   printf("left.x=%f,left.y=%f\n", left.x, left.y);
30   printf("rigth.x=%f,rigth.y=%f\n", rigth.x, rigth.y);
31     return 0;
32 }
```

ხაზზე 14-21 გენერირდება შემთხვევითი წერტილები და ემატება `m_points` მასივს.

24-28 ხაზზე ციკლში გავდივართ მასივს და მასივის წერილებიდან ვაგებთ `left`(მარცხენა დაბალი წერტილი) და `right`(მარჯვენა მაღალი წერტილი). ეს ორი წერილი გვაძლევს ისეთი მინიმალური მართკუთხედის ორ მოპირდაპირე წვეროს, რომელიც მოიცავს ყველა წერტილს.

2.19 ნაწარმოები კლასი (შთამომავლები)

C++ შეგვიძლია შევქმნათ კლასი(ეს იქნება ნაწარმოები კლასი) სხვა კლასის(ეწოდება საბაზო,აბსტრაქტული კლასი, წინაპარი კლასი) ან კლასების საფუძველზე(ანუ ნაწარმოები იყოს ორი ან რამდენიმე კლასისაგან). რის შედეგადაც ნაწარმოებ კლასს ექნება ყველა ის თვისება რაც აქვს საბაზო კლასს(კლასებს) და აუცილებლობა აღარაა განმარტებულ იქნას ეს თვისებები ნაწარმოები კლასისთვის.

ხშირად განიხილავენ ასეთ მაგალითს. ვთქვათ შევქმენით კლასი „ცხოველი“. ამ კლასში გვექნება თვისებები რაც ახასიათებს ზოგადად „ცხოველს“. „ცხოველი“ კლასიდან შეგვიძლია ვაწარმოოთ სხვადასხვა კლასები. მაგალითად, კლასი „ძაღლი“, „კატა“, „მგელი“, „ცხვარი“. თითოეულს ექნება ყველა თვისება და ფუნქცია რაც აქვს კლასს „ცხოველს“. ამას გარდა თითოეულ კლასს შეგვიძლია დავუმატოთ ამ კლასის ცხოველის დამახასიათებელი თვისებები და ფუნქციები (მაგალითად, შეგვიძლია გვქონდეს ფუნქცია „ხმა“ საბაზისო კლასში, რომელიც სხვადასხვა ცხოველის შემთხვევაში გახსნის და გაუშვებს(დაუკრავს) ცხოველის შესაბამის ხმის ფაილს).

შეგვიძლია ავიღოთ უფრო ზოგადი სქემა, მაგალითად: „სულიერი არსება“, „ფრინველი“ და „ცხოველი“ შეგვიძლია ვაწარმოოთ ამ კლასიდან. შემდეგ „ფრინველი“ კლასიდან ვაწარმოოთ სხვადასხვა ფრინველის შესაბამისი კლასები და „ცხოველი“ კლასიდან ვაწარმოოთ ცხოველის შესაბამისად სხვადასხვა კლასები. ასე რომ კლასი „ძაღლი“ პირველ შემთხვევაში თუ იყო ნაწარმოები „ცხოველიდან“, მეორე შემთხვევაში „ძაღლი“ ნაწარმოები იქნება „ცხოველიდან“, რომელიც თავის მხრივ ნაწარმოებია „სულიერი არსებიდან“.

ამოცანიდან გამომდინარე შეიძლება გვქონდეს აბსტრაქციის სხვადასხვა დონე. სანამ კლასებს გამოვაცხადებთ უნდა გავიაზროთ ამოცანა ზოგადად. რა ობიექტები გვექნება. რა ურთიერ-

თობაა ობიექტებს შორის. რა თვისებები შეიძლება განვაზოგადოთ და ა.შ. განზოგადების რაღაც დონე აუცილებელი იქნება მაგრამ ეს უნდა იყოს მინიმალური. როგორც წესი ერთი ან ორი თაობა (ერთი ან ორი „წინაპარი“) საკმარისია.

2.20 protected წევრები

{kodi:Solid2D_h}

კოდი 2.14 ფაილი Solid2D.h

```
1  #ifndef _SOLID2D_H
2  #define _SOLID2D_H
3  #include "Point2df.h"
4  #include <vector>
5  #include <string>
6  #define M_PI 3.14159265358979323846
7
8  class Solid2D
9  {
10 protected:
11   Point2df m_Position;
12   std::vector<Point2df>m_points;
13   std::string m_name;
14 public:
15   Solid2D():m_Position(0,0),m_name("Unnamed Solid") {};
16   Solid2D(const Point2df& pos) { m_Position = pos; };
17   Solid2D(const Solid2D& s)
18   {
19       m_Position = s.m_Position;
20       m_points = s.m_points;
21       m_name=s.m_name;
22       m_name.append("_copy");
23     };
24   Solid2D(float x, float y):m_name("Unnamed Solid") {
25       m_Position.x = x; m_Position.y = y;}
26   void setPosition(const Point2df& pos);
27   void setPosition(float x, float y) { m_Position.x = x; m_Position.y = y; }
28   const Point2df& getPosition() const;
29   std::string  getName()const{return m_name;}
30   void setName(std::string name){m_name=name;}
31   void setName(const char* name)
32   {
33       m_name.assign(name);
34     }
35   virtual float Perimeter() const;
36   virtual float Area() const = 0 ;
37
38    ~Solid2D() {};
39 };
40 #endif
```

განვიხილოთ პროექტი derivedClass

:Solid2D_cpp}

კოდი 2.15 ფაილი Solid2D.cpp

```
#include "Solid2D.h"
void Solid2D::setPosition(const Point2df& pos)
{
  m_Position = pos;
}
const Point2df& Solid2D::getPosition() const
{
  return m_Position;
}

float Solid2D::Perimeter() const
{
   const int n = m_points.size();
   const Point2df diff = m_points[n - 1] - m_points[0];
   float perimeter = diff.length();
   for (int i=0;i<n-1;i++)
   {
     const float dist = (m_points[i] - m_points[i + 1]).length();
     perimeter += dist;
   }
  return perimeter;
}
```

წრფივი ალგებრის ელემენტები

{refsection:3}

3.1 მატრიცა

რიცხვების ერთობლიობას $\mathcal{K}$ ველიდან (როგორც წესი ეს შეიძლება იყოს ნამდვილი ან კომპლექსური რიცხვების ველი) წარმოდგენილს მართკუთხა ცხრილის სახით m სტრიქონით და n სვეტით, ეწოდება $m \times n$ მატრიცა და აღვნიშნავთ როგორც

$$\mathbf{A}^{m\times n} = \begin{array}{c} \begin{matrix} \text{სვ.} & 1 & \cdots & k & \cdots & n \end{matrix} \\ \begin{pmatrix} a^1{}_1 & \cdots & a^1{}_k & \cdots & a^1{}_n \\ a^2{}_1 & \cdots & a^2{}_k & \cdots & a^2{}_n \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i{}_1 & \cdots & a^i{}_1 & \cdots & a^i{}_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^m{}_1 & \cdots & a^m{}_k & \cdots & a^m{}_n \end{pmatrix} \end{array} \begin{matrix} \text{სტრ.} \\ 1 \\ 2 \\ \vdots \\ i \\ \vdots \\ m \end{matrix} \tag{3.1}$$

ანუ უფრო მარტივად, მატრიცას ავღნიშნავთ მუქი დიდი ასო ნიშნით($\mathbf{A}$), ხოლო მატრიცის ელემენტს — a^i_j სადაც $i = \overline{1,m}$,$j = \overline{1,n}$.

ხშირად ზედა ინდექსს წერენ ქვემოთ და გვაქვს a_{ij}. ამ შემთხვევაში პირველი ინდექსი ნომრავს სტრიქონს, ხოლო მეორე სვეტს. ტენზორულ აღრიცხვაში მრავალი ინდექსის შემთხვევაში თანმიმდევრობის სწორად აღსანიშნავად იყენებენ ცარიელ ადგილს $a^i{}_j$ ან წერტილს $a^i{}_{\cdot j}$.

„$\mathbf{A}$ მატრიცის ელემენტი i,j“ ავღნიშნავთ როგორც $\{\mathbf{A}\}^i_j = a^i_j$.

„$\mathbf{A}$ მატრიცა ელემენტებით a^i_j “ ავღნიშნავთ როგორც $[a^i_j]$.

მატრიცას $\mathbf{B}^{m\times n} = [b^i_j]$ ეწოდება $\mathbf{A}^{n\times m} = [a^j_i]$ მატრიცის ტრანსპორირებული თუ $a^j_i = b^i_j$; $i = \overline{1,m}$,$j = \overline{1,n}$. ტრანსპორირებული მატრიცის აღსანიშნავად გამოვიყენებთ აღნიშვნას $\mathbf{A}^{\mathrm{T}}$,$\mathbf{A}^{\mathrm{t}}$.

ორი მატრიცისთვის გამრავლების ოპერაცია განმარტებულია, როცა პირველი მატრიცის სვეტების რაოდენობა მეორე მატრიცის სტრიქონების რაოდენობის ტოლია:

$$\mathbf{A}^{m\times p} \cdot \mathbf{B}^{p\times n} = \mathbf{C}^{m\times n}$$

და ნიშნავს შემდეგს:

{eq:appendrow}

$$\begin{pmatrix} a^1_1 & \cdots & a^1_2 & \cdots & a^1_p \\ a^2_1 & \cdots & a^2_2 & \cdots & a^2_p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i_1 & \cdots & a^i_k & \cdots & a^i_p \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a^m_1 & \cdots & a^m_k & \cdots & a^m_p \end{pmatrix} \cdot \begin{pmatrix} b^1_1 & \cdots & b^1_j & \cdots & b^1_n \\ b^2_1 & \cdots & b^2_j & \cdots & b^2_n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ b^k_1 & \cdots & b^k_j & \cdots & b^k_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b^p_1 & \cdots & b^p_j & \cdots & b^p_n \end{pmatrix} = \begin{pmatrix} c^1_1 & \cdots & c^1_j & \cdots & c^1_n \\ c^2_1 & \cdots & c^2_j & \cdots & c^2_n \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ c^i_1 & \cdots & c^i_j & \cdots & c^i_n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c^m_1 & \cdots & c^m_j & \cdots & c^m_n \end{pmatrix} \tag{3.2}$$

სადაც მიღებული მატრიცის ელემენტები გამოითვლება როგორც

{eq:matX}

$$c^i_j = \sum_{k=1}^{p} a^i_k \cdot b^k_j \tag{3.3}$$

მატრიცული გამრავლების აღნიშვნაში „$\cdot$“ ნიშანს გამოვტოვებთ და ვიგულისხმებთ რომ ჩანაწერი $\mathbf{AB}$ აღნიშნავს $\mathbf{A}\cdot\mathbf{B}$— ზემოთ განმარტებულ მატრიცულ ნამრავლს. ზოგიერთ ლიტერატურაში მატრიცული ნამრავლისთვის გამოიყენებენ ,,$\times$“ ნიშანს.

$i = j$ პოზიციების სიმრავლეს მატრიცაში ეწოდება მთავარი დიაგონალი. კვადრატულ მატრიცას(სვეტების და სტრიქონების რაოდენობა ტოლია) მთავარ დიაგონალზე ერთიანებით და სხვა ელემენტებით ნული, ეწოდება ერთეულოვანი მატრიცა და აღინიშნება როგორც $\mathbf{E}$ ან $\mathbf{I}$

{eq:}
$$\mathbf{I} \equiv \mathbf{E} = \mathbf{E}_n = \begin{pmatrix} 1 & 0 & \cdots 0 \\ 0 & 1 & \cdots 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & \cdots 1 \end{pmatrix} = [\delta^i_j]_{n\times n} = [\delta^i_j] \tag{3.4}$$

δ^i_j კრონეკერის სიმბოლო ეწოდება. სხვადასხვა შემთხვევებში ასევე შეგვიძლია შეგვხვდეს როგორც δ_{ij} ან δ^{ij}. δ^i_j გააჩნია ე.წ. ინდექსის ფილტრაციის თვისება $a_j = \sum_{i=1}^n \delta^i_j a_i$, $a^i = \sum_{j=1}^n \delta^{ij} a^j$ და $a_i = \sum_{j=1}^n \delta_{ij} a_j$ ანუ კრონეკერის სიმბოლოზე გამრავლება ერთი ინდექსის ჯამით ჯამის ინდექსს ცვლის კრონეკერის სიმბოლოს მეორე, თავისუფალი ინდექსით.

ინდექსს რის მიხედვითაც აიღება ჯამი მუნჯი ინდექსი ეწოდება. ხშირად ჯამის ნიშანს არ წერენ და განმეორებითი ინდექსის შემთხვევაში იგულისხმება ჯამი (აინშტაინის შეთანხმება). ანუ ზემოთ მოყვანილი ტოლობების შეგვიძლია ჩავწეროთ ჯამის ნიშნის გარეშე: $a_i = \delta_{ij} a_j$. განტოლება (3.3) შეგვიძლია დავწეროთ როგორც

{eq:matXnosum}
$$c^i_j = a^i_k \cdot b^k_j, \quad k = \overline{1,p} \text{ განმეორებითი ინდექსით იგულისხმება ჯამი}. \tag{3.5}$$

თუ $\mathbf{A}^{\mathrm{T}} = \mathbf{A}$, $\mathbf{A}$-ს სიმეტრიული მატრიცა ეწოდება, ხოლო როცა $\mathbf{A}^{\mathrm{T}} = -\mathbf{A}$ — ანტისიმეტრიული.

მატრიცების გამრავლების ოპერაციის თვისებები:

1. $\mathbf{A}(\mathbf{BC}) = (\mathbf{AB})\mathbf{C}$ ასოციატურობა
2. $(\mathbf{A}+\mathbf{B})\mathbf{C} = \mathbf{AC} + \mathbf{BC}$ და $\mathbf{A}(\mathbf{B}+\mathbf{C}) = \mathbf{AB} + \mathbf{AC}$
3. ნებისმიერი $x \in \mathcal{K}$ გვაქვს $x(\mathbf{AB}) = (x\mathbf{A})\mathbf{B} = \mathbf{A}(x\mathbf{B})$
4. $(\mathbf{AB})^{\mathrm{T}} = \mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$
5. $(\mathbf{A}^{\mathrm{T}})^{-1} = (\mathbf{A}^{-1})^{\mathrm{T}}$

3.2 ვექტორ სვეტი და ვექტორ სტრიქონი

ერთ სვეტიანი მატრიცის შემთხვევაში გამოვიყენებთ აღნიშვნას

$$\mathbf{A}^{m\times 1} \equiv \vec{\boldsymbol{a}} = \begin{pmatrix} a^1 \\ a^2 \\ \vdots \\ a^m \end{pmatrix} = [a^i]^m = [a^i] \tag{3.6}$$

რაც ნიშნავს, რომ ვექტორი მატრიცულ აღნიშვნებში წარმოადგენენ სვეტ მატრიცას. $\vec{\boldsymbol{a}}$ ვექტორში ვგულისხმობთ ფორმალურ ვექტორს — რაღაც ობიექტები რომლებიც შეგვიძლია ჩავწეროთ (გავაერთიანოთ) ერთ სვეტში. ეს შეიძლება იყოს რიცხვები, გეომეტრიული ვექტორის კომპონენტები, ფუნქციები, ოპერატორები (მაგ. სხვადასხვა რიგის წარმოებულები რაიმე ცვლადით)

ერთ სტრიქონიანი მატრიცის შემთხვეაში გამოვიყენებთ აღნიშვნას

$$\mathbf{A}^{1\times n} \equiv (\mathbf{A}^{n\times 1})^{\mathrm{T}} \equiv \vec{\boldsymbol{a}}^{\mathrm{T}} = \begin{pmatrix} a_1 a_2 \cdots a_n \end{pmatrix} = [a^{\mathrm{t}}_i]_n = [a^{\mathrm{t}}_i] \tag{3.7}$$

ტრანსპორირების ოპერაციის გამოყენებით შეგვიძლია სტრიქონ მატრიცა წარმოვადგინოთ როგორც ტრანსპორირებული სვეტ მატრიცა და პირიქით.

$$[a_i]^{\mathrm{T}} = [a^i] \quad [a_i] = [a^i]^{\mathrm{T}} \tag{3.8}$$

ან ვექტორული სახით

$$\vec{\boldsymbol{a}} \equiv [a^i] \quad \vec{\boldsymbol{a}}^{\mathrm{T}} \equiv [a_i^{\mathrm{t}}] \equiv \vec{\boldsymbol{a}}^{\mathrm{t}} \tag{3.9}$$

ტრანსპორირებულ სვეტს(ანუ სტრიქონს) ფორმალურად ავღნიშნავთ როგორც $[a^i]^{\mathrm{T}} \equiv [a^{i\mathrm{t}}]$. t ზედა ინდექსით ავღნიშნავთ რომ ობიექტი არის სტრიქონ ჩანაწერი.

3.3 სკალარული და პირდაპირი ნამრავლი

დავუშვათ მოცემულია მატრიცა სვეტი $[a^i]$ და მატრიცა სტრიქონი $[b_i]$. მაშინ

{eq:mat}

$$[a^i][b_i] = \begin{pmatrix} a^1 \\ a^2 \\ \vdots \\ a^m \end{pmatrix} (b_1 b_2 \cdots b_p) = \begin{pmatrix} a^1 b_1 & a^1 b_2 & \cdots a^1 b_p \\ a^2 b_1 & a^2 b_2 & \cdots a^2 b_p \\ \vdots & \vdots & \vdots \\ a^m b_1 & a^m b_2 & \cdots a^m b_p \end{pmatrix} \tag{3.10}$$

და

{eq:scalar}

$$[b_i][a^i] = (b_1 b_2 \cdots b_m) \begin{pmatrix} a^1 \\ a^2 \\ \vdots \\ a^m \end{pmatrix} = b_1 a^1 + b_2 a^2 + \cdots + b_m a^m = \sum_{j=1}^{m} b_j a^j \tag{3.11}$$

შემოტანილი აღნიშვნების თანახმად $[a^j]$ სვეტ მატრიცაა, ხოლო $[b_j]$ სტრიქონ მატრიცა. ამიტომ (3.10) და (3.11) ჩანაწერები არავითარ გაუგებრობას არ იწვევს. აღნიშვნები ნათლად აჩვენებს რომ $[a^j]$ და $[b_j]$ ურთიერთ ტრანსპორირებულია და ორივე შემთხვევაში ჩანაწერიდან ნათლად ჩანს რა იქნება შედეგი.

როგორც აღვნიშნეთ ზოგიერთ ლიტერატურაში გამოყენებულია მხოლოდ ქვედა ინდექსები(როგორც წესი ესაა წრფივი ალგებრის და ანალიზური გეომეტრიის, ვექტორული და ტენზორული აღრიცხვის სახელმძღვანელოების ის სექციები სადაც საუბარია ევკლიდის სივრცეზე და გამოყენებულია დეკარტის საკოორდინატო სისტემა).

თუ გვსურს ჩავწეროთ განტ. (3.11) მაშინ ვწერთ $[a^j]^{\mathrm{t}}[b^j]$. ანუ ვგულისხმობთ რომ გვაქვს სვეტ მატრიცები და (3.11) ჩანაწერი ნიშნავს, რომ ტრანსპორირებული სვეტ მატრიცა(რომელიც უკვე სტრიქონ მატრიცაა ტრანსპორირების შემდეგ) მრავლდება სვეტ მატრიცაზე და შედეგად გვაძლევს რიცხვს.

თუ გვსურს ჩავწეროთ განტ. (3.10) მაშინ ვწერთ $[a^j][b^j]^{\mathrm{t}}$. ანუ გვაქვს სვეტის[1] ნამრავლი სტრიქონზე და შედეგად მიიღება მატრიცა (3.10).

$\vec{\boldsymbol{a}} \otimes \vec{\boldsymbol{b}} \equiv [a^j][b^j]^{\mathrm{T}}$ აღნიშნავს ე.წ. "პირდაპირ ნამრავლს"(ტენზორული აღრიცხვის ტერმინია დიადური ნამრავლი, კრონეკერის ნამრავლი)[2].

თუ გვაქვს ორი სტრიქონ მატრიცა ამ შემთხვევაშიც (3.10) და (3.11) ოპერაციებისათვის საჭიროა ტრანსპორირება, რომ მატრიცული გამრავლების წესმა(სტრიქონი მრავლდება სვეტზე), რაც ყოველთვის დაცული უნდა იყოს, მოგვცეს ის შედეგები რაც გვაქვს (3.10) და (3.11) განტოლებებში.

განტ. (3.11) ბოლო ჯამში არ აქვს მნიშვნელობა b_j დავწერთ პირველს თუ a^j, ვინაიდან b_j და a^j რიცხვებია, მაგრამ მატრიცულ ნამრავლში მამრავლების თანმიმდევრობას აქვს მნიშვნელობა. როგორც განხილული მაგალითი აჩვენებს შედეგი შეიძლება იყოს რიცხვი ან მატრიცა. როცა მატრიცულ ნამრავლში მამრავლების გადანაცვლება შედეგს არ ცვლის ამბობენ, რომ მატრიცები კომუტირებენ.

[1] ანუ ისევ ვუშვებთ რომ გვაქვს სვეტ მატრიცები და სვეტის ტრანსპორირება გვაძლევს სტრიქონს

[2] როგორც ვხედავთ სვეტ მატრიცის აღსანიშნავად შემოვიტანეთ დაბალი დახრილი მუქი ასო ნიშანი. ხშირად სვეტ მატრიცას ვექტორს უწოდებენ(რაც ზოგადად სწორი არაა)

ზემოთმოყვანილ სვეტ და სტრიქონ მატრიცის აღნიშვნებში და მატრიცების გამრავლების წესის გამოყენებით ორი ვექტორის სკალარული ნამრავლი მატრიცულ აღნიშვნებში ჩაიწერება როგორც

{eq:scalProd1}
$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = \vec{\boldsymbol{a}}^{\mathrm{T}} \vec{\boldsymbol{b}} = [a_i][b^i] = a_i \cdot b^i \tag{3.12}$$

რომ არ გამოგვეყენებინა მატრიცული ჩაწერა, ან, თუ შევთანხმდებოდით, მატრიცაში ორივე ინდექსი ქვემოთაა ან ორივე ზემოთაა. შეგვეძლო დაგვეწერა

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = a^i \cdot b^i = a_i \cdot b_i \tag{3.13}$$

სვეტ და სტრიქონების ჩანაწერებისთვის ინდექსების განსხვავებულ პოზიციებზე (ზემოთ და ქვემოთ) წერა მოსახერხებელია და ბევრ შეცდომას გვარიდებს, ჩვენს შემთხვევაში (ვსაუბრობთ რიცხვებზე რომლებიც ჩაწერილია მატრიცაში და სხვა არაფერი) $a_i \cdot b^i$ ჯამში ინდექსები ზემოთ იქნება თუ ქვემოთ არ აქვს მნიშვნელობა, მაგრამ როცა შემოვიტანთ ბაზისის ცნებას და ვექტორის და ბაზისის გარდაქმნებს, ვნახავთ, რომ ზედა/ქვედა ინდექსები მოსახერხებელია. ამას გარდა მოსახერხებელია სვეტების და სტრიქონების, სვეტ მატრიცის და სტრიქონ მატრიცის კომპონენტების, სვეტ ვექტორის და სტრიქონ ვექტორს შორის განსხვავების აღსანიშნავად, რასაც ასევე ქვემოთ ვნახავთ. ახლა მხოლოდ ავღნიშნავთ რომ განტ. (3.12)-ში

{eq:scalarProduct}
$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = \vec{\boldsymbol{a}}^{\mathrm{T}} \vec{\boldsymbol{b}} \tag{3.14}$$

ყოველთვის სამართლიანია და იძლევა სწორ პასუხს. განტოლება

$$\vec{\boldsymbol{a}} \cdot \vec{\boldsymbol{b}} = [a_i][b^i] = a_i \cdot b^i$$

ასევე ყოველთვის სამართლიანია და იძლევა სწორ პასუხს. მაგრამ

$$[a_i] = [a^i]^{\mathrm{T}}$$

ვექტორის კომპონენტებისთვის სამართლიანია მხოლოდ დეკარტეს საკოორდინატო სისტემაში. ტრანსპორირება სვეტ მატრიცას აქცევს სტრიქონ მატრიცად, მაგრამ უნდა გვახსოვდეს, რომ ოპერაცია სახელად „ინდექსის ჩამოწევა“ არაა ტრანსპორირების შედეგი. ამიტომ ტრანსპორირებული სვეტის შემთხვევაში(ანუ მივიღეთ სტრიქონი).

ეს ნიშნავს შემდეგს:

- ტრანსპორირების ოპერაციას ავღნიშნავთ როგორც „T“
- სტრიქონ ჩანაწერს ავღნიშნავთ როგორც „t“

მატრიცის ერთ-ერთი საინტერესო მახასიათებელი არის ე.წ. "კვალი"(trace, spoor) $\mathrm{tr}(\mathbf{A}) = A^i_i$ — დიაგონალური ელემენტების ჯამი.

გვაქვს სკალარული ნამრავლის განზოგადება მატრიცისთვის. მატრიცისთვის სკალარული ნამრავლი განმარტებულია როგორც

$$\mathrm{tr}(\mathbf{A}^{\mathrm{T}}\mathbf{B}) = \mathrm{tr}(\mathbf{C}) = c^i{}_i = a^i{}_j b^j{}_i. \tag{3.15}$$

სადაც $a^i{}_j$, $b^i{}_j$, $c^i{}_j$ აღნიშნავს $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ მატრიცების ელემენტებს, ხოლო განმეორებული ინდექსებით ვგულისხმობთ ჯამს.

ამ განმარტების თანახმად თუ შევადარებთ (3.10) და (3.11), ვნახავთ, რომ (3.10)-ს დიაგონალური ელემენტების ჯამი მართლაც არის (3.11).

3.4 მატრიცის გამრავლების სხვადასხვა წარმოდგენა

განვიხილოთ მატრიცების გამრავლების წესი

{eq:rXc}
$$c^i_j = \sum_{k=1}^{p} a^i_k \cdot b^k_j = \vec{\boldsymbol{a}}^{i\mathrm{t}} \vec{\boldsymbol{b}}_j = \vec{\boldsymbol{a}}^i \cdot \vec{\boldsymbol{b}}_j, \quad i = \overline{1,m}; j = \overline{1,n} \tag{3.16}$$

ბოლო ჩანაწერი ტოლობაში ნიშნავს რომ „i“ სტრიქონზე ჩაწერილი ვექტორი (რომელიც სტრიქონია, რადგან ინდექსი ზემოთაა და ნომრავს სტრიქონს. სვეტების ინდექსია k, რომლიც „ჩამალულია“ ვექტორის აღნიშვნაში) სკალარულად (შიდა ნამრავლი/წერტილოვანი ნამრავლი) მრავლდება „j“ ნომერ სვეტ ვექტორზე(ინდექსი ქვემოთაა, რაც აღნიშნავს სვეტს მეორე მატრიცაში). ეს ინდექსები ნომრავენ არა ვექტორის კომპონენტებს არამედ თავად ვექტორებს. ვექტორის კომპონენტების ინდექსები არ ჩანს ვინაიდან ვექტორული ჩანაწერია. კომპონენტებს ნომრავს ჯამის ინდექსი „k“.

(3.16) არაა სტანდარტული აღნიშვნა. სახელმძღვანელოებში ნახავთ $\mathrm{row_i}(\mathbf{A}) \cdot \mathrm{col_j}(\mathbf{B})$

მატრიცული ნამრავლის სხვადასხვა წარმოდგენების განხილვამდე განვიხილოთ მატრიცების ნამრავლის ორი კერძო შემთხვევა:

1. მატრიცის გამრავლება სვეტზე.
2. სტრიქონის გამრავლება მატრიცაზე.

3.5 მატრიცის ნამრავლი სვეტზე

მატრიცის ნამრავლი სვეტზე არის სვეტი და გამოითვლება როგორც

$$\begin{pmatrix} c^1 \\ c^2 \\ \vdots \\ c^i \\ \vdots \\ c^m \end{pmatrix} = \begin{pmatrix} a^1{}_1 & \cdots & a^1{}_2 & \cdots & a^1{}_p \\ a^2{}_1 & \cdots & a^2{}_2 & \cdots & a^2{}_p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i{}_1 & \cdots & a^i{}_k & \cdots & a^i{}_p \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a^m{}_1 & \cdots & a^m{}_k & \cdots & a^m{}_p \end{pmatrix} \cdot \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^k \\ \vdots \\ b^p \end{pmatrix} = \begin{pmatrix} a^1{}_1 \cdot b^1 + \cdots + a^1{}_k \cdot b^k + \cdots + a^1{}_p \cdot b^p \\ a^2{}_1 \cdot b^1 + \cdots + a^2{}_k \cdot b^k + \cdots + a^2{}_p \cdot b^p \\ \vdots \\ a^i{}_1 \cdot b^1 + \cdots + a^i{}_k \cdot b^k + \cdots + a^i{}_p \cdot b^p \\ \vdots \\ a^m{}_1 \cdot b^1 + \cdots + a^m{}_k \cdot b^k + \cdots + a^m{}_p \cdot b^p \end{pmatrix} \tag{3.17}$$

$$\begin{pmatrix} c^1 \\ c^2 \\ \vdots \\ c^i \\ \vdots \\ c^m \end{pmatrix} = \begin{pmatrix} a^1{}_1 \\ a^2{}_1 \\ \vdots \\ a^i{}_1 \\ \vdots \\ a^m{}_1 \end{pmatrix} \cdot b^1 + \cdots + \begin{pmatrix} a^1{}_k \\ a^2{}_k \\ \vdots \\ a^i{}_k \\ \vdots \\ a^m{}_k \end{pmatrix} \cdot b^k + \cdots + \begin{pmatrix} a^1{}_p \\ a^2{}_p \\ \vdots \\ a^i{}_p \\ \vdots \\ a^m{}_p \end{pmatrix} \cdot b^p \tag{3.18}$$

$$\boxed{\vec{\boldsymbol{c}} = \mathbf{A} \cdot \vec{\boldsymbol{b}} = \begin{pmatrix} | & & | & & | \\ \vec{\boldsymbol{a}}_1 & \cdots & \vec{\boldsymbol{a}}_k & \cdots & \vec{\boldsymbol{a}}_p \\ | & & | & & | \end{pmatrix} \cdot \begin{pmatrix} b^1 \\ b^2 \\ \vdots \\ b^k \\ \vdots \\ b^p \end{pmatrix} = \vec{\boldsymbol{a}}_1 \cdot b^1 + \cdots + \vec{\boldsymbol{a}}_k \cdot b^k + \cdots + \vec{\boldsymbol{a}}_p \cdot b^p = \vec{\boldsymbol{a}}_i b^i}$$

{eq:matXcol} (3.19)

ცხადია თუ გამოვიყენებთ ზოგად წესს (სტრიქონი მრავლდება სვეტზე) მაშინ შედეგი ჩაიწე-

რება როგორც(შეადარეთ გან. (3.12))

{eq:matXcol_1}

$$\boxed{\vec{\boldsymbol{c}} = \mathbf{A}\cdot\vec{\boldsymbol{b}} = \begin{pmatrix} - & \vec{\boldsymbol{a}}^{1\mathrm{t}} & - \\ - & \vec{\boldsymbol{a}}^{2\mathrm{t}} & - \\ & \vdots & \\ - & \vec{\boldsymbol{a}}^{m\mathrm{t}} & - \end{pmatrix}\cdot\vec{\boldsymbol{b}} = \begin{pmatrix} \vec{\boldsymbol{a}}^{1\mathrm{t}}\cdot\vec{\boldsymbol{b}} \\ \vec{\boldsymbol{a}}^{2\mathrm{t}}\cdot\vec{\boldsymbol{b}} \\ \vdots \\ \vec{\boldsymbol{a}}^{m\mathrm{t}}\cdot\vec{\boldsymbol{b}} \end{pmatrix}} \tag{3.20}$$

ან უფრო კომპაქტურად

$$\boxed{c^i = \vec{\boldsymbol{a}}^{i\mathrm{t}}\cdot\vec{\boldsymbol{b}}} \quad \text{ცხადია იგივეა რაც (3.19)} \tag{3.21}$$

3.6 სტრიქონის გამრავლება მატრიცაზე.

მატრიცის ნამრავლი სვეტზე არის სვეტი და გამოითვლება როგორც

$$\Big(c_1\ c_2\cdots c_k\cdots c_p\Big) = \Big(b_1\ b_2\cdots b_i\cdots b_m\Big)\cdot\begin{pmatrix} a^1{}_1 & \cdots & a^1{}_2 & \cdots & a^1{}_p \\ a^2{}_1 & \cdots & a^2{}_2 & \cdots & a^2{}_p \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a^i{}_1 & \cdots & a^i{}_k & \cdots & a^i{}_p \\ \vdots & \vdots & \cdots & \vdots & \vdots \\ a^m{}_1 & \cdots & a^m{}_k & \cdots & a^m{}_p \end{pmatrix} \tag{3.22}$$

ადგილის დაზოგვის მიზნით ჩავწეროთ ტოლობა ყოველი c_i-სათვის

$$c_i = \sum_{k=1}^{m} b_k\cdot a^k_i$$

ეს ჯამი გვაქვს თითოეული c_i-სათვის. თუ გამოვიყენებთ მატრიცების შეკრების წესს. გავშლით მიღებულ ჯამს და დავაჯგუფებთ b_i წევრებს მივიღებთ

$$\Big(c_1\cdots c_k\cdots c_p\Big) = b_1\cdot\Big(a^1_1\cdots a^1_k\cdots a^1_p\Big) + \cdots + b_m\cdot\Big(a^m_1\cdots a^m_k\cdots a^m_p\Big) \tag{3.23}$$

ან ზემოთ მოყვანილ განტოლებებს თუ ჩავწერთ სტრიქონ ვექტორების გამოყენებით

$$\vec{\boldsymbol{c}}^{\,\mathrm{t}} = \vec{\boldsymbol{b}}^{\,\mathrm{t}}\cdot\begin{pmatrix} -\vec{\boldsymbol{a}}^{1\mathrm{t}}- \\ -\vec{\boldsymbol{a}}^{2\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{i\mathrm{t}}- \\ \vdots \\ -\vec{\boldsymbol{a}}^{m\mathrm{t}}- \end{pmatrix} \tag{3.24}$$

და

$$\boxed{\vec{\boldsymbol{c}}^{\,\mathrm{t}} = b_k\vec{\boldsymbol{a}}^{k\mathrm{t}}} \tag{3.25}$$

შეგახსენებთ რომ t აღნიშნავს რომ გვაქვს სტრიქონ ვექტორები.

3.7 მატრიცული ნამრავლის სხვადასხვა წარმოდგენა

ზემოთ აღწერილი შემთხვევების განხილვის შემდეგ მატრიცული ნამრავლი (ე.წ. შიდა ნამრავლი) შეგვიძლია განვიხილოთ ხუთი სხვადასხვა გზით:

1. მატრიცული ნამრავლის განმარტების თანახმად ესაა სტრიქონ ვექტორების სკალარული ნამრავლი სვეტ ვექტორებზე(მატრიცების გამრავლების წესის თანახმად $\vec{a}$-ს ზომა ტოლი უნდა იყოს $\vec{b}$-ს ზომის.

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} - & \vec{a}^{1\mathrm{t}} & - \\ - & \vec{a}^{2\mathrm{t}} & - \\ & \vdots & \\ - & \vec{a}^{m\mathrm{t}} & - \end{pmatrix} \begin{pmatrix} | & | & | & & | \\ \vec{b}_1 & \vec{b}_2 & \vec{b}_3 & \dots & \vec{b}_n \\ | & | & | & & | \end{pmatrix} \tag{3.26}$$

$$= \begin{pmatrix} \vec{a}^{1\mathrm{t}} \cdot \vec{b}_1 & \vec{a}^{1\mathrm{t}} \cdot \vec{b}_2 & \vec{a}^{1\mathrm{t}} \cdot \vec{b}_3 & \dots & \vec{a}^{1\mathrm{t}} \cdot \vec{b}_t \\ \vec{a}^{2\mathrm{t}} \cdot \vec{b}_1 & \vec{a}^{2\mathrm{t}} \cdot \vec{b}_2 & \vec{a}^{2\mathrm{t}} \cdot \vec{b}_3 & \dots & \vec{a}^{2\mathrm{t}} \cdot \vec{b}_t \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vec{a}^{m\mathrm{t}} \cdot \vec{b}_1 & \vec{a}^{m\mathrm{t}} \cdot \vec{b}_2 & \vec{a}^{m\mathrm{t}} \cdot \vec{b}_3 & \dots & \vec{a}^{m\mathrm{t}} \cdot \vec{b}_n \end{pmatrix}$$

2. მატრიცული ნამრავლი როგორც მატრიცის ნამრავლი სვეტზე

$$\mathbf{C} = \mathbf{AB} = \mathbf{A} \begin{pmatrix} | & & | \\ \vec{b}_1 & \dots & \vec{b}_n \\ | & & | \end{pmatrix} = \begin{pmatrix} | & & | \\ \mathbf{A}\vec{b}_1 & \dots & \mathbf{A}\vec{b}_n \\ | & & | \end{pmatrix} \tag{3.27}$$

აქედან გამომდინარეობს, რომ $\mathbf{C}$ მატრიცის i სვეტი წარმოადგენს $\mathbf{A}$ მატრიცის და $\mathbf{B}$-მატრიცის i სვეტის ნამრავლს.

$$\{\mathbf{C}\}_i \equiv \vec{c}_i = \mathbf{A}\vec{b}_i \tag{3.28}$$

3. მატრიცული ნამრავლი როგორც სტრიქონის ნამრავლი მატრიცაზე

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} - & \vec{a}^{1\mathrm{t}} & - \\ - & \vec{a}^{2\mathrm{t}} & - \\ & \vdots & \\ - & \vec{a}^{m\mathrm{t}} & - \end{pmatrix} \mathbf{B} = \begin{pmatrix} -\vec{a}^{1\mathrm{t}}\mathbf{B}- \\ -\vec{a}^{2\mathrm{t}}\mathbf{B}- \\ \vdots \\ -\vec{a}^{m\mathrm{t}}\mathbf{B}- \end{pmatrix} \tag{3.29}$$

$$\vec{c}^{\,i\mathrm{t}} = \vec{a}^{\,i\mathrm{t}}\mathbf{B} \tag{3.30}$$

4. მატრიცული ნამრავლი როგორც სვეტის და სტრიქონის პირდაპირი ნამრავლების ჯამი იხ. განტ (3.10)

$$\mathbf{C} = \mathbf{AB} = \begin{pmatrix} | & & | \\ \vec{a}_1 & \dots & \vec{a}_p \\ | & & | \end{pmatrix} \begin{pmatrix} - & \vec{b}^{1\mathrm{t}} & - \\ - & \vec{b}^{2\mathrm{t}} & - \\ & \vdots & \\ - & \vec{b}^{p\mathrm{t}} & - \end{pmatrix} = \left(\sum_i \left(\vec{a}_i \otimes \vec{b}^{\,i\mathrm{t}} \right) \right) \tag{3.31}$$

5. მატრიცული ნამრავლი, როგორც ბლოკებად დანაწევრებული მატრიცების ნამრავლი

$$\begin{aligned}\mathbf{C} = \mathbf{AB} &= \left(\begin{array}{c|c} \mathbf{A}_1 & \mathbf{A}_2 \\ \hline \mathbf{A}_3 & \mathbf{A}_4 \end{array}\right) \times \left(\begin{array}{c|c} \mathbf{B}_1 & \mathbf{B}_2 \\ \hline \mathbf{B}_3 & \mathbf{B}_4 \end{array}\right) \\ &= \left(\begin{array}{c|c} \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_3 & \mathbf{A}_3\mathbf{B}_1 + \mathbf{A}_4\mathbf{B}_3 \\ \hline \mathbf{A}_1\mathbf{B}_2 + \mathbf{A}_2\mathbf{B}_1 & \mathbf{A}_3\mathbf{B}_1 + \mathbf{A}_4\mathbf{B}_3 \end{array}\right)\end{aligned} \tag{3.32}$$