თემა 1. თარგიანი ფუნქცია და კლასი

განხილული საკითხები:

- *ორ ობიექტს შორის მაქსიმალურის განაზღვრის მაკროსი >>>*
- *იგივე შინაარსის უფრო ვრცელი მაკროსი >>>*
- *ორ ობიექტს შორის მაქსიმალურის განაზღვრა თარგიანი ფუნქციით >>>*
- *თარგიანი swap ფუნქცია >>>*
- *თარგიანი (templated) კლასი და მისი გამოყენების მაგალითი >>>*
- *თარგი, სპეციალიზაცია და ნაწილობრივი სპეციალიზაცია >>>*
- *სავარჯიშოები >>>*

ტერმინი template ქართულად ითარგმნება როგორც ყალიბი, თარგი, ან თუნდაც ფორმა. როგორც სხვადასხვა მასალისგან შეიძლება ჩამოისხას ერთი და იგივე ფორმა, ასევე სხადასხვა ტიპის მონაცემებზე შესაძლოა შესრულდეს ერთი და იგივე მოქმედებების მიმდევრობა.

თარგების საშუალებით იწერება განზოგადებული (genetic) კოდი და ფუნქციები, რომლებიც მუშაობენ განსხვავებულ მონაცემთა ტიპებზე. C++ ენა მოიცავს სპეციალურ ბიბლიოთეკას (STL) რომელიც იძლევა განზოგადებული პროგრამირების საშუალებებს.

თარგების შემოღებამდე იგივე საქმეს გარკვეულ დონემდე ასრულებდნენ მაკროსების საშუალებით. მაკროსის, თარგის, inline ფუნქციების გამოყენებით დაწერილი კოდი კომპილაციის დროს განიცდის სერიოზულ ცვლილებებს, ამიტომ მათ საერთო მექანიზმი აქვთ. მათ შორის განსხვავება ისაა, რომ თარგით შექმნილი კოდი ბევრად უფრო დაცულია შეცდომებისგან და ბევრად მეტი შესაძლებლობები აქვს.

წარმოვიდგინოთ, რომ გვინდა ორ ობიექტს შორის მაქსიმალურის ამორჩევა, იმ პირობით რომ ობიექტებზე განსაზღვრულია შედარების ოპერატორები. განვიხილოთ კოდი:

<<< ორ ობიექტს შორის მაქსიმალურის განაზღვრის მაკროსი *(ეს ნაკვეთი მხოლოდ გაცნობითი ხასიათისაა)*

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

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

int main()
{
    cout << MAX(-11,4) << endl;
    cout << MAX(11.5, 4.3) << endl;
    return 0;
}
```

აქ სტრიქონი `#define MAX(a,b)  ((a<b)?a:b)` *ეუბნება პრეპროცესორს, რომ მან პირველი წევრი* `MAX(a,b)` *შემდეგში ყველგან უნდა ჩაანაცვლოს მეორეთი* `((a<b)?a:b)`. *დიდი ასოები იმას ეუბნება პროგრამისტს, რომ ეს ფაქტიურად მუდმივია, ჩანაცვლებადი პარამეტრებით. კოდი ზოგადია, მაგრამ საკმაოდ დაუცველი შეცდომებისგან და შეზღუდული შესაძლებლობით, -* `#define` *დირექტივის შემდეგ მხოლოდ ერთი სტრიქონი უნდა იყოს ათვისებული.*

ამ ბოლო შეზღუდვის აცილება ადვილად შეიძლება. შემდეგი კოდი გვიჩვენებს როგორ გავაკეთოთ უფრო ვრცელი მაკროსი, რომელიც რამდენიმე სხადასხვა ფუნქციით შეიძლება ჩანაცვლდეს:

```
//  <<<  ვრცელი მაკროსი

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

```
#define def_max(type) type max(type d1,type d2) { \
    if(d1 > d2)                                   \
        return d1;                                \
    return d2;                                    \
    )                                             \

int main()
{
    cout << max(-11,4) << endl;
    cout << max(11.5,4.3) << endl;
    cout << max('S','a') << endl;
    system("pause");
    return 0;
}
```

აქ შებრუნებული სლეშები კომპილერს ეუბნება, რომ სტრიქონი ლოგიკურად გახლეჩილი არაა და უნდა განხილულ იქნას როგორც ერთი მთლიანი. ეს ტექნიკა სხვა ამოცანებშიც ხშირად გამოიყენება C++ენაში.

ამ მაკროსის ხუთი სტრიქონი არის მაკროსის განსაზღვრის ეტაპი.

ძველ კომპილერებს კიდევ სჭირდებათ ფუნქციების გენერირების ეტაპი, რაც ნიშნავს რომ მაკროსსა და `int main()` ფუნქციას შორის უნდა მოთავსდეს მითითებები, თუ რომელი ტიპებისთვის შესრულდება ეს მაკროსი. ჩვენს შემთხვევაში:

```
def_max(int);
def_max(double);
def_max(char);
```

ამ ყველაფრისთვის უმჯობესია გამოვიყენოთ თარგი, რაც ნიშნავს, რომ ფუნქციის განაცხადში უნდა დავამატოთ წინადადება (განაცხადი), რომელიც კომპილერს ეტყვის, რომ ერთი ან რამდენიმე ტიპი პარამეტრს წარმოადგენს ფუნქციისათვის. ჩვენს შემთხვევაში:

<<< ორ ობიექტს შორის მაქსიმალურის განაზღვრა თარგიანი ფუნქციით

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

template <typename T>
T maxim(const T d1, const T d2)
{
    if(d1 > d2)
        return d1;
    return d2;
}

int main()
{
    int a(5), b(7);
    cout << maxim(a,b) << endl;
    cout << maxim(11.5,4.3) << endl;
    cout << maxim('S','a') << endl;
}
```

ფუნქციის სახელია `maxim` და არა max, რათა თვიდან ავიცილოთ სახელების კონფლიქტი. აქ ერთი დამატებით სტრიქონი `template <typename T>` აფრთხილებს კომპილერს, რომ `T` არის მონაცემთა ტიპი და იგი შეიძლება იცვლებოდეს. ამ ფუნქციის თარგის გაკეთებისის უფრო ვრცლად იხ. მისამართზე https://en.cppreference.com/w/cpp/algorithm/max.

ზოგადად, თარგის პარამეტრი შესაძლოა იყოს ტიპიც და იყოს სხვა რამეც.

!) **ყურადღება მივაქციოთ**, რომ კომპილერი არაა პასუხისმგებელი შეამოწმოს ფუნქციის ტანში პარამეტრად გადაცემულ ტიპზე განხორციელებული მოქმედებები კორექტულია თუ არა.

!!) **ყურადღება მივაქციოთ** იმ გარემოებას, რომ თარგების გამოყენება სულაც არ გამორიცხავს ფუნქციის გადატვირთვის აუცილებლობას. მაგალითები შეგვიძლია ვნახოთ შემდეგ სავარჯიშოებში.

დასასრულ, შევქმნათ ფუნქცია, რომელიც მნიშვნელობას გაუცვლის ორ ობიექტს:

<<< თარგიანი swap ფუნქცია

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

template <typename T>
void Swap(T& d1, T& d2)
{
    T tmp(d1);
    d1 = d2;
    d2 = tmp;
}
int main()
{
    int a(5), b(7);
    cout << "Before swap:  a = " << a << ";  b = " << b << endl;
    Swap(a, b);
    cout << "After swap:  a = " << a << ";  b = " << b << endl;

    string s1 = ("Anna"), s2 = ("Ani");
    cout << "Before swap:  s1 = " << s1 << ";  s2 = " << s2 << endl;
    Swap(s1, s2);
    cout << "After swap:  s1 = " << s1 << ";  s2 = " << s2 << endl;
}
```

თარგიანი შეიძლება იყოს კლასიც. ვიდრე ასეთი კლასების აგებას ვისწავლით, განვიხილოთ მაგალითი. cpusplus.com ზე შეგიძლიათ დაწვრილებით გაეცნოთ კლასს pair<,>, რომელსაც აქვს ორი ღია ველი სახელად first, second, და რამდენიმე საინტერესო მეთოდი. განაცხადის გაკეთება ძალიან მარტივია, როგორც გვიჩვენებს შემდეგი მაგალითი.

<<< თარგიანი (templated) კლასი და მისი გამოყენების მაგალითი

განვიხილოთ მარტივი მაგალითი pair<string, int>- ის გამოყენებაზე. შემდეგ ჩვენ თვითონ შევქმნათ ასეთივე კლასი, ჯერ ზუსტად <string, int> სპეციალიზაციის მსგავსი, შემდეგ ზოგადი, თარგის გამოყენებით.

```
#include<iostream>
#include<vector>
using namespace std;
int main()
{
	pair<string, int> a;
	vector<pair<string, int> > v;

	a.first = "Gia";
	a.second = 2320942;
	v.push_back(a);

	a.first = "Omar";
	a.second = 5770941;
	v.push_back(a);
```

```
	v.push_back(pair<string, int>("Khatia", 1234567));
	v.push_back(make_pair("Gio", 1234321));

	for (int i = 0; i<v.size(); i++)
		cout << v[i].first << '\t' << v[i].second << endl;
}
```

შევქმნათ ასეთი კლასი, <string, int> სპეციალიზაციაზე მიმსგავსებული:

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

class myPair
{
public:
	string first;
	int second;
	myPair() {}
	myPair(const string &a, const  int b) :first(a), second(b) {}
	void printMyPairObject()
	{
		std::cout << first << "   " << second << std::endl;
	}
};

int main()
{
	myPair a;
	vector<myPair> v;

	a.first = "Gia";
	a.second = 2320942;
	v.push_back(a);

	a.first = "Omar";
	a.second = 5770941;
	v.push_back(a);

	v.push_back(myPair("Khatia", 1234567));

	for (int i = 0; i<v.size(); i++)
		cout << v[i].first << '\t' << v[i].second << endl;
}
```

აქ და შემდეგ მაგალითში, (ჯერ-ჯერობით) ვერ ვაკეთებთ ერთ წყვილს, რომელიც პირველ მაგალითში გავაკეთეთ ფაბრიკა-ფუნქციის გამოყენებით.

ახლა, გავაკეთოთ ასეთივე კლასი თარგით და გამოვიყენოთ ისევე, როგორც სტანდარტული ბიბლიოთეკის pair<,> კლასის მაგალითში:

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

template<typename T, typename S>
class myPair
{
public:
```

```
	T first;
	S second;
	myPair() {}
	myPair(const T &a, const  S b) :first(a), second(b) {}
	void printMyPairObject()
	{
		std::cout << first << "   " << second << std::endl;
	}
};


int main()
{
	myPair<string, int> a;
	vector<myPair<string, int>> v;

	a.first = "Gia";
	a.second = 2320942;
	v.push_back(a);

	a.first = "Omar";
	a.second = 5770941;
	v.push_back(a);

	v.push_back(myPair<string, int>("Khatia", 1234567));

	for (int i = 0; i<v.size(); i++)
		cout << v[i].first << '\t' << v[i].second << endl;
}
```

რადგან კლასს ელემენტის ბეჭდვაც დავამატეთ, შეგვიძლია პროგრამა-დრაივერის ბოლო შეტყობინება უფრო თანამედროვე სახით გავაფორმოთ:

```
	for (auto e : v)
		e.printMyPairObject();
```

როდესაც კლასის წევრ-ფუნქციებს (მეთოდებს) კლასის გარეთ ვანხორციელებთ, მაშინ ყველა მეთოდს სჭირდება თარგის პარამეტრების მითითება. თარგის დამატებითი პარამეტრების მითითება ხშირად კლასის შიგნით იმპლემენტირებულ მეთოდებშიც ხდება საჭირო. ასეთ შემთხვევებს შევხვდებით, მაგალითად, შეტანა-გამოტანის და არითმეტიკული ოპერატორების გადატვირის განხილვისას.

<<< თარგი, სპეციალიზაცია და ნაწილობრივი სპეციალიზაცია

ვთქვათ, დაგვჭირდა ნამდვილი რიცხვების ვექტორი. გავაკეთეთ განაცხადი:

```
	vector<double> v{ 1,2.2,4.01,2,1 };
```

ვექტორის არის კლასის თარგი. ტიპი არის თარგის პარამეტრი. ყოველ კონკრეტულ შემთხვევებში, როდესაც ასეთი კლასი იქმნება, ხდება ამ ობიექტის სპეციალიზაცია მითითებულ კონკრეტულ ტიპზე. ამ შემთხვევაში, v ობიექტის სპეციალიზაცია არის double. როდესაც კლასს ან ფუნქციის თარგს ერთი პარამეტრი აქვს, ყველაფერი მარტივია და ამ მაგალითს გავს.

ამავე დროს, ბევრად ხშირია შემთხვევა, როდესაც კლასის ან ფუნქციის თარგს ბევრი პარამეტრი აქვს. ვნახოთ თუ რაები შეიძლება მოხდეს ამ დროს. განვიხილოთ კოდი:

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

```
template<typename T, typename S>
void coolFunction(T a, S b)
{
	std::cout << "a and b  " << a << "  " << b << std::endl;
}

template<typename S>
void coolFunction(std::string a, S b)
{
	std::cout << "Here, First parameter is a string" << std::endl;
}


int main()
{
	string s{ "Koba" };
	coolFunction("Koba", 2020);
	coolFunction(s, 2020);
}
```

აქ coolFunction ფუნქციის ორი გადატვირთვა გვაქვს. პირველი არის სრული სპეციალიზაციით. მეორე - ნაწილობრივი სპეციალიზაციით. თარგის ორი პარამეტრიდან მხოლოდ ერთი სპეცილიზდება, მეორე - დაფიქსირებულია. გამოძახებაც არაა ძალიან გულუბრყვილო: პირველ გამოძახებაში თარგის პარამეტტრის ამოცნობა ხდება სიმბოლოების მუდმივ მასივად და არა სტრინგად. ამიტომ სრულ სპეციალიზაციას აკეთებს.

სულ ესაა. რაც არ უნდა რთული ჩანდეს თარგის პარამეტრების სპეციალიზაციის კოდი, პრინციპი ისაა რაც აქ ვთქვით.

თუმცა არის ერთი რამ. თარგების საშუალებით პროგრამისტი ქმნის ე.წ. განზოგადებულ (generic) კოდს. ხშირად, როდესაც ენას ემატება გარკვეული ინსტრუმენტი გარკვეული მიზნით, სინამდვილეში მას კიდევ უფრო მეტის გაკეთებაც შეუძლია. მაგალითად, თარგის პარამეტრი შეიძლება იყოს ტიპიც (თუმცა ზუსტად ტიპისთვის გაკეთდა ეს), შეიძლება იყოს სხვა რაღაცეებიც. განვიხილოთ კარგად ცნობილი მაგალითი:

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

template <int N>
struct Factorial
{
	static int getValue() { return N*Factorial<N - 1>::getValue(); }
};

template <>
struct Factorial<0>
{
	static int getValue() { return 1; }
};

// Factorial<4>::value == 24
// Factorial<0>::value == 1
int main()
{
	int x = Factorial<4>::getValue(); // == 24
	int y = Factorial<0>::getValue(); // == 1
	cout << x << "  " << y << endl;
}
```

აი ასეც შეიძლება ფაქტორიალის გამოთვლა. თურმე.

<<< სავარჯიშოები:

1. გადატვირთეთ თარგიანი ფუნქცია, რომელმაც უნდა დააბრუნოს მინიმუმი ორ, ან სამ ობიექტს შორის.
2. შექმენით თარგიანი ფუნქცია, რომელმაც უნდა დააბრუნოს მაქსიმუმი ორ ან სამ ობიექტს შორის.
3. შექმენით თარგიანი ფუნქცია, რომელმაც უნდა დააბრუნოს პარამეტრად გადაცემული სამი რიცხვის საშუალო არითმეტიკული.
4. შექმენით თარგიანი ფუნქცია, რომელმაც უნდა დააბრუნოს პარამეტრად გადაცემული ვექტორის საშუალო არითმეტიკული.
5. გადატვირთეთ თარგიანი ფუნქცია, რომელმაც უნდა დააბრუნოს პარამეტრად გადაცემული სიის საშუალო არითმეტიკული.
6. შექმენით თარგიანი ფუნქცია, რომელმაც უნდა დააბრუნოს ვექტორის ელემენტების ნამრავლი.
7. შექმენით თარგიანი ფუნქცია, რომელმაც უნდა დაბეჭდოს სხვადასხვა ტიპის ვექტორები.
8. შექმენით თარგიანი კლასი სამეულებისთვის, რომელიც ბიბლიოთეკის `pair<,>` კლასის ანალოგიური იქნება.