ŠABLONAI IR PARAMETRIZAVIMAS

Kaip jau minėta, šablonai leidžia aprašyti apibendrintas funkcijas ir klases, kurias galima panaudoti su įvairiais duomenimis. Konkrečių duomenų tipų pateikimas šabloninei funkcijai ar klasei vadinamas parametrizavimu, o pačios funkcijos (klasės) vadinamos parametrizuotomis.

Funkcijos šablono (angl. *template*) skelbimo sintaksė:

template <class T_1, class T_2, …, class T_n>
<tipas> <funkcijos_vardas> (parametrų sąrašas)
{
// funkcijos tekstas
}

Kiekvienas šablono parametras aprašomas arba baziniu žodžiu *class* ir tipo vardu, arba tipo vardu ir identifikatoriumi. Parametrizuotiems tipams aprašyti vietoj *class* galima naudoti bazinį žodį *typename*. Naujuose standartuose (ir naujuose kompiliatoriuose) vartojamas būtent pastarasis bazinis žodis.

Šablonai leidžia sukurti apibendrintas funkcijas ir klases, neprisirišant prie konkrečių duomenų tipų. Jie taip pat naudojami kuriant universalias bibliotekas (pvz., STL) ir tokiu būdu užtikrina daugkartinį funkcijų ar klasių panaudojimą.

Pavyzdys 3.21. Šabloninių funkcijų naudojimas C++.

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

template <class T>
T Sqr(T x)
{
    return x*x
}

template <class T>
T* Swap(T* t, int ind1, int ind2)
{
    T temp = t[ind1];
    t[ind1] = t[ind2];
    t[ind2] = temp
```

```
    return t;
}

int main ()
{
    int n=10, sq_n, i = 2, j = 5;
    double d = 20.10, sq_d;
    char* str = „Šablonas“;

    sq_n = Sqr(n);
    sq_d = Sqr(d);

    cout << “reiksme n =” << n <<  endl << “n kvadratas =”<<sq_n << endl;
    cout << “reiksme d =” << d <<  endl << “d kvadratas =”<<sq_d << endl;
    cout << “pradine eilute = ‘ ” << str <<  “ ‘ “ << endl;
    cout << “pakeista eilute = ‘ ” << Swap(str, i, j) <<  “ ‘ “ << endl;
    return 0;
 }
```

Šios programos vykdymo rezultatas:
reiksme n = 10
n kvadratas = 100
reiksme d = 20.10
d kvadratas = 404.01
pradine eilute = ‘Šablonas‘
pakeista eilute = ‘Šanlobas‘

Kiekvienas šabloninės funkcijos iškvietimas – tai nurodymas kompiliatoriui sugeneruoti funkcijos kodą konkrečiam duomenų tipui. Todėl šis procesas, pvz., *sq_d = Sqr(d),* vadinamas šabloninės funkcijos *konkretizavimu*.

Apibendrinant, matome, kad šablonai leidžia aprašyti apibendrintas funkcijų ir klasių specifikacijas, t.y. parametrizuojamas funkcijas ir klases (angl. *generic functions*, *generic classes*). Konkrečios kompiliatoriaus kuriamos funkcijos realizacijos procesas vadinamas konkretizacija (angl. *instantiating*).

PARAMETRIZAVIMO PASKIRTIS. ALGORITMŲ PARAMETRIZAVIMAS.

Yra nemažai algoritmų, kurių logika nepriklauso nuo duomenų tipų, t.y. tokį algoritmą galima būtų įvykdyti su įvairių tipų duomenimis. Pvz., tipiniuose rūšiavimo algoritmuose vyksta dviejų elementų palyginimas, jų tvarkos nustatymas ir, jei reikia, elementų apsikeitimas vietomis. Dviejų

kintamųjų reikšmių apsikeitimas iš esmės nepriklauso nuo duomenų tipo *DataType*:

```
DataType tmp, a, b;
{       tmp=a;
        a=b;
        b=tmp;
}
```

Tačiau, nors algoritmas ir bendras, daugumoje programavimo kalbų kiekvienam tipui reikės sukurti atskirą šios procedūros versiją. Šią problemą buvo bandoma spręsti dar iki šablonų sukūrimo, juolab, kad ir C++ kalboje šablonų pradžioje nebuvo, jie atsirado kiek vėliau ir buvo įtraukti į ANSI C++ kalbos standartą tik 90-tais metais.

Pirmas parametrizavimo būdas – tai funkcijos pavidalo makro apibrėžimų (makrosų) naudojimas, pvz., mūsų atveju:

```
#define swap(a,b) { tmp=a; a=b; b=tmp; }
```

Kadangi kompiliatorius vykdo makroišplėtimą tik keisdamas tekstą, šis makrosas (ir kiti panašūs) gali būti naudojamas su skirtingais duomenų tipais. Tačiau, tai nėra labai patogu ir sunku užtikrinti tokio makroso darbą su vartotojo apibrėžtais duomenų tipais. Bet, blogiausia tai, kad čia neveikia jokia tipų kontrolė, tad galimas klaidingas taikymas.

Antras parametrizuotų funkcijų konstravimo būdas – perduoti jai parametrais ne tik pačius duomenis, bet ir jų duomenų tipą (t.y. duomenų ilgį baitais). Tačiau ir čia duomenų tipų kontrole turi rūpintis pats programuotojas, o papildomi funkcijos parametrai reiškia perteklinio kodo generavimą. Jei tokia funkcija dažnai naudojama (pvz., rekursinės funkcijos iškvietimai), programos efektyvumas ženkliai krenta.

Galu gale, galima būtų parašyti perkraunamas funkcijas kiekvienam reikalingam duomenų tipui (kaip buvo parodyta anksčiau). Tačiau, vietoj kelių tokių funkcijų žymiai patogiau aprašyti vieną šabloninę funkciją. Tokia funkcija apibrėžia algoritmą realizuojantį operacijų rinkinį, kuris gali būti taikomas įvairių tipų duomenims. Konkretus tipas perduodamas funkcijai kaip parametras kompiliavimo metu (jis priskiriamas apibendrintam šablonіniam tipui, pvz. 3.21 – tipui T). Kompiliatorius automatiškai sugeneruos teisingą kodą, kuris atitinka tipą, naudojamą

funkcijos iškvietime (todėl svarbu, kad kompiliatorius galėtų jį tiksliai nustatyti).

Pavyzdžiui, funkcijos *swap*, kurią galima iškviesti su bet kokio tipo duomenimis, šablonas:

```
template <class SwapType>
void swap(SwapType &a, SwapType &b)
{       SwapType tmp;
        tmp=a;
        a=b;
        b=tmp;
}
```

Tokiu būdu, lyginant su kitais aptartais parametrizavimo būdais, šablonų sukūrimas leido užtikrinti:

- tipų kontrolę ir apsaugą,
- aukštą parametrizuotų programų efektyvumą,
- naudojimo paprastumą, standartizavimą ir kodo skaidrumą.

Pavyzdys 3.22. Rūšiavimo algoritmų parametrizavimas.

Realizuokime elementų rūšiavimą paprastu „burbulo“ metodu. Priminsime, kad jame mažiausi masyvo elementai (“lengviausi”) išplaukia į paviršių, sunkiausi – skenda. Burbulo metodo algoritmas peržiūri visą masyvą “iš apačios į viršų” ir keičia greta esančius elementus vietomis, jei apatinis elementas mažesnis už viršutinį. Taip į paviršių “išplauks” mažiausias masyvo elementas. Toliau šis algoritmas kartojamas kitiems *N-1* elementams.

```
//Paprasta burbulo metodo algoritmo realizacija simbolių rūšiavimui
#include <iostream>
#include <cstring>

using namespace std;

void bubble(char* elem, int skaicius)
{
        register int a,b;
        char tmp;
        for (a=1; a<skaicius; ++a)
```

```
                    for (b=skaicius-1; b>=a; --b)
                        { if (elem[b-1] > elem[b])
                            { tmp = elem[b-1];
                              elem[b-1] = elem[b];
                              elem[b] = tmp; }
                        }
}
int main ()
{
char str[ ] = “ecfabdhg“;
bubble (str, (int) strlen(str));
cout << “Surūšiuota eilutė: ” << str <<  endl;
return 0;
 }
```

Šios programos vykdymo rezultatas:
Surūšiuota eilutė: abcdefgh

Dabar transformuokime burbulo metodo algoritmą į šabloną ir parametrizuokime rūšiuojamų duomenų tipą.

```
//Parametrizuota burbulo metodo algoritmo realizacija
#include <iostream>
#include <cstring>

using namespace std;

//Paremetrizuota funkcija
template <class Stype>
void bubble(Stype *elem, int skaicius)
{
        register int a,b;
        Stype tmp;
        for (a=1; a<skaicius; ++a)
                    for (b=skaicius-1; b>=a; --b)
                        { if (elem[b-1] > elem[b])
                            { tmp = elem[b-1];
                              elem[b-1] = elem[b];
                              elem[b] = tmp; }
                        }
}
int main ()
{
//simbolių eilutės rūšiavimas
char str[ ] = “ecfabdhg“;
bubble (str, (int) strlen(str));
```

```
cout << “Surūšiuota eilutė: ” << str <<  endl;

//sveikojo tipo masyvo elementų rūšiavimas
int sv_mas[ ] = {7, 6, 1, 5, 10, 2, 5, 4, 3};
int i;
bubble (sv_mas, 9);
cout << “Surūšiuotas sveikas masyvas: ”;
for (i=0; i<9; i++) cout<< sv_mas[i]  <<  “   ”;
return 0;
 }
```

Šios programos vykdymo rezultatas:

Surūšiuota eilutė: abcdefgh
Surūšiuotas sveikas masyvas: 1 2 3 4 5 5 6 7 10

PARAMETRIZUOTOS KLASĖS.

Šablonai leidžia kurti ne tik apibendrintas funkcijas bet ir klases. Tokia parametrizuota klasė apibrėžiama vieną kartą, o duomenų tipas (tipai), kuriuo ji operuoja, nurodomas kaip parametras (-ai) kuriant šios klasės objektą.

Bendra klasės šablono forma:

template <class Ttipas >
class *klasės vardas*
{
// klasės aprašas
}

Kai parametrizuota klasė yra apibrėžta, jos konkreti realizacija kuriama tokia forma

klasės vardas <tipas> *objekto vardas;*

čia <tipas> – tai duomenų tipas, su kuriuo duomenimis klasė faktiškai dirba, t.y. tipas pakeičia parametrą Ttipas. Parametrizuotos klasės funkcijos-nariai automatiškai tampa parametrizuotomis funkcijomis (jas parašant nebūtina naudoti *template*),

Klasių šablonai vis dažniau naudojami kuriant universalias bibliotekas (pvz., STL). Tai yra todėl, kad jos labai patogu taikyti kuriant vadinamas *konteinerių klases*. *Konteineris* – tai klasė, kurios objektuose saugomas

organizuotas elementų (duomenų) rinkinys. Prieiga prie elementų yra reglamentuota tam tikromis apibrėžtomis taisyklėmis. Paprastai konteineriai realizuoja sudėtines duomenų struktūras: masyvus, vektorius, sąrašus, eiles, stekus, dekus, medžius ir kt. Vieną kartą apibrėžtas ir suderintas parametrizuotas konteineris gali būti daug kartų panaudotas be pakeitimų, pvz., rišlaus sąrašo konteineris gali būti taikomas kuriant pašto adresų, knygų kartotekų, atsarginių dalių ir t.t. sąrašus.

Pavyzdys 3.23. Parametrizuota klasė. Steko realizacija.

Stekas – tai konteinerio (sąrašo) duomenų struktūra su LIFO (Last In First Out) aptarnavimo disciplina. Bazinė steko realizacija įgyvendina tokius veiksmus:

- tuščio steko sukūrimas (konstravimas, inicializacija);
- elemento išsaugojimas steke - operacija *push;*
- elemento ištraukimas iš steko - operacija *pop.*

Steką galima realizuoti įvairiais būdais (dinaminiu sąrašu, masyvu ir kt.). Čia parametrizuotoje klasėje *stack* steką talpiname dinaminėje atmintyje, masyve, į kurį rodo rodyklė *stck*. Kintamasis *tos* – tai steko viršūnė, *length* – steko dydis.

```
//Parametrizuota steko klasė
#include <iostream>
#include <cstdlib>
using namespace std;

//Parametrizuota klasė
template <class Stype>
class stack
{
        Stype *stck;
        int tos;
        int length;
public
        stack(int size);
        ~ stack() {delete[] stck;};
        void push(Stype i);
        Stype pop();
};
//Steko konstruktorius
template <class Stype>
stack< Stype >::stack(int size)
{       stck = new Stype[size];
```

```
        if (!stck) { cout << “Negalima sukurti steko ” << endl; exit(1);}
        length = size;
        tos = 0;
}

template <class Stype>
void stack< Stype >::push(Stype elem)
{       if (tos==length) { cout << “Stekas uzpildytas ” << endl; return;}
        stck[tos] = elem;
        tos++;
}

template <class Stype>
Stype stack< Stype >::pop()
{       if (tos==0) { cout << “Stekas tuscias ” << endl; return 0;}
        tos--;
        return stck[tos];
}

int main ()
{
stack<int> a(10);           //sveikasis stekas
stack<double> b(10);        //double tipo stekas
stack<char> c(10);          //simbolių stekas
int i;

//skaiciu stekai
a.push(10);
b.push(50.25);
a.push(20);
b.push(100-20.75);

cout << a.pop() << “  ”;
cout << a.pop() << “  ”;
cout << b.pop() << “  ”;
cout << b.pop() << “  ”;

//simboliu stekas
for (i=0; i<10; i++) c.push((char)‘A‘ + i);
for (i=0; i<10; i++) cout  <<  c.pop();
cout <<  endl;
return 0;
 }
```

Šios programos vykdymo rezultatas:

20 10 79.25 50.25 JIHGFEDCBA