თემა 12: სახელების სივრცე (namespace). გამონაკლისები (exceptions)

საკითხები:

სახელების სივრცე (namespace)

სახელების კონფლიქტი. ხილვადობა (visibility). დიდ პროექტებში ე.წ. სახელების კონფლიქტის ალბათობა მაღალია, ამიტომ საჭირო ხდება სათანადო ზომების მიღება. სახელების კონფლიქტი ნიშნავს, რომ ერთდროულად გამოყენებულია ერთი სახელის და ტიპის მქონე რამდენიმე ცვლადი ან ობიექტი, ან ერთი და იმავე ხელწერის მქონე რამდენიმე ფუნქცია.

განვიხილოთ სახელების კონფლიქტის თავიდან აცილების რამდენიმე გზა. მათ შორის ერთ-ერთი ძირითადი არის სახელების სივრცის გამოყენება.

როდესაც კომპილერი მიყვება კოდს და ადგენს სახელებისა და ფუნქციების სიებს, იგი ამავდროულად ცდილობს სახელების კონფლიქტის აღმოჩენას. თუმცა, თუ სახელები გაბნეულია object ფაილსა და ტრანსლაციის სხვა ბლოკებს შორის, კომპილერი ვერ აღმოაჩენს სახელების კონფლიქტს. ამ შემთხვევაში ეს უკვე ლინკერის საქმეა.

განვიხილოთ მაგალითი. ვთქვათ, პროექტი შედგება მინიმუმ ორი cpp ფაილისგან:

```
//file main.cpp
int intValue = 0;
int main()
{
	int intValue =0;
}
```

და

```
//file second.cpp
int intValue;
```

ლინკერის დიაგნოზი შემდეგია:

```
>second.obj : error LNK2005: "int intValue" (?intValue@@3HA) already defined in main.obj
```

პრობლემა იმაშია, რომ ორ გლობალურ ცვლადს აქვს ერთნაირი ხილვადობის ფარგლები.

ტერმინი ხილვადობა გამოიყენება ობიექტის (ცვლადი, კლასი, ფუნქცია) მოქმედების ფარგლების განსაზღვრისთვის.

მაგალითად, თუ ობიექტი განსაზღვრულია ყველა ფუნქციის გარეთ, მისი მოქმედების ფარგლები არის გლობალური, ანუ ფაილი. ეს ობიექტი ხილვადია მისი განსაზღვის მომენტიდან ფაილის დასასრულამდე. ამისგან განსხვავებით, ობიექტი, რომელიც განსაზღვრულია ბლოკში, მოქმედებს ბლოკის, ანუ ლოკალურ ფარგლებში. განვიხილოთ მარტივი მაგალითი:

```
int globalInt = 5;
void f(void)
{
	int localInt = 11;
}
int main()
{
	int localInt = 55;
```

```
        {
                int otherLocalInt = 20;
                int localInt = 30;
        }
}
```
პირველი განაცხადი int ტიპის ცვლადზე კეთდება globalInt ცვლადზე, რომელიც ჩანს f() ფუნქციიდან და მთავარი ფუნქციიდან. მეორე განაცხადი int ტიპის ცვლადზე კეთდება f() ფუნქციაში. ამ ცვლადს აქვს ლოკალური ფარგლები და ჩანს იმ ბლოკში, რომელშიც განსაზღვრულია. Main() ფუნქციას არ შეუძლია f() ფუნქციის localInt ცვლადზე მიწვდომა და მისი მნიშვნელობების მანიპულირება. როცა f() ფუნქცია დასრულდება, მისი ცვლადი გადის ხილვადობის ფარგლებიდან.

მესამე განაცხადი კეთდება main() -ში int ტიპის localInt ცვლადზე ბლოკის ფარგლებში. შემდეგი ორი ცვლადიც ბლოკის ფარგლებში მოქმედებს. როგორც კი დაიხურება ბლოკის } ფრჩხილი, ეს ცვლადები კარგავენ ხილვადობას. თუმცა ამ ბოკის შიგნით ხილვადია ის localInt ცვლადი, რომლის მნიშვნელობაც არის 20.

<<< **კავშირი (linkage).** სახელს შეიძლება ჰქონდეს შიგა კავშირი (internal linkage) ან გარე კავშირი (external linkage). ეს ორი ტერმინი ახასიათებს სახელზე წვდომას, ტრანსლაციის მრავალ ბლოკს შორის ან ტრანსლაციის მხოლოდ ერთ ბლოკში. თუ სახელს აქვს შიგა კავშირი, იგი მიწვდომადია მხოლოდ ტრანსლაციის იმ ბლოკის შიგნით, რომელშიც არის განსაზღვრული. კავშირების ილუსტრირებისთვის განვიხილოთ მაგალითი. ვთქვათ, პროექტი შედგება ორი ფაილისგან:

```
//file: main.cpp
int externalInt = 55;
const int j = 11;

int main(){}
```
და

```
//file: second.cpp
extern int externalInt;
int anExternalInt = 99;
const int j = 11;
```

ცვლადს externalInt აქვს გარე კავშირი, ამიტომ მეორე ფაილსაც აქვს მასზე წვდომა. მუდმივი ცვლადები იგულისხმება შიგა კავშირის მქონედ. თუმცა შეგვიძლია ცხადი აღწერა დავამატოთ და გავხადოთ გარე კავშირის მქონედ. მაგალითად:

```
//file: first.cpp
const int j = 11;
```
და

```
//file: second.cpp
extern const int j;
#include<iostream>
int main()
{
    std::cout << "j=" << j << std::endl;
}
```
როგორც ვხედავთ, გარკვეული მსგავსებაა ვირტუალობასა და გარე კავშირს შორის. გარე კავშირის მქონე ობიექტი ერთადერთია, ამიტომ მისი ინიციალიზება მხოლოდ ერთხელ ხდება. Extern საკმარისია მხოლოდ იმ განაცხადს მივუწეროთ, სადაც ინიციალიზება არ ხდება.

<<< **სახელების სივრცის შექმნა.** სახელების სივრცე აჯგუფებს ერთმანეთთან დაკავშირებულ აღწერებს და განაცხადებს. მიზანი არის სახელების კონფლიქტის თავიდან აცილება.

სახელების სივრცის შექმნა ძალიან გავს კლასის შექმნას, მაგრამ არის რამდენიმე პრინციპული განსხვავება: სახელების სივრცის სტრუქტურა მარტივია:

```
namespace სახელი
```

```
{
    // declarations and statement
}
```
გავს კლასისას, მაგრამ არ მთავრდება წერტილმძიმით.

შემდეგ, შესაძლოა არსებობდეს სახელების სივრცე სახელის გარეშე ან პირიქით, ერთი და იმავე სივრცეს ერქვას რამდენიემ სახელი (ასეთ მაგალითებს არ განვიხილავთ).

ბოლოს, რაც მთავარია, ერთი და იგივე სახელების სივრცე შესაძლოა რამდენიმე განსხვავებულ ფაილში იქმნებოდეს.

განვიხილოთ მაგალითი, რომელიც მცირე ცვლილებებით არის აღებული სტრაუსტრუპის წინიდან. ვთქვათ, გვაქვს ორი ფაილისგან შედგენილი პროექტი:

```
//file: first.h
namespace Foo
{
       int a;
}
```
და

```
//file: main.cpp
#include<iostream>
#include"first.h"
using namespace std;

int a(9);
namespace Foo
{
       void f(int i)
       {
              a+=i;
       }

}
void f(int i)
{
       cout << "i=" << i << endl;
}

int main()
{
       int a = 7;
       ::a = 99;
       cout <<  "::a =" << ::a << endl;
       cout <<  "a =" << a << endl;
       cout <<  "Foo::a =" << Foo::a << endl;

       f(2);
       Foo::f(3);
       cout <<  "after f(2) and Foo::f(3) " << endl;
       cout <<  "a=" << a << endl;
       cout <<  "Foo::a =" << Foo::a << endl;
       Foo::a = 17;
       Foo::f(5);
       cout <<  "after Foo::a = 17; and Foo::f(5) " << endl;
       cout <<  "Foo::a =" << Foo::a << endl;

       ::f(4);
}
```

პროგრამის მუშაობის შედეგი არის:

```
::a =99
a =7
Foo::a =0
i=2
after f(2) and Foo::f(3)
a=7
Foo::a =3
after Foo::a = 17; and Foo::f(5)
Foo::a =22
i=4
Press any key to continue . . .
```

როგორც ვხედავთ, სახელები ცალსახად კვლიფიცირდება მათი სახელების სივრცის სახელით (მაგ. Foo::f3), ან :: ოპერატორის გამოყენებით, რაც მიუთითებს გლობალური ხილვადობის მქონე სახელს.

<<< გამონაკლისები (exceptions)

გამონაკლისების საშუალებით ხდება რეაგირება გამონაკლის შემთხვევებზე (მაგალითად, შესრულების დროს წარმოქმნილ შეცდომებზე). ამ მიზნით, გამონაკლის შემთხვევებში მართვა გადაეცემა კერძო სახის ფუნქციებს, ე.წ. დამმუშავებლებს (hamdler).

იმისათვის რომ დავიჭიროთ (არ გამოგვეპაროს) გამონაკლისი სიტუაცია, კოდის ნაწილი თავსდება გამონაკლის შემთხვევაზე შემოწმების ბლოკში. როცა ამ ბლოკში წარმოიშობა გამონაკლისი სიტუაცია, ხდება გამონაკლისის „გამოსროლა“, ანუ მართვის გადაცემა გამონაკლისის დამმუშავებელზე. თუ გამონაკლისი სიტუაცია არ იქმნება, მაშინ კოდი სრულდება ნორმალურად და დამმუშავებელი ფუნქციები არ გამოიყენება.

ამ ყველაფერს უზრუნველყოფს შემდეგი სინტაქსი. გამონაკლისზე შემოწმების ბლოკს ჰქვია try - ბლოკი. ამ ბლოკში გამონაკლისის აღმოჩენის შემთხვევაში ხდება მართვის გადაცემა (ანუ გამონაკლისის გამოსროლა) დამმუშვებლებზე. დამმუშავებლის ბლოკს ჰქვია `catch`- ბლოკი. იგი მართვას მიიღებს გასროლილი გამონაკლისის ტიპის მიხედვით. ფორმალურად, გამოაკლისის გამოსროლა შეიძლება მოხდეს უპირობოდაც. მაგ:

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

int main () {
  try
  {
    throw 20;
  }
  catch (int e)
  {
    cout << "An exception occurred. Exception Nr. " << e << '\n';
  }
}
```

პროგრამის მუშაობის შედგად გვაქვს:

An exception occurred. Exception Nr. 20
Press any key to continue . . .

გამონაკლისი შემთხვევების გათვალისწინების აუცილებლობა ძალიან ხშირად წარმოიშობა. მაგალითად, როდესც ორ რიცხვს ვყოფთ ერთმანეთზე, ყოველთვის არის საფრთხე, რომ შესაძლოა მოხდეს ნულზე გაყოფა. ამ მარტივ მაგალითზე ვნახოთ, თუ როგორ შეგვიძლია გამონაკლისების საშუალებით შევქმნათ უფრო დაცული კოდი. განვიხილოთ:

```
// trying and catching
#include <iostream>
using namespace std;

int main ()
{
       int top(55);
       int bottom(0);
    try
    {
              cout << "top/2=" << (top/2) << endl;
              cout << "top devided by bottom = ";
              if( bottom == 0)
                     throw "Division by zero!";
              cout << ( top / bottom ) << endl;
              cout << "top/3=" << (top/3) << endl;
    }
    catch (const char* ex)
    {
          cout << "\n*** " << ex << " ***" << endl;
    }
       cout << "Done." << endl;
}
```

მისი მუშაობის შედეგია:

```
top/2=27
top devided by bottom =
*** Division by zero! ***
Done.
Press any key to continue . . .
```

როგორც ვხედავთ, შემოწმების ბლოკში იმ ადგილზე სადაც შესაძლოა პრობლემა შეიქმნას, პრობლემის შექმნის შემთხვევაში გათვალისწინებულია პროგრამის ნორმალური მსვლელობის დარღვევა და მართვის გადაცემა დამმუშვებელზე.

გამონაკლისი შემთხვევების დამუშავება ძალიან ვრცელი თემაა, და იგი სულ უფრო იხვეწება პროგრამირების ენების ახალ-ახალ სტანდარტებში.

ბოლო მაგალითი გვიჩვენებს, თუ როგორ შეიძლება ფუნქციისთვის უფრო დაცული კოდის შექმნა:

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

double division(int a, int b)
{
   if( b == 0 )
   {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main ()
{
   int x = 50;
   int y = 10;
   double z = 0;

   try {
      z = division(x, y);
      cout << z << endl;
   }catch (const char* msg) {
```

```
        cerr << msg << endl;
    }
}
```

როდესაც ფუნქციის არგუმენტები რეგულარულია, შედეგი მიიღება პროგრამის ნორმალური მსვლელობის შედეგად:

5
Press any key to continue . . .

თუ ავიღებთ

```
int y = 0;
```

შედეგი მიიღება გამონაკლისის გათვალისწინებით:

Division by zero condition!
Press any key to continue . . .

<<< სავარჯიშოები:

1. გამონაკლისი სიტუაციების გათვალისწინებით, შექმენით ფუნქცია, რომელიც იპოვის და დააბრუნებს ნამდვილი რიცხვების ვექტორის ელემენტების საშუალო არითმეტიკულს.
2. იგივე ამოცანა გადაჭერით იტერატორების საშუალებით. გაითვალისწინეთ მესამე არგუმენტი, რომელშიც უნდა ჩაიწეროს საშუალო არითმეტიკული.
3. გამონაკლისი სიტუაციების გათვალისწინებით, შექმენით ფუნქცია, რომელიც იპოვის და დააბრუნებს ნამდვილი დადებითი რიცხვების ვექტორის ელემენტების საშუალო გეომეტრიულს.
4. იგივე ამოცანა გადაჭერით იტერატორების საშუალებით. გაითვალისწინეთ მესამე არგუმენტი, რომელშიც უნდა ჩაიწეროს საშუალო გეომეტრიული.
5. შექმენით ფუნქცია, რომელიც გვერდების მიხედვით გამოითვლის და დააბრუნებს მართკუთხედის ფართობს. თუ რომელიმე გვერდის სიგრძე უარყოფითია, ფუნქციამ უნდა გამოისროლოს გამონაკლისი. მოიყვანეთ ფუნქციის გამოძახების მაგალითი `try { }` ბლოკის შიგნით. მოიყვანეთ აგრეთვე ამ გამონაკლისის დამუშვების კოდი.
6. შექმენით ფუნქცია, რომელიც გვერდების მიხედვით გამოითვლის და დააბრუნებს მართკუთხედის პერიმეტრს. თუ რომელიმე გვერდის სიგრძე უარყოფითია, ფუნქციამ უნდა გამოისროლოს გამონაკლისი. მოიყვანეთ ფუნქციის გამოძახების მაგალითი `try { }` ბლოკის შიგნით. მოიყვანეთ აგრეთვე ამ გამონაკლისის დამუშვების კოდი.